Trusted Postgres Architect

Version 23.44.0

Built at 2026-06-10T16:56:45

1 Trusted Postgres Architect

Introduction

TPA is an orchestration tool that uses Ansible to deploy Postgres clusters according to EDB's recommendations.

TPA embodies the best practices followed by EDB, informed by many years of hard-earned experience with deploying and supporting Postgres. These recommendations are as applicable to quick testbed setups as to production environments.

What can TPA do?

TPA is built around a declarative configuration mechanism that you can use to describe a Postgres cluster, from its topology right down to the smallest details of its configuration.

Start by running `tpaexec configure` to generate an initial cluster configuration based on a few high-level choices (e.g., which version of Postgres to install). The default configuration is ready to use as-is, but you can edit it to suit your needs (the generated configuration is just a text file, config.yml).

Using this configuration, TPA can:

1. Provision servers (e.g., AWS EC2 instances or Docker containers) and any other resources needed to host the cluster (or you can deploy to existing servers or VMs just by specifying connection details).

2. Configure the operating system (tweak kernel settings, create users and SSH keys, install packages, define systemd services, set up log rotation, and so on).

3. Install and configure Postgres and associated components (e.g., PGD, Barman, pgbouncer, repmgr, and various Postgres extensions).

4. Run automated tests on the cluster after deployment.

5. Deploy future changes to your configuration (e.g., changing Postgres settings, installing and upgrading packages, adding new servers, and so on).

How do I use it?

To use TPA, you need to install it and run the `tpaexec setup` command. Follow the installation instructions for your platform.

TPA operates in four distinct stages to bring up a Postgres cluster:

- Generate a cluster configuration
- Provision servers (VMs, containers) to host the cluster
- Deploy software to the provisioned instances
- Test the deployed cluster


```
# 1. Configuration: decide what kind of cluster you want
tpaexec configure clustername --architecture M1 --platform aws \
        --postgresql 14 \
        --failover-manager repmgr

# 2. Provisioning: create the servers needed to host the cluster
tpaexec provision clustername

# 3. Deployment: install and configure the necessary software
tpaexec deploy clustername

# 4. Testing: make sure everything is working as expected
tpaexec test clustername
```

You can run TPA from your laptop, an EC2 instance, or any machine that can reach the cluster's servers over the network.

Here's a list of capabilities and supported software.

Configuration

The `tpaexec configure` command generates a simple YAML configuration file to describe a cluster, based on the options you select. The configuration is ready for immediate use, but you can modify it to better suit your needs. Editing the configuration file is the usual way to make any configuration changes to your cluster, both before and after it's created.

At this stage, you must select an architecture and a platform for the cluster. An**architecture** is a recommended layout of servers and software to set up Postgres for a specific purpose. Examples include "M1" (Postgres with a primary and streaming replicas) and "PGD-Always-ON" (EDB Postgres Distributed 5 in an Always On configuration). A **platform** is a means to host the servers to deploy any architecture, e.g., AWS, Docker, or bare-metal servers.

Provisioning

The `tpaexec provision` command creates instances and other resources required by the cluster. The details of the process depend on the architecture (e.g., M1) and platform (e.g., AWS) that you selected while configuring the cluster.

For example, given AWS access with the necessary privileges, TPA will provision EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, etc.

You can also "provision" existing servers by selecting the "bare" platform and providing connection details. Whether these are bare metal servers or those provisioned separately on a cloud platform, they can be used just as if they had been created by TPA.

You are not restricted to a single platform—you can spread your cluster out across some AWS instances (in multiple regions) and some on-premise servers, or servers in other data centres, as needed.

At the end of the provisioning stage, you will have the required number of instances with the basic operating system installed, which TPA can access via SSH (with sudo to root).

Deployment

The `tpaexec deploy` command installs and configures Postgres and other software on the provisioned servers (which may or may not have been created by TPA; but it doesn't matter who created them so long as SSH and sudo access is available). This includes setting up replication, backups, and so on.

At the end of the deployment stage, Postgres will be up and running.

Testing

The `tpaexec test` command executes various architecture and platform-specific tests against the deployed cluster to ensure that it is working as expected.

At the end of the testing stage, you will have a fully-functioning cluster.

Incremental changes

TPA is carefully designed so that provisioning, deployment, and testing are idempotent. You can run through them, make a change to config.yml, and run through the process again to deploy the change. If nothing has changed in the configuration or on the instances, then rerunning the entire process will not change anything either.

Cluster management

Once your cluster is up and running, TPA provides convenient cluster management functions, including configuration changes, switchover, and zero-downtime minor-version upgrades. These features make it easier and safer to manage your cluster than making the changes by hand.

Extensible through Ansible

TPA supports a variety of configuration options, so you can do a lot just by editing config.yml and re-running provision/deploy/test. If you do need to go beyond what TPA already supports, you can write

- Custom commands, which make it simple to write playbooks to run on the cluster. Just create `commands/xyz.yml` in your cluster directory, and invoke it using `tpaexec xyz /path/to/cluster`. Ideal for any management tasks or processes that you need to automate.

- Custom tests, which augment the builtin tests with in-depth verifications specific to your environment and application. Using `tpaexec test` to run all tests in a uniform, repeatable way ensures that you will not miss out on anything important, either when dealing with a crisis, or just during routine cluster management.

- Hook scripts, which are invoked during various stages of the deployment. For example, tasks in `hooks/pre-deploy.yml` will be run before the main deployment; there are many other hooks, including `post-deploy`. This places the full range of Ansible functionality at your disposal.

It's just Postgres

TPA can create complex clusters with many features configured, but the result is just Postgres. The installation follows some conventions designed to make life simpler, but there is no hidden magic or anything standing in the way between you and the database. You can do everything on a TPA cluster that you could do on any other Postgres installation.

Versioning in TPA

TPA previously used a date-based versioning scheme whereby the major version was derived from the year. From version 23 TPA transitioned to semantic versioning, initially using a two-part `major-minor` pattern, then adopting full three-part semantic versioning in version 23.34.1. Under this scheme, the major version is only incremented where required to comply with the backward compatibility principle below.

Backwards compatibility

A key development principle of TPA is to maintain backwards compatibility so there is no reason for users to need anything other than the latest version of TPA. We define backwards compatibility as follows:

- A config.yml created with TPA X.a will be valid with TPA X.b where b>=a
- The cluster created from that config.yml will be maintainable and re-deployable with TPA X.b

  Therefore, a new major version implies a break in backward compatibility. As such, we aim to avoid releasing major versions and will only do so in exceptional circumstances.

Getting started

Follow the TPA installation instructions for your system, then configure your first cluster.

2 Trusted Postgres Architect release notes

The Trusted Postgres Architect documentation describes the latest version of Trusted Postgres Architect 23.

Trusted Postgres Architect version	Release Date
23.44.0	10 Jun 2026
23.43.0	18 Mar 2026
23.42.0	25 Feb 2026
23.41.0	26 Nov 2025
23.40.1	28 Oct 2025
23.40.0	02 Oct 2025
23.39.0	22 Aug 2025
23.38.1	25 Jun 2025
23.38.0	09 Jun 2025
23.37.0	24 Mar 2025
23.36.0	19 Feb 2025
23.35.0	25 Nov 2024
23.34.1	09 Sep 2024
23.34	22 Aug 2024
23.33	24 Jun 2024
23.32	15 May 2024
23.31	19 Mar 2024
23.30	19 Mar 2024
23.29	15 Feb 2024
23.28	23 Jan 2024
23.27	19 Dec 2023
23.26	30 Nov 2023
23.25	14 Nov 2023
23.24	17 Oct 2023
23.23	21 Sep 2023
23.22	06 Sep 2023
23.21	05 Sep 2023
23.20	01 Aug 2023
23.19	12 Jul 2023
23.18	23 May 2023
23.17	10 May 2023
23.16	21 Mar 2023
23.15	15 Mar 2023
23.14	23 Feb 2023
23.13	22 Feb 2023
23.12	21 Feb 2023
23.1-11	21 Jun 2023 to 31 Jan 2023

2.1 Trusted Postgres Architect 23.44.0 release notes

Released: 10 June 2026

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.44.0 include the following:

Highlights

- Support for upgrading components in a PGD-X cluster
- Improved robustness of PGD upgrades
- The EFM user is no longer a superuser by default

Enhancements

Description	Addresses
TPA now supports upgrade of components in the PGD-X architecture. In the upgrade playbook for PGD-X, additional steps that ensure the upgrade of selected components (PgBouncer, pg-backup-api, Barman, PEM server) via tpaexec (`tpaexec upgrade (...) --components=<component>`) have been included, so individual components can be upgraded without the need to upgrade the entire cluster. This upgrade process is included as a standalone feature for component upgrades, as well as minor PGD-X upgrades and major upgrades to the PGD-X architecture.	
TPA can now deploy the EFM database user with only the minimum required permissions. On a regular EFM deployment, TPA granted the EFM user SUPERUSER privileges. A new variable called `efm_user_is_superuser` has been introduced, where TPA deploys the EFM user as a Postgres SUPERUSER, or as a regular login user that is a member of `efm_role` , a role created by TPA with only the specific catalog functions and predefined roles (pg_monitor, pg_read_all_settings, pg_read_all_stats) that EFM requires (the variable is set to `false` by default when creating new clusters via `tpaexec configure` ; removing this variable from the configuration file causes `efm_user_is_superuser` to be treated as `true`). The variable acts as a switch and can be toggled with a regular deployment, and can be used on already deployed clusters to switch the privileges of the EFM user.	
TPA now supports Ansible Automation Platform 2.6. TPA is now tested with AAP 2.6. Existing execution environments are compatible with both AAP 2.4 and AAP 2.6.	
Added support for proxy monitoring during PGD 5 to PGD 6 upgrades. The `enable_proxy_monitoring=yes` option to `tpaexec upgrade` can now be used during upgrades from PGD-Always-ON (PGD 5) to PGD-X (PGD 6). Previously it would cause the upgrade to fail immediately because the monitor attempted to connect to the Connection Manager port on witness nodes, which do not run Connection Manager. Witness nodes are now excluded from the list of endpoints the monitor targets.	

Description	Addresses
TPA now supports non-default hugepage sizes. TPA's hugepages settings previously assumed the architecture's default page size (2MB on x86_64), so a configuration that wanted Postgres to use 1GB hugepages required the user to set the kernel command line, the `vm.nr_hugepages` sysctl, and the `huge_page_size` GUC by hand, and the values TPA generated worked against them. TPA now supports a `huge_page_size` variable (a Postgres-style memory string such as `1GB`). When set, TPA reserves pages of that size on the kernel command line with `hugepagesz=`, omits `vm.nr_hugepages` from `/etc/sysctl.conf` (which only ever applies to the architecture's default-size pool), and sets `huge_page_size` in `postgresql.conf` so Postgres draws from the chosen pool. The page count can be overridden with a new `nr_hugepages` variable; for backwards compatibility, an existing `sysctl_values['vm.nr_hugepages']` is still honoured when `nr_hugepages` is not set explicitly. Because every reservation is taken out of normal memory at boot, `huge_page_size` is usually best set on individual Postgres instances rather than cluster-wide.	
Added an environment variable option to the repmgr service unit file. A new variable called `repmgr_service_environment` has been added to the repmgr service unit template file for repmgr. This variable allows users to specify custom environment variables that will be set for the repmgr service. The `repmgr_service_environment` variable can be defined in the cluster configuration file, as a dictionary where the keys are the names of the environment variables and the values are the corresponding values for those variables. This is useful in environments where the repmgr process requires access to specific runtime values, such as custom library paths or authentication credentials.	56200
Added supported cluster_vars overrides for TPA's choice of Python interpreter and packages. Two new (or newly-effective) cluster_vars now control how TPA uses Python on cluster nodes: 1. `script_python_interpreter` — templates the shebang line of TPA- installed Python scripts (for example `/etc/tpa/postgres-monitor`). Defaults to `/usr/bin/env {{ python }}`, preserving the previous PATH-based behaviour. Users can set it in `cluster_vars` to an absolute interpreter path to pin scripts deterministically — useful on hosts that have more than one Python installed and where the default `env` lookup would find the wrong one. 2. `python_pkg_prefix` — controls the name prefix TPA uses when constructing OS package names (for example `python3-psycopg2` versus `python311-psycopg2`). It was already an internal fact, but setting it in `cluster_vars` previously had no effect because `minimal_setup`'s output overwrote it. It is now passed through to `minimal_setup` as a module parameter, so a user-supplied value is honoured. See `docs/src/python.md` for details, including the recommendation to couple the two variables on any host that has more than one Python interpreter installed.	
TPA now supports enable_proxy_monitoring during PGD 6 minor upgrades. The `enable_proxy_monitoring=yes` option to `tpaexec upgrade` now works for PGD 6 to PGD 6 minor upgrades, in addition to the major upgrade paths where it was already supported. When enabled, TPA records Connection Manager downtime on each data node during the rolling fence/restart cycle.	
Improved restart and service excluded_tasks coverage. TPA now respects the `restart` and `service` values of `excluded_tasks` across all roles that perform service restarts or systemd service management, including Barman, beacon-agent, EFM, etcd, harp, Patroni, PEM (agent and server), pgbackupapi, PgBouncer, PGD Proxy, Postgres, repmgr (restart, switchover, replica final), rsyslog, and OpenVPN. Previously, users excluding `restart` or `service` tasks could still see unintended restarts in these areas during a deploy. With this change, all restart and direct service operations are guarded consistently, so excluded_tasks reliably suppresses them.	

Description	Addresses
`tpaexec reconfigure` now accepts --bdr-package-version. tpaexec reconfigure now accepts --bdr-package-version on BDR-Always-ON, PGD-Always-ON, PGD-X, and PGD-S clusters, recording the value in config.yml and applying any version-gated configuration options the chosen version requires. Currently the only such option is read_listen_port in default_pgd_proxy_options for PGD 5.5+, matching the behaviour of `tpaexec configure` for PGD 5.5 clusters. The new logic also runs as a require() of the BDR-Always-ON → PGD-Always-ON and PGD-Always-ON → PGD-X architecture changes, so those upgrade paths pick up the version-gated options when no --bdr-package-version is supplied.	
TPA now supports enable_proxy_monitoring for PGD 5 minor upgrades. The enable_proxy_monitoring option now works during PGD 5 minor version upgrades in PGD-Always-ON clusters. Previously, proxy monitoring was only supported during BDR 4 to PGD 5 major upgrades. When enabled, the proxy monitor tracks connection availability through pgd-proxy endpoints throughout the upgrade process and reports any interruptions.	

Changes

Description	Addresses
Replaced upgrade_legacy.yml with a dedicated upgrade_major_4to5.yml playbook. The BDR 4→5 major upgrade logic has been extracted from the monolithic upgrade_legacy.yml into a dedicated playbook. This simplifies the upgrade path by removing all conditional branching, hardcoding harp as the failover manager and upgrade_from as version 4. The now-orphaned upgrade_legacy.yml files have been removed from PGD-Always-ON and Lightweight architectures.	
Added a dedicated upgrade_minor_5.yml playbook for BDR 5.x minor upgrades. This new playbook handles all BDR 5.x minor upgrades. This uses the relevant logic previously embedded in the monolithic upgrade_legacy.yml, making the upgrade process easier to maintain.	
TPA no longer passes empty --team and --owner to pemworker. When registering a Postgres server with PEM, TPA previously passed `--team ""` and `--owner ""` to `pemworker --register-server` if the `monitoring_team` or `monitoring_server_owner` variables were not set. Earlier PEM versions treated these empty strings as equivalent to omitting the option, but PEM 10.5 tightens CLI validation and rejects empty strings, causing the registration task to fail. TPA now omits these flags entirely when the corresponding variables are unset, so server registration succeeds against PEM 10.5 without requiring any configuration change.	
Documented how to run custom playbooks on AAP. The AAP user documentation now explains how to add custom playbooks to a cluster and run them through AAP, using the `commands/` subdirectory convention and AAP's Duplicate template action.	
Added missing flags to the tpaexec help output. 'tpaexec help' was missing some flags that are actually supported. This change adds those flags to the help output.	
TPA now rejects pgaudit on EPAS clusters at the start of deploy. Including `pgaudit` in `extra_postgres_extensions` on an EPAS cluster used to result in a deployment that failed late at Postgres startup, because pgaudit is not loadable on EPAS. TPA now detects this combination at the start of deploy and fails with a message recommending EPAS's built-in audit features.	

Description	Addresses
Added {stop,start,list}-containers to the tpaexec --help output. The already implemented options for managing docker containers in tpaexec were featured in the documentation but not in the output of `tpaexec --help`. This change ensures that the `tpaexec --help` command contains a brief summary of what these commands do.	

Bug Fixes

Description	Addresses
Fixed an issue whereby tpaexec reconfigure would not add postgres_distributed during BDR 4 to BDR 5 upgrades. The `tpaexec reconfigure` command, when upgrading from BDR 4 to BDR 5, didn't add the `postgres_distributed` repository, which would result in a failure when trying to download the new BDR 5 packages. The fix ensures that the `postgres_distributed` repository is included in the configuration.	
Fixed an issue whereby standby nodes could be promoted to BDR primary candidates. An issue was found where standby nodes could be selected as BDR primary candidates during deployment, which could lead to unintended consequences in some cluster scenarios (for example, when joining BDR node groups via standby nodes). This fix ensures that standby nodes are excluded from the list of potential BDR primary candidates.	
Fixed an issue whereby SLES 15 deployments failed when featuring patroni with EDBPGE and EPAS. When selecting patroni as the failover manager with the EDBPGE/EPAS PostgreSQL flavour, the modules needed to install etcd-related packages were missing, and some additional adjustments were required to make the deployment work correctly. This fix ensures that the required packages are installed when deploying on SLES 15 with patroni as the failover manager, alongside some minor adjustments.	
Fixed an issue where requesting PEM agent <9.6 would cause deployment to fail. TPA uses the `pemworker --enable-probe` option to enable EFM probes where required. However this option does not exist before PEM 9.6, so trying to call it causes an error. This fix adds a version check to this task meaning it will be skipped when the agent version is <9.6.	
Fixed an issue whereby pgd-cli tasks would run when pgd-cli was not upgraded. During a postgres-only upgrade (the default), the tasks that log pgd-cli diagnostic output and wait for write leader elections were always executed on the first BDR primary, even when pgd-cli had not been upgraded as part of that run. This caused the upgrade to fail with "unknown command show-groups" because the old pgd-cli binary does not support that subcommand. Both tasks are now skipped unless pgdcli or all components are included in the upgrade.	
Updated default AWS AMIs to currently-available images. The default AWS AMIs that `tpaexec configure` selects for officially supported platforms have been updated to currently-available images. Previously several AMIs (notably Ubuntu 22.04, Ubuntu 24.04, RHEL 8.10 and SLES 15 SP7) had been deregistered by their vendors, causing `tpaexec provision` to fail for new clusters on those platforms. RHEL 9 and Rocky 9 also move from minor version 9.5 to 9.7, since 9.5 is no longer the current minor and updated 9.5 AMIs are not being published.	

Description	Addresses
TPA now derives bdr_version_num automatically from installed BDR/PGD packages. TPA uses bdr_version_num to make precise decisions about CAMO configuration. Previously this value was obtained only by querying the running database, so on an initial deploy (when Postgres is not yet running) parts of the CAMO configuration could not be rendered correctly, and users had to set bdr_version_num manually in config.yml as a workaround. TPA now derives bdr_version_num from the version of the installed BDR or PGD package on each node, so no manual setting is required. A value set in config.yml continues to take precedence, and once the database is running the precise value reported by bdr.bdr_version_num() remains authoritative.	
Stabilised the post-upgrade health check for PGD 6 minor upgrades. After the last rolling restart of a PGD 6 minor upgrade, BDR replication slots can briefly remain inactive while they re-handshake with the just-updated node. The post-upgrade `pgd cluster show --health` check could fire during that window and report a spurious "Replication Slots Critical" failure. TPA now polls `bdr.group_replslots_details` for up to two minutes waiting for all slots to become active before running the strict health check. If the window is exhausted, the health check runs anyway so genuine problems are still reported.	
Fixed an issue with custom rc-local service creation. TPA creates a custom systemd service file to ensure the rc-local script is running on startup on distributions that don't support it out of the box. This fix changes the location of the service file to comply with conventions and avoid failures with a missing parent folder that could happen with the previously chosen path. This fix also ensures that this service file creation task is only applied when it is actually needed.	
TPA now normalises Postgres Extended to `pgextended` and `edbpge` based on architecture. A fix has been introduced to normalise Postgres Extended flavours based on architecture. When choosing any Postgres Extended variant via `tpaexec configure` (--pgextended, --edbpge, --edb-postgres-extended, or --postgres-flavour pgextended/edbpge), BDR-Always-ON or older architectures now normalise to `pgextended`, whereas newer architectures normalise to `edbpge`. Trying to deploy a `pgextended` flavour cluster on an incompatible architecture will result in an error, preventing misconfiguration.	
Fixed an issue whereby read_listen_port was added to config for BDR versions below 5.5. tpaexec configure added read_listen_port to default_pgd_proxy_options even for older BDR versions. A deploy-time assertion was also added to catch a missing read_listen_port when upgrading to PGD 5.5 or later.	
TPA now rejects invalid cluster names at configure time. Previously, `tpaexec configure` accepted cluster names that contained characters such as periods (for example, when the cluster directory name embedded a version string like `v1.1.0-rc.1`). The configuration was written successfully, but a later `tpaexec provision` then failed with a fatal assertion because the cluster name did not match the required pattern `^[_a-zA-Z0-9-]+$`. `tpaexec configure` now applies the same check up front and rejects invalid names before any cluster directory is created, so the problem is reported immediately rather than several steps later.	
Upgrade playbooks now use systemctl instead of pgrep. Major and minor PGD upgrade playbooks used `pgrep` to decide whether harp-proxy or harp-manager was still running on a node. On hosts where procps was not installed, two of those checks silently misread "binary missing" as "process not running": the BDR 4 → 5 upgrade then skipped `harpctl unmanage cluster` before stopping harp-manager, and the PGD 5 minor upgrade ran the wrong proxy health check. The checks now use `systemctl is-active --quiet`, which works on every distro TPA supports and reflects the authoritative service state.	

Description	Addresses
TPA now omits pgaudit from CIS compliance on EPAS clusters. Previously, `tpaexec configure --compliance cis` unconditionally added `pgaudit` to `extra_postgres_extensions`. This caused startup failures on EPAS, which has built-in audit logging, so `pgaudit` is now omitted on EPAS clusters.	
TPA now fails deploy when pgbouncer and pgd-proxy share a listen port. When an instance had both `pgbouncer` and `pgd-proxy` in its role list and both services were configured to listen on the same port (the default, 6432, on each), only one could bind. pgBouncer typically won the race and pgd-proxy was left in a failed state, but TPA's deploy returned success. The misconfiguration only became visible during the first upgrade, as a `FATAL: SSL required` error from the TLS-protection test connecting to pgBouncer instead of pgd-proxy. TPA now rejects this configuration at deploy time with a clear message naming the affected host and the two variables involved (`pgbouncer_port` and `pgd_proxy_options.listen_port`). Users who co-host pgbouncer and pgd-proxy must set one of them to a non-default value in `config.yml` so the two listeners don't collide.	
TPA now installs pgd-cli on pgd-proxy nodes during the 4to5 upgrade. The BDR-Always-ON to PGD-Always-ON upgrade adds the pgd-proxy role to former harp-proxy nodes, but pgd-cli was never installed on those nodes during the upgrade. The post-upgrade diagnostic then failed with "No such file or directory". The pgdcli-upgrade plays now also target pgd-proxy hosts, so pgd-cli is in place when the diagnostic runs.	
Fixed an issue whereby automatic witness node addition did not work for PGD-X clusters with even data nodes. When configuring a PGD-X cluster with an even number of `--data-nodes-per-location`, `tpaexec configure` now correctly adds a witness node to each location. This ensures Raft consensus can be established without requiring the user to explicitly pass `--add-witness-node-per-location`. The behaviour is now consistent with the PGD-Always-ON architecture and the documented behaviour for PGD-X. `tpaexec configure` now also rejects `--data-nodes-per-location` values below 2, which previously produced an invalid cluster with no data nodes.	59694
TPA now includes the correct mod_wsgi module for PEM server version 10.4.0 and higher. This change ensures that the correct mod_wsgi module is included for PEM server versions 10.4.0 and higher, while preserving the existing behaviour for earlier versions.	
TPA now includes additional modules to deploy EPAS in SLES 15. When deploying a SLES 15 cluster, PackageHub registration is now performed as part of the initial system registration for both EPAS and PEM installations. Additionally, two new modules (sle-module-desktop-applications and sle-module-development-tools) are now enabled for EPAS deployments in order to install dependencies such as libclang13.	
TPA now auto-creates the switch2cm.yml link on older cluster directories. `tpaexec switch2cm` failed with "the playbook: commands/switch2cm.yml could not be found" on clusters that were originally configured with a TPA release predating the `switch2cm` command, because the cluster directory had no link to `architectures/PGD-Always-ON/commands/switch2cm.yml`. `tpaexec switch2cm` now calls `tpaexec relink` automatically when the link is missing, matching the behaviour `tpaexec upgrade` already had for its own command link.	

Description	Addresses
Fixed an issue whereby HARP manager symlink creation failed on nodes 2+ during upgrade. During a rolling upgrade to edbpge, the task that creates a symlink at /var/run/postgresql/.s.PGSQL.<port> for HARP manager to connect to the database would fail on all nodes after the first with "refusing to convert from file to symlink". The destination path could hold a real socket file left behind by the pre-upgrade postgres (either from a flavour migration or from an unclean stop). Adding force: true to the symlink task in all affected upgrade playbooks ensures the symlink is created correctly regardless of the prior state of that path.	
Introduced retry logic between BDR replication slots when upgrading. When upgrading the postgresql package version in a BDR cluster, replication slots may take a few seconds to reconnect, and the upgrade process can have an exit error due to not allocating enough time to the recovery process. The fix adds retry logic to the health check on the upgrade process to allow time for the slots to recover and avoid timing-related failures.	
TPA now reports a clear error when --location-names doesn't match the architecture's requirement. Previously, supplying `--location-names` with the wrong number of names (for example, a single location for a `BDR-Always-ON` cluster using the `bronze` layout, which requires two) caused `tpaexec configure` to abort with an unhandled Python traceback. `tpaexec configure` now compares the supplied location names against the number of locations the chosen architecture and layout actually need, and reports a clear error if they differ. The change applies to the `M1`, `BDR-Always-ON` and `PGD-Always-ON` architectures.	
Fixed an issue whereby PGD-X demanded too many hostnames in --hostnames-from. Configuring a PGD-X cluster with --hostnames-from used to fail with "found only N/16 names matching ..." whenever the supplied file held fewer than 16 names, even when the cluster only needed a handful. The underlying PGD-X architecture now computes the real number of instances it will build (data + witness + barman nodes per location, plus an optional witness-only location and a pemserver if requested), so --hostnames-from accepts a correctly-sized file and no longer requires padding it out to 16 entries.	
Fixed an issue whereby CAMO configuration was incorrectly added to config.yml. When configuring a PGD-X cluster with --data-nodes-per-location 2 without explicitly enabling CAMO via --enable-camo, the generated config.yml would incorrectly include CAMO commit scopes and partner assignments. This resulted in unexpected CAMO configuration being applied to clusters that did not request it. CAMO configuration is now only generated when --enable-camo is explicitly passed to tpaexec configure.	59694
Fixed an issue whereby PGD 5/6 minor and 5-to-6 major upgrades occasionally left nodes fenced. During a node-by-node upgrade, the "Wait for write leader elections to complete" task could time out because nodes that had been fenced earlier in the loop were never unfenced: the unfence step did not always see the node's updated state and was wrongly skipped, leaving the node fenced. TPA now waits for the node to rejoin Raft consensus and refreshes its state before unfencing, so nodes are reliably unfenced and the upgrade can proceed.	
Improved error output for `--primary-location` in `tpaexec configure`. An error in the argument parsing logic between the `--primary-location` and the `--location-names` arguments when the primary location is not found in the list of locations caused a built-in Python error to be thrown, which could be misleading to users, as it is not a TPA-related error. This fix properly handles the mismatch between the provided primary location and the list of known locations, avoiding confusing error messages.	
Fixed an issue whereby tpaexec configure crashed when --overrides-from was supplied. Running `tpaexec configure` with `--overrides-from <file>` aborted with "An error was encountered during execution of tpaexec configure: name 'reduce' is not defined" and no cluster directory was produced. `tpaexec configure --overrides-from` now completes successfully and the values from the supplied YAML file are merged into the generated cluster configuration as documented.	

Description	Addresses
`tpaexec switch2cm` now works when ansible_user is not root. Three plays in the PGD-Always-ON switch2cm command did not declare become at the play level, so the command only worked when ansible_user was root. With a non-root ansible_user it failed with "Permission denied" on /pgdata/data/conf.d. Those plays now explicitly become root, matching the pattern used elsewhere in PGD-Always-ON. Per-task become overrides for SQL queries that run as the postgres user are unaffected.	
TPA now configures max_active_replication_origins for PostgreSQL 18+ PGD clusters. TPA now sets the max_active_replication_origins parameter for PGD clusters on PostgreSQL 18 and above, using "3 * number_of_nodes + 3" — a safety-margin formula above the "3 origins per peer node" minimum recommended by the EDB PGD documentation. This prevents PGD node join failures caused by the default value being too low for multi-node clusters.	
TPA now sets path_prefix per backup server in the barman configuration. Barman's path_prefix was previously set only as a global value in barman.conf, using the postgres binary directory of the barman host itself. This caused incorrect behaviour in mixed-version scenarios (during rolling upgrades, or when the barman host runs a different PostgreSQL major version than the nodes it backs up). path_prefix is now also written per backup server in /etc/barman.d/<backup>.conf, resolving from the backed-up node's own postgres_bin_dir. This ensures barman uses the correct client binaries for each node it backs up. The global path_prefix in barman.conf is unchanged and continues to serve as the default for same-version scenarios. When the backed-up node's PostgreSQL version differs from the barman host's, the matching client packages are now installed on the barman host automatically so the per-server path_prefix resolves to real binaries. The barman host must have repository access for the additional PostgreSQL version (typically the case with EDB enterprise repositories). An explicit per-instance override is also available by setting barman_path_prefix in a node's vars in config.yml.	

Deprecations

Description	Addresses
Removed the deprecated `--cohost-proxies` and `--add-proxy-nodes-per-location` configure flags for PGD-X. The --cohost-proxies and --add-proxy-nodes-per-location flags in PGD-X have now been removed and will no longer be valid options in `tpaexec configure` for the PGD-X architecture. In PGD 6 all proxies are cohosted, making both options obsolete.	

2.2 Trusted Postgres Architect 23.43.0 release notes

Released: 18 March 2026

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.43.0 include the following:

Highlights

- Support for SLES 15 SP7
- Improved SSH configuration options

Enhancements

Description	Addresses
Added support for SLES 15 SP7 across all platforms. TPA now supports SLES 15 SP7, upgrading from SP6. This support extends to all platforms, including bare-metal, AWS, and Docker. TPA can now be used as a deployment node on SLES 15 SP7 and can also provision and manage target nodes running this Service Pack. The update includes new AMI images, Docker base images, and validated package builds.	
Improved SSH configuration, supporting jump host. TPA now fully populates its generated `ssh_config` with all the options needed by ansible. Ansible uses the config instead of adding options to the command line. The configuration can now include a jump host, which will therefore be automatically used when deploying.	
Added task selector to skip cluster SSH key installation in authorized_keys. Added a new task selector `ssh-cluster-key-config` that allows skipping the 'Ensure authorized_keys allows admin access' task during deployment. This is useful in environments where the `authorized_keys` file is managed externally or is read-only (e.g. Oracle Cloud Infrastructure).	

Changes

Description	Addresses
Support `--keep-compression` in `barman-wal-restore`. Replaced the deprecated `-z` flag with `--keep-compression` in the Barman `restore_command` for installations using Barman 3.12+. TPA implements a conditional fallback to the deprecated flag for legacy versions, maintaining backward compatibility while adopting modern CLI standards based on the detected or specified Barman version.	
Standardize `pemworker` argument ordering for PEM 10.4. Refactored `pemworker` command-line usage to place subcommands (e.g., `--enable-probe`) as the first argument. This update addresses changes in the PEM 10.4 CLI, which no longer supports arbitrary argument positioning, ensuring that TPA continues to successfully configure and manage PEM agents.	

Description	Addresses
Ensure Docker platform compatibility with recent Docker versions. Recent versions of Docker no longer populate the top-level `NetworkSettings.IPAddress` field when inspecting containers - the IP is now only available within `NetworkSettings.Networks[<name>].IPAddress`. This caused container provisioning to fail when TPA attempted to determine the IP of containers. TPA now checks the top-level field for compatibility with older Docker versions, falls back to the configured `docker_networks` entry if defined, and otherwise uses the first available network IP.	
Updated OpenJDK version used in EFM SLES 15 deployments. The OpenJDK version used in EFM for SLES 15 deployments has been updated to a newer version (from version 11 to 21). The reason behind is to ensure the availability of an OpenJDK version in the SLES 15 repositories used on the docker images.	
Updated EFM-dependency OpenJDK in RHEL, Debian and Ubuntu. This change ensures that EFM uses the latest available version of OpenJDK in the supported distributions: • RHEL 8 and 9, from version 11 to 21. • Debian 11, from version 11 to 17. • Ubuntu 22.04 and 24.04, from version 11 to 21.	
TPA will now create an rc-local service on systems without rc-local. A change has been introduced to support distributions that do not have rc-local by default. The change includes creating a custom TPA rc-local service and ensuring that it is enabled on the system's boot. This allows TPA to run `/etc/tpa/rc.local` at startup, providing a consistent way to execute TPA startup tasks across different distributions.	
TPA now blocks use of the pgextended flavour with PGD 4 or later. Deploy now fails with a clear error if `postgres_flavour` is set to `pgextended` with `bdr_version` 4 or later. The `pgextended` flavour is only supported with BDR 3. Use `edbpge` instead for PGD 4 and later.	51418

Bug Fixes

Description	Addresses
TPA will now retry EFM's health check after an upgrade. In a busy system, the cluster can take a bit longer to recover making just a single health check insufficient. This change introduces a retry mechanism for efm's health check after an upgrade to give the cluster more time before deciding that the upgrade has failed.	
Fixed PEM version pinning issues on Debian/Ubuntu. Resolved a dependency conflict on Debian/Ubuntu systems that occurred when pinning the PEM version. TPA now correctly pins both `edb-pem-cli` and `edb-pem-agent` dependencies to the same version, preventing the package manager from rejecting the installation due to mismatched version requirements.	
Fixed EFM switchover failures on busy systems with standby lag. Fixed an issue where EFM switchover operations could fail during upgrades or maintenance operations on busy systems when the standby server was lagging behind the primary. Previously, TPA's pre-switchover health checks would immediately fail if standby nodes were not fully synchronised with the primary, causing upgrade procedures to abort even when the nodes would catch up within seconds. TPA now implements a wait-and-retry mechanism in the EFM health check that monitors cluster status and waits for all nodes to synchronise before allowing the switchover to proceed. This ensures switchover operations only begin when the cluster is in a safe state for promotion. This resolves spurious failures on distributed systems where temporary replication lag exists during upgrade workflows on production systems with active workloads.	

Description	Addresses
AWS provisioning now validates that specified security groups exist. When provisioning AWS clusters with `ec2_groups` configured in `config.yml`, TPA now explicitly validates that the specified security groups actually exist. Previously, if a non-existent security group was specified (for example, due to a typo in the group name), provisioning would continue silently and EC2 instances would be created using the VPC's default security group instead. This could result in instances having incorrect or unexpected network access rules without the user realizing it. Provisioning now fails immediately with a clear error message indicating which region has the problem, what filters were used, and guidance on how to fix the issue. This prevents security misconfigurations and makes troubleshooting easier. Additionally, TPA now provides explicit validation with actionable error messages when creating optional AWS resources such as Internet Gateways, Route Tables, and Elastic IP associations, improving error visibility during provisioning.	

2.3 Trusted Postgres Architect 23.42.0 release notes

Released: 25 February 2026

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.42.0 include the following:

Highlights

- Experimental support for RHEL 10 (open source clusters only until EDB packages are available)
- Improved control over Postgres log settings
- New `privilege_escalation_command` option to accommodate a wider range of user environments

Enhancements

Description	Addresses
Improved and documented the handling of log destination and log directory. By default, TPA configures Postgres to log to `syslog`. The log location for `rsyslog` can be set with `postgres_log_file`. If the user selects a destination other than `syslog`, TPA will enable the Postgres logging collector configure it to log to `postgres_log_file` Previously this behavior was only implemented if the user selected `stderr`. Now it will work for any setting other than syslog, including `jsonlog` and `csvlog`. This change adds a new cluster variable `postgres_log_directory_mode` which can be used to set the directory permissions. It also adds new documentation for all logging-related cluster variables. Logrotate will also be configured to target the log file regardless of how logging is collected.	50711
Support running TPA on RHEL 10. The TPA controller can now be a system running RHEL 10 or an equivalent OS (Rocky Linux, AlmaLinux, or Oracle Linux 10). On these systems, the system Python packages (Python 3.12) can be used.	
Experimental support for deployment to RHEL 10 nodes. TPA can now deploy to nodes running RHEL 10 or distributions based on RHEL 10 (Rocky Linux, AlmaLinux, or Oracle Linux). Suitable docker images or ec2 AMIs are selected if the docker or aws platforms are being used. This feature is considered experimental because deployment depends on the availability of the various packages for RHEL 10, so not all architectures or software options will work until packages are uploaded to upstream repositories.	

<table>
<tr><th>Description</th><th>Addresses</th></tr>
<tr><td>
Added the <code>privilege_escalation_command</code> configuration variable for alternative privilege escalation commands.
This release introduced a new cluster_vars variable <code>privilege_escalation_command</code> that allows users to configure which privilege escalation command managed applications use for cluster operations at runtime. The default value is <code>sudo</code>, which maintains the current behaviour where TPA automatically installs the sudo package if not present and configures sudoers files for various components (Postgres user, EFM user, failover managers). Users can now specify an alternative privilege escalation command by setting <code>privilege_escalation_command</code>.
When using an alternative privilege escalation command (anything other than <code>sudo</code>), TPA will:

Skip sudo package installation
Skip sudoers configuration for the Postgres user
Skip sudoers configuration for the EFM user
Use the configured command for all failover manager operations (repmgr, HARP, EFM)

This feature is useful for environments that:

Use alternative privilege escalation commands
Require manual management of sudo configuration
Have strict policies around privilege escalation commands

Important: When using alternative privilege escalation commands, users must ensure:

The chosen command is pre-installed and configured on all target systems
The command allows the postgres user to run systemctl commands for service management
The command allows the EFM user to run <code>efm_db_functions</code> (if using EFM)
Ansible's become mechanism is properly configured (via <code>ansible_become_method</code>)

Component-specific changes:

Postgres user sudoers: When using alternative commands, TPA skips configuring sudoers for the postgres system user. Users must configure privilege escalation for service management commands.
EFM sudoers: When using alternative commands, TPA skips configuring sudoers for the EFM user. Users must configure privilege escalation to allow the EFM user to run <code>efm_db_functions</code> as the postgres user.
Failover managers: repmgr and EFM now use the configured privilege escalation command for service management commands instead of hardcoding sudo.
Barman SELinux: Barman SELinux configuration tasks now use Ansible's become mechanism instead of explicit sudo commands, respecting the configured privilege escalation command. Refer to the documentation in ansible-and-sudo.md and configure-cluster.md for detailed usage examples and configuration guidelines.

</td><td></td></tr>
<tr><td>
The <code>download-packages</code> command now has a 'refresh repository metadata' option.
A new option has been included on the <code>tpaexec download-packages</code> command (<code>--refresh-repository</code>, requires Docker & its corresponding Docker image to work). Prior to this change, we relied on the user to have the commands required to manually recreate the local repository, depending on its cluster and OS. This fix provides an automated way of upgrading the repository with <code>tpaexec upgrade</code> once the packages are being placed and the new version of said packages is selected on the configuration file. It also includes documentation covering the steps that need to be taken, given the different air-gapped conditions that can be found, and minor fixes that were discovered related to the DNF repository in an air-gapped scenario.
</td><td>52313</td></tr>
</table>

Changes

Description	Addresses
Prevent concurrent installation of multiple EFM versions. TPA now prevents the installation of multiple EFM versions, resolving service startup conflicts. The version check now correctly detects the `upgrade_in_progress` state, allowing `tpaexec upgrade --components=efm` to replace the existing version while continuing to block accidental multi-version installations during normal deployment operations.	48997
TPA now rejects unsupported Postgres 13 for PGD 6 clusters. Updated `tpaexec configure` to explicitly reject configurations combining Postgres 13 with PGD 6. As Postgres 13 is not supported by PGD 6, this check prevents users from proceeding with incompatible setups that would otherwise fail later during the deployment phase.	
Default to "enterprise" repository for PGD-S architecture. A cluster configured to use the PGD-S architecture now defaults to the "enterprise" repository not the "postgres_distributed" repository. Thus the default repositories now match the required subscription levels.	
Removed mandatory EDB token check for local-only deployments. Resolved an issue where `tpaexec deploy` incorrectly required an EDB_SUBSCRIPTION_TOKEN even when the `--use-local-repo-only` option was enabled. TPA now bypasses this validation when deploying from local repositories, provided that all necessary packages are available locally. Note that a token is still required to initially download packages and populate the local repository.	
Removed unused monitoring-server role. The monitoring-server role was previously used to set up a centralised monitoring server for clusters, but this functionality is no longer used or supported. This change removes all references to the monitoring-server role from the codebase, including: • Removal of monitoring-server role from group creation in roles/init • Removal of the task that sets the monitoring_server variable • Removal of monitoring-server from test fixtures This cleanup removes dead code and prevents confusion about supported roles in TPA deployments.	
Separate requirements files for RH8. TPA now uses separate requirements files when installing python packages, depending on whether it is running on RH8. In addition, it checks at deploy-time that the ansible version in use is 2.16 or later if any of the instances in the cluster is running RH8. This will enable TPA to continue to support deployments to RH8 while using the latest version of ansible-core for other distributions.	
Removed documented dependency on patch. As TPA no longer requires the patch utility, removed mention of it from the installation instructions, and removed unreachable code which called it.	

Bug Fixes

Description	Addresses

Description	Addresses
Fixed upgrade failures for clusters using mTLS. Resolved an issue where `tpaexec upgrade` failed on clusters with mTLS enabled. Previously, required TLS variables were missing during the upgrade phase, causing "certificate required" errors on etcd and undefined variable errors on Patroni. This fix ensures that all mTLS configuration is correctly propagated during upgrades to maintain secure connectivity.	
Removed 'file' as a dependency for Barman. Barman now requires 'file' as package dependency, so it's installed by the package manager. There's no need to track it as a separate dependency any more.	
Fixed an issue that resulted in duplicate `bdr_node_groups` in PGD-X configuration. Resolved an issue where specifying Connection Manager port options during `tpaexec configure` for PGD-X clusters resulted in duplicate `bdr_node_groups` entries in `config.yml`. The configuration logic now correctly handles these options to ensure a clean and valid cluster definition.	
Fixed `--edb-repositories` in `tpaexec configure` command to correctly handle PGDG repositories. Prior to this version of TPA, using the `--edb-repositories` option in `tpaexec configure`, caused the code responsible for defining the `{apt,yum,suse}_repository_list` variables to be wrongfully skipped. This generated incorrect configurations with both EDB and PGDG repositories defined, causing conflicts in packages resolution. A fix has been introduced to ensure the correct configuration is generated in every situation.	
TPA now accounts for Parallel Apply in the `max_replication_slots` calculation. Updated the formula for `max_replication_slots` to support PGD 5 Parallel Apply. Since Parallel Apply requires an additional replication origin per writer, TPA now incorporates the `bdr.writers_per_subscription` setting into its automated calculations to prevent slot exhaustion in PGD 5 clusters. IMPORTANT: This change will result in postgres cluster restart on the next deploy when using PGD version 5 and above, as TPA will apply the updated calculation results. To avoid downtime, verify and define the current value for `max_replication_slots` and `max_wal_senders` in your `cluster_vars`.	51548, 57005
TPA now supports running tests when ansible_user is not root. "tpaexec tests" runs database queries which must be run either as root or as the correct user in order to be able to read SSL certificates. These tests were previously run without specifying a user, and therefore failed if the user was not root. They now run with the "become_user" attribute and run as the postgres or pgd_proxy user according to the node type.	96876
TPA now allows Docker instance IP addresses to be set using `--hostnames-from`. Previous versions of TPA automatically generated `ip_address` even if it was supplied in a hostnames file. This fix means TPA will only generate an IP address at configure time if one has not been specified already. Note that TPA will not automatically select a cluster network that contains the provided IP addresses, so the `--network` option should be used to specify a matching CIDR.	
Fixed an issue with undefined Patroni SSL facts during upgrade and test phases. SSL-related facts set during deployment were not persisted across playbook runs, causing errors such as `'patroni_ssl_ca_file' is undefined` when running `upgrade` or `test` against a Patroni cluster with SSL enabled. These facts are now set by a dedicated `patroni/facts` role included at the relevant points in the upgrade playbook and test tasks, ensuring they are always available regardless of which phase is being executed.	

Description	Addresses
Corrected Patroni `ignore_slots` data structure to prevent DCS parsing errors. Previously, the `ignore_slots` parameter was generated as a dictionary instead of the required list format. This caused an AttributeError during DCS parsing, which not only prevented the `ignore_slots` configuration from taking effect but also blocked the application of subsequent dynamic and static configuration updates. This change ensures that `ignore_slots` is correctly formatted as a list of dictionaries, resolving the parsing error and ensuring all configuration changes are applied successfully.	
TPA will now validate that instance roles list doesn't contain empty entries. Fix a bug whereby either adding an empty entry or forgetting to remove the dash when manually editing the instance's roles list in `config.yml` would result in a failed deployment. An empty entry in the instance roles list is invalid and should be caught early in the deployment process. This adds a task that validate the content of the role list for each nodes as soon as possible during deployment.	46041
Added missing `--enable-camo` option for PGD-X in `tpaexec configure`. Introduced the `--enable-camo` flag for PGD-X architectures within `tpaexec configure`. This update enables Commit At Most Once (CAMO) support for the cluster, including automated compatibility checks and the necessary configuration adjustments required for a successful deployment.	
Added validation to catch missing etcd roles. When deploying a BDR 4 cluster with HARP using `harp_consensus_protocol: etcd`, if no instances have the etcd role defined, deployment would fail late with cryptic and obscure errors about missing etcd endpoints. Debugging such issues is expensive but this validation now catches the issue early and provides some clues to the user about what went wrong.	46041
TPA now bypassed "Record PEM server credentials" task during shared PEM deployment. Previously, when deploying a shared PEM server, the task "Record PEM server credentials" would fail because it required the 'postgres_password' attribute. This attribute is only defined when a node with the 'postgres' role is deployed alongside the shared PEM node. This caused failures in specific scenarios, such as deploying an independent Barman node with shared PEM where no 'postgres' role exists, leading to a missing attribute error. The fix skips this task when shared PEM is in use, as the 'postgres_password' is instead derived from the 'EDB_PEM_CREDENTIALS_FILE' in a subsequent step.	
Fixed `barman_role` privilege grants for BDR clusters. In BDR clusters, privileges required by Barman were incorrectly granted to the BDR database instead of the `postgres` database, which is where Barman actually connects. This caused the "no access to backup functions: FAILED" error after upgrading to TPA 23.40. After deploying this fix, the grants are correctly applied to the `postgres` database. Existing affected clusters can be fixed by re-running `tpaexec deploy`.	54698
Fixed downloader to ensure pgd-proxy and pgdcli package version matches `bdr_package_version` when only `bdr_package_version` is set. Selecting a 'bdr_package_version' on an air-gapped environment downloads the selected version for BDR, but not for PGD CLI and PGD Proxy, which later results on an error in 'tpaexec deploy', since there's a package mismatch. This is not the case for non air-gapped environments, where choosing a specific version of 'bdr_package_version' will also select the according versions for PGD CLI and PGD Proxy. This fix ensures that when executing 'tpaexec download-packages', we first check if 'pgd_proxy_package_version' or 'pgdcli_package_version' are defined, and if not, we will reuse the 'bdr_package_version' variable to download the same version specified, since BDR package mismatching can lead to critical issues in a production environment.	

Description	Addresses
Standardized PGD-X port flags in `tpaexec configure`. Updated `tpaexec configure` to remove `--proxy-listen-port` and `--proxy-read-only-port` as valid options for PGD-X. Documentation was adjusted to reflect these removals and to add coverage for the supported `--read-write-port` and `--read-only-port` flags, ensuring alignment between the CLI and the manual.	57408
TPA now ignores PEM server instances in slot default calculations. Modified the logic for `num_postgres_instances` to ignore instances dedicated to the PEM server backend. Since this variable determines default values for `max_replication_slots`, excluding PEM nodes prevents over-allocation. IMPORTANT: Deployments on clusters with a PEM server will trigger a restart on the next deploy if `max_replication_slots` is not explicitly defined in `config.yml`, as TPA will apply the updated calculation results. To avoid downtime, verify and define the current value in your `cluster_vars`.	
Fixed a crash that occurred when running tpaexec deploy with external variable files. Fixed a crash that occurred when running tpaexec deploy with external variable files (e.g., `tpaexec deploy --extra-vars "@./vars.yml"`). Previously, the internal provisioning step triggered by the deploy command did not inherit these extra variables, causing the execution to fail. This fix ensures that all specified variables are correctly passed to tpaexec provision before the deployment begins.	56864
Fixed an issue where PGD-X deployment failed when a commit scope was selected. PGD-X by default creates a series of immutable commit scopes. This change prevents TPA attempting to delete these commit scopes for PGD-X when adding commit scopes not defined previously in the configuration file (e.g. CAMO).	
Fixed RHEL 8 deployment failures in Patroni clusters. Resolved a deployment failure on RHEL 8 by updating the `patroni_compare_config.py` script for Python 3.6+ compatibility. This fix ensures that Patroni configuration validation runs correctly in modern Python environments while maintaining the script's core logic.	
Fixed KeyError on etcd_protocol whilst upgrading Patroni. This is a race condition when the first etcd node tried to build endpoints for all etcd nodes using pyformat_hostvars, it looked for etcd_protocol in each host's hostvars, however, only the current host had set this fact during an upgrade. This caused a KeyError when accessing other hosts.	
Fixed an issue with 'basic' authentication in 'etcd_authentication_mode'. When selecting 'basic' authentication for 'etcd_authentication_mode', the execution of the deployment would fail on the task that grants permissions to the 'patroni' role in etcd, responsible for making calls to the etcd cluster. This was due to a fact variable not being properly defined in the scope of the task, causing the task's condition to fail and thus the permission not being granted. The fix involved ensuring that the fact variable is correctly set and accessible, allowing the permission granting task to execute as intended when 'basic' authentication is selected.	
Fixed package conflicts during PGD upgrade on RHEL-based systems. Resolved a dependency resolution (depsolve) issue encountered when upgrading from PGD 5 to PGD 6 on RHEL-based operating systems. The upgrade process now conditionally utilizes the `allowerasing` option, permitting the necessary replacement of conflicting PGD 5 packages during dry run check before upgrade to ensure a successful transition to PGD 6.	

Description	Addresses
Fixed Barman deployment failures when using version pinning. Resolved a dependency conflict on Debian/Ubuntu systems that occurred when pinning the Barman version via `barman_package_version`. TPA now correctly pins both `barman-cli` and its `python3-barman` dependency to the same version, preventing the package manager from rejecting the installation due to mismatched version requirements.	

2.4 Trusted Postgres Architect 23.41.0 release notes

Released: 26 November 2025

> **Known Issue**
>
> When using TPA 23.35 or later to `deploy` to a PGD/BDR cluster created with an earlier version of TPA the `barman` user will have its superuser permissions revoked. This means that attempts by this user to connect to the `postgres` database will fail. This will cause the Barman `check` command to report a failure. The workaround for this issue is to restore these permissions with a post-deploy hook.

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.41.0 include the following:

Highlights

- Perform updates of all cluster components - not just Postgres and PGD
- Hardening of the default Patroni configuration, with many new options to fine-tune security measures
- Support for minor version upgrades of PGD 6 (PGD-X and PGD-S)
- Support for Postgres 18

Enhancements

Description	Addresses
Secure etcd and Patroni defaults for new `tpaexec configure` clusters. When creating a new cluster configuration using `tpaexec configure`, the generated `config.yml` file will now automatically enable SSL/TLS encryption and mutual TLS (mTLS) authentication for both `etcd` and `Patroni` by default. This ensures that new deployments start with a secure communication baseline out-of-the-box. The following variables are now set by default for new configurations: • `etcd_ssl_enabled: true` • `etcd_authentication_mode: mtls` • `patroni_ssl_enabled: true` • `patroni_authentication_mode: mtls` Existing cluster configurations are unaffected by this change.	
Support upgrade of components M1 and PGD/BDR clusters. All cluster software components (failover managers, PEM, Barman, etc.) can now be updated, either to a specific version by specifying the component package version in `config.yml` or to the latest available when it is not provided. A list of components for upgrade can be passed to the `--components` flag as a comma-separated list. If this flag is not passed, `postgres` will be updated. Specifying `--components=all` will update all applicable software components.	
Improved automation of PGD5.9+ to PGD6 upgrade. For a PGD v5.9 to v6 migration, Connection Manager is a requirement instead of PGD proxy. A new command `switch2cm` has been introduced to facilitate this transition. The new command requires that the cluster's `config.yml` must be prepared by running `tpaexec reconfigure <cluster> --enable-connection-manager`. Upgrading from PGD-Always-ON to PGD-X	

Description	Addresses
Communication with the `etcd` cluster can now be secured using SSL/TLS. A new configuration parameter, `etcd_ssl_enabled`, has been introduced to enable TLS encryption for all `etcd` communication. This significantly hardens the security of the cluster's distributed control plane. When `etcd_ssl_enabled` is set to `true`: • The deployment process automatically generates the required TLS certificates for each `etcd` node. These certificates now include both DNS and IP Subject Alternative Names (SANs) for robust validation. • Both peer-to-peer and client-server `etcd` connections are configured to use HTTPS. • Downstream components, including **Patroni** and internal health checks using `etcdctl`, are automatically configured to connect to `etcd` securely over HTTPS. To maintain backward compatibility with existing deployments, this feature is disabled by default (`etcd_ssl_enabled: false`).	
Mutual TLS (mTLS) Authentication for Patroni REST API. Support for mutual TLS has been added to provide a more secure, certificate-based authentication method for the Patroni REST API. A new configuration variable, `patroni_authentication_mode`, is now available. You can set this to `mtls` to enable this feature. When enabled (which requires `patroni_ssl_enabled: true`), the deployment automation will: • Configure the Patroni REST API server to require and validate client certificates. • Automatically configure clients, including `patronictl` and HAProxy health checks, with the necessary client certificates and keys to connect securely. The default authentication mode remains `basic` (username and password) to ensure backward compatibility with existing deployments.	
Improve security of Patroni REST API with access `allowlist`. The security of the Patroni REST API has been enhanced by configuring an `allowlist` for unsafe API endpoints (which accept `POST`, `PUT`, `PATCH`, `DELETE`). This measure prevents unauthorized nodes from performing administrative actions, such as restarting a database node or changing the cluster configuration. The `allowlist` is dynamically populated with the hostnames of the Patroni cluster members based on the Ansible inventory, ensuring that only authorized nodes can perform requests to these sensitive API endpoints.	
TPA now supports minor version upgrades for PGD-X. The `tpaexec upgrade` command can now perform minor version upgrades of postgres and PGD on a cluster running the PGD-X architecture. The upgrade process checks cluster health and upgrades the nodes one at a time. This scenario supports the use of `update_hosts` variable to upgrade a subset of the cluster when used accordingly to the best practice recommendations found in the documentation.	
TPA now supports minor version upgrades for PGD-S. The `tpaexec upgrade` command can now perform minor version upgrades of postgres and PGD on a cluster running the PGD-S architecture. The upgrade process checks cluster health and upgrades the nodes one at a time. This scenario supports the use of `update_hosts` variable to upgrade a subset of the cluster when used accordingly to the best practice recommendations found in the documentation.	

Description	Addresses
Enable etcd Basic Authentication and Role-Based Access Control. This release introduces a new capability to secure the etcd distributed key-value store with client authentication. You can now enable basic authentication and configure a dedicated user for Patroni with granular, least-privilege permissions. A new configuration variable, `etcd_authentication_mode`, has been added to control the feature. When set to `basic`, the deployment automation will: • Set up an administrative `root` user. • Create a dedicated etcd user for Patroni, given by the new `patroni_etcd_user` configuration option, and grant it read-write permissions limited to its cluster's key prefix (e.g., `/tpa/cluster_name`). • Configure Patroni to automatically use these credentials to securely connect and authenticate with etcd. To maintain backward compatibility, this feature is disabled by default (`etcd_authentication_mode: none`).	
Mutual TLS (mTLS) Authentication for `etcd`. Support for mutual TLS has been added to secure all `etcd` communication. A new `mtls` option is now available for the `etcd_authentication_mode` variable. When `etcd_authentication_mode` is set to `mtls` (which requires `etcd_ssl_enabled: true`), the deployment automation will enforce certificate-based authentication for: • Peer-to-peer communication between all `etcd` nodes. • Client-server communication from clients like Patroni. TPA automatically configures both the `etcd` servers and the Patroni clients with the necessary certificates and keys to ensure a fully secure and validated connection.	
Introduced new instance-level variables to control how servers and agents are displayed in PEM. Introduced four new instance variables for use with the `pem-agent` role: • `monitoring_group` - `monitoring_agent_group` - `monitoring_cluster` - `monitoring_agent_cluster` Also introduce the instance variable `pem_agent_group` for use with the `pem-server` role alongside the existing variable `pem-server-group`. Collectively, these variables determine which group and cluster the servers and agents are assigned to in PEM. See the documentation for a full explanation.	
Added support for validity period in OpenSSL generated certificates. Previously, some SSL certificates (OpenVPN, PEM and TPA itself) used in TPA didn't come with a way of specifying the number of days before expiration. This change adds a variable called `openssl_certificate_validity` (default 3650 days), that can be used to determine the number of days an SSL certificate will last before expiring. In order to force TPA to reload a new certificate with a new expiration date, you must pass the appropriate option to `tpaexec deploy` `-e pem_web_server_renew_tls_certificates=true"` or `-e tpa_tls_renew_tls_certificates=true")`.	

Changes

Description	Addresses

Description	Addresses
Changed the default EFM password encryption to scram-sha-256. TPA now defaults to `efm_user_password_encryption: scram-sha-256` for all new clusters configured with EFM as the failover manager. This change applies when running `tpaexec configure` with `--enable-efm` or `--failover-manager efm`. This addresses the deprecation of MD5 password encryption in PostgreSQL 18 and provides improved security for all PostgreSQL versions. Existing clusters are unaffected. Users can override this default by explicitly setting `efm_user_password_encryption: md5` in `config.yml` if needed, though MD5 support will be removed from TPA in a future release.	
The `tpaexec configure` command now adds `efm_version` to `config.yml`. Support for explicitly adding the `efm_version` if introduced when executing `tpaexec configure` and choosing EFM as the failover manager. This can be seen when invoking `tpaexec` as follows: `tpaexec configure my-cluster-dir -a M1 --enable-efm --efm-version 5.2 (\u2026)`. Additionally, documentation covering the `efm_version` selection has been added.	
TPA now runs `pgd-proxy` instead of `journalctl` for PGD Proxy version detection. There are some edge cases where `journalctl` does not return the expected version. Hence, TPA now runs `pgd-proxy` directly to get the version information instead.	46753, 46426
Moved the pre-deploy hook outside repositories role. In previous versions, this hook was part of the repositories role. When TPA introduced the possibility to exclude this role from deployments, it created a situation whereby this hook could be unintentionally skipped. To avoid this situation, we moved the hook outside the role so it's always executed.	52412
Added `auto.basebackup` property for EFM 5.2 and above. Starting with EFM 5.2, there will be a new property `auto.basebackup` that adds on to the new auto-rebuild capability. Users can use the existing pg_rewind feature, pg_basebackup, or tell EFM to use both (pg_basebackup will be run if pg_rewind fails).	
Updated the internal dictionary of the AWS regions. The change updates the internal dictionary of the AWS regions to expand the coverage of the supported regions and corresponding AZs.	
Documented `postgres_port` in TPA documentation. Added detail of the parameter `postgres_port` in the TPA documentation. `postgres_port` is the variable allowing the customization of the listening port for Postgres. This variable ensures consistent use of the set value across all components of the cluster relying on the information.	
Disabled TRACK and TRACE methods on the PEM HTTPD server. PEM 10.3 contains a fix for HTTPD configuration, moving the directive to rewrite TRACK/TRACE requests to the <VirtualHost> section for the default virtual host on port defined by the `pem_server_ssl_port` variable. This change applies the same fix to all new deployments of PEM, regardless of version.	54529
Added `efm-post-config` hook. A new `efm-post-config` hook has been introduced to be able to execute new tasks after the efm configuration has taken place. New documentation explaining how to use it has also been developed under the 'TPA hooks' section.	

Description	Addresses
Added support for pgaudit and postgis plugins on PG18. Adding PG18 support for those plugins. The version of those depends on the Postgres version being supported as well. We have removed versions 11 and 12 from the plugin lists as they are no longer widely supported.	
Start PGD proxy after the network is available. This service needs the network service to be up and running before it can be started. `Wants` and `After` directives have been added to the PGD Proxy service unit to ensure that.	54251, 52731
Install `libpq5` on PEM server v10+ for RHEL. PEM 10.1 added `libpq5` as a dependency to fix `psycopg` errors related to `kerberos` authentication on RHEL 9 servers. To ensure `libpq5` is installed, TPA now explicitly installs it on RHEL servers when the PEM server version is 10 or greater.	
Install `libcurl-full` when required. `libcurl-minimal` is installed certain RHEL-esque images, and while it satisfies the package dependencies for `edb-pem-agent`, it does NOT provide the complete set of APIs required (namely for SMTP) If it is installed, we remove it and install the `libcurl-full` meta-package to ensure full `libcurl` functionality exists for the PEM agent. Because the `edb-pem` package depends on the `edb-pem-agent` package, the `libcurl-minimal` package must also be removed on the PEM server in order to avoid dependency resolution conflicts. This is done by installing the package with the `allowerase` parameter.	

Bug Fixes

Description	Addresses
Patroni REST API SSL/TLS connections are now correctly configured and secure. Previously, enabling SSL for the Patroni REST API (`patroni_ssl_enabled: true`) used a self-signed certificate. This resulted in `CERTIFICATE_VERIFY_FAILED` errors for clients like `patronictl`, making the feature unusable in a secure configuration. This has been fixed by implementing a proper certificate generation process. The Patroni REST API server is now configured with a server certificate signed by a Certificate Authority (CA) created by TPA for the TPA cluster. Additionally, the `ctl` section of the `patroni.yml` configuration has been corrected to ensure `patronictl` is properly configured to use the CA for server validation and to present its own client certificate and key for authentication (mTLS). As a result, enabling SSL for the Patroni REST API now provides a fully secure and verifiable communication channel for all clients.	
Fixed race conditions during concurrent updates to the Barman ``.pgpass` file.  Previously, when multiple backed-up nodes attempted to update the Barman node's `.pgpass` file concurrently, a race condition could occur, resulting in partial or incomplete entries. TPA now serializes this process, ensuring that the Barman node's `.pgpass` file is updated sequentially to prevent loss.	52460
Fix `unwanted_packages` to support 'common' key that applies to all distribution. Previously, when setting `unwanted_packages` using the `common` key was not supported, which differs from the behavior of other package dictionary that can be set in `config.yml`. This fix introduces the support for the `common` key, ensuring any package listed under this key will be removed if present on any distribution in use in the cluster.	
Fixed task selection for pgdcli. Fixed a bug whereby the `pgdcli` role would be run on `bdr` nodes even when task selectors were used which should implicitly exclude it.	

Description	Addresses
Fixed a fatal error during Docker container provisioning. When creating containers on modern versions of docker using user-defined networks, `provision` would sometime crash with a 'dict object' has no attribute 'IPAddress' error. This was caused by an outdated method of discovering the container's IP address. The logic has been updated to be compatible with modern Docker network structures, making container creation more robust.	
Fixed `etcd` package installation on SLES. Deployments on SLES-based distributions that require `etcd` (such as those using Patroni) would previously fail because the `etcd` package could not be found in the standard repositories. This has been resolved by enabling the PGDG `extras` repository, which provides the necessary package and allows the installation to succeed.	
Fixed an issue whereby `tpaexec test` did not remove data after testing in PGD clusters. After deploying a cluster and running `tpaexec test` on that cluster, creates an additional entry in `bdr.replication_set_table` which is not later removed. The fix makes the test run on a temporary schema called `tpatest`, which is later removed after the test is finished.	53224
Fixed issue with missing PGD CLI on proxy nodes when performing upgrade (3.7 to 5). When performing a regular deploy, or an upgrade from a 'BDR-Always-ON' setup to 'PGD-Always-ON', the proxy nodes (previously deployed with the 'harp-proxy' role), will not come shipped with the PGD CLI when trying to install it with the 'pgd-proxy' role, necessary to perform the upgrade. This fix ensures that the PGD CLI will be installed on the proxy nodes and nodes with a 'bdr' role, also on the nodes featuring only the 'pgdcli' role.	

2.5 Trusted Postgres Architect 23.40.1 release notes

Released: 28 October 2025

This is a bug fixes release. The issues resolved in Trusted Postgres Architect 23.40.01 include the following:

Bug Fixes

Description	Addresses
Fixed an issue whereby the PGD `alter_node_option` task would run for physical replicas. In a `PGD-AlwaysON` cluster with a BDR node that is also a physical replica, the upgrade process would fail during the `alter_node_option` task because a BDR node is expected to have RAFT enabled, but the physical replica BDR node does not. TPA now skips this task on any node with the `replica` role, allowing upgrade to complete successfully.	51997
Fixed unnecessary output after deploy by removing the post-deploy check for pgbouncer_auth_function in pg_catalog. TPA recently relocated the pgbouncer_auth_function used to authenticate users when pgbouncer is in use. This was done in an effort to better follow the best practices and avoid issues during upgrade where pg_catalog function would not be part of the dump of the database. TPA introduced a post deploy check to help users still relying on the auth function defined in pg_catalog to identify the problem and make sure that they are able to fix it before removing the function from their systems. This check was used as a transition mechanism and is now being removed. This resolves issues with unwanted output generated by that check.	100065
Fixed a misconfiguration of the `.pgpass` file when `backup_name` is used. Previously, when using the `backup_name` variable, TPA would incorrectly use this name in `pgpass` for the Barman server. This has been fixed and the documentation has been updated.	
TPA now ensures EDB repos are used when Beacon Agent is included in a cluster. TPA can deploy the Beacon Agent to monitor the health of a system. However, it needs access to a valid EDB repository. This change ensures that the Standard repository is enabled when the Beacon Agent is to be deployed.	
Fixed issue whereby Barman could not configure the log_file parameter correctly. When setting up a barman node, the variable 'barman_log_file' was not setting up an alternative directory to place the log file. Instead, it used the default location (`/var/log/barman/barman.log`). Bug and fix reported by voxnyx (https://github.com/voxnyx).	

2.6 Trusted Postgres Architect 23.40.0 release notes

Released: 2 October 2025

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.40.0 include the following:

Enhancements

Description	Addresses
Added support for upgrading EDB Postgres Distributed (PGD) v5 clusters to v6. TPA now provides a workflow for upgrading an existing PGD-Always-ON cluster to a PGD-X cluster. Due to the significant differences between these architectures, this is a multi-step process whereby you must first upgrade to PGD 5.9, then enable connection manager, then finally upgrade to PGD 6. This process will be further automated in a future TPA release. Please refer to the TPA docs for full details.	
Extended support for configuring PGD-S clusters. TPA now supports a full set of dedicated options for configuring a PGD-S cluster. The `--layout` option can be set to `standard` for a one-location or `near-far` for a two-location layout. The `--add-subscriber-only-nodes` option adds up to 10 subscriber-only nodes. The `--read-write-port` , `--read-only-port` , `--http-port` , and `--use-https options` control Connection Manager's ports and HTTP API.	
Introduced a `pgd-proxy-config` hook. Introduced a hook to run tasks from `pgd-proxy-config.yml` at the end of PGD Proxy config, after core directories and files are set but before the PGD Proxy service starts. Developers of the hook must ensure task idempotency. This hook will come in handy for various post-configuration tasks e.g. making adjustments to PGD Proxy configuration not supported by TPA interface and other relevant cluster-specific settings.	49911
Added three `release.vip.*` properties for EFM 5.1 and above. Starting with EFM 5.1, there will be three new properties that control the timing of when the VIP is released by the primary agent (if a VIP is used). This allows the user to avoid having the VIP released at the same time new database connections are being made, which can lead to connection failures and a failure of switchover in some environments. The new properties are: `release.vip.background` `release.vip.pre.wait` and `release.vip.post.wait` .	
TPA now allows the RSA key size to be set for certificates. Added a variable `postgres_rsa_key_size` , (default value is 2048) that can configure the size of the RSA key size for self-signed TLS keys and certificates later used on the docker clusters. Added a variable called `pem_rsa_key_size` , (default value is 4096) that can configure the size of the RSA key size for self-signed TLS key and certificate later used on the PEM Server. Added a variable called `pem_db_ca_certificate_key_size` , (default value is 4096) that can configure the size of the database CA RSA key size used on the PEM Server. Added a variable called `ha_proxy_dhparams_key_size` , (default value is 2048) that can configure the size of the RSA key size used for ssl-dh-param-file within haproxy. Added a variable called `openvpn_rsa_key_size` , (default value is 4096) that can configure the size of the RSA key size for self-signed TLS key and certificate later used on OpenVPN. Added a variable called `openvpn_dhparams_numbits_size` , (default value is 2048) that can configure the number of the DH `numbits` in `dh2048.pem` .	

Description	Addresses
Documented how TPA generates Postgres server TLS certificates and keys and how to override this. Previously, the TPA docs mentioned that TPA generates default certs but not their names or paths. This change adds that information and also explains how to replace these files with your own. We now document the previously undocumented `ssl_*` cluster vars for this purpose. The new content is placed in `postgresql.conf.md` and linked from the original location `postgres_user.md`.	52049

Changes

Description	Addresses
TPA now supports EPAS clusters with repmgr as failover manager. Until now, this combination wasn't supported by EDB. This support starts with EPAS 14 and is available as of now up to EPAS 17.	

Bug Fixes

Description	Addresses
Fixed an issue whereby backup wasn't configured in PGD 6 clusters. Previously, when creating a new configuration for PGD 6, the barman node created wasn't used by any node. This change ensures that when running "tpaexec configure" for PGD6 based architectures, each barman node configured will be used by a BDR node in the same location.	
Fixed support for `update_host` variable on PGD-Always-ON. TPA is now able to honor the `update_hosts` option for minor Postgres and minor PGD5 upgrades when using PGD-Always-ON architecture. By specifying the `update_hosts` option in the `tpaexec upgrade` command, TPA will now upgrade this subset of nodes and leave the remaining nodes on the currently installed versions. This allows for more controlled upgrade that can be split into multiple runs in order to better control and test the cluster along the upgrade process. When using this feature keep in mind best practices such as updating write leaders last and testing in a dev/staging environment first.	52558, 45181
HAProxy health checks now correctly use HTTPS for SSL-enabled Patroni clusters. Fixed a bug whereby HAProxy was incorrectly configured to use HTTP for health checks against Patroni nodes, even when the Patroni REST API was secured with SSL (`patroni_ssl_enabled: true`). This misconfiguration caused health checks to fail, leading HAProxy to mark healthy Patroni nodes as down. The HAProxy configuration logic has been updated to use `check-ssl verify none` when `patroni_ssl_enabled` is true, ensuring that health checks are correctly performed over HTTPS.'	
Fixed an issue with `pgaudit` extension when used with `edbpge` Postgres. Previously, adding `pgaudit` to the list of Postgres extensions with the `edbpge` Postgres flavour would fail because the dictionary for handling recognized extensions was missing an entry for `edbpge` and its corresponding package names. This entry has been added, and additional logic incorporated to handle naming change for `pgaudit` packages from version 16 onwards.	
Fixed an issue where global proxy routing was not set up correctly in PGD 5.9. When creating a new configuration file via 'tpaexec configure -a PGD-Always-ON --pgd-proxy-routing global (...)', the configuration by default will create a top group node with enable_proxy_routing & enable_raft equal to true, and subgroups enable_proxy_routing & enable_raft equal to false. When creating a new configuration file via 'tpaexec configure -a PGD-Always-ON --pgd-proxy-routing local (...)', the configuration by default will create a top group node with enable_proxy_routing equal to false & enable_raft equal to true, subgroups have enable_proxy_routing & enable_raft equal to true. and subgroups enable_proxy_routing & enable_raft equal to false.	

Description	Addresses
TPA will now correctly skip the `postgres/pgpass` task on PEM-agent Barman nodes The `postgres/pgpass` task runs on Postgres nodes with the pem-agent role in order to add the `postgres_user` to the `.pgpass` file in this user's home directory. In previous versions, the task would also attempt to run on Barman nodes that have the pem-agent role. This would fail since the directory does not exist. This task is now excluded for Barman nodes.	

2.7 Trusted Postgres Architect 23.39.0 release notes

Released: 22 August 2025

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.39.0 include the following:

Highlights

- Support for s390x architectures
- Support for `edb_stat_monitor` extension
- EFM property support for `auto.rewind`, `jdbc.properties`, `jdbc.loglevel`, and `check.vip.timeout`

Enhancements

Description	Addresses
Added support for PEM on Ubuntu 24.04. There was no specific package defined for this distro which caused deployment errors. Additionally, this ensures that either mod_wsgi or edb's mod_wsgi module is enabled by default where it applies.	
Added `edb_stat_monitor` to recognized extensions. When a user specifies `edb_stat_monitor` as an entry in either `postgres_extensions` or the list of extensions named under `postgres_databases`, TPA will handle installing the correct package, creating the extension and including it in the `shared_preload_libraries`.	
Add support for s390x (aka IBM Z and LinuxONE) TPA now supports the s390x CPU architecture. Supported s390x operating systems are RHEL 8 and 9 and SLES 15. TPA can run on these systems and use them as target hosts for cluster deployment. Because s390x binaries are not available for all PyPI packages, it is highly recommended that you install the `tpaexec-deps` package from EDB rather than rely on installation from PyPI during setup. Similarly, because there is not a PGDG RPM repo for s390x you must have access to EDB Repos for TPA to install Postgres and other cluster components from packages.	
Added `auto.rewind` property for EFM 5.1 and above. Starting with EFM 5.1, there will be a new property `auto.rewind` that is part of a new feature to attempt to rebuild failed primary db servers.	
Add EFM `jdbc.properties` property support This adds support for the `jdbc.properties` property in EFM 5.0 and later.	
Add `jdbc.loglevel` property for EFM 5.1 and above. Starting with EFM 5.1, there will be a new property `jdbc.loglevel` that is used to increase information logged from the JDBC driver. This property can be used to get more information when there are connection problems, e.g. when using ssl for database connections.	

Description	Addresses
Add `check.vip.timeout property` for EFM 5.1 and above Starting with EFM 5.1, there will be a new property `check.vip.timeout` that is used to control how long EFM will keep checking is the VIP (if used) is reachable before promoting a standby.	

Changes

Description	Addresses
Ubuntu 20.04 (Focal) is now a legacy distribution. Ubuntu 20.04 is now out of support upstream and is no longer fully supported by TPA.	
Removed support for obsolete EDB/2q repositories. Now that the old 2q repositories and EDB Repos 1.0 are no longer available, TPA will not try to use them or check for configuration related to them. All packages that were formerly available in those repositories are now available in the EDB Repos 2.0 repositories.	
TPA will now suppress PGDG repos when using EDB repos on SLES. When running `tpaexec configure` for a SLES system, use either EDB repositories or PGDG repositories, not both, matching the behavior of other operating systems. This ensures that packages such as barman, which are available in both places, are consistently sourced from the same repository as other packages.	
TPA will now use `bdr_package_version` as default PGD Proxy version. `pgd_proxy_package_version` and `pgdcli_package_version` may be explicitly defined in config.yml. If they are not explicitly defined, they now default to the value of `bdr_package_version`, if it is set. If none of them is set, the latest available versions of packages are installed. Setting these package versions to different values is not supported and is only useful for testing.	49150
Reduced default for `vacuum_cost_delay` to 2ms In recent Postgres versions, the suggested default for `vacuum_cost_delay` has been reduced to 2ms. TPA now matches this.	
Documented node promotability logic and the `efm-not-promotable` role for EFM. Added a new section to the `efm.md` documentation that explains how TPA determines whether a node is eligible for promotion during failover. The update clarifies the rules for promotability, including the roles of witness nodes, cascading standbys, and nodes explicitly marked with the `efm-not-promotable` role. This enhancement should help users understand and control failover behavior, reducing the risk of unintended promotions in EFM-managed clusters.	
Document `include_vars` behavior for templated variables. Since the `include_vars` module immediately parses and evaluates expressions, nested variables do not exist at the point they are loaded from `config.yml` and thus are undefined when evaluated in templated expressions. This is now documented with an example so it is clearer for users.	45722
Use Docker images from Rocky Linux organization. TPA now uses the Rocky Linux organization's docker images rather than docker hub's "official" ones. These are more frequently updated and hence less likely to cause dependency problems with newer packages.	

Description	Addresses
Updated architecture info for PGD 6 architectures. The PGD6 architectures now have correct metadata and therefore appear as expected in the output of `tpaexec info architectures`.	

Bug Fixes

Description	Addresses
Fixed an issue whereby `replication` user was missing from `.pgpass` on the primary node. The `replication_user` entry is required the `.pgpass` file for both `replica` and `primary` nodes managed by either `efm` or `patroni` as the `failover_manager`. Previously, it was only added to the .pgass file for `replica` nodes, resulting in connection issues to the primary node after a switchover.	48224
Fixed an issue whereby TPA would attempt to enable routing for subscriber-only groups in PGD6 clusters. Previously, TPA would mistakenly try to enable routing for subscriber-only groups during PGD6 deployments. This property is not editable anymore as it forms core part of the definition of subscriber-only groups in PGD6.	49673
Fixed an issue whereby PEM deployment would fail with PEM 10.1.1. The TPA task `Register PEM backend database server for monitoring and configuration` explicitly calls the `pem.setup` SQL function. This is not considered part of the public API of PEM and the signature changed between 10.1.0 and 10.1.1 causing TPA's call to this function to fail. This fix addresses the issue by adjusting the function call according to the installed PEM server version. To accommodate this change we have introduced a new `pem/server/facts` Ansible role that is responsible for collecting facts about the installed PEM version. This also means that TPA will no longer attempt to run `postgresexpert.sql` on PEM 10, where Postgres Expert is no longer present.	
Fixed an issue whereby \`harp_dcs_user\` was not created when using multiple failover managers	49150
Fixed an issue whereby `bdr_node_groups` could not be overridden. If a file supplied in the --overrides-from argument to `tpaexec configure` sets `bdr_node_groups` in `cluster_vars`, the contents of this will now be added to the node groups automatically created.	50550
Fixed issue whereby TPA would use `pgdproxy` user in DSN on PGD6 nodes. Since PGD version 6 has a built-in Connection Manager which replaces PGD Proxy, the `pgdproxy` Postgres user should not be used in any DSNs. A new variable `bdr_connection_manager_route_dsn` is introduced for defining connection strings to Connection Manager. Users can additionally define `bdr_connection_manager_dsn_attributes` in their `config.yml` to specify additional connection parameters in the DSN.	
Fixed an issue with `ignore_slots` for Patroni clusters with multiple Barman servers. For Patroni clusters TPA will now dynamically generate the ignore_slots setting corresponding to Barman nodes in the cluster. This change ensures that Patroni will not remove the physical slots created for each of the Barman nodes. This is especially important for the Patroni clusters with more than one backup servers; including shared Barman nodes for example.	50748

Description	Addresses
Fixed usability and improve reliability of the TPA Docker image. Upgraded Base Image to Debian Trixie to meet TPA's Python 3.12 minimum requirement, ensuring compatibility with modern dependencies. • Removed `--use-community-ansible` flag as it is no longer supported. • Included the `openssh-client` package to prevent `tpaexec` failures due to missing `ssh-keygen`, enhancing out-of-the-box functionality. • Introduced Build-Time Versioning: Added an `ARG` variable to pass the output of `git describe` from the host during build, storing it as `/opt/EDB/TPA/VERSION` for improved version tracking. • Revised `docker/README.md` and `docs/src/INSTALL-docker.md` to reflect the latest setup instructions and best practices.	
Fixed an issue whereby Barman could not run `switch-wal`. For Postgres 15+, the Barman user is now created with the `pg_checkpoint` role. This allows Barman to run CHECKPOINT without the need of being superuser, `barman switch-wal --force` will not fail any more.	

2.8 Trusted Postgres Architect 23.38.1 release notes

Released: 25 June 2025

Trusted Postgres Architect 23.38.1 is a bug-fix release that resolves the following issues:

Bug Fixes

Description	Addresses
Fixed an issue with the configure command for PGD6 using bare platform. Fixed a bug with tpaexec configure command on the newly released architecture for PGD6 (PGD-S and PGD-X) whereby trying to generate a cluster using `--platform bare` would result in an Unknown Platform error. This fix ensures that the configure command successfully generates a PGD6 configuration file for the bare platform.	49673
Fixed an issue whereby pgpass_users were not correctly added to the .pgpass file. Previously, any user in the `postgres_users` list specified with `generate_password: true` AND included in the `pgpass_users` list would NOT be added to the `~postgres/.pgpass` file on the initial deploy because the user's password did not yet exist when the pgpass task was executed, thus the user was skipped. This is fixed by invoking the pgpass task once more after all the Postgres users have been created. The `repmgr` and `replication` users were previously included in the `default_pgpass_users` list. They are now added to `pgpass_users` and hence to the .pgpass file as required by the replication manager in use. The `postgres_user` (`postgres` or `enterprisedb` by default) is still part of the `default_pgpass_users` list. If this is overridden by a `pgpass_users` list in `config.yml` that does NOT include `postgres_user` , a PEM-enabled cluster will fail to register agents as it needs the encrypted `postgres_user` password. This is fixed by adding the `postgres_user` to the `~postgres/.pgpass` file as part of the PEM agent tasks.	
Fixed an issue whereby an invalid cluster_vars dictionary would lead to a failed deploy or an unexpected configuration. Previously, if an invalid dictionary is set as the `cluster_vars` dictionary in `config.yml` (such as `cluster_vars` variables referencing other `cluster_vars` variables), TPA would swallow any Ansible errors by falling back to the default value of an empty dictionary. This resulted in every `cluster_vars` variable being undefined, so each was set to it's TPA-default value. The resulting cluster would be entirely different than what the user specified in their `config.yml` file. This is fixed by asserting that the `cluster_vars` dictionary is defined and non-empty when the configuration file is loaded. Also as a final bailout, the `cluster_vars` variable now no longer defaults to an empty dictionary. This allows Ansible to throw an error when creating `group_vars` and terminate.	

2.9 Trusted Postgres Architect 23.38.0 release notes

Released: 9 June 2025

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.38.0 include the following:

Highlights

- Support for deploying EDB Postgres Distributed version 6 in both Expanded and Essential architectures
- Improvements to PgBouncer implementation including a mitigation for CVE-2025-2291
- Support for PEM 10.1

Enhancements

Description	Addresses
Support for PGD6 architectures. TPA can now configure and deploy clusters using the PGD-X and PGD-S architectures based on PGD6 The PGD-S architecture implements PGD Essential and the PGD-X architecture implements PGD Expanded. These architectures have sensible default configurations and also accept various configure options to customize their behavior. PGD 6 deployments no longer include pgd-proxy; instead, PGD's built-in Connection Manager is configured. Testing support for the new architectures is added.	
Configure PEM to monitor Barman when both are present in a cluster. When a cluster is configured with PEM enabled (using the `--enable-pem` option) and includes a Barman node, the following actions are now performed automatically: • `enable_pg_backup_api` is set to `true` in `config.yml` • The `pem-agent` role is assigned to the Barman node • The Barman endpoint is registered with the local PEM agent. These changes simplify setup and ensure seamless integration between PEM and Barman.	
Added support for Rocky Linux 9 on AWS. TPA now supports configuring a cluster using Rocky Linux 9.5 on the `aws` platform. This is now the default version for Rocky Linux on AWS if a version is not specified.	
Added support for pg_backup_api on SLES 15. TPA will now configure pg_backup_api is on SUSE Linux Enterprise Server 15 (SLES 15) when PEM monitoring is enabled and a Barman node is present in the cluster.	

Changes

Description	Addresses
Treat `PEM_DB_PORT` as a string for PEM 10.1 and above. PEM 10.1 adds support for multi-host connection strings from the web application to the backend servers. To support this change, the `PEM_DB_PORT` parameter in PEM's `config_setup.py` file is now a string rather than an integer. While TPA does not yet support deploying HA PEM configurations, TPA will now correctly set this parameter as a string when the PEM version is 10.1 or greater.	
TPA will now skip repository checks when `repo` is excluded from tasks. The `repo` tag is available for exclusion, but previously would only skip tasks under the `sys/repositories` role. Now it also skips over the initialization tasks which check which repositories to use and the verifies the credentials to access them are provided.	
Improved the behavior of `postgres_package_version`. Setting `postgres_package_version` will now cause TPA to install the selected version of various postgres-related components on Debian or Ubuntu systems installing EDB Postgres Advanced Server or EDB Postgres Extended Server. This avoids dependency resolution problems when newer package versions are visible in repositories.	
Added a new task selectors `create_postgres_system_user` and `create_pgd_proxy_system_user`. Added new task selectors that allow to skip the postgres_user and pgd_proxy_user operating system user. This allows clusters to use remote users created by a centralized user management such as NIS. This can be set in config.yml: `cluster_vars: excluded_tasks: - create_postgres_system_user - create_pgd_proxy_system_user`	48601,44388
TPA will now redirect PgBouncer to the new primary in M1 repmgr clusters during switchover. TPA will now ensure that PgBouncer instances are redirected to the new primary node after using the `switchover` command in a repmgr + PgBouncer cluster that has `repmgr_redirect_pgbouncer` set to true. The `tpaexec switchover` command will now ensure that PgBouncer instance connect to the new primary node. A new `revert_redirect` variable can also be set as extra-variable after a first switchover is done to revert back to the initial primary node.	
Made `key_id` / `gpgkey` optional in custom repository definitions. The `key_id` and `gpgkey` parameters (for apt and yum custom repositories definition) are not required by the underlying modules, there are use cases where this is not easy to provide ahead of installation. With this change, TPA does not make it mandatory to provide those in custom repository definitions.	
Fixed `verify-settings` check in `tpaexec test` for PGD CLI 5.7.0+. The output for the PGD CLI command `pgd verify-settings` changed in PGD 5.7.0. TPA now correctly parses the output when using version above 5.7.0 of PGD CLI. Note, verify-settings will be deprecated along with other commands in future PGD releases. those commands are now wrapper calling the new commands until the deprecation occurs.	
Improved logic for granting permissions to `barman_role`. TPA now uses the `postgresql_privs` module to apply the grant on `barman_role`, so that changes are only applied when needed. In addition, in PGD clusters, TPA will use the `bdr_database` on second deploys so the DDL is replicated across the cluster by PGD.	
Documented `cluster_vars` variable templating in config.yml Added documentation to explain correct templating procedure for variables defined under `cluster_vars` with a worked example in order to avoid confusion from unexpected behavior associated with inventory variables not being defined when improperly templated in `config.yml`.	48797

Description	Addresses
TPA will now raise an ArchitectureError for if an invalid CIDR is passed to `--network` flag during configure. By default, the Python standard library `ipaddress` package enforces 'strict' interpretation of the CIDR, whereby the IP used should be the network address of the range. Previously, any IP passed to the `--network` flag that contained host bits would dump a stacktrace due to the raised ValueError. That exception is now caught and an ArchitectureError is raised to display a clear message to the user about the `--network` parameter.	
Added a default value for EFM application.name property. If the EFM `application.name` property is not set for a node, TPA will use the Postgres `cluster_name` property as a default. EFM uses this value when performing a switchover or when building a new standby database.	
TPA now uses the EDB repository setup script on SUSE. Previously, TPA did not use the EDB repository setup script on SUSE because it did not work on repeat deploys. Zypper would raise because the repositories that the script attempts to install already exist, and require unique names. Now that the repository setup script task is skipped if the repositories are already installed, this issue is not encountered.	

Bug Fixes

Description	Addresses
TPA will now create the `pgbouncer_get_auth()` function in dedicated database. The `pgbouncer_get_auth()` function was created in the `pg_catalog` schema and execute granted to the `pgbouncer_auth_user`. This function was created in every database, but this was not necessary for PgBouncer. A failure may be encountered during the `pgd node upgrade` process when this function was created in the `pg_catalog` schema as it is not included in the dump created by `pg_dump`. A later task attempts to run a `GRANT` on this function and fails, as the function is not restored since it was not originally dumped. Now this function is only created in a single database, named under the `pgbouncer_auth_database` variable in `config.yml`, which defaults to `pgbouncer_auth_database` if not included. It is only created if at least one instance with `pgbouncer` role is included in the cluster. A warning is also issued during deploy and upgrade if any databases define this function under the `pg_catalog` schema, as a future TPA release may remove the function from that schema. The `pgbouncer_get_auth()` function itself used by PgBouncer `auth_query` has been updated to address CVE-2025-2291. This vulnerability allowed for authentication using expired passwords, potentially granting unauthorized access because the auth_query mechanism did not consider the `VALID UNTIL` attribute set in PostgreSQL for user passwords.	42911, 45068
Fixed an issue whereby some tasks were incorrectly skipped when the `--check` option was used. In PGD clusters without HARP, the `Read current configuration file if exists` task needs to run in check mode to ensure we have the information available to correctly skip the following HARP check task. However, by default Ansible skips tasks using the the `shell` module during check mode, meaning this task did not run, resulting in a spurious failure on subsequent tasks. We now let Ansible know that this task has to be run.	
Fixed a bug whereby settings added to \`ignore_slots\` via \`cluster_vars['patroni_conf_settings']['bootstrap']['dcs']\` were not merged into the eventual config.	
Fixed an issue where RAFT checks for BDR nodes with replica role were not skipped during upgrade. Physical replication of a `subscriber-only` node can be achieved in a `PGD` cluster by installing `repmgr` as a failover-manager and designating the `subscriber-only` node as the `primary` and listing another BDR data node as the `backup`; this backup node is given the `replica` role. This configuration would result in the PGD upgrade process failing, since TPA expects BDR data nodes to have RAFT enabled, but the physical replica BDR data node (with both `replica` and `bdr` roles) by design does not. As a fix, certain BDR-specific tasks in the upgrade process now skip any node that has a `replica` role, allowing for a successful upgrade.	46186

2.10 Trusted Postgres Architect 23.37.0 release notes

Released: 24 March 2025

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.37.0 include the following:

Highlights

- The `upgrade` command now supports minor-version upgrades of Postgres in Patroni and EFM clusters
- Upgrades in repmgr clusters have also been improved

Enhancements

Description	Addresses
Minor-version Postgres upgrades for M1 + Patroni clusters TPA can now upgrade Postgres to the latest minor version on an M1 cluster which uses Patroni as the failover manager. The upgrade process stops Barman on any Barman server in the cluster, then upgrades the replicas in the cluster. Then it switches to one replica as a temporary primary, upgrades Postgres on the original primary, and switches back to the original primary. Patroni's handling of the cluster is paused during the process and resumed afterwards. Then Barman is restarted and cluster health checks are run.	102662
Improved minor-version Postgres upgrade for M1 + repmgr clusters Witness nodes were previously omitted from upgrade, they are now upgraded along with replicas. Postgres service restart is now more reliable and always run right after the package upgrade on the node is finished.	
Minor-version Postgres upgrades for M1 + EFM clusters TPA can now upgrade Postgres to the latest minor version on an M1 cluster which uses EFM as the failover manager. The upgrade process stops Barman on any Barman server in the cluster, then upgrades the replicas in the clusters. Then it switches to one replica as a temporary primary, upgrades Postgres on the original primary, and switches back to the original primary. The EFM agent is started and stopped on the different servers at the correct times. Then Barman is restarted and cluster health checks are run.	103595
Separated changed from unchanged tasks in output In TPA's default output plugin, tasks which return "ok" but with no changes are now separated from ones that have reported changes, which are now highlighted in yellow.	
Improved the speed of Docker instance deprovisioning When deprovisioning docker instances, TPA now kills the container instead of stopping it, and does so to all the instances in parallel.	
Added `check.num.sync.period` property for EFM 5.x Starting with EFM 5.0, there is a new property `check.num.sync.period` that defines how often a primary agent will check to see if num_sync needs to change on the primary database. This can now be specified in `config.yml`.	

Changes

Description	Addresses
TPA will now copy EFM config files if they are removed, even if no configuration changes If either the `efm.nodes` or `efm.properties` configuration files do not exist in the top-level EFM directory, the `efm upgrade-conf` command copies them from the `/raw` directory, even if there have been no configuration changes. This amends previous behavior that required a configuration change before the `upgrade-conf` command would run and copy files.	
TPA will now verify that the URI for EDB repository setup is accessible The EDB repos are set up using the setup script following the EDB Repos 2.0 documentation: piping the cURL output to bash for execution. However, if a user passes a nonexistent `EDB_SUBSCRIPTION_TOKEN` or repository to cURL, the exit code gets silently swallowed and replaced with a 0 because bash executes an empty input. In this version, a request is dispatched to a repository's GPG key endpoint to ensure a `404` response is not returned *before* continuing to download the setup script. Additionally, tasks related to EDB repository set up are now skipped if the repository has already been set up.'	
TPA will now raise an ArchitectureError when BDR-Always-ON is configured with BDR version 5 An architecture error is now raised during `tpaexec configure` if `--bdr-version 5` is passed with `-a BDR-Always-ON` alerting the user that BDR version 5 should be used with `PGD-Always-ON`.	

Bug Fixes

Description	Addresses
Fixed `shared_preload_libraries` computation during deploy Fixed a limitation of Ansible's handling of list ordering that would trigger unneeded and uncontrolled rewriting of the `shared_preload_libraries` and subsequently require a Postgres service restart, even on second deployment scenarios with no changes to the configuration.	
Fixed an issue whereby unnecessary replication slots were created when configuring Patroni When setting up a Patroni cluster, a replication slot was created for each etcd-only node. This caused problems because the unused slots cause the WAL to accumulate. Slots are now only created for the DB servers.	
Fixed `patronictl switchover` command usage TPA will now correctly use `--leader` instead of the deprecated `--master` parameter when using `patronictl switchover` command.	
Fixed an issue with counting instances in BDR-Always-ON clusters containing PEM Fixed a bug whereby in certain circumstances, TPA would incorrectly calculate the number of instances in a BDR-Always-ON cluster with a PEM server, causing "tpaexec configure" to fail with "StopIteration".	
Fixed an issue whereby a PEM server could be incorrectly picked as a cluster primary in EFM cluster. If a cluster is created with a pem-server, that backend is not monitored by EFM, hence, that node shouldn't be included when discovering a Postgres primary for the entire cluster. This behavior is now correctly implemented by TPA.	45279
Fixed duplicated lines in .pgpass files Fixed a bug whereby extra lines could be added to .pgpass for the same user when re-running 'tpaexec deploy'.	

2.11 Trusted Postgres Architect 23.36.0 release notes

Released: 19 February 2025

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.36.0 include the following:

Highlights

- Support for Ubuntu 24.04 and SLES 15 SP6
- Version specifiers for all cluster components
- Better support for EDB Failover Manager (EFM) 5

Enhancements

Description	Addresses
Added support for package version specifiers for all cluster components The following software packages now accept an `--xxx-package-version` option to the `tpaexec configure` command, which populates `xxx_package_version` in the generated `config.yml` - barman - pgbouncer - beacon-agent - etcd - patroni - pem-server - pem-agent - pg_backup_api - pgd_proxy - pgdcli - repmgr.	
Added support for Ubuntu Noble 24.04 TPA runs on Ubuntu 24.04 and supports this OS as a target host. **EDB packages for Ubuntu 24.04 will be generally available at the end of March 2025. Before that, deployments needing EDB packages will likely fail.**	
Added support for EFM 5 "auto resume" properties Starting with EFM 5.0, the `auto.resume.period` property has been broken into two properties, one for the startup case and one for the db failure case. This change adds the correct properties based on the `efm_version` being used.	
Added support for EFM 5 'backup.wal' property. Starting with EFM 5.0, a new property 'backup.wal' has been added. This change adds the new property if the version of EFM is 5 or higher.	
Added support for SLES 15 SP6 When SLES is requested at configure-time, TPA will now install SLES 15 SP6. The Docker and EC2 images are now SP6, and the systemd-sysvcompat package is installed on SLES, so that local boot-time scripts continue to work.	

Changes

Description	Addresses
Use the latest Barman from PGDG on RHEL-like systems TPA previously defaulted to Barman 3.9 when installing from PGDG on a RHEL-like system, as a workaround for broken packages. More recent barman packages are OK, so we now let yum install the latest packages.	

Description	Addresses
Set up EDB repositories via setup script for RedHat and Debian This change simplifies EDB repository setup on TPA nodes into a single task for `dnf` , `yum` and `apt` package managers. This should reduce the occurrence of unexpected HTTP errors when adding EDB repositories to target hosts.	
Fixed documentation for `efm_conf_settings` Previously, documentation stated *"You can use efm_conf_settings to set any parameters, whether recognized by TPA or not. Where needed, you need to quote the value exactly as it would appear in efm.properties"* However, the `efm.properties.j2` template uses the values from `efm_conf_settings` as an Ansible dictionary, so the entries must be written in `key: value` form. `yaml cluster_vars: efm_conf_settings: notification.level: WARNING ping.server.ip: <well known address in the network>`	
Bump Python dependency to version 3.12 TPA now requires Python 3.12. When installing TPA from EDB Repos 2.0, an appropriate Python interpreter will be installed automatically.	
Wait for protocol version update during PGD upgrade During upgrade from PGD3 to PGD5, the protocol version update may take some time. The PGD 5 specific config changes will fail if they are attempted before the protocol version change. We have added a wait to avoid such failures.	
Update `<clustername>.nodes` when new nodes are added to an existing EFM cluster In previous versions of TPA, when a new EFM node was added to `config.yml` , it was not listed in the `Allowed node host list` on the existing EFM nodes in the cluster. The task which executes `efm upgrade-conf` and propagates the changes from `/raw/<clustername>.properties` and `/raw/<clustername>.nodes` is now run when EITHER of these files are changed. This results in the new EFM node being written to the `<clustername>.nodes` file and `efm cluster-status` including it in the `Allowed node host list` .	
Include `PGPORT` in the postgres user's `.bashrc` file The `PGPORT` environment variable will now be exported as part of the postgres user's `.bashrc` file. It defaults to the port value used by the selected `postgres_flavour` , or `postgres_port` if specified in the config file.	

Bug Fixes

Description	Addresses
Fixed an issue whereby databases were created with the default parameters rather than as configured In previous versions of TPA, in order to suppress the `CREATE EXTENSION` statement for extensions that do not require it (aka 'modules'), the entire `postgres_databases` hash was modified. This introduced a bug, since a new hash was created that ONLY contained the database `name` and list of `extensions` , ignoring all other configuration settings for the database (`owner` , `template` , `encoding` etc). This resulted in databases being created with the default parameters rather than as configured. To fix this, the modules are removed from the list of extensions and the resulting list is passed to the task which runs CREATE EXTENSION.	44539
Fixed an issue with `shared_preload_libraries` on Patroni clusters Entries in shared_preload libraries are now treated correctly by Patroni. This fixes a bug whereby adding the pglogical extension to a Patroni cluster via `config.yml` would fail.	

Description	Addresses
Fixed a bug whereby deploys would fail for distributions which have no additional repository setup commands for extensions In previous versions of TPA, when `postgis` was added to `extra_postgres_extensions` or the `extensions` list of a database in `postgres_databases`, deploys would fail for Debian, SLES and Ubuntu because their list of `repository_setup_commands` was empty (only RHEL has an additional command to run `crb enable`). This empty list was passed to the `command` module, which would fail with `no command given`, resulting in deployment failure. The `Automatically run additional repository setup commands for recognized extensions` task is now skipped if the distribution has no additional commands to run.	
Fixed an issue whereby TPA attempted to create replication slots even when `repmgr_use_slot` was set to 0	41776
Set `bdr_client_dsn_attributes` as the default for `pgd_proxy_dsn_attributes` and `pgd_cli_dsn_attributes` Because pgd-proxy and pgd-cli are written in Go and use a Go driver, they do not support the full set of parameter keywords supported by libpq. In the case a cluster has installed pgd-proxy and/or pgd-cli and has configured `bdr_client_dsn_attributes` with parameters that the Go driver does *not* support, two new configuration variables must be included: `pgd_proxy_dsn_attributes` and `pgd_cli_dsn_attributes`, containing only additional DSN parameters that the Go driver supports. Conversely, if pgd-proxy and pgd-cli are installed and `bdr_client_dsn_attributes` does not include any Go-incompatible parameters, the connection strings for these tools will be configured with the attributes in `bdr_client_dsn_attributes`. This amends unexpected behavior where the `pgd_proxy_dsn_attributes` and `pgd_cli_dsn_attributes` were defaulting to empty strings when not defined, even though the user was expecting the parameters in `bdr_client_dsn_attributes` to be used.	44819
Use standard form of home directory for etcd When creating the etcd user, TPA now refers to its home directory without a trailing slash, matching the usage of other tools like 'useradd'.	
Fixed an issue whereby the `cluster_ssh_port` setting was not respected by Barman The Barman configuration is now able to use custom ssh port set via the `cluster_ssh_port` in `config.yml`, which defaults to 22 if it is not set. The `-p` / `--port` flags are now included in the `ssh` command in `barman.d.conf` and `barman-wal-restore` /`barman-wal-archive' commands respectively.	

2.12 Trusted Postgres Architect 23.35.0 release notes

Released: 25 November 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.35.0 include the following:

Highlights

- Options for STIG/CIS compliance.
- Support for PGD Lightweight architecture
- Postgis is now a recognized extension.
- Docker `configure` creates named networks with static IP addresses.
- Support for RedHat Enterprise Linux 9 for ARM architectures.
- Support for PostgreSQL, EDB Postgres Extended, and EDB Postgres Advanced Server 17.

Enhancements

Description	Addresses
Support STIG/CIS compliance TPA now supports command-line options to create a cluster configured to conform to many of the requirements of the STIG and CIS security standards. These options cause TPA to set postgresql.conf settings as defined in the relevant standards, to install required extensions, to configure other aspects of system behaviour such as filesystem permissions and user connection limits, and to check for other requirements such as FIPS crypto standards which TPA can't directly impose. The clusters thus generated are not certified by TPA to conform to the standards, but much of the groundwork of creating a conforming cluster is now automated.	
Add support for PGD Lightweight architecture TPA is now able to generate a PGD Lightweight architecture comprised of three nodes in two locations (2 nodes in Primary and one in Disaster Recovery) designed to ease migrations from physical replication. Users can now run `tpaexec configure lw -a Lightweight --postgresql 15`.	
Have `configure` create a user-defined network on Docker The configure command will now automatically add a named network and static IP addresses to config.yml when Docker is the selected platform. The network name is the same as the cluster name and the address range follows the existing semantics of the --network option with the exception that only one subnet is used for the whole cluster rather than one per location. If a subnet prefix is not specified by the user, TPA will attempt to select a prefix which results in a subnet large enough to fit the whole cluster. The key `ip_address` may now be used to specify a static IP for a Docker instance as long as a named network is specified in the config.yml.	
Added experimental support for using an existing Barman node as backup node in new cluster When using an existing Barman node as a backup node in a new cluster, users can set `barman_shared: true` in the Barman instance's vars with the platform set to `bare` and other information supplied as usual for bare instances. This change allows TPA to skip some configuration steps that would otherwise fail due to usermod issues, as the Barman user already has running processes from previous deployments. The shared Barman instance is treated as a bare instance, so the required access, including the Barman user's access to the target PostgreSQL instances, must be already in place. Copying the Barman user's keys from the original cluster to the new cluster can be used to achieve this, see the Barman section of the TPA documentation for detailed information.	

Description	Addresses
Add `postgis` to list of recognized extensions The PostGIS package will automatically be added when a user specifies `postgis` as an entry in either `postgres_extensions` or the list of extensions named under `postgres_databases`. Also enables the CRB (Code Ready Builder) repository for RHEL-compatible distributions so PostGIS dependencies can be installed.	
Enable EFM probes when a PEM agent is registered on an EFM node The `--efm-install-path` and `--efm-cluster-name` flags are set when a PEM server is registered on an EFM node. The `Streaming Replication`, `Failover Manager Node Status` and `Failover Manager Cluster Info` probes are enabled when a PEM agent is registered on an EFM node.	
Support RedHat Enterprise Linux 9 for ARM architectures Packages are now published targeting RHEL 9 ARM64, and TPA supports deployments using this architecture and OS. Also updated the list of supported AWS images to include the RedHat 9 ARM64 AMI provided by Amazon. The default `instance_type` for ARM64 EC2 instances has been updated from `a1` to `t4g`, which is the current generation processor available for burstable general purpose workloads.	
Support PostgreSQL, EDB Postgres Extended, and EDB Postgres Advanced Server 17 Clusters can be configured to use PostgreSQL, EDB Postgres Extended and EDB Postgres Advanced Server version 17. Barman no longer needs to install the postgres server package to get the `pg_receivewal` binary when using EDB Postgres Advanced Server 17 or EDB Postgres Extended 17 since the binary has been added to the client package for these versions. TPA raises an architecture error when a cluster is configured with `repmgr` as the failover_manager as it is not available for Postgres 17. Updated documentation to reflect supported versions.	
Make `password_encryption` algorithm for `efm` Postgres user configurable. Expose a configurable `efm_user_password_encryption` variable which should be set to either `'md5'` or `'scram-sha-256'` depending on user requirements. This controls the `auth-method` for the `efm` Postgres user in `pg_hba.conf` and the algorithm used for generating it's encrypted password. In clusters deployed with `compliance` configured to `stig`, the 'efm' Postgres user's `auth-method` in `pg_hba.conf` will be set to `scram-sha-256` since FIPS-enabled operating systems do not allow `md5` to be used.	
Allow multiple addresses to be supplied with hostnames When using the `--hostnames-from` option to `tpaexec configure`, you can now include two ip addresses on each line, which will be included in the generated config.yml as public_ip and private_ip.	

Changes

Description	Addresses
Remove deprecated `PermissionStartOnly` in postgres.service.j2 template `PermissionsStartOnly` has been deprecated and is now achieved via `ExecStartPost=+/bin/bash...` syntax	

Description	Addresses
The `barman` Postgres user is no longer a superuser Certain required privileges are granted to Postgres role, `barman_role`, which is then granted to the `barman` Postgres user. This avoids creating the `barman` user as a superuser. This role can also be granted to other Postgres users by adding it to their `granted_roles` list using `postgres/createuser`. The `barman_role` is created as part of the Barman tasks; if Barman is not used, this role will not be created. Therefore, the task that grants privileges to this role is only executed if the `barman_role` username is in the list of Postgres users that are created. The 'barman' user now has `NOSUPERUSER` explicitly specified as a role attribute. If a cluster was deployed with a previous TPA version (which created the 'barman' user as a superuser), deploying with this version will remove the `superuser` role attribute from the `barman` user.	
Add new option `harp_local_etcd_only` when using etcd with HARP Add new optional var `harp_local_etcd_only` available when using etcd with HARP. This option tells HARP manager to connect to local etcd node. This recommendation follows the best practices learnt by doing the same when `bdr` as consensus procotol is being used. The default mode of adding multiple endpoints can lead to performance issues in some cases. This option is added to give more control to the user.	
Improve postgres-monitor script Improve postgres-monitor script to better manage recoverable errors and add retries on network errors to ensure that it won't return failure when it just didn't allow enough time for postgres service to be fully started.	
Only add nodes with `efm` role to cluster `efm.nodes` file Previously the `pemserver` and `barman` nodes were added to the `Allowed node host list` in EFM when they were not relevant to EFM functions. Refactored the task that writes the `efm.node` configuration to only include those nodes that have `efm` in their list of roles.	

Bug Fixes

Description	Addresses
Fix tpaexec test for pgd-proxy config verification Fixed a bug whereby the test that ensures the current pgd-proxy configuration matches the expected configuration would fail for version < 5.5.0. This fix ensures that TPA won't try to query configuration keys added in version 5.5.0.	
Fix case where `primary_slot_name` added for EFM compatibility interferes with `bdr_init_physical` A `primary_slot_name` is configured on the primary node to ensure the old primary uses a physical slot for replication during an EFM switchover. However, 'bdr_init_physical' attempts to use it for node initialisation and hangs indefinitely since the slot does not exist in a PGD installation. This `primary_slot_name` is now conditionally set explicitly when the `failover_manager` is EFM to avoid setting it unnecessarily.	
Download correct `bash-completion` package version If the `pgdcli_package_version` is specified in `config.yml`, the `bash-completion` package is incorrectly named because the `packages_for` filter erroneously appends the `pgdcli_package_version` to the package name. This results in an attempt to download a nonexistant package. The `bash-completion` package is now appended to the list after the `packages_for` filter, since it's version is independent from the `pgdcli_package_version`.	

Description	Addresses
Fix an issue whereby in some cases error messages would be repeated even after successful tasks. TPA now clears the error message stack after each task to ensure messages are not spuriously repeated	
Fix issue that prevented the addition of replicas to Patroni clusters Fixed an issue whereby new replicas in Patroni clusters would fail with errors related to replication slots.	
Add `pem-agent` role on barman nodes at most once for M1 architecture If `--enable-pem` and `--enable-pg-backup-api` are passed to `tpaexec configure`, `pem-agent` is added twice to the `barman` node if it is also a `witness`. Fixed by consolidating both `if` statements together to only evaluate the conditions once.	
Set `pem_python_executable` outside of the `pkg` role Fixed a bug whereby if the user excluded the `pkg` selector, later PEM-related tasks would fail because the `pem_python_executable` fact had not been set.	

2.13 Trusted Postgres Architect 23.34.1 release notes

Released: 9 September 2024

Trusted Postgres Architect 23.34.1 is a bug fix release which resolves the following issues:

Type	Description
Bug Fix	Fixed an issue whereby running deploy after a switchover fails for nodes with `efm-witness` role. The `upstream-primary` for EFM nodes is determined using the facts gathered from Postgres. This previously failed for nodes with `efm-witness` roles since they do not have Postgres. The task to determine upstream-primary is now run only on nodes with `primary` or `replica` roles.

2.14 Trusted Postgres Architect 23.34 release notes

Released: 22 August 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.34 include the following:

Type	Description	
Enhancement	TPA now has an `efm-pre-config` hook which runs after efm has been installed and its configuration directory and user have been created, but before efm is configured. It can be used to install custom efm helper scripts.	
Enhancement	TPA now has its own output plugin, which shows one line of information per task, omitting tasks for which even one line would be uninformative. The lines are indented to enable TPA's control flow to be visible, and include color-coded counts of successful, skipped, and ignored hosts. The fuller output can be turned on by setting TPA_USE_DEFAULT_OUTPUT=true in your environment, or by adding the -v switch to the command line.	
Enhancement	TPA now allows you to specify additional options when registering PEM agents by listing them under `pemagent_registration_opts` in `cluster_vars`.	
Enhancement	You can now provide your own web server certificates for use with the PEM server by including the names of the certificate and key pair for use on the PEM server in `config.yml` under the cluster_vars or pem-server instance vars `pem_server_ssl_certificate` and `pem_server_ssl_key`. TPA will copy them from the `ssl/pemserver` directory of the cluster directory to the PEM server and configure Apache/httpd accordingly.	
Enhancement	TPA now runs the EFM `upgrade-conf` command on new cluster deployments to benefit from the comments and inline documentation that are added to both `<cluster_name>.properties` and `<cluster_name>.nodes` files.	
Enhancement	TPA can now set up EFM clusters using hostname resolution instead of IP addresses for `bind.address` value. This can be invoked with tha `--efm-bind-by-hostname` option for the configure command or `efm_bind_by_hostname: true	false` in cluster_vars section of config.yml.
Enhancement	TPA now supports setting the EFM properties that added in EFM 4.9: `enable.stop.cluster: boolean, default true`, `priority.standbys: default ''`, `detach.on.agent.failure: boolean, default false`, `pid.dir: default ''`.	
Enhancement	TPA can now configure pgBouncer to use `cert` authentication for connections from pgBouncer to Postgres. This is enabled by setting `pgbouncer_use_cert_authentication` to true in cluster_vars. When enabled, the authentication method for users connecting to pgBouncer is also changed from `md5` to `scram-sha-256`.	
Change	TPA no longer supports RAID creation on AWS.	
Change	Removed EFM dependency for resolving upstream_primary. Previously, EFM was queried for the current primary on a deploy after a switchover. If EFM is not running, this will fail. Now the cluster_facts collected through Postgres are used to determine the current primary after a switchover, removing the dependency on EFM.	
Change	In EFM clusters, the `upstream_primary` is now correctly updated after switchover, resulting in the correct `auto.reconfigure` setting be set on replicas. Standbys now follow the new primary.	
Bug Fix	Fixed an issue whereby TPA would incorrectly apply proxy settings when accessing the Patroni API. The Ansible default is to use a proxy, if defined. This does not work in the (rather common) case of an airgapped environment that needs a proxy to download packages from the internet, because the proxy also intercepts (and disrupts) calls to the Patroni API.	
Bug Fix	Fixed an issue whereby TPA would set PEM agent parameters on all instances that were only appropriate for the `pemserver` instance.	
Bug Fix	Added missing entries for pgd-proxy and pgdcli default package name when using SLES operating system as target for cluster nodes.	
Bug Fix	Fix an issue whereby TPA would fail to reload/restart postgres on existing nodes to re-read configuration changes and the new node would therefore fail to connect to the cluster.	
Bug Fix	Fixed an issue whereby when taking backups from a replica, barman could fail when taking its initial backup by timing out waiting for WAL files. This is fixed by waiting for barman to complete its base backup before forcing a WAL segment switch.	

Type	Description
Bug Fix	Ensure that `repmgr witness register` command is used with the correct postgres_port value even when using non-default postgres port for the upstream_primary postgres.
Bug Fix	Fixed an issue whereby failover_manager override to `repmgr` would not work correctly when set at instance level for subscriber-only nodes and their replicas in PGD clusters.
Bug Fix	Fixed two cases of incorrect cgroup detection: on MacOSX, we no longer try to read `/proc/mounts`. On systems where `/sys/fs/cgroup` is `ro` but mounts under it are `rw`, TPA now correctly detects this.
Bug Fix	Ensure we can verify the actual config set on pgd-proxy nodes for the newly added `read_listen_port` option in pgd-proxy.
Bug Fix	Fixed an issue that would prevent deployment with PEM 9.7.0. PEM 9.7.0 no longer depends on Apache at a package level therefore to use Apache as the web server we install the packages explicitly.

2.15 Trusted Postgres Architect 23.33 release notes

Released: 24 Jun 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.33 include the following:

Type	Description
Enhancement	TPA now supports Debian 12 Bookworm on the ARM64 CPU architecture.
Enhancement	PGD version 5.5 allows for proxy nodes to be configured as read endpoints, which direct read-only queries to a shadow node. TPA supports this configuration option by setting a `read_listen_port` parameter under `default_pgd_proxy_options` and `pgd_proxy_options` in `config.yml`. This parameter is included by default when the PGD version is 5.5 or greater. Users can also specify the port numbers by passing `--proxy-listen-port` and `proxy-read-listen-port` arguments to the `tpaexec configure` command.
Enhancement	TPA now supports deployment and configuration of the Beacon Agent on any Postgres node by assigning the role 'beacon-agent' or using the `--enable-beacon-agent` option with `configure`.
Enhancement	Added support for `postgres_wal_dir` in Patroni deployments. When a custom `postgres_wal_dir` is specified in TPA configuration, TPA will make sure to relay that option to the corresponding settings in the Patroni configuration file. That way, if Patroni ever needs to rebuild a standby on its own, out of TPA, the standby will be properly set up with a custom WAL directory.
Enhancement	When adding PgBouncer nodes in a Patroni cluster, TPA now configures Patroni with a `on_role_change` callback. That callback takes care of updating the primary connection info in the PgBouncer nodes in response to failover and switchover events.
Enhancement	EDB now produces its own `edb-patroni` package instead of rebuilding the `patroni` packages from PGDG. TPA now allows users to select between `patroni` and `edb-patroni` packages. The selection is made through the new TPA setting `patroni_package_flavour`.
Change	To work around broken Barman 3.10 packages in the PGDG repos, TPA now installs version 3.9 of Barman if using PGDG repos on an RHEL-family system. This behavior can be overridden by explicitly setting barman_package_version in config.yml .
Change	The `haproxy_bind_address` is now set to `0.0.0.0` when Patroni is the failover manager. This resolves an issue with the general default of `127.0.0.1` preventing communication between Postgres nodes and HA Proxy nodes. Users should change this value to something more restrictive and appropriate for their cluster networking.
Change	Task selectors are now consistently applied in the final stage of deployment. Consistency of task selectors in the tests is improved and the examples of task selectors in the docs are now correct. All deploy-time hooks now have corresponding task selectors.
Change	If `barman_package_version` is set, TPA will now look at it when looking for the barman-cli package as well as for Barman itself. This resolves an inconsistency which caused clusters using the downloader to fail when `barman_package_version` was used.
Bug Fix	Fixed an issue whereby required permissions on functions in the BDR database were not being granted to the HARP DCS user on a witness node.
Bug Fix	Fixed an issue whereby docker provisioning failed with "read-only file system". On host systems running cgroup1 with docker containers running recent OS images, `tpaexec provision` could fail to provision containers with an error message like "mkdir /sys/fs/cgroup/tpa.scope: read-only file system". TPA will now detect this case and avoid it.
Bug Fix	TPA now provides a clear error message if the user runs `tpaexec cmd` or `tpaexec ping` before provision.

2.16 Trusted Postgres Architect 23.32 release notes

Released: 15 May 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.32 include the following:

Type	Description
Enhancement	The M1 architecture now supports the following additional arguments to `tpaexec configure`: `--location-names`, `--primary-location`, `--data-nodes-per-location`, `--witness-only-location`, and `--single-node-location`. By combining these arguments, most common layouts can be specified without needing to edit `config.yml`.
Enhancement	TPA now installs chrony during deploy, keeping the default config upon all except on AWS where we point to Amazon Time Sync service.
Enhancement	TPA now supports RHEL 8 and 9 on IBM Power (PPC64le).
Enhancement	Added a `--force` option to `tpaexec relink`. By default, relink doesn't modify targeted files if they already exist. With `--force`, relink removes all existing targeted files then recreates them.
Enhancement	TPA now supports Debian 12 x86.
Enhancement	`pg_failover_slots` is now a recognized extension
Enhancement	The `sql_profiler`, `edb_wait_states` and `query_advisor` extensions are now automatically included for any `pem-agent` node. The list of default extensions for pem-agent nodes can be overridden by including a list of `pemagent_extensions` in `config.yml`. If this list is empty, no extensions will be automatically included.
Change	TPA can now provision Docker clusters on hosts running cgroups 2 for all systems except RHEL 7. On newer systems (RHEL 9 or Ubuntu 22.04), TPA will use cgroups 2 scopes for additional isolation between the host and the containers.
Change	Updated AWS AMI versions to the latest versions.
Bug Fix	Fixed an issue whereby deploying to Debian 10 on AWS would fail with the message `The repository 'http://cdn-aws.deb.debian.org/debian buster-backports Release' does not have a Release file`. The backports repository for debian 10 (buster) is no longer available on deb.debian.org but the standard AWS AMI still refers to it, so we modify /etc/apt/sources.list accordingly before attempting apt operations.
Bug Fix	Fixed an issue whereby deployment would fail on AWS when `assign_public_ip:no` was set.
Bug Fix	Fixed problems with various roles that caused mixed errors when trying to use custom users for barman and postgres, thereby resulting in a failed deployment.
Bug Fix	Fixed an issue whereby deployments after the initial one could fail with an error like `Unrecognised host=... in primary_conninfo` if the key `ip_address` was used to define the IP address.
Bug Fix	Fixed an error whereby `tpaexec upgrade` could invoke the relink script in a way which caused an error and showed an unhelpful usage message for `tpaexec relink`.
Bug Fix	Fixed an issue whereby a task to reload Postgres was skipped resulting in the restore_command override not being removed from `postgresql.auto.conf`
Bug Fix	Fixed an issue whereby TPA did not change to the source directory before attempting to compile BDR from source.
Bug Fix	Fixed an issue whereby TPA would require a valid 2ndQuadrant token even if one was not needed for the specified cluster.
Documentation	Updated the tower/AAP documentation to include instructions on creating an AAP Execution Environment.

2.17 Trusted Postgres Architect 23.31 release notes

Released: 19 Mar 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.31 include the following:

Type	Description
Bug Fix	Fixed a critical bug whereby deployments could fail due to a syntax error.

2.18 Trusted Postgres Architect 23.30 release notes

Released: 19 Mar 2024

> **End-of-support for 2ndQuadrant Ansible**
>
> Please note that, per the previously issued deprecation notice, this release completely removes support for 2ndQuadrant Ansible, which is no longer maintained. In addition, after Ansible 8 became the default in version 23.29, this version requires Ansible 8 or newer. To ensure you have a compatible Ansible version, please run tpaexec setup after updating TPA as detailed in the documentation.

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.30 include the following:

Type	Description
New Feature	TPA now provides a custom 'Execution Environment' image to be used in Ansible Automation Platform 2.4+ (Controller version 4+). this image contains everything needed to run deployments via AAP. This image is built using ansible-builder and a python-alpine lightweight base image.
Enhancement	TPA now automatically adds package names and shared preload library entries for a subset of extensions. For these specific extensions, only the extension name is needed in the `extra_postgres_extensions` list or the the `extensions` list of a database entry in `postgres_databases` .
Enhancement	The EDB Advanced Storage Pack package and shared preload library entry will automatically be added for `bluefin` when a user specifies it as an extension and the `postgres_version` is 15 or greater.
Enhancement	Added a new 'provision_only' option for instances. If an instance has `provision_only: true` in config.yml, it will be provisioned as normal but not added to the inventory which is seen by `tpaexec deploy` .
Change	Previous versions of TPA used to synchronize the source node's database structure to witness nodes. This was not necessary and the synchronized schema was never be used or updated. To prevent this happening, TPA now explicitly sets "synchronize_structure" to "none" when calling `bdr.join_node_group()` for witness nodes.
Change	Selective execution of tasks is now supported using custom selectors rather than Ansible tags. To run only tasks matching a certain selector: `tpaexec deploy . --included_tasks=barman` . To skip tasks matching a certain selector: `tpaexec deploy . --excluded_tasks=ssh` Task selectors can also be used by specifying the `excluded_tasks` or `included_tasks` variables in config.yml .
Change	Ansible 2.9 is no longer supported, neither the community distribution nor the 2ndQuadrant fork. Users who have been using the `--skip-tags` option to `tpaexec deploy` should move to the new `--excluded_tasks` option.

2.19 Trusted Postgres Architect 23.29 release notes

Released: 15 Feb 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.29 include the following:

Type	Description
Enhancement	Added support for storing the cluster vault password in the system keyring. This leverages python keyring module to store vault password in the supported system keyring when `keyring_backend` is set to `system` (default for new clusters). This change does not impact existing clusters or any clusters that set `keyring_backend` to `legacy` in config.yml.
Enhancement	The `--ansible-version` argument to `tpaexec setup` now accepts `8` or `9` as valid ansible versions, as well as the existing `2q` or `community`, both of which imply ansible 2.9. The default is now `8`. Support for ansible 9 is experimental and requires python 3.10 or above.
Bug Fix	Fixed an issue whereby edb_repositories already defined in config.yml are not kept during reconfigure. Fixes bdr4 to pgd5 upgrade scenario in air gapped environment.
Bug Fix	TPA's `postgres-monitor` will now recognize the message "the database system is not yet accepting connections" as a recoverable error.
Bug Fix	TPA now correctly skips the `postgres/config/final` role on replicas when upgrading.
Bug Fix	Fixed an issue whereby wildcards in package names were not respected when using package downloader on Debian and Ubuntu systems.
Bug Fix	The downloader now runs apt-get update before fetching packages on Debian and Ubuntu systems.
Bug Fix	TPA now disables transaction streaming when CAMO is enabled in PGD clusters.
Bug Fix	TPA now correctly configures Barman servers where selinux is enabled.

2.20 Trusted Postgres Architect 23.28 release notes

Released: 23 Jan 2024

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.28 include the following:

Type	Description
Enhancement	Added a new option `postgres_log_file`. This option sets the Postgres log file, whether logging through stderr or syslog. The default is '/var/log/postgres/postgres.log', the previously hard-coded value.
Enhancement	Added a new hook `barman-pre-config`. This hook is invoked after Barman is installed and its user is set up but before it is configured. It can be used for installing certificate files or other tasks which need the barman user to exist but which must be done before Barman is started.
Enhancement	The key `elastic_ip` on an AWS instance in `config.yml` can be set to an elastic IP address that has already been allocated in order to assign it to this instance.
Change	In Patroni clusters, TPA now sets up replicas before handing over control of the cluster to Patroni, rather than setting up the primary only and letting Patroni set up the replicas.
Change	For new clusters, TPA will create the user specified by setting `harp_manager_user` (by default `harpmanager`), belonging to the `bdr_superuser` role, and set HARP Manager to operate as this user instead of `postgres` superuser. This does not affect the existing clusters where TPA will keep using `postgres` as the HARP Manager user, unless the user overrides this behavior by explicitly setting `harp_manager_user` to a different value in `config.yml`.
Bug Fix	Fixed an issue whereby TPA would erroneously attempt to install repmgr on an EFM cluster.
Bug Fix	Fixed an issue whereby the TPA would return a non-zero exit code when the warning about 2q repositories was displayed despite deploy having succeeded.
Bug Fix	TPA will now interpret wildcards correctly on Debian-family systems when downloading packages for offline use.
Bug Fix	Fixed an issue whereby TPA would attempt to use incorrect package names for repmgr when installing from PGDG repositories.
Bug Fix	Fixed barman connection failure when using selinux and a custom barman home directory.
Bug Fix	TPA will now use the correct cluster name in `show-password` and `store-password` commands when it is different from the directory name
Bug Fix	TPA will now error out cleanly if unavailable 2ndQuadrant repository keys are required.
Bug Fix	TPA will now sanitize hostnames correctly when the --cluster-prefixed-hostnames option is used.
Bug Fix	TPA will now ensure packages are correctly copied to the remote host when upgrading a cluster using a local repo.

2.21 Trusted Postgres Architect 23.27 release notes

Released: 19 Dec 2023

> **Migration to EDB repositories**
>
> This release of TPA lays the groundwork for the decommissioning of the legacy 2ndQuadrant repositories. Existing configurations that use the legacy repositories will continue to function until they are decommissioned, but a warning will be displayed. To update an existing configuration to use EDB Repos 2.0, you may use `tpaexec reconfigure --replace-2q-repositories`.

> **Python interpreter**
>
> TPA now runs using a Python interpreter provided by the `edb-python-39` package, which will be automatically installed as a dependency of the `tpaexec` package. This allows us to keep TPA updated with security patches on older systems where the Python version is no longer widely supported. This is a completely standard build of Python 3.9. If you prefer, you may run TPA using another interpreter. We recommend 3.9, versions older than 3.9 or newer than 3.11 are not supported.

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.27 include the following:

Type	Description
Enhancement	TPA now supports Oracle Linux 7, 8 and 9 on Docker.
Change	TPA now requires Python 3.9-3.11 and depends on the package `edb-python-39` to provide a suitable interpreter.
Change	TPA will no longer configure any 2ndQuadrant repositories by default, instead it will select suitable repositories from EDB Repos 2.0.
Change	TPA now provides a new `--replace-2q-repositories` argument to `tpaexec reconfigure` that will remove 2q repositories from an existing config.yml and add suitable EDB repositories for the cluster's postgres flavour and BDR version.
Change	TPA now sets file system permissions explicitly on more objects.
Change	A new variable `disable_repository_checks` can be set to true in config.yml to bypass the usual check for EDB repositories when deploying the PGD-Always-ON architecture.
Change	TPA will now generate a primary_slot_name also on primary node to be used in case of switchover, to ensure the switched primary will have a physical slot on the new primary.
Change	TPA will now ensure that commit_scope for CAMO enabled partners is generated using existing config options from older BDR versions when running tpaexec reconfigure command to prepare for major PGD upgrade. It also choses better defaults.
Bug fix	Fixed an issue whereby postgres variables were rejected by Patroni due to validation rules.
Bug fix	Fixed an issue whereby a user could not set a single `barman_client_dsn_attributes` with `sslmode=verify-full`.
Bug Fix	TPA will now assign a lower default `maintenance_work_mem` to avoid out-of-memory errors.

2.22 Trusted Postgres Architect 23.26 release notes

Released: 30 Nov 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.26 include the following:

Type	Description
Enhancement	TPA now supports Oracle Linux 9 on the Docker platform.
Enhancement	Added `--cluster-prefixed-hostnames` option to `tpaexec configure`. This makes it easy to avoid hostname clashes on machines hosting more than one docker cluster.
Change	Added packages to enable Docker builds on Mac OS X.
Change	When there are multiple PEM servers in a cluster, the agent running on a PEM server registers to its local server.
Change	For PGD 5 clusters with CAMO. TPA will set timeout to 60s and require_write_lead to true by default.
Bug Fix	Fixed an issue whereby CAMO config was not correctly set up when upgrading a PGD 3 cluster to PGD 5. Upgrade is now fully supported for CAMO clusters.
Bug Fix	Fixed an issue whereby `hostname` rather than `bdr_node_name` was used when fencing or unfencing a HARP node.
Bug Fix	Fixed an issue whereby `provision` would be automatically run when `deploy` was invoked with options that suppress deployment.

2.23 Trusted Postgres Architect 23.25 release notes

Released: 14 Nov 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.25 include the following:

Type	Description
Enhancement	TPA now supports automated upgrades from PGD 3.7 to PGD 5.3 or above. *Note, upgrading clusters with CAMO is not yet supported.*
Enhancement	TPA now supports EDB Advanced Server 16 and EDB Extended Server 16.
Change	Various improvements to the upgrade process introduced with PGD 4 to PGD 5 upgrades have been backported to BDR-Always-ON upgrades.
Change	TPA now supports installing PEM on SLES.
Change	TPA now explicitly sets permissions when creating some filesystem objects. This will be extended to all filesystem objects in a future release.
Change	TPA now adds a symlink to the pgd-cli config file for v1 so it can be run without having to specify the path via `-f` switch.
Change	TPA now calls the `alter_node_kind` PGD function to ensure node kind is set correctly for BDR-Always-ON clusters using BDR version 4.3 and above.
Change	Default cluster configuration from now selects SLES 15 SP5 when SLES 15 is requested (previously SP4).
Bug Fix	Fixed an issue which resulted in a checksum failure during `tpaexec setup` command for tpaexec-deps users.
Bug Fix	Fixed an issue whereby `pem_server_group` was not correctly applied when pemworker was invoked meaning servers were not grouped as expected in PEM.
Bug Fix	Fixed an issue with the sys/sysstat role whereby `sar` was not scheduled to run on instances other than the Barman instance.

2.24 Trusted Postgres Architect 23.24 release notes

Released: 17 Oct 2023

> **2ndQuadrant/ansible deprecation**
>
> 2ndQuadrant/ansible is now deprecated and `tpaexec setup` now defaults to Community Ansible.
>
> Support for using the 2ndQuadrant Ansible fork will be removed from TPA in April 2024 and the GitHub repository will be archived.
>
> You should switch to Community Ansible, which is now the default. For the vast majority of users, this change will be transparent.
>
> If you are using `--skip-tags` with 2ndQuadrant Ansible, be aware that this is not supported with TPA and Community Ansible. We plan to provide an alternative to `--skip-tags` compatible with Community Ansible before the removal of 2ndQuadrant Ansible.

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.24 include the following:

Type	Description
Change	`tpaexec setup` now defaults to using community ansible rather than 2ndQuadrant ansible. The option `--use-2q-ansible` can be used to force the use of 2ndQuadrant ansible, which is now deprecated and will be removed in a future release. If you are using `--skip-tags`, see the install documentation.
Change	When a repository has been removed from `edb_repositories` in config.yml, `tpaexec deploy` now removes it from the nodes.
Change	TPA will now detect when harp-proxy and harp-manager are running on the same node and use a different config file for harp-proxy.
Change	The `upgrade` command will now update local repositories on target instances.
Bug Fix	Fixed an issue whereby TPA did not respect `postgres_wal_dir` in pg_basebackup invocation
Bug Fix	TPA will now accept repmgr as a failover manager for subscriber-only nodes in PGD clusters, allowing physical replication of such nodes.
Bug Fix	Fixed a typo which prevented TPA building Ubuntu 22.04 Docker images.
Bug Fix	TPA will now reject unsupported combination of the BDR-Always-ON architecture, the EDB Postgres Extended flavour, and PEM at configure-time.

2.25 Trusted Postgres Architect 23.23 release notes

Released: 21 Sep 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.23 include the following:

Type	Description
Enhancement	TPA now supports PostgreSQL 16. Please note, PostgreSQL 16 packages are not yet available in all supported repos, so not all configurations will work until this is the case.
Change	When Postgres 16 or above is selected, TPA will not add any 2ndQuadrant repos by default. TPA will explicitly set `tpa_2q_repositories: []` in this case.
Change	EFM is now configured to use JDK 11 by default on platforms where it is available.
Change	Where no EDB Repositories are use, TPA will not exclude any packages from PGDG (previously Barman and psycopg2 were excluded).
Change	Added package names for etcd and Patroni to support installation on SLES.
Bug Fix	Fixed an issue whereby Apache HTTPD service for PEM Server would not start on boot.
Bug Fix	Fixed an issue whereby pg_backup_api tests were run with incorrect permissions causing them to fail.
Bug Fix	Fixed an issue whereby Apache HTTPD service for pg_backup_api would not start on boot.
Bug Fix	Fixed an issue whereby `bdr.standby_slot_names` and `bdr.standby_slots_min_confirmed` checks used the incorrect schema on bdr3 clusters.
Bug Fix	Fixed an issue whereby configuration keys for extensions were passed to Patroni in the incorrect format, resulting in 'WARNING: Removing unexpected parameter'.
Bug Fix	Fixed an issue when using the intermediate base image option for `docker_images` whereby the resulting image name was incorrect.

2.26 Trusted Postgres Architect 23.22 release notes

Released: 6 Sep 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.22 include the following:

Type	Description
Change	TPA is now an open source project! You can clone the source under the GPLv3 license from GitHub.

2.27 Trusted Postgres Architect 23.21 release notes

Released: 5 Sep 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.21 include the following:

Type	Description
Change	The default M1 configuration now uses EDB Repos 2.0 if any EDB software is selected, otherwise PGDG is used. This only affects new clusters.
Change	You must now choose a failover manager explicitly when running `tpaexec configure` with the M1 architecture.
Bug fix	Fixed an issue with creation of PGD subscriber-only nodes whereby TPA incorrectly required 'subscriber-only' to be set on the replica instead of the upstream instance.
Bug fix	TPA will now skip inapplicable tasks when deploying to containers even if you are using the 'bare' platform option (previously these were skipped only if 'docker' was selected).
Bug fix	Fixed an issue with permissions on `/etc/edb` whereby if you added the pgd-proxy role to a data node in a deployed PGD5 cluster, pgd-proxy would fail to start because it did not have permissions to open pgd-proxy-config.yml.
Bug fix	Fixed an issue whereby `/var/log/postgres` could end up with inappropriate permissions (0600) if a strict umask was set
Bug fix	Fixed an issue whereby repeating `tpaexec deploy` on a Barman instance correctly registered with PEM would lose the PEM Agent Barman configuration.

2.28 Trusted Postgres Architect 23.20 release notes

Released: 01 Aug 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.20 include the following:

Type	Description
New feature	TPA now supports upgrades from PGD 4 to PGD 5 by running the new command `tpaexec reconfigure` to generate a revised config.yml and then `tpaexec upgrade` to perform the upgrade.
Enhancement	Added a new subcommand `tpaexec info validate` that runs a checksum over the TPA installation and confirms that it matches the one distributed with the package.
Change	The `update-postgres` command has been replaced with the more general `upgrade` command.
Change	TPA now explicitly adds `tzdata-java` when installing OpenJDK for Failover Manager on RHEL 8 or 9. This is a workaround for this OpenJDK bug.
Change	TPA now uses the latest available Debian AMIs on AWS (latest at the time of this release).
Change	TPA now runs `tpaexec provision` automatically as part of `tpaexec deploy` or `tpaexec upgrade` if config.yml has changed.
Bug fix	Fixed a bug whereby TPA could attempt to use a non-existent user when running pgd-cli on pgd-proxy nodes.
Bug fix	Fixed a bug whereby changes made by `tpaexec relink` were not committed to the Git repository correctly.

2.29 Trusted Postgres Architect 23.19 release notes

Released: 12 Jul 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.19 include the following:

Type	Description
New feature	TPA now allows the creation of physical replicas of subscriber-only PGD nodes.
New feature	TPA now supports the configuration of HTTP(S) HARP and PGD Proxy health probes.
New feature	TPA now allows you to select Patroni as a failover manager with the M1 architecture. This support is experimental and not yet recommended for use in production.
Enhancement	TPA now allows you to set specific versions for edb-pgd-proxy and edb-bdr-utilities rather than always using the latest version.
Change	On Debian-like systems, the package selection code now uses -dbg rather than -dbgsym for certain packages where applicable.
Change	When configuring replication slots, TPA will now ensure that only valid characters are used in the primary_slot_name. Previously TPA would use the inventory_hostname as a default, which could contain hyphens; these are now replaced with underscores.
Change	The default Failover Manager version is now 4.7.
Bug fix	Fixed an issue whereby PGD 3.7 to 4 upgrades would fail in TPA 23.18.
Bug fix	Fixed an issue whereby TPA would include underscores in TLS certificate Common Names when deploying PEM. This is invalid and would result in failure on some platforms.
Bug fix	Fixed an issue whereby an incorrect etcd service name would be used on Debian-like platforms, preventing TPA from starting etcd.
Bug fix	Fixed an issue whereby TPA could not install etcd packages on RHEL 8.
Bug fix	Fixed an issue whereby the message `Failed to commit files to git: b''` would be displayed during configure.
Bug fix	Fixed an issue whereby TPA would erroneously generate and overwrite Postgres user passwords when `generate_password: false`.
Bug fix	Fixed an issue whereby volume map creation on AWS failed to take account of region resulting in failures when using regions other than eu-west-1.

2.30 Trusted Postgres Architect 23.18 release notes

Released: 23 May 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.18 include the following:

Type	Description
Enhancement	TPA now uses pg_basebackup instead of repmgr for initial replica creation.
Enhancement	TPA now supports SLES 15, excluding creation of local repositories for air-gapped deployments.
Enhancement	TPA now supports minor-version upgrades of PGD5.
Enhancement	TPA now runs improved tests when `tpaexec test` is executed.
Bug fix	Fixed an issue whereby TPA attempted to use legacy 2ndQuadrant repositories on unsupported distributions.
Bug fix	Fixed an issue whereby TPA didn't install pg_receivewal on Barman instances where it was required.
Bug fix	Fixed an issue whereby TPA intermittently failed to create symlinks to block devices on AWS hosts during provisioning, causing deploy to fail.

2.31 Trusted Postgres Architect 23.17 release notes

Released: 10 May 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.17 include the following:

Type	Description
Enhancement	Added a new `--pgd-proxy-routing` parameter to the configure command. This can be set to `global` or `local`. Local routing will make every PGD-Proxy route to a write leader within its own location. Global routing will make every proxy route to a single write leader, elected amongst all available data nodes across all locations.
Change	Removed the `--active-locations` parameter from the configure command.
Enhancement	TPA now supports Ubuntu 22.04
Change	Updated the AWS AMIs used for RHEL 7 and 8.
Bug fix	Fixed an issue whereby TPA would incorrectly remove groups from existing Postgres users.
Bug fix	Fixed an issue whereby TPA would print an unhelpful error message when a git commit failed.
Bug fix	Fixed an issue whereby group names were incorrectly sanitized and uppercase letters were converted to underscores rather than lowercase ones.
Bug fix	Fixed an issue whereby Postgres was not restarted when required after CAMO configuration.
Bug fix	Fixed an issue with etcd changes, ensuring that they are now idempotent and avoiding unnecessary restarts of etcd on subsequent deployments.

2.32 Trusted Postgres Architect 23.16 release notes

Released: 21 Mar 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.16 include the following:

Type	Description
Change	The default PGD-Always-ON cluster is now one location with an associated subgroup containing two data nodes and one witness node.
Change	TPA now deploys pgd-proxy on all data nodes by default.
Enhancement	Added a new option, `--add-proxy-nodes-per-location N`, which creates separate proxy instances
Enhancement	TPA now adds a witness node automatically if `--data_nodes_per_location` is even and prints a warning if you specify a cluster with only two locations
Change	The parameter `--add-witness-only-location` has been renamed to `--witness-only-location` because we're NOT adding a location, but designating an already-named (in `--location-names`) location as witness-only.
Change	You must now specify Postgres flavour and version explicitly at `tpaexec configure` time
Enhancement	Added new CLI abbreviations for Postgres flavour and version, for example `--postgresql 14` or `--edbpge 15`
Enhancement	Improved handling and documentation of the various supported EDB software repositories
Change	TPA no longer includes the PGDG repository by default for PGD-Always-ON clusters
Bug fix	Fixed an issue whereby EDB Failover Manager was not selected as the failover manager for EPAS by default
Bug fix	Fixed an issue whereby pglogical was unnecessarily installed in the M1 architecture

2.33 Trusted Postgres Architect 23.15 release notes

Released: 15 Mar 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.15 include the following:

Type	Description
Minor change	Changes to dependency mappings.

2.34 Trusted Postgres Architect 23.14 release notes

Released: 23 Feb 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.14 include the following:

Type	Description
Bug fix	Fixes an error whereby package lists weren't correctly populated for PGD 3 and 4 configurations. (TPA-365)
Change	Use multi-line BDR DCS configuration in HARP's config.yaml (TPA-360, RT90034)

2.35 Trusted Postgres Architect 23.13 release notes

Released: 22 Feb 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.13 include the following:

Type	Description
Bug fix	Don't enable old EDB repo with PGD-Always-ON and `--epas`.
Bug fix	Fix error with PGD-Always-ON and `--postgres-version 15`.

2.36 Trusted Postgres Architect 23.12 release notes

Released: 21 Feb 2023

New features, enhancements, bug fixes, and other changes in Trusted Postgres Architect 23.12 include the following:

Type	Description
Feature	Introduce full support for EDB Postgres Distributed 5, including Commit At Most Once (CAMO) configuration support based on commit scopes.
Feature	Introduce support for EDB Postgres Extended repository and packages.
Enhancement	Preliminary support for configuring multi-region AWS clusters. Multi-region clusters require manual setup of VPCs and VPC.
Enhancement	Enable proxy routing (and, therefore, subgroup RAFT) automatically for `--active-locations`. Removes the configure option to enable subgroup RAFT globally.
Bug fix	Ensure the EDB_SUBSCRIPTION_TOKEN is not logged.
Bug fix	Allow the user to suppress addition of the products/default/release repo to tpa_2q_repositories.
Bug fix	Ensure that nodes subscribe to bdr_child_group, if available. In clusters with multiple subgroups, TPA did not expect instances to be subscribed to the replication sets for both the top group and the subgroup, so it would incorrectly remove the latter from the node's subscribed replication sets.
Bug fix	Fail reliably with a useful error if Postgres doesn't start. Due to an Ansible bug, the deployment wouldn't fail if Postgres did not start on some instances, but did start on others (for example, due to a difference in the configuration). Continuing on with the deployment resulted in errors when trying to access cluster_facts for the failed hosts later.
Bug fix	Don't call `bdr.alter_node_replication_sets()` on witnesses for BDR 4.3 and later. This adjusts to a new restriction in BDR versions where witness nodes are not handled with a custom replication set configuration.
Bug fix	Replace hardcoded "barman" references to enable use of the barman_{user,group} settings to customize the barman user and home directory.
Bug fix	Add shared_preload_libraries entries, where appropriate, for extensions mentioned under postgres_databases[*].extensions.
Bug fix	Ensure that `pgaudit` does not appear before `bdr` in shared_preload_libraries (to avoid a known crash).
Bug fix	Fix syntax error (DSN quoting) in pgd-cli config file.
Bug fix	Sort endpoints in pgd-proxy config to avoid file rewrites. This will likely require a pgd-proxy restart on the next deploy (but it will avoid unnecessary future rewrites/restarts on subsequent deploys).
Bug fix	Fix an error while installing rsync from a local-repo on RH systems.
Bug fix	Fix an error with Apache WSGI module configuration for PEM 9 on Debian systems.
Bug fix	Don't remove the bdr extension if it has been created on purpose, even if it is unused.

2.37 Trusted Postgres Architect 23.1 to 23.11 release notes

TPA 23.11

Released: 2023-01-31

Notable changes

- TPA-180 Introduce experimental support for PGD-Always-ON architecture (to be released later this year). PGD-Always-ON architecture will use the upcoming BDR version 5. Initial support has been added for internal purposes and will be improved in upcoming releases.

Minor changes

- TPA-349 Bump dependency versions Bump cryptography version from 38.0.4 to 39.0.0 Bump jq version from 1.3.0 to 1.4.0
- TPA-345 Change TPAexec references to TPA in documentation. Update the documentation to use 'TPA' instead of 'TPAexec' when referring to the product.

TPA 23.10

Released: 2023-01-04

Minor changes

- TPA-161 Introduce `harp_manager_restart_on_failure` setting (defaults to false) to enable process restart on failure for the harp-manager systemd service

Bug Fixes

- TPA-281 Delete FMS security groups when deprovisioning an AWS cluster Fixes a failure to deprovision a cluster's VPC because of unremoved dependencies.
- TPA-305 Add enterprisedb_password to pre-generated secrets for Tower
- TPA-306 Prefer PEM_PYTHON_EXECUTABLE, if present, to /usr/bin/python3 Fixes a Python module import error during deployment with PEM 9.0.
- TPA-219 Make pem-agent monitor the bdr_database by default on BDR instances

TPA 23.9

Released: 2022-12-12

Bugfixes

- TPA-301 Fix auto-detection of cluster_dir for Tower clusters When setting cluster_dir based on the Tower project directory, we now correctly check for the existence of the directory on the controller, and not on the instances being deployed to.
- TPA-283 Add dependency on psutil, required for Ansible Tower.
- TPA-278 Remove "umask 0" directive from rsyslog configuration, which previously resulted in the creation of world-readable files such as rsyslogd.pid .
- TPA-291 Respect the postgres_package_version setting when installing the Postgres server package to obtain pg_receivewal on Barman instances.

TPA 23.8

Released: 2022-11-30

Notable changes

- TPA-18 Support Ansible Tower 3.8 This release supports execution of `deploy.yml` (only) on a `bare` cluster (i.e., with existing servers) through Ansible Tower 3.8. Install TPAexec on the Tower server and run `tpaexec setup` to create a virtual environment which can be used in Tower Templates to run TPAexec playbooks. Use the `--use-ansible-tower` and `--tower-git-repository` configure options to generate a Tower-compatible cluster configuration. For details, see Ansible Tower.

Minor changes

- TPA-238 Initialise the cluster directory as a git repository If git is available on the system where you run TPAexec, `tpaexec configure` will now initialise a git repository within the cluster directory by default. If git is not available, it will continue as before. To avoid creating the repository (for example, if you want to store the cluster directory within an existing repository), use the `--no-git` option.

TPA 23.7

Released: 2022-11-09

Notable changes

- TPA-234 Support the community release of Ansible 2.9 TPAexec used to require the 2ndQuadrant/ansible fork of Ansible 2.9. In this release, you may instead choose to use the community release of Ansible with the `tpaexec setup --use-community-ansible`. For now, the default continues to be to use 2ndQuadrant/ansible. This will change in a future release; support for 2ndQuadrant/ansible will be dropped, and Ansible will become the new default.

Minor changes

- TPA-209 Accept `--postgres-version 15` as a valid `tpaexec configure` option, subsequent to the release of Postgres 15
- TPA-226 Accept IP addresses in the `--hostnames-from` file Formerly, the file passed to `tpaexec configure` was expected to contain one hostname per line. Now it may also contain an optional IP address after each hostname. If present, this address will be set as the `ip_address` for the corresponding instance in config.yml. (If you specify your own `--hostnames-from` file, the hostnames will no longer be randomised by default.)
- TPA-231 Add a new bdr-pre-group-join hook This hook is executed before each node joins the BDR node group. It may be used to change the default replication set configuration that TPAexec provides.
- TPA-130 Use the postgresql_user module from community.postgresql The updated module from the community.postgresql collection is needed in order to correctly report the task status when using a SCRAM password (the default module always reports `changed`).
- TPA-250 Upgrade to the latest versions of various Python dependencies

Bugfixes

- TPA-220 Ensure LD_LIBRARY_PATH in .bashrc does not start with ":"
- TPA-82 Avoid removing BDR-internal ${group_name}_ext replication sets
- TPA-247 Fix "'str object' has no attribute 'node_dsn'" errors on AWS The code no longer assigns `hostvars[hostname]` to an intermediate variable and expects it to behave like a normal dict later (which works only sometimes). This fixes a regression in 23.6 reported for AWS clusters with PEM enabled, but also fixes other similar errors throughout the codebase.
- TPA-232 Eliminate a race condition in creating a symlink to generated secrets in the inventory that resulted in "Error while linking: [Errno 17] File exists" errors
- TPA-252 Restore code to make all BDR nodes publish to the witness-only replication set This code block was inadvertently removed in the v23.6 release as part of the refactoring work done for TPA-193.

TPA 23.6

Released: 2022-09-28

Notable changes

- TPA-21 Use boto3 (instead of the unmaintained boto2) AWS client library for AWS deployments. This enables SSO login and other useful features.
- TPA-202 Add harp-config hook. This deploy-time hook executes after HARP is installed and configured and before it is started on all nodes where HARP is installed.

Bugfixes

- TPA-181 Set default python version to 2 on RHEL 7. Formerly, tpaexec could generate a config.yml with the unsupported combination of RHEL 7 and python 3.
- TPA-210 Fix aws deployments using existing security groups. Such a deployment used to fail at provision-time but will now work as expected.
- TPA-189 Remove group_vars directory on deprovision. This fixes a problem that caused a subsequent provision to fail because of a dangling symlink.
- TPA-175 Correctly configure systemd to leave shared memory segments alone. This only affects source builds.
- TPA-160 Allow version setting for haproxy and PEM. This fixes a bug whereby latest versions of packages would be installed even if a specific version was specified.
- TPA-172 Install EFM on the correct set of hosts. EFM should be installed only on postgres servers that are members of the cluster, not servers which have postgres installed for other reasons, such as PEM servers.
- TPA-113 Serialize PEM agent registration. This avoids a race condition when several hosts try to run pemworker --register-agent at the same time.

TPA 23.5

Released: 2022-08-23

Notable changes

- TPA-81 Publish tpaexec and tpaexec-deps packages for Ubuntu 22.04 Jammy
- TPA-26 Support harp-proxy and harp-manager installation on a single node. It is now possible to have both harp-proxy and harp-manager service running on the same target node in a cluster.

TPA 23.4

Released: 2022-08-03

Bugfixes

- TPA-152 fix an issue with locale detection during first boot of Debian instances in AWS Hosts would fail to complete first boot which would manifest as SSH key negotiation issues and errors with disks not found during deployment. This issue was introduced in 23.3 and is related to TPA-38

TPA 23.3

Released: 2022-08-03

Notable changes

- TPA-118 Exposed two new options in harp-manager configuration. The first sets HARP `harp_db_request_timeout` similar to dcs request_timeout but for database connections and the second `harp_ssl_password_command` specifies a command used to de-obfuscate sslpassword used to decrypt the sslkey in SSL enabled database connection

Minor changes

- TPA-117 Add documentation update on the use of wildcards in `package_version` options in tpaexec config.yml. This introduces a warning that unexpected package upgrades can occur during a `deploy` operation. See documentation in `tpaexec-configure.md` for more info
- TPA-38 Add locale files for all versions of Debian, and RHEL 8 and above. Some EDB software, such as Barman, has a requirement to set the user locale to `en_US.UTF-8`. Some users may wish to also change the locale, character set or language to a local region. This change ensures that OS files provided by libc are installed on AWS instances during firstboot using user-data scripts. The default locale is `en_US.UTF-8`. See `platform_aws.md` documentation for more info
- TPA-23 Add log config for syslog for cluster services Barman, HARP, repmgr, PgBouncer and EFM. The designated log server will store log files received in `/var/log/hosts` directories for these services
- TPA-109 Minor refactoring of the code in pgbench role around choosing lock timeout syntax based on a given version of BDR

Bugfixes

- TPA-147 For clusters that use the source install method some missing packages for Debian and Rocky Linux were observed. Debian receives library headers for krb5 and lz4. On RedHat derived OSs the mandatory packages from the "Development Tools" package group and the libcurl headers have been added
- TPA-146 Small fix to the method of package selection for clusters installing Postgres 9.6
- TPA-138 Addresses a warning message on clusters that use the "bare" platform that enable the local-repo configure options. As the OS is not managed by TPAexec in the bare platform we need to inform the user to create the local-repo structure. This previously caused an unhandled error halting the configure progress
- TPA-135 When using `--use-local-repo-only` with the "docker" platform and the Rocky Linux image initial removal of existing yum repository configuration on nodes would fail due to the missing commands `find` and `xargs`. This change ensures that if the `findutils` package exists in the source repo it will be installed first
- TPA-111 Remove a redundant additional argument on the command used to register agents with the PEM server when `--enable-pem` option is given. Previously, this would have caused no problems as the first argument, the one now removed, would be overridden by the second
- TPA-108 Restore SELinux file context for postmaster symlink when Postgres is installed from source. Previously, a cluster using a SELinux enabled OS that is installing postgres from source would fail to restart Postgres as the systemd daemon would be unable to read the symlink stored in the Postgres data bin directory. This was discovered in tests using a recently adopted Rocky Linux image in AWS that has SELinux enabled and in enforcing mode by default

TPA 23.2

Released: 2022-07-13

Notable changes

- Add support for Postgres Backup API for use with Barman and PEM. Accessible through the `--enable-pg-backup-api` option.
- SSL certificates can now be created on a per-service basis, for example the server certificate for Postgres Backup API proxy service. Certificates will be placed in `/etc/tpa/<service>/<hostname>.cert` These certificates can also be signed by a CA certificate generated for the cluster.

- Placement of Etcd for the BDR-Always-ON architecture When using 'harp_consensus_protocol: etcd', explicitly add 'etcd' to the role for each of the following instances:
  - BDR Primary ('bdr' role)
  - BDR Logical Standby ('bdr' + 'readonly' roles)
  - only for the Bronze layout: BDR Witness ('bdr' + 'witness' roles)
  - only for the Gold layout: Barman ('barman' role) Credit: Gianni Ciolli gianni.ciolli@enterprisedb.com

Minor changes

- Replace configure argument `--2q` with `--pgextended` to reflect product branding changes. Existing configuration will retain expected behaviour.
- Improve error reporting on Docker platform compatibility checks when using version 18 of docker, which comes with Debian old stable.
- Add some missing commands to CLI help documentation.
- Improved error reporting of configure command.
- Add initial support for building BDR 5 from source. Credit: Florin Irion florin.irion@enterprisedb.com
- Changes to ensure ongoing compatibility for migration from older versions of Postgres with EDB products.

Bugfixes

- Fixed an issue which meant packages for etcd were missing when using the download-packages command to populate the local-repo.
- Fixed an issue affecting the use of efm failover manager and the selection of its package dependencies

TPA 23.1

Released: 2022-06-21

This release requires you to run `tpaexec setup` after upgrading (and will fail with an error otherwise)

Changes to package installation behavior

In earlier versions, running `tpaexec deploy` could potentially upgrade installed packages, unless an exact version was explicitly specified (e.g., by setting postgres_package_version). However, this was never a safe, supported, or recommended way to upgrade. In particular, services may not have been safely and correctly restarted after a package upgrade during deploy.

With this release onwards, `tpaexec deploy` will never upgrade installed packages. The first deploy will install all required packages (either a specific version, if set, or the latest available), and subsequent runs will see that the package is installed, and do nothing further. This is a predictable and safe new default behavior.

If you need to update components, use `tpaexec update-postgres`. In this release, the command can update Postgres and Postgres-related packages such as BDR or pglogical, as well as certain other components, such as HARP, pgbouncer, and etcd (if applicable to a particular cluster). Future releases will safely support upgrades of more components.

Notable changes

- Run "harpctl apply" only if the HARP bootstrap config is changed WARNING: This will trigger a single harp service restart on existing clusters when you run `tpaexec deploy`, because config.yml is changed to ensure that lists are consistently ordered, to avoid unintended changes in future deploys
- Add `tpaexec download-packages` command to download all packages required by a cluster into a local-repo directory, so that they can be copied to cluster instances in airgapped/disconnected environments. See air-gapped.md and local-repo.md for details
- Require `--harp-consensus-protocol <etcd|bdr>` configure option for new BDR-Always-ON clusters TPAexec no longer supplies a default value here because the choice of consensus protocol can negatively affect failover performance, depending on network latency between data centres/locations, so the user is in a better position to select the protocol most suitable for a given cluster. This affects the configuration of newly-generated clusters, but does not affect existing clusters that use the former default of `etcd` without setting harp_consensus_protocol explicitly

Minor changes

- Install openjdk-11 instead of openjdk-8 for EFM on distributions where the older version is not available
- Accept `harp_log_level` setting (e.g., under cluster_vars) to override the default harp-manager and harp-proxy log level (info)
- Configure harp-proxy to use a single multi-host BDR DCS endpoint DSN instead of a list of individual endpoint DSNs, to improve resilience
- Omit extra connection attributes (e.g., ssl*) from the local (Unix socket) DSN for the BDR DCS for harp-manager

Bugfixes

- Ensure that harp-manager and harp-proxy are restarted if their config changes
- Fix harp-proxy errors by granting additional (new) permissions required by the readonly harp_dcs_user
- Disable BDR4 transaction streaming when CAMO is enabled If bdr.enable_camo is set, we must disable bdr.default_streaming_mode, which is not compatible with CAMO-protected transactions in BDR4. This will cause a server restart on CAMO-enabled BDR4 clusters (which could not work with streaming enabled anyway).

3 TPA installation

To use TPA, you need to install from packages or source and run the `tpaexec setup` command. This document explains how to install TPA packages. If you have an EDB subscription plan, and therefore have access to the EDB repositories, you should follow these instructions. To install TPA from source, please refer to Installing TPA from Source.

See Distribution support for information on what platforms are supported.

> **Info**
>
> Please make absolutely sure that your system has the correct date and time set, because various things will fail otherwise. We recommend you use a network time, for example `sudo ntpdate pool.ntp.org`

Quickstart

First, subscribe to an EDB repository.

Install TPA

```
sudo <your-package-manager> install tpaexec
```

Install additional dependencies

```
sudo /opt/EDB/TPA/bin/tpaexec setup
```

Verify installation (run as a normal user)

```
/opt/EDB/TPA/bin/tpaexec selftest
```

More detailed explanations of each step are given below.

Where to install TPA

As long as you are using a supported platform, TPA can be installed and run from your workstation. This is fine for learning, local testing or demonstration purposes. TPA supports deploying to Docker containers should you wish to perform a complete deployment on your own workstation.

For production use, we recommend running TPA on a dedicated, persistent virtual machine. We recommend this because it ensures that the cluster directories are retained and available to your team for future cluster management or update. It also means you only have to update one copy of TPA and you only need to provide network access from a single TPA host to the target instances.

Installing TPA packages

To install TPA, you must first subscribe to an EDB repository. TPA is available in all EDB repositories.

Install TPA as follows:

Install on Debian or Ubuntu

```
sudo apt-get install tpaexec
```

Install on RHEL, Rocky, AlmaLinux or Oracle Linux

```
sudo dnf install tpaexec
```

SLES

```
sudo zypper install tpaexec
```

This will install TPA into `/opt/EDB/TPA`. It will also ensure that other required packages (e.g., Python 3.12 or later) are installed.

We mention `sudo` here only to indicate which commands need root privileges. You may use any other means to run the commands as root.

Setting up the TPA Python environment

Next, run `tpaexec setup` to create an isolated Python environment and install the correct versions of all required modules.

> **Note**
>
> On Ubuntu versions prior to 20.04, please use `sudo -H tpaexec setup` to avoid subsequent permission errors during `tpaexec configure`

```
sudo /opt/EDB/TPA/bin/tpaexec setup
```

`tpaexec setup` will automatically use a suitable version of the requirements.txt file to install the correct modules for your system.

You must run this as root because it writes to `/opt/EDB/TPA`, but the process will not affect any system-wide Python modules you may have installed (including Ansible).

Add `/opt/EDB/TPA/bin` to the `PATH` of the user who will normally run `tpaexec` commands. For example, you could add this to your .bashrc or equivalent shell configuration file:

```
export PATH=$PATH:/opt/EDB/TPA/bin
```

Installing TPA without internet or network access (air-gapped)

This section describes how to install TPA onto a server which cannot access either the EDB repositories, a Python package index, or both. For information on how to use TPA in such an environment, please see Managing clusters in a disconnected or air-gapped environment

Downloading TPA packages

If you cannot access the EDB repositories directly from the server on which you need to install TPA, you can download the packages from an internet-connected machine and transfer them. There are several ways to achieve this.

If your internet-connected machine uses the same operating system as the target, we recommend using `yumdownloader` (RHEL-like) or `apt download` (Debian-like) to download the packages.

Alternatively, you can download packages for any platform from your browser by visiting EDB Repos and selecting either **Enterprise**, **Standard** or **Community 360** under the heading **Download EDB software packages from your browser**. To install TPA you need these packages:

- tpaexec
- tpaexec-deps
- edb-python312

Once you have transferred the downloaded packages to the target server, you must install them using the appropriate tool for your platform.

Installing without access to a Python package index

When you run `tpaexec setup`, it will ordinarily download the Python packages from a Python package index. Unless your environment provides a different index the default is the official PyPI. If no package index is available, you should install the `tpaexec-deps` package in the same way your installed `tpaexec`. The `tpaexec-deps` package (available from the same repository as tpaexec) bundles everything that would have been downloaded, so that they can be installed without network access. Just install the package before you run `tpaexec setup` and the bundled copies will be used automatically.

Verifying your TPA installation

Once you're done with all of the above steps, run the following command to verify your local installation:

```
tpaexec selftest
```

If that command completes without any errors, your TPA installation is ready for use.

Upgrading TPA

To upgrade to a later release of TPA, you must:

1. Install the latest `tpaexec` package
2. Install the latest `tpaexec-deps` package (if required; see above)
3. Run `tpaexec setup` again

If you have subscribed to the TPA package repository as described above, running `apt-get update && apt-get upgrade` or `yum update` should install the latest available versions of these packages. If not, you can install the packages by any means available.

We recommend that you run `tpaexec setup` again whenever a new version of `tpaexec` is installed. Some new releases may not strictly require this, but others will not work without it.

Ansible versions

TPA uses ansible-core 2.16 by default.

4 Open source TPA

What is Trusted Postgres Architect (TPA)?

TPA is an orchestration tool developed by EnterpriseDB (EDB) that uses Ansible to deploy Postgres clusters according to EDB's recommendations.

TPA embodies the best practices followed by EDB, informed by many years of hard-earned experience with deploying and supporting Postgres. These recommendations are as applicable to quick testbed setups as to production environments.

Next Steps

- Installing TPA from Source
- Deploying your first cluster
- TPA's full documentation online

TPA Open Source FAQs

Can I use this if I'm not an EDB customer?

Yes, TPA is an open source project under the GPLv3 license. It supports deploying clusters comprised of open source software, or EDB's proprietary products, or combinations.

Can I report an issue?

Yes, if you're an EDB customer then please contact support. Otherwise please open a GitHub Issue.

Can I contribute?

Sure, we'd love to hear from you but please open an issue before you start coding. We are quite selective with what TPA can/should do so bug fixes are more likely to get accepted than new features.

5 Known issues

This page lists known issues affecting the current version of TPA. Where one is known, a workaround for each issue is provided.

EFM clusters with more than one location are missing inter-location HBA rules (TPA-1247)

Details

If you deploy an EFM cluster with more than one location, then perform the `efm cluster-status` command on one node, you will observe that `FATAL: password authentication failed for user "efm"` is displayed in the **Promote Status** section for all nodes not in the same location as the node on which the command was performed.

This occurs because TPA fails to add the necessary HBA rules for the EFM user to connect from nodes in one location to nodes in another location.

Workaround

Use the `postgres_hba_settings` cluster variable to specify one HBA rule for each node such that the `efm` user and `replication` user can connect from any node to any other node. For example:

```
postgres_hba_settings:
  - hostssl all efm 192.168.57.12/32
md5
  - hostssl all efm 192.168.57.13/32
md5
  - hostssl all efm 192.168.58.11/32
md5
  - hostssl all efm 192.168.58.12/32
md5
  - hostssl all efm 192.168.58.13/32
md5

  - hostssl replication replication 192.168.57.12/32
md5
  - hostssl replication replication 192.168.57.13/32
md5
  - hostssl replication replication 192.168.58.11/32
md5
  - hostssl replication replication 192.168.58.12/32
md5
  - hostssl replication replication 192.168.58.13/32
md5
```

PGD clusters deployed with TPA 23.34 or earlier have primary_slot_name set preventing bdr_init_physical from working (TPA-1229)

Details

TPA 23.35 resolved an issue where a replication slot name intended for use with EFM would be incorrectly set as the `primary_slot_name` on PGD clusters. Clusters deployed with TPA 23.35 and above will not have this issue. However, existing clusters deployed with earlier versions will retain the incorrect `primary_slot_name` setting, even after deployment run on newer TPA version.

The impact of this incorrect `primary_slot_name` is that if you attempt to add a new node to the cluster (or rebuild a node) using bdr_init_physical, it will hang on `Waiting for PostgreSQL to accept connections...`, the logs will show a fatal error `replication slot ... does not exist`.

Workaround

You can manually remove this incorrect setting from each Postgres node in your cluster by deleting the file `/opt/postgres/data/conf.d/8901-primary_slot_name.conf` and reloading Postgres. As long as you are now using TPA 23.35 or later, it will not be reinstated.

SSH configuration templating error in clusters using `forward_ssh_agent: true`

Details

TPA 23.43 changed the templating for the ssh_config file in the cluster directory and introduced a bug whereby if `forward_ssh_agent` is set to True, the generated config file will be invalid. The default for this setting is False, and it is usually only set to True when installing components of the cluster from source code for testing.

Workaround

After running `tpaexec provision` to generate the ssh_config file, edit the file in a text editor. Find the line reading `IdentitiesOnly yes ForwardAgent yes`, usually line 7 in the file, and split it two lines by adding a newline after the first "yes". The file will be overwritten by `tpaexec provision` so it must be re-edited after provisioning when a change is made to config.yml.

PEM agent registration options provided by `pemagent_registration_opts` are not reflected in `agent.cfg` and therefore do not take effect (TPA-1376)

Details

The `pemagent_registration_opts` option is used to pass additional command line options to the pemworker utility during PEM agent registration. Some of these options affect the immediate act of registration, some affect the data written to the PEM database, and others affect the content of the generated `agent.cfg` file which is used to determine agent behaviour. In TPA's current implementation, the `agent.cfg` file is overwritten immediately after registration with a templated version, meaning any changes made to this file as a result of `pemagent_registration_opts` are lost.

Workaround

Avoid using `pemagent_registration_opts` unless you are sure the options you specify do not interact with `agent.cfg`. Apply any changes required to `agent.cfg` using a post-deploy hook.

PEM 10.4.x web application shows "Internal Server Error" when deployed using TPA (TPA-1415)

Details

PEM 10.4 uses a newer Python version than previous PEM versions on most operating systems. TPA installs the required Python version because it is a package dependency of PEM, but still installs the `mod_wsgi` modules for the previous version, meaning the web application cannot run.

Workaround

The most complete workaround is to explicitly add the required `mod_wsgi` package to your PEM server instance. You can find the required package in the PEM documentation. For example for RHEL 9:

```
instances:
...
- name: pem-server

...
  packages:
    RedHat:
    - edb-python312-mod-
wsgi
```

6 Installing TPA from source

This document explains how to use TPA from a copy of the source code repository.

> **Note**
>
> EDB customers must install TPA from packages in order to receive EDB support for the software.

To run TPA from source, you must install all of the dependencies (e.g., Python 3.12+) that the packages would handle for you, or download the source and run TPA in a Docker container. (Either way will work fine on Linux and macOS.)

Prerequisites

Before installing TPA from source, ensure that `git` is available on your system.

You must also have a **Python 3.12** interpreter, including the `pip` and `venv` modules, present on the TPA controller host.

- For **RHEL 9**, you may need to enable the CodeReady Linux Builder (CRB) repository to install necessary build dependencies.
- For **SLES**, ensure the Python 3.12 module is enabled.
- If your operating system repositories do not provide Python 3.12, we recommend using pyenv to install it.

> **Note**
>
> Legacy dependencies such as `openvpn` and `patch` are no longer required for standard TPA installations and have been removed from these instructions.

Clone and setup

With prerequisites installed, you can now clone the repository.

```
git clone https://github.com/enterprisedb/tpa.git ~/tpa
```

This creates a `tpa` directory in your home directory.

If you prefer to checkout with ssh use:

```
git clone ssh://git@github.com/EnterpriseDB/tpa.git ~/tpa
```

Add the bin directory, found within in your newly created clone, to your path with:

```
export PATH=$PATH:$HOME/tpa/bin
```

Add this line to your `.bashrc` file (or other profile file for your preferred shell).

You can now create a working tpa environment by running:

```
tpaexec setup
```

This will create the Python virtual environment that TPA will use in future. All needed packages are installed in this environment. To test this configured correctly, run the following:

```
tpaexec selftest
```

You now have tpaexec installed.

Dependencies

Python 3.12+

TPA requires Python 3.12 or later, available on most modern distributions. If you don't have it, you can use pyenv to install any version of Python you like without affecting the system packages.

```
# First, install pyenv and activate it in ~/.bashrc
# See https://github.com/pyenv/pyenv#installation
# (e.g., `brew install pyenv` on MacOS X)

pyenv install 3.12.0
```

output

```
Downloading Python-3.12.0.tar.xz...
-> https://www.python.org/ftp/python/3.12.0/Python-3.12.0.tar.xz
Installing Python-3.12.0...
Installed Python-3.12.0 to /home/ams/.pyenv/versions/3.12.0
```

```
pyenv local 3.12.0
pyenv version
```

output

```
3.12.0 (set by /home/ams/pyenv/.python-version)
```

```
pyenv which python3
```

output

```
/home/ams/.pyenv/versions/3.12.0/bin/python3
```

```
python3 --version
```

output

```
3.12.0
```

If you were not already using pyenv, please remember to add `pyenv` to your PATH in .bashrc and call `eval "$(pyenv init -)"` as described in the pyenv documentation.

Virtual environment options

By default, `tpaexec setup` will use the builtin Python 3 `-m venv` to create a venv under `$TPA_DIR/tpa-venv`, and activate it automatically whenever `tpaexec` is invoked.

You can run `tpaexec setup --venv /other/location` to specify a different location for the new venv.

We strongly suggest sticking to the default venv location. If you use a different location, you must also set the environment variable TPA_VENV to its location, for example by adding the following line to your .bashrc (or other shell startup scripts):


```
export TPA_VENV="/other/location"
```

7 A First Cluster Deployment

In this short tutorial, we are going to work through deploying a simple M1 architecture deployment onto a local Docker installation. By the end you will have four containers, one primary database, two replicas and a backup node, configured and ready for you to explore.

For this example, we will run TPA on an Ubuntu system, but the considerations are similar for most Linux systems.

Installing TPA

If you're an EDB customer, you'll want to follow the EDB Repo instructions which will install the TPA packages straight from EDB's repositories.

If you are an open source user of TPA, there's instructions on how to build from the source which you can download from Github.com.

Follow those guides and then return here.

Installing Docker

As we said, We are going to deploy the example deployment onto Docker and unless you already have Docker installed we'll need to set that up.

On Debian or Ubuntu, install Docker by running:

```
sudo apt update
sudo apt install docker.io
```

For other Linux distributions, consult the Docker Engine Install page.

You will want to add your user to the docker group with:

```
sudo usermod -aG docker <yourusername>
newgrp docker
```

> **Warning**
>
> Giving a user the ability to speak to the Docker daemon lets them trivially gain root on the Docker host. Only trusted users should have access to the Docker daemon.

Creating a configuration with TPA

The next step in this process is to create a configuration. TPA does most of the work for you through its `configure` command. All you have to do is supply command line flags and options to select, in broad terms, what you want to deploy. Here's our `tpaexec configure` command:

```
tpaexec configure demo --architecture M1 --platform docker --postgresql 15 --enable-repmgr --no-git
```

This creates a configuration called `demo` which has the M1 architecture. It will therefore have a primary, replica and backup node.

The `--platform docker` tells TPA that this configuration should be created on a local Docker instance; it will provision all the containers and OS requirements. Other platforms include AWS, which does the same with Amazon Web Services and Bare, which skips to operating system provisioning and goes straight to installing software on already configured Linux hosts.

With `--postgresql 15`, we instruct TPA to use Community Postgres, version 15. There are several options here in terms of selecting software, but this is the most straightforward default for open-source users.

Adding `--enable-repmgr` tells TPA to use configure the deployment to use Replication Manager to hand replication and failover.

Finally, `--no-git` turns off the feature in TPA which allows you to revision control your configuration through git.

Run this command, and apparently, nothing will happen on the command line. But you will find a directory called `demo` has been created containing some files including a `config.yml` file which is a blueprint for our new deployment.

Provisioning the deployment

Now we are ready to create the containers (or virtual machines) on which we will run our new deployment. This can be achieved with the `provision` command. Run:

```
tpaexec provision demo
```

You will see TPA work through the various operations needed to prepare for deployment of your configuration.

Deploying

Once provisioned, you can move on to deployment. This installs, if needed, operating systems and system packages. It then installs the requested Postgres architecture and performs all the needed configuration.

```
tpaexec deploy demo
```

You will see TPA work through the various operations needed to deploy your configuration.

Testing

You can quickly test your newly deployed configuration using the tpaexec `test` command which will run pgbench on your new database.

```
tpaexec test demo
```

Connecting

To get to a psql prompt, the simplest route is to log into one of the containers (or VMs or host depending on configuration) using docker or SSH. Run

```
tpaexec ping demo
```

to ping all the connectable hosts in the deployment: You will get output that looks something like:

```
output
unfair | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
uptake | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
quondam | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
uptight | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Select one of the nodes which responded with `SUCCESS`. We shall use `uptake` for this example.

If you are only planning on using docker, use the command `docker exec -it uptake /bin/bash`, substituting in the appropriate hostname.

Another option, that works with all types of TPA deployment is to use SSH. To do that, first change current directory to the created configuration directory.

For example, our configuration is called demo, so we go to that directory. In there, we run `ssh -F ssh_config ourhostname` to connect.

```
cd demo
ssh -F ssh_config uptake
```

```
output
Last login: Wed Sep  6 10:08:01 2023 from 172.17.0.1
[root@uptake ~]#
```

In both cases, you will be logged in as a root user on the container.

We can now change user to the `postgres` user using `sudo -iu postgres`. As `postgres` we can run `psql`. TPA has already configured that user with a `.pgpass` file so there's no need to present a password.

```
sudo -iu postgres
psql
```

```
output
[root@uptake ~]# sudo -iu postgres
postgres@uptake:~ $ psql
psql (15.4)
Type "help" for help.

postgres=#
```

And we are connected to our database.

You can connect from the host system without SSHing into one of the containers. Obtain the IP address of the host you want to connect to from the `ssh_config` file.

```
grep "^ *Host" demo/ssh_config
```

```
output
Host *
Host uptight
    HostName 172.17.0.9
Host unfair
    HostName 172.17.0.4
Host quondam
    HostName 172.17.0.10
Host uptake
    HostName 172.17.0.11
```

We are going to connect to uptake, so the IP address is 172.17.0.11.

You will also need to retrieve the password for the postgres user too. Run `tpaexec show-password demo postgres` to get the stored password from the system.

```
tpaexec show-password demo postgres
```

```
output
a9LmI1X^uMOpPoEnLuRdL%L$oRQak3om
```

Assuming you have a Postgresql client installed, you can then run:

```
psql --host 172.17.0.11 -U postgres
```

```
output
Password for user postgres:
```

Enter the password you previously retrieved.

```
output
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1), server 15.4)
WARNING: psql major version 14, server major version 15.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

You are now connected from the Docker host to Postgres running in one of the TPA deployed Docker containers.

8 Cluster configuration

The `tpaexec configure` command generates a YAML cluster configuration file that is required by subsequent stages in the provision/deploy/test cycle.

Quickstart

```
tpaexec configure ~/clusters/speedy --architecture M1 \
        --postgresql 14 \
        --failover-manager repmgr
```

This command will create a directory named `~/clusters/speedy` and generate a configuration file named `config.yml` that follows the layout of the architecture named M1 (single primary, N replicas). It will create a git repository in the new directory and make an initial commit containing the generated `config.yml`.

The command also accepts various options (some specific to the selected architecture or platform) to modify the configuration, but the defaults are sensible and intended to be usable straightaway. You are encouraged to read the generated config.yml and fine-tune the configuration to suit your needs. (Here's an overview of configuration settings that affect the deployment.)

It's possible to write config.yml entirely by hand, but it's much easier to edit the generated file.

Configuration options

The first argument must be the cluster directory, e.g., `speedy` or `~/clusters/speedy` (the cluster will be named speedy in both cases). We recommend that you keep all your clusters in a common directory, e.g., `~/clusters` in the example above.

The next argument must be `--architecture <name>` to select an architecture, e.g., M1 or BDR-Always-ON or PGD-X. For a complete list of architectures, run `tpaexec info architectures`.

Next, you must specify a flavour and version of Postgres to install.

The arguments above are always mandatory. The rest of the options described here may be safely omitted, as in the example above; the defaults will lead to a usable result.

Run `tpaexec help configure-options` for a list of common options.

Architecture-specific options

The architecture you select determines what other options are accepted. Typically, each architecture accepts some unique options as well as the generic options described below.

For example, with M1 you can use `--location-names l1 l2` to create a cluster with nodes in two named locations. Please consult the documentation for an architecture for a list of options that it accepts (or, in some cases, requires).

Platform options

Next, you may use `--platform <name>` to select a platform, e.g., aws or bare.

An architecture may or may not support a particular platform. If not, it will fail to configure the cluster.

The choice of platform affects the interpretation of certain options. For example, if you choose aws, the arguments to `--region <region>` and `--instance-type <type>` must be a valid AWS region name and EC2 instance type respectively. Please refer to the platform documentation for more details.

If you do not explicitly select a platform, the default is currently aws.

Note: TPA fully supports creating clusters with instances on different platforms, but `tpaexec configure` cannot currently generate such a configuration. You must edit config.yml to specify multiple platforms.

Owner

Specify `--owner <name>` to associate the cluster (by some platform-specific means, e.g., AWS tags) with the name of a person responsible for it. This is especially important for cloud platforms. By default, the owner is set to the login name of the user running `tpaexec provision`.

(You may use your initials, or "Firstname Lastname", or anything else that identifies you uniquely.)

Region

Specify `--region <region>` to select a region.

This option is meaningful only for cloud platforms. The default for AWS is eu-west-1.

Note: TPA fully supports creating clusters that span multiple regions, but `tpaexec configure` cannot currently generate such a configuration. You must edit config.yml to specify multiple regions.

Network configuration

> **Note**
>
> These options are not meaningful for the "bare" platform, where TPA will not alter the network configuration of existing servers.

By default, each cluster will be configured with a number of randomly selected `/28` subnets from the CIDR range `10.33.0.0/16`, depending on the selected architecture.

Specify `--network 192.168.0.0/16` to assign subnets from a different network. On AWS clusters, this corresponds to the VPC CIDR. See aws documentation for details.

Specify `--subnet-prefix 26` to assign subnets of a different size, /26 instead of /28 in this case.

> **Note**
>
> When the "docker" platform is selected, TPA will always place the entire cluster in a single subnet regardless of the architecture. This subnet is generated according to the logic described here with the exception that if the `subnet-prefix` is not specified, TPA will automatically select a subnet size large enough to accommodate the number of instances in `config.yaml`.

Specify `--no-shuffle-subnets` to allocate subnets from the start of the network CIDR range, without randomisation, e.g. `10.33.0.0/28`, then `10.33.0.16/28` and so on.

Specify `--exclude-subnets-from <directory>` to exclude subnets that are already used in existing cluster config.yml files. You can specify this argument multiple times for each directory.

Instance type

Specify `--instance-type <type>` to select an instance type.

This option is meaningful only for cloud platforms. The default for AWS is t3.micro.

Disk space

Specify `--root-volume-size 64` to set the size of the root volume in GB. (Depending on the platform, there may be a minimum size required for the root volume.)

The `--postgres-volume-size <size>` and `--barman-volume-size <size>` options are available to set the sizes of the Postgres and Barman volumes on those architectures and platforms that support separate volumes for Postgres and Barman.

None of these options is meaningful for the "bare" platform, where TPA has no control over volume sizes.

Hostnames

By default, `tpaexec configure` will randomly select as many hostnames as it needs from a pre-approved list of several dozen names. This should be enough for most clusters.

Specify `--hostnames-from <filename>` to select hostnames from a file with one name per line. The file must contain at least as many valid hostnames as there are instances in your cluster. Each line may contain an optional IP address after the name; if present, this address will be set as the `ip_address` for the corresponding instance in `config.yml`. If two ip addresses are present, the first will be set as `public_ip` and the second as `private_ip`.

> **Note**
>
> When you explicitly specify IP addresses, whether through a hostnames file or directly in `config.yml`, you must ensure that they fall within the cluster network CIDR range. For Docker and AWS platform, this means that you must specify a cluster network if you specify IP addresses.
>
> If you do not specify a cluster network CIDR range, TPA will select one randomly with no reference to the specified IP addresses. This will result in a failed deployment or a cluster with different IP addresses than you were expecting.

Use `--hostnames-pattern '…pattern…'` to limit the selection to lines matching an egrep pattern.

Use `--hostnames-sorted-by="--dictionary-order"` to select a sort(1) option other than `--random-sort` (which is the default).

Use `--hostnames-unsorted` to not sort hostnames at all. In this case, they will be assigned in the order they are found in the hostnames file. This is the default when a hostnames file is explicitly specified.

Use `--cluster-prefixed-hostnames` to make each hostname begin with the name of the cluster. This can be useful to avoid hostname clashes when running more than one docker cluster on the same host.

Hostnames may contain only letters (a-z), digits (0-9), and '-'. They may be FQDNs, depending on the selected platform. Hostnames should be in lowercase; any uppercase characters will be converted to lowercase internally, and any references to these hostnames in config.yml (e.g., `upstream: hostname`) must use the lowercase version.

Software selection

Distribution

Specify `--distribution <name>` to select a distribution.

The selected platform determines which distributions are available, and which one is used by default.

In general, you should be able to use "Debian", "RedHat", "Ubuntu", and "SLES" to select the right images.

Optionally, you can include the version of the distribution by passing `--os-version <version number>`.

This option is not meaningful for the "bare" platform, where TPA has no control over which distribution is installed.

EDB repositories

TPA can enable any EDB software repository that you have access to through a subscription. By default, TPA will install any product repositories that the architecture requires.

More detailed explanation of how TPA uses EDB repositories is available here and on the page for each architecture.

Specify `--edb-repositories repository …` to specify the complete list of EDB repositories to install on each instance.

Use this option with care. TPA will configure the named repositories with no attempt to make sure the combination is appropriate.

To use this options, you must `export EDB_SUBSCRIPTION_TOKEN=xxx` before you run TPA. You can get an EDB token from enterprisedb.com/repos.

Local repository support

Use `--enable-local-repo` to create a local package repository from which to ship packages to target instances.

In environments with restricted network access, you can instead use `--use-local-repo-only` to create a local repository and disable all other package repositories on target instances, so that packages are installed only from the local repository.

The page about Local repository support has more details.

Software versions

Postgres flavour and version

TPA supports PostgreSQL, EDB Postgres Extended, and EDB Postgres Advanced Server (EPAS) versions 13 through 18.

You must specify both the flavour (or distribution) and major version of Postgres to install, for example:

- `--postgresql 14` will install PostgreSQL 14
- `--edb-postgres-extended 15` will install EDB Postgres Extended 15
- `--edb-postgres-advanced 15 --redwood` will install EPAS 15 in "Redwood" mode

- `--edb-postgres-advanced 15 --no-redwood` will install EPAS 15 in non-Redwood mode

If you are installing EPAS, you must specify whether it should operate in `--redwood` or `--no-redwood` mode, i.e., whether to enable or disable its Oracle compatibility features.

Installing EDB Postgres Extended or Postgres Advanced Server requires a valid EDB repository subscription.

Package versions

By default, we always install the latest version of every package. This is usually the desired behaviour, but in some testing scenarios, it may be necessary to select specific package versions using any of the following options:

1. `--postgres-package-version 10.4-2.pgdg90+1`
2. `--repmgr-package-version 4.0.5-1.pgdg90+1`
3. `--barman-package-version 2.4-1.pgdg90+1`
4. `--pglogical-package-version '2.2.0*'`
5. `--bdr-package-version '3.0.2*'`
6. `--pgbouncer-package-version '1.8*'`
7. `--beacon-agent-package-version 1.56.2-1`
8. `--etcd-package-version 9.8.0-1.el8`
9. `--patroni-package-version 4.0.0-1PGDG.rhel8`
10. `--pem-server-package-version 9.7.0-1.el9`
11. `--pem-agent-package-version 9.7.0-1.el9`
12. `--pg-backup-api-package-version 2.0.0-1.el8`
13. `--pgd-proxy-package-version 5.0.0-1`
14. `--pgdcli-package-version 5.6.1`

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

You may also specify `--extra-packages p1 p2 …` or `--extra-postgres-packages p1 p2 …` to install additional packages. The former lists packages to install along with system packages, while the latter lists packages to install later along with postgres packages. (If you mention packages that depend on Postgres in the former list, the installation will fail because Postgres will not yet be installed.) The arguments are passed on to the package manager for installation without any modifications.

The `--extra-optional-packages p1 p2 …` option behaves like `--extra-packages`, but it is not an error if the named packages cannot be installed.

Known issue with wildcard use

Please note that the use of wildcards in `*_package_version` when added permanently to `config.yml`, can result in unexpected updates to installed software during `tpaexec deploy` on nodes with RHEL 8 and above (or derivative OSs which use dnf such as Rocky Linux). When deploy runs on an existing cluster that already has packages installed ansible may be unable to match the full package version. For example, if the value for `bdr_package_version` was set to `3.6*` then ansible would not be able to match this to an installed version of PGD, it would assume no package is installed, and it would attempt to install the latest version available of the package with the same name in the configured repository, e.g. 3.7.

We are aware of this limitation as an ansible dnf module bug and hope to address this in a future release of TPA.

Building and installing from source

If you specify `--install-from-source postgres`, Postgres will be built and installed from a git repository instead of installed from packages. By default, this will build the appropriate `REL_nnn_STABLE` branch.

You may use `--install-from-source postgres bdr5` to build and install both components from source, or just use `--install-from-source bdr5` to use packages for Postgres, but build and install PGD v5 from source. By default, this will build the `main` branch of PGD.

To build a different branch, append `:branchname` to the corresponding argument. For example `--install-from-source 2ndqpostgres:dev/xxx`, or `pglogical:bug/nnnn`.

You may not be able to install packages that depend on a package that you chose to replace with a source installation instead. For example, PGD v3 packages depend on pglogical v3 packages, so you can't install pglogical from its source repository and PGD from packages. Likewise, you can't install Postgres from source and pglogical from packages.

Overrides

You may optionally specify `--overrides-from a.yml …` to load one or more YAML files with settings to merge into the generated config.yml.

Any file specified here is first expanded as a Jinja2 template, and the result is loaded as a YAML data structure, and merged recursively into the arguments used to generate config.yml (comprising architecture and platform defaults and arguments from the command-line). This process is repeated for each additional override file specified; this means that settings defined by one file will be visible to any subsequent files.

For example, your override file might contain:

```
cluster_tags:
  some_tag: "{{ lookup('env', 'SOME_ENV_VAR')
}}"

cluster_vars:
  synchronous_commit: remote_write
  postgres_conf_settings:
    effective_cache_size:
4GB
```

These settings will augment `cluster_tags` and `cluster_vars` that would otherwise be in config.yml. Settings are merged recursively, so `cluster_tags` will end up containing both the default Owner tag as well as `some_tag`. Similarly, the `effective_cache_size` setting will override that variable, leaving other `postgres_conf_settings` (if any) unaffected. In other words, you can set or override specific subkeys in config.yml, but you can't empty or replace `cluster_tags` or any other hash altogether.

The merging only applies to hash structures, so you cannot use this mechanism to change the list of `instances` within config.yml. It is most useful to augment `cluster_vars` and `instance_defaults` with common settings for your environment.

That said, the mechanism does not enforce any restrictions, so please exercise due caution. It is a good idea to generate two configurations with and without the overrides and diff the two config.yml files to make sure you understand the effect of all the overrides.

Ansible Tower

Use the `--use-ansible-tower` and `--tower-git-repository` options to create a cluster adapted for deployment with Ansible Tower. See Ansible Tower for details.

PGAI agent

Use the `--enable-beacon-agent` and `--beacon-agent-project-id` options to install the PGAI agent (packaged as `beacon-agent`), which enables you to view your cluster in the EDB Postgres AI Console. See Configuring the PGAI agent for details.

Git repository

By default, a git repository is created with an initial branch named after the cluster, and a single commit is made, with the configure options you used in the commit message. If you don't have git in your `$PATH`, tpaexec will not raise an error but the repository will not be created. To suppress creation of the git repository, use the `--no-git` option. (Note that in an Ansible Tower cluster, a git repository is required and will be created later by `tpaexec provision` if it does not already exist.)

Keyring backend for vault password

TPA generates a cluster-specific Ansible vault password. This password is used to encrypt other sensitive variables generated for the cluster: Postgres user password, Barman user password and so on.

Keyring backend `system` will leverage the best keyring backend on your system from the list of backends supported by the Python keyring module including gnome-keyring and secret-tool.

The default is to store the vault password using `system` keyring for new cluster. removing `keyring_backend: system` in config.yml file **before** any `provision` will revert previous default to store vault password in plaintext file.

Using `keyring_backend: system` also generates a `vault_name` entry in config.yml used to store the vault password unique storage name. TPA generates a UUID by default but there is no naming scheme requirement.

The keyring backend can be specified during `tpaexec configure` by supplying the `--keyring-backend` option. Permitted values are `legacy` and `system`.

> **Note**
>
> When using `keyring_backend: system` and the same base config.yml file for multiple clusters with same `cluster_name`, by copying the config file to a different location, ensure the value pair (`vault_name`, `cluster_name`) is unique for each cluster copy.

> **Note**
>
> Note: When using `keyring_backend: system` and moving an already provisioned cluster folder to a different tpa host, ensure that you export the associated vault password on the new machine's system keyring. vault password can be displayed via `tpaexec show-vault <cluster_dir>`.

Security standards compliance

Use the `--compliance stig` or `--compliance cis` options to generate a cluster with configuration suitable for complying with the STIG or CIS standard. See Compliance for details. Note that these options do not guarantee that the cluster fulfils the relevant standard; they only cause TPA to generate a configuration designed to comply with those aspects of the standard that can be controlled by TPA.

Examples

Let's see what happens when we run the following command:

```
tpaexec configure ~/clusters/speedy --architecture M1 \
        --distribution Debian \
        --os-version 12 \
        --platform aws --region us-east-1 --network 10.33.0.0/16 \
        --instance-type t2.medium --root-volume-size 32 \
        --postgres-volume-size 64 --barman-volume-size 128 \
        --postgresql 14 \
        --failover-manager repmgr
```

There is no output, so there were no errors. The cluster directory has been created and populated.

```
ls -lh ~/clusters/speedy/
```

output

```
total 8.0K
drwxrwxr-x 2 haroon haroon 4.0K Aug 17 02:33 commands
-rw-rw-r-- 1 haroon haroon 1.5K Aug 17 02:33 config.yml
lrwxrwxrwx 1 haroon haroon   53 Aug 17 02:33 deploy.yml -> /home/haroon/tpa/architectures/M1/deploy.yml
```

The cluster configuration is in config.yml, and its neighbours are links to architecture-specific support files that you need not interact with directly. Here's what the configuration looks like:

```yaml
---
architecture: M1
cluster_name:
speedy
cluster_tags: {}

keyring_backend:
system
vault_name: cfae3da3-ec00-46cd-ab05-e153f1c788db

cluster_rules:
- cidr_ip: 0.0.0.0/0
  from_port: 22
  proto:
tcp
  to_port: 22
- cidr_ip: 10.33.120.80/28
  from_port: 0
  proto:
tcp
  to_port: 65535
ec2_ami:
  Name: debian-11-amd64-20240104-1616
  Owner: '136693071363'
ec2_instance_reachability:
public
ec2_vpc:
  us-east-1:
    Name: Test
    cidr: 10.33.0.0/16

cluster_vars:
  edb_repositories: []
  failover_manager:
repmgr
```

```
  postgres_flavour: postgresql
  postgres_version: '14'
  preferred_python_version: python3

locations:
- Name: main
  az: us-east-1a
  region: us-east-1
  subnet: 10.33.120.80/28

instance_defaults:
  default_volumes:
  - device_name: root
    encrypted: true
    volume_size: 32
    volume_type: gp2
  - device_name: /dev/sdf
    encrypted: true
    vars:
      volume_for: postgres_data
    volume_size: 64
    volume_type: gp2
  platform: aws
  type: t2.medium
  vars:
    ansible_user: admin

instances:
- Name: uproar
  backup: kinsman
  location: main
  node: 1
  role:
  - primary
- Name: unravel
  location: main
  node: 2
  role:
  - replica
  upstream: uproar
- Name: kinsman
  location: main
  node: 3
  role:
  - barman
  - log-server
  - witness
  upstream: uproar
  volumes:
  - device_name: /dev/sdf
    encrypted: true
    vars:
```

```
      volume_for:
barman_data
    volume_size: 128
    volume_type:
gp2
```

The next step is to run `tpaexec provision` or learn more about how to customise the configuration of the cluster as a whole or how to configure an individual instance.

9 tpaexec provision

Provision creates instances and other resources required by the cluster.

The exact details of this process depend both on the architecture (e.g. M1) and platform (e.g. AWS) that you selected while configuring the cluster.

At the end of the provisioning stage, you will have the required number of instances with the basic operating system installed, which TPA can access via ssh (with sudo to root).

Prerequisites

Before you can provision a cluster, you must generate the cluster configuration with `tpaexec configure` (and edit config.yml to fine-tune the configuration if needed).

You may need additional platform-dependent steps. For example, you need to obtain an AWS API access keypair to provision EC2 instances, or set up LXD or Docker to provision containers. Consult the platform documentation for details.

Quickstart

```
tpaexec provision ~/clusters/speedy
```

output

```
PLAY [Provision cluster] **********************************************************
...

TASK [Set up EC2 instances] *******************************************************
changed: [localhost] => (item=us-east-1:uproar)
changed: [localhost] => (item=us-east-1:unravel)
changed: [localhost] => (item=us-east-1:kinsman)
...

TASK [Generate ssh_config file for the cluster] ***********************************
changed: [localhost]

PLAY RECAP ************************************************************************
localhost                  : ok=163  changed=35   unreachable=0    failed=0    skipped=44   rescued=0
ignored=2


real    4m42.726s
user    0m39.101s
sys     0m15.687s
```

This command will produce lots of output (append `-v` , `-vv` , etc. to the command if you want even more verbose output). The output is also logged to `ansible.log` in the cluster directory. This can be overriden by setting the environment variable `ANSIBLE_LOG_PATH` to the path and name of the desired logfile.

If it completes without error, you may proceed to run `tpaexec deploy` to install and configure software.

Options

When provisioning cloud instances, it is especially important to make sure instances are directly traceable to a human responsible for them. By default, TPA will tag EC2 instances as being owned by the login name of the user running `tpaexec provision`.

Specify `--owner <name>` to change the name (e.g., if your username happens to be something generic, like postgres or ec2-user). You may use initials, or "Firstname Lastname", or anything else to uniquely identify a person.

Any other options you specify are passed on to Ansible.

Accessing the instances

After provisioning completes, you should be able to SSH to the instances (after a brief delay to allow the instances to boot up and install their SSH host keys). As shown in the output above, tpaexec will generate an ssh_config file for you to use.

```
cd ~/clusters/speedy
cat ssh_config
```

output

```
Host *
    Port 22
    IdentitiesOnly yes
    IdentityFile "id_speedy"
    UserKnownHostsFile known_hosts tpa_known_hosts
    ServerAliveInterval 60

Host uproar
    User admin
    HostName 3.88.255.205
Host unravel
    User admin
    HostName 54.80.99.142
Host kinsman
    User admin
    HostName 54.165.229.179
```

To login to a host, use the command `ssh -F ssh_config` followed by the hostname. For example `ssh -F ssh_config uproar`.

If you need to connect to the instances through a jump host, specify its details in the `jumphost` top-level section of `config_yml`:

output

```
jumphost:
    name: myjumphost
    user: my_username
    identity_file: ~/.ssh/id_ed25519
```

TPA will then create a `Host` section in the generated ssh config for the jump host, and add the `ProxyJump` directive to each host.

You can run `tpaexec deploy` immediately after provisioning. It will wait as long as required for the instances to come up. You do not need to wait for the instances to come up, or ssh in to them before you start deployment.

Generated files

During the provisioning process, a number of new files will be created in the cluster directory:

```
ls ~/clusters/speedy
```

```
output
total 240
-rw-r--r-- 1 ams ams 193098 Aug  4 17:59 ansible.log
drwxr-xr-x 2 ams ams   4096 Aug  4 17:38 commands
-rw-r--r-- 1 ams ams   1442 Aug  4 17:54 config.yml
lrwxrwxrwx 1 ams ams     51 Aug  4 17:38 deploy.yml ->
                             /opt/EDB/TPA/architectures/M1/deploy.yml
drwxr-xr-x 2 ams ams   4096 Aug  4 17:38 hostkeys
-rw------- 1 ams ams   1675 Aug  4 17:38 id_speedy
-rw------- 1 ams ams   1438 Aug  4 17:38 id_speedy.ppk
-rw-r--r-- 1 ams ams    393 Aug  4 17:38 id_speedy.pub
drwxr-xr-x 4 ams ams   4096 Aug  4 17:50 inventory
-rw-r--r-- 1 ams ams   2928 Aug  4 17:50 tpa_known_hosts
-rw-r--r-- 1 ams ams    410 Aug  4 17:50 ssh_config
-rw-r--r-- 1 ams ams   3395 Aug  4 17:59 vars.json
drwxr-xr-x 2 ams ams   4096 Aug  4 17:38 vault
```

We've already studied the ssh*config file, which refers to the `id**` `files (an SSH keypair generated for the cluster) and` tpa_known_hosts `(the signatures of the` hostkeys/` installed on the instances).

The `vars.json` file may be used by `tpaexec provision` on subsequent invocations with `--cached`.

The `inventory/` directory contains static and dynamic inventory files as well as group and host variable definitions from config.yml.

```
cat inventory/00-speedy
```

```
output
[tag_Cluster_speedy]
uproar ansible_host=3.88.255.205 node=1 platform=aws
unravel ansible_host=54.80.99.142 node=2 platform=aws
kinsman ansible_host=54.165.229.179 node=3 platform=aws
```

```
cat inventory/group_vars/tag_Cluster_speedy/01-speedy.yml
```

```
output
cluster_name: speedy
cluster_tag: tag_Cluster_speedy
edb_repositories: []
failover_manager: repmgr
keyring_backend: system
postgres_flavour: postgresql
postgres_version: '14'
preferred_python_version: python3
ssh_key_file: id_speedy
tpa_version: v23.33-24-g4c0909d1
```

```
cat inventory/host_vars/kinsman/01-instance_vars.yml
```

```
output
ansible_user: admin
location: main
region: us-east-1
role:
- barman
- log-server
- witness
upstream: uproar
volumes:
- device: /dev/xvda
- device: /dev/sdf
  volume_for: barman_data
```

If you now change a variable in config.yml and rerun provision, these files will be updated. If you don't change the configuration, it won't do anything. If you add a new instance in config.yml and rerun, it will bring up the new instance without affecting the existing ones.

10 tpaexec deploy

Deployment is the process of installing and configuring Postgres and other software on the cluster's servers. This includes setting up replication, backups, and so on.

At the end of the deployment stage, Postgres will be up and running along with other components like repmgr, Barman, pgbouncer, etc. (depending on the architecture selected).

Prerequisites

Before you can run `tpaexec deploy`, you must have already run `tpaexec configure` to generate the cluster configuration and then provisioned the servers with `tpaexec provision`.

Before deployment, you must `export EDB_SUBSCRIPTION_TOKEN=xxx` if you are using any EDB repositories. If you forget to do this, an error message will soon remind you.

Quickstart

```
tpaexec deploy ~/clusters/speedy -v
```

```
output
Using /opt/EDB/TPA/ansible/ansible.cfg as config file

PLAY [Basic initialisation and fact discovery] ***************************************
...

PLAY [Set up TPA cluster nodes] ******************************************************
...

PLAY RECAP ***************************************************************************
zealot                     : ok=281  changed=116  unreachable=0    failed=0
keeper                     : ok=284  changed=96   unreachable=0    failed=0
quaver                     : ok=260  changed=89   unreachable=0    failed=0
quavery                    : ok=260  changed=88   unreachable=0    failed=0
quirk                      : ok=262  changed=100  unreachable=0    failed=0


real    7m1.907s
user    3m2.492s
sys     1m5.318s
```

This command produces a great deal of output and may take a long time (depending primarily on the latency between the host running tpaexec and the hosts in the cluster, as well as how long it takes the instances to download the packages they need to install). We recommend that you use at least one `-v` during deployment. The output is also logged to `ansible.log` in the cluster directory.

The exact number of hosts, tasks, and changed tasks may of course vary.

The deploy command takes no options itself—any options you provide after the cluster name are passed on unmodified to Ansible (e.g., `-v`).

Those who are familiar with Ansible may be concerned by the occasional red "failed" task output scrolling by. Rest assured that if the process does not stop soon afterwards, the error is of no consequence, and the code will recover from it automatically.

When the deployment is complete, you can run `tpaexec test` to verify the installation.

Selective deployment

You can limit the deployment to a subset of your hosts by setting `deploy_hosts` to a comma-separated list of instance names:

```
tpaexec deploy ~/clusters/speedy -v -e deploy_hosts=keeper,quaver
```

This will run the deployment on the given instances, though it will also initially execute some tasks on other hosts to collect information about the state of the cluster.

(Setting `deploy_hosts` is the recommended alternative to using Ansible's `--limit` option, which TPA does not support.)

deploy.yml

The deployment process is architecture-specific. Here's an overview of the various configuration settings that affect the deployment. If you are familiar with Ansible playbooks, you can follow along as tpaexec applies various roles to the cluster's instances.

Unlike config.yml, deploy.yml is not designed to be edited (and is usually a link into the architectures directory). Even if you want to extend the deployment process to run your own Ansible tasks, you should do so by creating hooks. This protects you from future implementation changes within a particular architecture.

11 tpaexec test

Now we run architecture-specific tests against a deployed cluster to verify the installation. At the end of this stage, we have a fully-functioning cluster.

You must have already run `tpaexec configure`, `tpaexec provision`, and `tpaexec deploy` successfully before you can run `tpaexec test`.

Quickstart

```
tpaexec test ~/clusters/speedy -v
```

Output is once again logged to `ansible.log` in the cluster directory.

If this command succeeds, your cluster works.

Congratulations.

12 PGD-S

> **Note**
>
> This architecture is for Postgres Distributed 6 only. If you require PGD 5 please use PGD-Always-ON.

EDB Postgres Distributed 6 in a PGD Essential (PGD-S) configuration suitable for use in test and production.

This architecture requires an EDB subscription. All software is sourced from EDB Repos 2.0.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.

```
tpaexec configure ~/clusters/pgd-s \
    --architecture PGD-S
    --edb-postgres-extended 15 \
    --platform aws --instance-type t3.micro \
    --distribution Debian \
```

You can list all available options using the help command.

```
tpaexec configure --architecture PGD-S --help
```

The table below describes the mandatory options for PGD-S and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Options	Description
`--architecture` (`-a`)	Must be set to `PGD-S`
Postgres flavour and version (e.g. `--postgresql 15`)	A valid flavour and version specifier. Supports Postgres 14-18.

Additional Options

Options	Description	Behaviour if omitted
`--platform`	One of `aws` , `docker` , `bare` .	Defaults to `aws` .
`--bdr-database`	The name of the database to be used for replication.	Defaults to `bdrdb` .
`--layout`	`standard` or `near-far`	Defaults to `standard`
`--add-subscriber-only-nodes`	The number of subscriber-only nodes to add, up to 10.	Defaults to 0

Options	Description	Behaviour if omitted
`--read-write-port`	The port for Connection Manager to listen on for read-write connections.	Left empty in config.yml, allowing default of the postgres port + 1000
`--read-only-port`	The port for Connection Manager to listen on for read-only connections.	Left empty in config.yml, allowing default of the read-write port + 1
`--http-port`	The port for Connection Manager to listen on for http api connections.	Left empty in config.yml, allowing default of the read-write port + 2
`--use https`	Enable https for Connection Manager's http api	https is not enabled

More detail about PGD-S configuration

A PGD-S cluster has three data nodes. In the `standard` layout the nodes are all in the same location; in the `near-far` layout two nodes are in the primary location and the other is in a secondary location. See the PGD documentation for more information about the two layouts.

The cluster also contains one barman node and up to 10 subscriber-only nodes, controlled by the `--add-subscriber-only-nodes` parameter. These are always in the primary location.

You may optionally specify `--bdr-database dbname` to set the name of the database with BDR enabled (default: bdrdb).

You may also specify any of the options described by `tpaexec help configure-options`.

13 PGD-X

> **Note**
>
> This architecture is for Postgres Distributed 6 only. If you require PGD 5 please use PGD-Always-ON.

EDB Postgres Distributed 6 in a PGD Expanded (PGD-X) configuration suitable for use in test and production.

This architecture requires an EDB subscription. All software is sourced from EDB Repos 2.0.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.

```
tpaexec configure ~/clusters/pgd-x \
         --architecture PGD-X \
         --edb-postgres-extended 15 \
         --platform aws --instance-type t3.micro \
         --distribution Debian \
         --pgd-routing global \
         --location-names dc1 dc2 dc3 \
         --witness-only-location dc3 \
         --data-nodes-per-location 2
```

You can list all available options using the help command.

```
tpaexec configure --architecture PGD-X --help
```

The table below describes the mandatory options for PGD-X and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Options	Description
`--architecture` (`-a`)	Must be set to `PGD-X`
Postgres flavour and version (e.g. `--postgresql 15`)	A valid flavour and version specifier. Supports Postgres 14-18.
`--pgd-routing`	Must be either `global` or `local` .

Additional Options

Options	Description	Behaviour if omitted
`--platform`	One of `aws`, `docker`, `bare`.	Defaults to `aws`.
`--location-names`	A space-separated list of location names. The number of locations is equal to the number of names supplied.	TPA will configure a single location with three data nodes.
`--witness-only-location`	A location name, must be a member of `location-names`.	No witness-only location is added.
`--data-nodes-per-location`	The number of data nodes in each location, must be at least 2.	Defaults to 3.
`--enable-camo`	Sets two data nodes in each location as CAMO partners.	CAMO will not be enabled.
`--bdr-database`	The name of the database to be used for replication.	Defaults to `bdrdb`.
`--enable-pgd-probes`	Enable http(s) api endpoints for pgd-proxy such as `health/is-ready` to allow probing proxy's health.	Disabled by default.
`--read-write-port`	The port for Connection Manager to listen on for read-write connections.	Left empty in config.yml, allowing default of the postgres port + 1000
`--read-only-port`	The port for Connection Manager to listen on for read-only connections.	Left empty in config.yml, allowing default of the read-write port + 1

More detail about PGD-X configuration

A PGD-X cluster comprises a number of locations, preferably odd, each with the same number of data nodes, again preferably odd. If you do not specify any `--location-names`, the default is to use a single location with three data nodes.

Location names for the cluster are specified as `--location-names dc1 dc2 …`. A location represents an independent data center that provides a level of redundancy, in whatever way this definition makes sense to your use case. For example, AWS regions, your own data centers, or any other designation to identify where your servers are hosted.

> **for AWS users**
>
> If you are using TPA to provision an AWS cluster, the locations will be mapped to separate availability zones within the `--region` you specify. You may specify multiple `--regions`, but TPA does not currently set up VPC peering to allow instances in different regions to communicate with each other. For a multi-region cluster, you will need to set up VPC peering yourself.

Use `--data-nodes-per-location N` to specify the number of data nodes in each location. The minimum number is 2, the default is 3.

If you specify an even number of data nodes per location, TPA will add an extra witness node to each location automatically. This retains the ability to establish reliable consensus while allowing cost savings (a witness has minimal hardware requirements compared to the data nodes).

A cluster with only two locations would entirely lose the ability to establish global consensus if one of the locations were to fail. We recommend adding a third witness-only location (which contains no data nodes, only a witness node, again used to reliably establish consensus). Use `--witness-only-location loc` to designate one of your locations as a witness.

Depending on your use-case, you must specify `--pgd-routing local` or `global` to configure how Connection Manager will route connections to a write leader. Local routing will make every Connection Manager route to a write leader within its own location (suitable for geo-sharding applications). Global routing will make every Connection Manager route to a single write leader, elected amongst all available data nodes across all locations.

You may optionally specify `--bdr-database dbname` to set the name of the database with BDR enabled (default: bdrdb).

You may optionally specify `--enable-camo` to set two data nodes in each region as CAMO partners.

You may optionally specify `--enable-pgd-probes [{http, https}]` to enable http(s) api endpoints that will allow to easily probe proxy's health.

You may also specify any of the options described by `tpaexec help configure-options`.

14 PGD-Always-ON

Note

This architecture is for Postgres Distributed 5 only. If you require PGD 4 or 3.7 please use BDR-Always-ON.

EDB Postgres Distributed 5 in an Always-ON configuration, suitable for use in test and production.

This architecture requires an EDB subscription. All software will be sourced from EDB Repos 2.0.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.

```
tpaexec configure ~/clusters/pgd-ao \
         --architecture PGD-Always-ON \
         --edb-postgres-extended 15 \
         --platform aws --instance-type t3.micro \
         --distribution Debian \
         --pgd-proxy-routing global \
         --location-names dc1 dc2 dc3 \
         --witness-only-location dc3 \
         --data-nodes-per-location 2
```

You can list all available options using the help command.

```
tpaexec configure --architecture PGD-Always-ON --help
```

The table below describes the mandatory options for PGD-Always-ON and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Options	Description
`--architecture` (`-a`)	Must be set to `PGD-Always-ON`
Postgres flavour and version (e.g. `--postgresql 15`)	A valid flavour and version specifier.
`--pgd-proxy-routing`	Must be either `global` or `local` .

Additional Options

Options	Description	Behaviour if omitted
`--platform`	One of `aws`, `docker`, `bare`.	Defaults to `aws`.
`--location-names`	A space-separated list of location names. The number of locations is equal to the number of names supplied.	TPA will configure a single location with three data nodes.
`--witness-only-location`	A location name, must be a member of `location-names`.	No witness-only location is added.
`--data-nodes-per-location`	The number of data nodes in each location, must be at least 2.	Defaults to 3.
`--add-proxy-nodes-per-location`	The number of proxy nodes in each location.	PGD-proxy will be installed on each data node.
`--enable-camo`	Sets two data nodes in each location as CAMO partners.	CAMO will not be enabled.
`--bdr-database`	The name of the database to be used for replication.	Defaults to `bdrdb`.
`--enable-pgd-probes`	Enable http(s) api endpoints for pgd-proxy such as `health/is-ready` to allow probing proxy's health.	Disabled by default.
`--proxy-listen-port`	The port on which proxy nodes will route traffic to the write leader.	Defaults to 6432
`--proxy-read-only-port`	The port on which proxy nodes will route read-only traffic to shadow nodes.	Defaults to 6433

More detail about PGD-Always-ON configuration

A PGD-Always-ON cluster comprises a number of locations, preferably odd, each with the same number of data nodes, again preferably odd. If you do not specify any `--location-names`, the default is to use a single location with three data nodes.

Location names for the cluster are specified as `--location-names dc1 dc2 …`. A location represents an independent data centre that provides a level of redundancy, in whatever way this definition makes sense to your use case. For example, AWS regions, your own data centres, or any other designation to identify where your servers are hosted.

for AWS users

If you are using TPA to provision an AWS cluster, the locations will be mapped to separate availability zones within the `--region` you specify. You may specify multiple `--regions`, but TPA does not currently set up VPC peering to allow instances in different regions to communicate with each other. For a multi-region cluster, you will need to set up VPC peering yourself.

Use `--data-nodes-per-location N` to specify the number of data nodes in each location. The minimum number is 2, the default is 3.

If you specify an even number of data nodes per location, TPA will add an extra witness node to each location automatically. This retains the ability to establish reliable consensus while allowing cost savings (a witness has minimal hardware requirements compared to the data nodes).

A cluster with only two locations would entirely lose the ability to establish global consensus if one of the locations were to fail. We recommend adding a third witness-only location (which contains no data nodes, only a witness node, again used to reliably establish consensus). Use `--witness-only-location loc` to designate one of your locations as a witness.

By default, every data node (in every location) will also run PGD-Proxy for connection routing. To create separate PGD-Proxy instances instead, use `--add-proxy-nodes-per-location 3` (or however many proxies you want to add).

Depending on your use-case, you must specify `--pgd-proxy-routing local` or `global` to configure how PGD-Proxy will route connections to a write leader. Local routing will make every PGD-Proxy route to a write leader within its own location (suitable for geo-sharding applications). Global routing will make every proxy route to a single write leader, elected amongst all available data nodes across all locations.

You may optionally specify `--bdr-database dbname` to set the name of the database with BDR enabled (default: bdrdb).

You may optionally specify `--enable-camo` to set two data nodes in each region as CAMO partners.

You may optionally specify `--enable-pgd-probes [{http, https}]` to enable http(s) api endpoints that will allow to easily probe proxy's health.

You may also specify any of the options described by `tpaexec help configure-options`.

15 PGD Lightweight

> **Note**
>
> This architecture is for Postgres Distributed 5 only. If you require PGD 4 or 3.7 please use BDR-Always-ON.
>
> EDB Postgres Distributed 5 in a Lightweight configuration, suitable for use in test and production.
>
> This architecture requires an EDB subscription. All software will be sourced from EDB Repos 2.0.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.

```
tpaexec configure ~/clusters/pgd-lw \
    --architecture Lightweight \
    --edb-postgres-extended 15 \
    --platform aws --instance-type t3.micro \
    --distribution Debian \
    --location-names main dr \
```

You can list all available options using the help command.

```
tpaexec configure --architecture Lightweight --help
```

The table below describes the mandatory options for PGD-Always-ON and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Options	Description
`--architecture` (`-a`)	Must be set to `Lightweight`
Postgres flavour and version (e.g. `--postgresql 15`)	A valid flavour and version specifier.

Additional Options

Options	Description	Behaviour if omitted
`--platform`	One of `aws` , `docker` , `bare` .	Defaults to `aws` .
`--location-names`	A space-separated list of location names. The number of locations is equal to the number of names supplied.	TPA will configure a single location with three data nodes.

Options	Description	Behaviour if omitted
`--add-proxy-nodes-per-location`	The number of proxy nodes in each location.	PGD-proxy will be installed on each data node.
`--bdr-database`	The name of the database to be used for replication.	Defaults to `bdrdb`.
`--enable-pgd-probes`	Enable http(s) api endpoints for pgd-proxy such as `health/is-ready` to allow probing proxy's health.	Disabled by default.
`--proxy-listen-port`	The port on which proxy nodes will route traffic to the write leader.	Defaults to 6432
`--proxy-read-only-port`	The port on which proxy nodes will route read-only traffic to shadow nodes.	Defaults to 6433

More detail about Lightweight configuration

A PGD Lightweight cluster comprises 2 locations, with a primary active location containing 2 nodes and a disaster recovery (dr) location with a single node.

Location names for the cluster are specified as `--location-names primary dr`. A location represents an independent data centre that provides a level of redundancy, in whatever way this definition makes sense to your use case. For example, AWS regions, your own data centres, or any other designation to identify where your servers are hosted.

> **for AWS users**
>
> If you are using TPA to provision an AWS cluster, the locations will be mapped to separate availability zones within the `--region` you specify. You may specify multiple `--regions`, but TPA does not currently set up VPC peering to allow instances in different regions to communicate with each other. For a multi-region cluster, you will need to set up VPC peering yourself.

By default, every data node (in every location) will also run PGD-Proxy for connection routing. To create separate PGD-Proxy instances instead, use `--add-proxy-nodes-per-location 3` (or however many proxies you want to add).

Global routing will make every proxy route to a single write leader, elected amongst all available data nodes across all locations.

You may optionally specify `--bdr-database dbname` to set the name of the database with BDR enabled (default: bdrdb).

You may optionally specify `--enable-pgd-probes [{http, https}]` to enable http(s) api endpoints that will allow to easily probe proxy's health.

You may also specify any of the options described by `tpaexec help configure-options`.

16 BDR-Always-ON

EDB Postgres Distributed 3.7 or 4 in an Always-ON configuration, suitable for use in test and production.

This architecture requires an EDB subscription. All software will be sourced from EDB Repos 2.0.

The BDR-Always-ON architecture has four variants, which can be selected with the `--layout` configure option:

1. bronze: 2×bdr+primary, bdr+witness, barman, 2×harp-proxy
2. silver: bronze, with bdr+witness promoted to bdr+primary, and barman moved to separate location
3. gold: two symmetric locations with 2×bdr+primary, 2×harp-proxy, and barman each; plus a bdr+witness in a third location
4. platinum: gold, but with one bdr+readonly (logical standby) added to each of the main locations

You can check EDB's Postgres Distributed Always On Architectures whitepaper for the detailed layout diagrams.

This architecture is meant for use with PGD versions 3.7 and 4.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.

```
tpaexec configure ~/clusters/bdr \
         --architecture BDR-Always-ON \
         --platform aws --region eu-west-1 --instance-type t3.micro \
         --distribution Debian \
         --edb-postgres-advanced 14 --redwood
         --layout gold \
         --harp-consensus-protocol bdr
```

You can list all available options using the help command.

```
tpaexec configure --architecture BDR-Always-ON --help
```

The tables below describe the mandatory options for BDR-Always-ON and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Option	Description
`--architecture` (`-a`)	Must be set to `BDR-Always-ON` .
Postgres flavour and version (e.g. `--postgresql 14`)	A valid flavour and version specifier.
`--layout`	One of `bronze` , `silver` , `gold` , `platinum` .
`--harp-consensus-protocol`	One of `bdr` , `etcd` .

Additional Options

Option	Description	Behaviour if omitted
`--platform`	One of `aws`, `docker`, `bare`.	Defaults to `aws`.
`--enable-camo`	Sets two data nodes in each location as CAMO partners.	CAMO will not be enabled.
`--bdr-database`	The name of the database to be used for replication.	Defaults to `bdrdb`.
`--enable-harp-probes`	Enable http(s) api endpoints for harp such as `health/is-ready` to allow probing harp's health.	Disabled by default.

More detail about BDR-Always-ON configuration

You must specify `--layout layoutname` to set one of the supported BDR use-case variations. The permitted arguments are bronze, silver, gold, and platinum. The bronze, gold and platinum layouts have a PGD witness node to ensure odd number of nodes for Raft consensus majority. Witness nodes do not participate in the data replication.

You must specify `--harp-consensus-protocol protocolname`. The supported protocols are bdr and etcd; see `Configuring HARP` for more details.

You may optionally specify `--bdr-database dbname` to set the name of the database with PGD enabled (default: bdrdb).

You may optionally specify `--enable-camo` to set the pair of PGD primary instances in each region to be each other's CAMO partners.

You may optionally specify `--enable-harp-probes [{http, https}]` to enable http(s) api endpoints that will allow to easily probe harp's health.

Please note we enable HARP2 by default in BDR-Always-ON architecture.

You may also specify any of the options described by `tpaexec help configure-options`.

17 M1

A Postgres cluster with a single primary node and physical replication to a number of standby nodes including backup and failover management.

This architecture is suitable for production and is also suited to testing, demonstrating and learning due to its simplicity and ability to be configured with no proprietary components.

If you select subscription-only EDB software with this architecture it will be sourced from EDB Repos 2.0 and you will need to provide a token.

Failover management

The M1 architecture always includes a failover manager. Supported options are repmgr, EDB Failover Manager (EFM) and Patroni. In all cases, the failover manager will be configured by default to ensure that a replica will be promoted to take the place of the primary should the primary become unavailable.

Application failover

The M1 architecture does not generally provide an automatic facility to reroute application traffic to the primary. There are several ways you can add this capability to your cluster.

In TPA:

- If you choose repmgr as the failover manager and enable PgBouncer, you can include the `repmgr_redirect_pgbouncer: true` hash under `cluster_vars` in `config.yml`. This causes repmgr to automatically reconfigure PgBouncer to route traffic to the new primary on failover.
- If you choose Patroni as the failover manager and enable PgBouncer, Patroni will automatically reconfigure PgBouncer to route traffic to the new primary on failover.
- If you choose EFM as the failover manager, you can use the `efm_conf_settings` hash under `cluster_vars` in `config.yml` to configure EFM to use a virtual IP address (VIP). This is an additional IP address which will always route to the primary node.
- Place an appropriate proxy or load balancer between the cluster and you application and use a TPA hook to configure your selected failover manager to update it with the route to the new primary on failover.
- Handle failover at the application itself, for example by using multi-host connection strings.

Backup failover

TPA does not configure any kind of 'backup failover'. If the Postgres node from which you are backing up is down, backups will simply halt until the node is back online. To manually connect the backup to the new primary, edit `config.yml` to add the `backup` hash to the new primary instance and re-run `tpaexec deploy`.

Cluster configuration

Overview of configuration options

An example invocation of `tpaexec configure` for this architecture is shown below.


```
tpaexec configure ~/clusters/m1 \
         --architecture M1 \
         --platform aws --region eu-west-1 --instance-type t3.micro \
         --distribution Debian \
         --postgresql 14 \
         --failover-manager repmgr \
         --data-nodes-per-location 3
```

You can list all available options using the help command.

```
tpaexec configure --architecture M1 --help
```

The tables below describe the mandatory options for M1 and additional important options. More detail on the options is provided in the following section.

Mandatory Options

Parameter	Description
`--architecture` (`-a`)	Must be set to `M1`.
Postgres flavour and version (e.g. `--postgresql 15`)	A valid flavour and version specifier.
One of: - `--failover-manager {efm, repmgr, patroni}` - `--enable-efm` - `--enable-repmgr` - `--enable-patroni`	Select the failover manager from `efm`, `repmgr` and `patroni`.

Additional Options

Parameter	Description	Behaviour if omitted
`--platform`	One of `aws`, `docker`, `bare`.	Defaults to `aws`.
`--location-names`	A space-separated list of location names. The number of locations is equal to the number of names supplied.	A single location called "main" is used.
`--primary-location`	The location where the primary server will be. Must be a member of `location-names`.	The first listed location is used.
`--data-nodes-per-location`	A number from 1 upwards. In each location, one node will be configured to stream directly from the cluster's primary node, and the other nodes, if present, will stream from that one.	Defaults to 2.
`--witness-only-location`	A location name, must be a member of `location-names`. This location will be populated with a single witness node only.	No witness-only location is added.
`--single-node-location`	A location name, must be a member of `location-names`. This location will be populated with a single data node only.	No single-node location is added.
`--enable-haproxy`	Two additional nodes will be added as a load balancer layer. Only supported with Patroni as the failover manager.	HAproxy nodes will not be added to the cluster.
`--enable-pgbouncer`	PgBouncer will be configured in the Postgres nodes to pool connections for the primary.	PgBouncer will not be configured in the cluster.

Parameter	Description	Behaviour if omitted
`--patroni-dcs`	Select the Distributed Configuration Store backend for patroni. Only option is `etcd` at this time. Only supported with Patroni as the failover manager.	Defaults to `etcd`.
`--efm-bind-by-hostname`	Enable efm to use hostnames instead of IP addresses to configure the cluster `bind.address`.	Defaults to use IP addresses

More detail about M1 configuration

You may also specify any of the options described by `tpaexec help configure-options`.

18 aws

TPA fully supports provisioning production clusters on AWS EC2.

API access setup

To use the AWS API, you must:

- Obtain an access keypair
- Add it to your configuration

For example,

```
cat > ~/.aws/credentials
```

```
output
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

The IAM user should at least have following set of permissions so tpaexec can use it to provision ec2 resources.

```
ec2:AssociateRouteTable
ec2:AttachInternetGateway
ec2:AuthorizeSecurityGroupIngress
ec2:CreateInternetGateway
ec2:CreateRoute
ec2:CreateRouteTable
ec2:CreateSecurityGroup
ec2:CreateSubnet
ec2:CreateTags
ec2:CreateVpc
ec2:DeleteKeyPair
ec2:DeleteRouteTable
ec2:DeleteSecurityGroup
ec2:DeleteSubnet
ec2:DeleteVpc
ec2:DescribeImages
ec2:DescribeInstanceStatus
ec2:DescribeInstances
ec2:DescribeInternetGateways
ec2:DescribeKeyPairs
ec2:DescribeRouteTables
ec2:DescribeSecurityGroups
ec2:DescribeSubnets
ec2:DescribeTags
ec2:DescribeVolumes
ec2:DescribeVpcAttribute
ec2:DescribeVpcClassicLink
ec2:DescribeVpcClassicLinkDnsSupport
ec2:DescribeVpcs
ec2:DisassociateRouteTable
ec2:ImportKeyPair
ec2:ModifyVpcAttribute
ec2:RevokeSecurityGroupIngress
ec2:RunInstances
```

```
ec2:TerminateInstances
iam:AddRoleToInstanceProfile
iam:CreateInstanceProfile
iam:CreateRole
iam:DeleteInstanceProfile
iam:DeleteRole
iam:DeleteRolePolicy
iam:GetInstanceProfile
iam:GetRole
iam:GetRolePolicy
iam:ListAttachedRolePolicies
iam:ListGroups
iam:ListInstanceProfiles
iam:ListInstanceProfilesForRole
iam:ListRolePolicies
iam:ListRoles
iam:ListUsers
iam:PassRole
iam:PutRolePolicy
iam:RemoveRoleFromInstanceProfile
kms:CreateGrant
kms:GenerateDataKeyWithoutPlaintext
s3:CreateBucket
s3:GetBucketVersioning
s3:GetObject
s3:GetObjectTagging
s3:ListAllMyBuckets
s3:ListBucket
s3:ListBucketVersions
s3:PutBucketOwnershipControls
s3:PutObject
s3:PutObjectAcl
```

Introduction

The service is physically subdivided into regions and availability zones. An availability zone is represented by a region code followed by a single letter, e.g., eu-west-1a (but that name may refer to different locations for different AWS accounts, and there is no way to coordinate the interpretation between accounts).

AWS regions are completely isolated from each other and share no resources. Availability zones within a region are physically separated, and logically mostly isolated, but are connected by low-latency links and are able to share certain networking resources.

Networking

All networking configuration in AWS happens in the context of a Virtual Private Cloud within a region. Within a VPC, you can create subnets that is tied to a specific availability zone, along with internet gateways, routing tables, and so on.

You can create any number of Security Groups to configure rules for what inbound and outbound traffic is permitted to instances (in terms of protocol, a destination port range, and a source or destination IP address range).

Instances

AWS EC2 offers a variety of instance types with different hardware configurations at different price/performance points. Within a subnet in a particular availability zone, you can create EC2 instances based on a distribution image known as an AMI, and attach one or more EBS volumes to provide persistent storage to the instance. You can SSH to the instances by registering an SSH public key.

Instances are always assigned a private IP address within their subnet. Depending on the subnet configuration, they may also be assigned an ephemeral public IP address (which is lost when the instance is shut down, and a different ephemeral IP is assigned when it is started again). You can instead assign a static region-specific routable IP address known as an Elastic IP to any instance.

For an instance to be reachable from the outside world, it must not only have a routable IP address, but the VPC's networking configuration (internet gateway, routing tables, security groups) must also align to permit access.

Configuration

Here's a brief description of the AWS-specific settings that you can specify via `tpaexec configure` or define directly in config.yml.

Regions

You can specify one or more regions for the cluster to use with `--region` or `--regions`. TPA will generate the required vpc entries associated to each of them and distribute locations into these regions evenly by using different availability zones while possible.

`regions` are differents from `locations`, each location belongs to a region (and an availability zone inside this region). `regions` are AWS specific objects, `locations` are cluster objects.

Note: When specifying multiple regions, you need to manually edit network configurations:

- `ec2_vpc` entries must have non-overlaping cidr networks to allow use of AWS vpc peering. by default TPA will set all cidr to `10.33.0.0/16`. See VPC for more informations.
- each `location` must be updated with `subnet` that match the `ec2_vpc` `cidr` they belong to. See Subnets for more informations.
- TPA creates security groups with basic rules under `cluster_rules` and those need to be updated to match `ec2_vpc` cidr for each `subnet` cidr. see Security groups for more informations.
- VPC peering must be setup manually before `tpaexec deploy`. We recommand creating VPCs and required VPC peerings before running `tpaexec configure` and using `vpc-id` in config.yml. See VPC for more informations.

VPC (required)

You must specify a VPC to use:

```
ec2_vpc:
  Name: Test
  cidr: 10.33.0.0/16
```

This is the default configuration, which creates a VPC named Test with the given CIDR if it does not exist, or uses the existing VPC otherwise.

To create a VPC, you must specify both the Name and the cidr. If you specify only a VPC Name, TPA will fail if a matching VPC does not exist.

If TPA creates a VPC, `tpaexec deprovision` will attempt to remove it, but will leave any pre-existing VPC alone. (Think twice before creating new VPCs, because AWS has a single-digit default limit on the number of VPCs per account.)

If you need more fine-grained matching, or to specify different VPCs in different regions, you can use the expanded form:

```
ec2_vpc:
  eu-west-1:
    Name: Test
    cidr: 172.16.0.0/16
  us-east-1:
    filters:
      vpc-id: vpc-nnn
  us-east-2:
    Name: Example
    filters:
      [filter expressions]
```

AMI (required)

You must specify an AMI to use:

```
ec2_ami:
  Name: xxx
  Owner: self
```

You can add filter specifications for more precise matching:

```
ec2_ami:
  Name: xxx
  Owner: self
  filters:
    architecture: x86_64
    [more key/value filters]
```

(By default, `tpaexec configure` will select a suitable `ec2_ami` for you based on the `--distribution` argument.)

Subnets (optional)

Every instance must specify its subnet (in CIDR form, or as a subnet-xxx id). You may optionally specify the name and availability zone for each subnet that we create:

```
ec2_vpc_subnets:
  us-east-1:
    192.0.2.0/27:
      az: us-east-1b
      Name: example1
    192.0.2.100/27:
      az: us-east-1b
      Name: example2
```

Security groups (optional)

By default, we create a security group for the cluster. To use one or more existing security groups, set:

```
ec2_groups:
  us-east-1:
    group-name:
      - foo
```


If you want to customise the rules in the default security group, set `cluster_rules`:

```
cluster_rules:
- cidr_ip: 0.0.0.0/0
  from_port: 22
  proto: tcp
  to_port: 22
- cidr_ip: 192.0.2.0/27
  from_port: 0
  proto: tcp
  to_port: 65535
- cidr_ip: 192.0.2.100/27
  from_port: 0
  proto: tcp
  to_port: 65535
```

This example permits ssh (port 22) from any address, and TCP connections on any port from specific IP ranges. (Note: from_port and to_port define a numeric range of ports, not a source and destination.)

If you set up custom rules or use existing security groups, you must ensure that instances in the cluster are allowed to communicate with each other as required (e.g., allow tcp/5432 for Postgres).

Internet gateways (optional)

By default, we create internet gateways for every VPC, unless you set:

```
ec2_instance_reachability: private
```

For more fine-grained control, you can set:

```
ec2_vpc_igw:
  eu-west-1: yes
  eu-central-1: yes
  us-east-1: no
```

SSH keys (optional)

```
# Set this to change the name under which we register our SSH
key.
# ec2_key_name:
tpa_cluster_name
#
# Set this to use an already-registered
key.
# ec2_instance_key:
xxx
```


S3 bucket (optional)

TPA requires access to an S3 bucket to provision an AWS cluster. This bucket is used to temporarily store files such as SSH host keys, but may also be used for other cluster data (such as backups).

By default, TPA will use an S3 bucket named `edb-tpa-<aws-account-user-id>` for any clusters you provision. (If the bucket does not exist, you will be asked to confirm that you want TPA to create it for you.)

To use an existing S3 bucket instead, set

```
cluster_bucket: name-of-bucket
```

(You can also set `cluster_bucket: auto` to accept the default bucket name without the confirmation prompt.)

TPA will never remove any S3 buckets when you deprovision the cluster. To remove the bucket yourself, run:

```
aws s3 rb s3://<bucket> --force
```

The IAM user you are using to provision the instances must have read and write access to this bucket. During provisioning, tpaexec will provide instances with read-only access to the cluster_bucket through the instance profile.

Elastic IP addresses

To use elastic IP addresses, set `assign_elastic_ip` to `true` in config.yml, either in `instance_defaults` to affect all the instances in your cluster or individually on the separate instances as required. By default, this will allocate a new elastic ip address and assign it to the new instance. To use an elastic IP address that has already been allocated but not yet assigned, use `elastic_ip: 34.252.55.252`, substituting in your allocated address.

Instance profile (optional)

```
# Set this to change the name of the instance profile role we
create.
# cluster_profile:
cluster_name_profile
#
# Set this to use an existing instance profile (which must have all
the
# required permissions assigned to
it).
# instance_profile_name:
xxx
```


19 bare(-metal servers)

Set `platform: bare` in config.yml

This platform is meant to support any server that is accessible via SSH, including bare-metal servers as well as already-provisioned servers on any cloud platform (including AWS).

You must define the IP address(es) and username for each target server:

```yaml
instances:
  - node: 1
    Name: igor
    platform: bare
    public_ip: 192.0.2.1
    private_ip: 192.0.2.222
    vars:
      ansible_user: xyzzy
```

You must ensure that

1. TPA can ssh to the instance as `ansible_user`
2. The `ansible_user` has sudo access on the instance

SSH access

In the example above, TPA will ssh to `xyzzy@192.0.2.1` to access the instance.

By default, TPA will run `ssh-keygen` to generate a new SSH keypair in your cluster directory. The private key is named `id_cluster_name` and the public key is stored in `id_cluster_name.pub`.

You must either set `ssh_key_file: /path/to/id_keyname` to use a different key that the instance will accept, or configure the instance to allow access from the generated key (e.g., use `ssh-copy-id`, which will append the contents of `id_cluster_name.pub` to `~xyzzy/.ssh/authorized_keys`).

You must also ensure that ssh can verify the host key(s) of the instance. You can either add entries to the `known_hosts` file in your cluster directory, or install the TPA-generated host keys from `hostkeys/ssh_host_*_key*` in your cluster directory into `/etc/ssh` on the instance (the generated `tpa_known_hosts` file contains entries for these keys).

For example, to ssh in with the generated user key, but keep the existing host keys, you can do:

```
cd ~/clusters/speedy
ssh-copy-id -i id_speedy xyzzy@192.0.2.1
ssh-keyscan -H 192.0.2.1 >> tpa_known_hosts
```

Run `tpaexec ping ~/clusters/speedy` to check if it's working. If not, append `-vvv` to the command to look at the complete ssh command-line. (Note: Ansible will invoke ssh to execute a command like `bash -c 'python3 && sleep 0'` on the instance. If you run ssh commands by hand while debugging, replace this with a command that produces some output and then exits instead, e.g., `'id'`.)

For more details:

- Use a different ssh key
- Manage ssh host keys for bare instances

Distribution support

TPA will try to detect the distribution running on target instances, and fail if it is not supported. TPA currently supports Debian (11/12; or bullseye/bookworm), Ubuntu (22.04/24.04; or jammy/noble), and RHEL/Rocky/AlmaLinux (8.x/9.x) on `bare` instances.

IP addresses

You can specify the `public_ip`, `private_ip`, or both for any instance.

TPA uses these IP addresses in two ways: first, to ssh to the instance to execute commands during deployment; and second, to set up communications within the cluster, e.g., for `/etc/hosts` or to set `primary_conninfo` for Postgres.

If you specify a `public_ip`, it will be used to ssh to the instances during deployment. If you specify a `private_ip`, it will be used to set up communications within the cluster. If you specify both, the `public_ip` will be used during deployment, and the `private_ip` for cluster communications.

If you specify only one or the other, the address will be used for both purposes. For example, you could set only `public_ip` for servers on different networks, or only `private_ip` if you're running TPA inside a closed network. (Instead of using public/private, you can set `ip_address` if you need to specify only one IP address.)

Starting afresh

To start afresh with a cluster on the `bare` platform, use the appropriate external tools to reinstall, reimage, or reprovision the servers, and repeat the process described in this document. If your new servers have different IP addresses or if you have a complex ssh setup, it may be easier to run tpaexec deprovision to remove all the locally created files and then tpaexec provision to recreate them, followed by repeating the process from this document, as above.

20 Docker

TPA can create Docker containers and deploy a cluster to them. At present, it sets up containers to run systemd and other services as if they were ordinary VMs.

Deploying to docker containers is an easy way to test different cluster configurations. It is not meant for production use.

Synopsis

Just select the platform at configure-time:

```
tpaexec configure clustername --platform docker […]
tpaexec provision clustername
tpaexec deploy clustername
```

Operating system selection

Use the standard `--os Debian/Ubuntu/RedHat/SLES` configure option to select which distribution to use for the containers. TPA will build its own systemd-enabled images for this distribution. These images will be named with a `tpa/` prefix, e.g., `tpa/redhat:8`.

Use `--os-image some/image:name` to specify an existing systemd-enabled image instead. For example, thecentos/systemd image (based on CentOS 7) can be used in this way.

Installing Docker

We test TPA with the latest stable Docker-CE packages.

This documentation assumes that you have a working Docker installation, and are familiar with basic operations such as pulling images and creating containers.

Please consult the Docker documentation if you need help to install Docker and get started with it.

On MacOS X, you can install "Docker Desktop for Mac" and launch Docker from the application menu.

Cgroups

All currently-supported operating systems provide support cgroups version 2. You should always use cgroups version 2 on the host machine when deploying to Docker unless you are using a legacy operating system.

> **Important**
>
> Instructions for using legacy systems are provided purely for testing or to support migrating away from those platforms.

For some legacy platforms, in particular RHEL 7, you will need to configure your host to use version 1. You can switch to cgroups version 1 as follows.

On Debian-family Linux distributions:

```
echo 'GRUB_CMDLINE_LINUX=systemd.unified_cgroup_hierarchy=false' > \
  /etc/default/grub.d/cgroup.cfg
update-grub
reboot
```

On RedHat-family Linux distributions:

```
grubby --args=systemd.unified_cgroup_hierarchy=false --update-kernel=ALL
reboot
```

On MacOS:

1. Edit ~/Library/Group\ Containers/group.com.docker/settings.json and make the following replacement `"deprecatedCgroupv1": false` → `"deprecatedCgroupv1": true`
2. Restart Docker Desktop app

Permissions

TPA expects the user running it to have permission to access to the Docker daemon (typically by being a member of the `docker` group that owns `/var/run/docker.sock`). Run a command like this to check if you have access:

```
docker version --format '{{.Server.Version}}'
```

```
output
19.03.12
```

> **Warning**
>
> Giving a user the ability to speak to the Docker daemon lets them trivially gain root on the Docker host. Only trusted users should have access to the Docker daemon.

Docker container privileges

Privileged containers

By default TPA provisions Docker containers in unprivileged mode, with no added Linux capabilities flags. Such containers cannot manage host firewall rules, file systems, block devices, or most other tasks that require true root privileges on the host.

If you require your containers to run in privileged mode, set the `privileged` boolean variable for the instance(s) that need it, or globally in `instance_defaults`, e.g.:

```
  instance_defaults:
    privileged: true
```

> **Warning**
>
> Running containers in privileged mode allows the root user or any process that can gain root to load kernel modules, modify host firewall rules, escape the container namespace, or otherwise act much as the real host "root" user would. Do not run containers in priviliged mode unless you really need to.

See `man capabilities` for details on Linux capabilities flags.

`security_opts` and the `no-new-privileges` flag

tpaexec can start docker containers in a restricted mode where processes cannot increase their privileges. setuid binaries are restricted, etc. Enable this in tpaexec with the `instance_defaults` or per-container variable `docker_security_opts`:

```
instance_defaults:
  docker_security_opts:
    - no-new-
privileges
```

Other arguments to `docker run`'s `--security-opts` are also accepted, e.g. SELinux user and role.

Linux capabilities flags

tpaexec exposes Docker's control over Linux capabilities flags with the `docker_cap_add` list variable, which may be set per-container or in `instance_defaults`. See `man capabilities`, the `docker run` documentation and the documentation for the Ansible `docker_containers` module for details on capabilities flags.

Docker's `--cap-drop` is also supported via the `docker_cap_drop` list.

For example, to run a container as unprivileged, but give it the ability to modify the system clock, you might write:

```
instance_defaults:
  privileged: false
  docker_cap_add:
    -
sys_time
  docker_cap_drop:
    -
all
```

Docker storage configuration

The default Docker configuration on many hosts uses `lvm-loop` block storage and is not suitable for production deployments. Run `docker info` to check which storage driver you are using. If you are using the loopback scheme, you will see something like this:

```
output
Storage Driver: devicemapper
 …
 Data file: /dev/loop0
```

Consult the Docker documentation for more information on storage configuration:

- Storage Drivers
- Configuring lvm-direct for production

Docker MTU settings

By default, Docker networks have a Maximum Transmission Unit (MTU) of 1500 bytes. If this is greater than the MTU of your host system's network interface you may experience problems routing connections through that interface to Docker containers. You can check the MTU of your network interfaces using the command `ipconfig | grep mtu`, `ip |grep mtu` or similar. You can change the MTU of a Docker network provisioned by TPA by adding the appropriate driver options to the network in `config.yml` as shown below.

```
docker_networks:
- ipam_config:
  - subnet:
10.33.214.192/28
  name: tpa-docker
  driver_options:
    com.docker.network.driver.mtu: 1400
```

> **Warning**
>
> The MTU can only be set when the Docker network is first provisioned. Subsequent changes in config.yml will have no effect.

You can verify the MTU of a Docker network by running one of the commands above from inside a container attached to that network. You can also use `docker network inspect <network-name> | grep mtu` but this only works when the MTU has been explicitly set.

Docker container management

All of the docker containers in a cluster can be started and stopped together using the `start-containers` and `stop-containers` commands:

```
tpaexec start-containers clustername
tpaexec stop-containers clustername
```

These commands don't provision or deprovision containers, or even connect to them; they are intended to save resources when you're temporarily not using a docker cluster that you need to keep available for future use.

For a summary of the provisioned docker containers in a cluster, whether started or stopped, use the `list-containers` command:

```
tpaexec list-containers clustername
```

21 Cluster configuration

With TPA, the way to make any configuration change to a cluster is to edit config.yml and run the provision/deploy/test cycle. The process is carefully designed to be idempotent, and to make changes only in response to a change in the configuration or a change on the instances.

The `tpaexec configure` command will generate a sensible config.yml file for you, but it covers only the most common topology and configuration options. If you need something beyond the defaults, or you need to make changes after provisioning the cluster, you will need to edit config.yml anyway.

This page is an overview of the configuration mechanisms available. There's a separate page with more details about the specific variables you can set to customise the deployment process.

config.yml

Your `config.yml` file is a YAML format text file that represents all aspects of your desired cluster configuration. Here's a minimal example of a cluster with two instances:

```
cluster_name:
speedy

cluster_vars:
  postgres_version: 14

instances:
- node: 1
  Name:
one
  role: primary
  platform:
docker
  vars:
    ansible_user: root
    x: 42

- node: 2
  Name:
two
  role: replica
  platform:
docker
  upstream:
one
  vars:
    ansible_user: root
    x: 53
```

These three definitions are central to your cluster configuration. The file may contain many other definitions (including platform-specific details), but the list of `instances` with `vars` set either for one instance or for the whole cluster are the basic building blocks of every TPA configuration.

All `tpaexec configure` options translate to config.yml variables in some way. A single option may affect several variables (e.g., `--bdr-version` could set `postgres_version`, `edb_repositories`, `extra_postgres_extensions`, and so on), but you can always accomplish with an editor what you could by running the command.

In terms of YAML syntax, config.yml as a whole represents a hash with keys such as `cluster_vars` and `instances`. **You must ensure that each key is defined only once.** If you were to inadvertently repeat the `cluster_vars`, say, the second definition would completely override the former, and your next deployment could make unintended changes because of missing (shadowed) variables.

TPA checks the consistency of the overall cluster topology (for example, if you declare an instance with the role "replica", you must also declare the name of its upstream instance, and that instance must exist), but it will not prevent you from setting any variable you like on the instances. You must exercise due caution, and try out changes in a test environment before rolling them out into production.

Variables

In Ansible terminology, most configuration settings are “inventory variables”—TPA will translate `cluster_vars` into `group_vars` (that apply to the cluster as a whole) and each instance's `vars` into `host_vars` in the inventory during provisioning, and deployment will use the inventory values. After you change config.yml, **you must remember to run** `tpaexec provision` **before** `tpaexec deploy`.

Any variable can be set for the entire cluster, or an individual host, or both; host variables override group variables. In practice, setting `x: 42` in `cluster_vars` is no different from setting it in every host's `vars`. A host that needs `x` during deployment will see the value 42 either way. A host will always see the most specific value, so it is convenient to set some default value for the group and override it for specific instances as required.

Whenever possible, defining variables in `cluster_vars` and overriding them for specific instances results in a concise configuration that is easier to review and change (less repetition). Beyond that, it's up to you to decide whether any given setting makes more sense as a group or host variable.

Cluster variables

The keys under `cluster_vars` may map to any valid YAML type, and will be translated directly into group variables in the Ansible inventory:

```
cluster_vars:
  postgres_version: 14
  edb_repositories:
  - enterprise
  - postgres_distributed
  postgres_conf_settings:
    bdr.trace_replay: true
```

> **Warning**
>
> Any variables used in templates must be defined at the top-level of `config.yml` (the same level as the `cluster_name` variable)
>
> Please refer to the following example below.

```
cluster_name: 'speedy'
postgres_version: '14' # Defined at top-
level
cluster_vars:
  postgres_version: "{{ postgres version }}" # Templated with top-level variable
  postgres_data_dir: "/data/{{ cluster_name }}/edb{{ postgres_version }}/data" # Templated with top-level
variable
  privilege_escalation_command: sudo
```

In this case, `tpaexec provision` will write three variables (a string, a list, and a hash) to the inventory in `group_vars/tag_Cluster_name/01-cluster_name.yml`.

Instance variables

This documentation uses the term “instance variables” to refer to any variables that are defined for a specific instance in config.yml. For example, here's a typical instance definition:

```
instances:
- Name:
unwind
  node: 1
  backup: unkempt
  location:
a
  role:
  - primary
  -
bdr
  volumes:
  - device_name: root
    encrypted: true
    volume_size: 16
    volume_type:
gp2
  - device_name: /dev/xvdf
    encrypted: true
    vars:
      volume_for: postgres_data
    volume_size: 64
    volume_type:
gp2
  platform:
aws
  type:
t3.micro
  vars:
    ansible_user: ec2-user
    postgres_conf_directory: /opt/postgres/conf
```

The variables defined in this instance's `vars` will all become host variables in the inventory, but all host vars in the inventory do not come from `vars` alone. Some other instance settings, including `platform`, `location`, `volumes`, and `role` are also copied to the inventory as host vars (but you cannot define these settings under `vars` or `cluster_vars` instead).

The settings outside `vars` may describe the properties of the instance (e.g., `Name` and `node`) or its place in the topology of the cluster (e.g., `role`, `backup`) or they may be platform-specific attributes (e.g., instance `type` and `volumes`). Other than knowing that they cannot be defined under `vars`, it is rarely necessary to distinguish between these instance “settings” and instance “variables”.

In this case, `tpaexec provision` will write a number of host variables to the inventory in `host_vars/unwind/01-instance_vars.yml`.

instance_defaults

This is a mechanism to further reduce repetition in config.yml. It is most useful for instance settings that cannot be defined as `cluster_vars`. For example, you could write the following:

```
instance_defaults:
  platform:
aws
  type:
t3.micro
  tags:
    AWS_ENVIRONMENT_SPECIFIC_TAG_KEY:
some_mandated_value

instances:
- node: 1
  Name:
one
- node: 2
  Name:
two
-
…
```

Whatever you specify under `instance_defaults` serves as the default for every entry in `instances`. In this example, it saves spelling out the `platform` and `type` of each instance, and makes it easier to change all your instances to a different type. If any instance specifies a different value, it will of course take precedence over the default.

It may help to think of `instance_defaults` as being a macro facility to use in defining `instances`. What is ultimately written to the inventory comes from the (expanded) definition of `instances` alone. If you're trying to decide whether to put something in `cluster_vars` or `instance_defaults`, it probably belongs in the former unless it *cannot* be defined as a variable (e.g., `platform` or `type`), which is true for many platform-specific properties (such as AWS resource tags) that are used only in provisioning, and not during deployment.

The `instance_defaults` mechanism does nothing to stop you from using it to fill in the `vars` for an instance (default hash values are merged with any hash specified in the `instances` entry). However, there is no particular advantage to doing this rather than setting the same default in `cluster_vars` and overriding it for an instance if necessary. When in doubt, use `cluster_vars`.

Locations

You can also specify a list of `locations` in config.yml:

```
locations:
- Name: first
  az: eu-west-
1a
  region: eu-west-
1
  subnet:
10.33.110.128/28

- Name:
second
  az: us-east-
1b
  region: us-east-
1
  subnet: 10.33.75.0/24

instances:
- node: 1
  Name:
one
  location: first
…
```

If an instance specifies `location: first` (or `location: 0`), the settings under that location serve as defaults for that instance. Again, just like `instance_defaults`, an instance may override the defaults that it inherits from its location. And again, you can use this feature to fill in `vars` for an instance. This can be useful if you have some defaults that apply to only half your instances, and different values for the other half (as with the platform-specific settings in the example above).

Locations represent a collection of settings that instances can “opt in” to. You can use them to stand for different data centres, AWS regions, Docker hosts, or something else entirely. TPA does not expect or enforce any particular interpretation.

22 Instance configuration

This page presents an overview of the various controls that TPA offers to customise the deployment process on cluster instances, with links to more detailed documentation.

Before you dive into the details of deployment, it may be helpful to readan overview of configuring a cluster to understand how cluster and instance variables and the other mechanisms in config.yml work together to allow you to write a concise, easy-to-review configuration.

System-level configuration

The first thing TPA does is to ensure that Python is bootstrapped and ready to execute Ansible modules (a distribution-specific process). Then it completes various system-level configuration tasks before moving on to Postgres configuration below.

- Distribution support
- Python environment (`preferred_python_version`)
- Environment variables (e.g., `https_proxy`)

Package repositories

You can use the pre-deploy hook to execute tasks before any package repositories are configured.

- Configure YUM repositories (for RHEL, Rocky and AlmaLinux)
- Configure APT repositories (for Debian and Ubuntu)
- Configure EDB repositories (on any system)
- Configure a local package repository (to ship packages to target instances)

You can use the post-repo hook to execute tasks after package repositories have been configured (e.g., to correct a problem with the repository configuration before installing any packages).

Package installation

Once the repositories are configured, packages are installed at various stages throughout the deployment, beginning with a batch of system packages:

- Install non-Postgres packages (e.g., acl, openssl, sysstat)

Postgres and other components (e.g., Barman, repmgr, pgbouncer) will be installed separately according to the cluster configuration; these are documented in their own sections below.

Other system-level tasks

- Create and mount filesystems (including RAID, LUKS setup)
- Upload artifacts (files, directories, tar archives)
- Set sysctl values
- Configure /etc/hosts
- Manage ssh_known_hosts entries
- Add system locale

Skipping deployment completely

To prevent TPA from doing any part of the deployment process on an instance - in other words, if you want TPA to provision the instance and then leave it alone - set the `provision_only` setting for the instance to `true` in `config.yml`. This setting will make TPA omit the instance entirely from the inventory which `tpaexec deploy` sees.

Postgres

Postgres configuration is an extended process that goes hand-in-hand with setting up other components like repmgr and pgbouncer. It begins with installing Postgres itself.

Version selection

Use the configure options to select a Postgres flavour and version, or set `postgres_version` in config.yml to specify which Postgres major version you want to install.

That's all you really need to do to set up a working cluster. Everything else on this page is optional. You can control every aspect of the deployment if you want to, but the defaults are carefully tuned to give you a sensible cluster as a starting point.

Installation

The default `postgres_installation_method` is to install packages for the version of Postgres you selected, along with various extensions, according to the architecture's needs:

- Install Postgres and Postgres-related packages (e.g., pglogical, BDR, etc.)
- Build and install Postgres and extensions from source (for development and testing)

Whichever installation method you choose, TPA can give you the same cluster configuration with a minimum of effort.

Configuration

- Configure the postgres Unix user
- Run initdb to create the PGDATA directory
- Configure pg_hba.conf
- Configure pg_ident.conf
- Configure postgresql.conf

You can use the postgres-config hook to execute tasks after the Postgres configuration files have been installed (e.g., to install additional configuration files).

Once the Postgres configuration is in place, TPA will go on to install and configure other components such as Barman, repmgr, pgbouncer, and haproxy, according to the details of the architecture.

Other components

- Configure Barman
- Configure pgbouncer
- Configure etcd
- Configure Patroni
- Configure haproxy
- Configure HARP
- Configure EFM

Configuring and starting services

TPA will now install systemd service unit files for each service. The service for Postgres is named `postgres.service`, and can be started or stopped with `systemctl start postgres`.

In the first deployment, the Postgres service will now be started. If you are running `tpaexec deploy` again, the service may be reloaded or restarted depending on what configuration changes you may have made. Of course, if the service is already running and there are no changes, then it's left alone.

In any case, Postgres will be running at the end of this step.

After starting Postgres

- Create Postgres users

- Create Postgres tablespaces

- Create Postgres databases

- Configure pglogical replication

- Configure .pgpass

You can use the postgres-config-final hook to execute tasks after the post-startup Postgres configuration has been completed (e.g., to perform SQL queries to create objects or load data).

- Configure BDR

You can use the post-deploy hook to execute tasks after the deployment process has completed.

23 Building from source

> **Warning**
>
> This option is intended for developers and advanced users. Only software built and tested by EDB is supported by EDB. Please refer to Self-Managed Supported Open Source Software.

TPA can build Postgres and other required components from source and deploy a cluster with exactly the same configuration as with the default packaged installation. This makes it possible to deploy repeatedly from source to quickly test changes in a realistic, fully-configured cluster that reproduces every aspect of a particular setup, regardless of architecture or platform.

You can even combine packaged installations of certain components with source builds of others. For example, you can install Postgres from packages and compile pglogical and PGD from source, but package dependencies would prevent installing pglogical from source and PGD from packages.

Source builds are meant for use in development, testing, and for support operations.

Quickstart

Spin up a cluster with 2ndQPostgres, pglogical3, and bdr all built from stable branches:

```
tpaexec configure ~/clusters/speedy -a BDR-Always-ON          \
    --layout bronze                                             \
    --harp-consensus-protocol etcd                              \
    --install-from-source                                       \
      2ndqpostgres:2QREL_13_STABLE_dev                          \
      pglogical3:REL3_7_STABLE                                  \
      bdr3:REL3_7_STABLE
```

As above, but set up a cluster that builds 2ndQPostgres source code from the official git repository and uses the given local work trees to build pglogical and BDR. This feature is specific to Docker:

```
tpaexec configure ~/clusters/speedy                           \
    --architecture BDR-Always-ON --layout bronze                \
    --harp-consensus-protocol etcd                              \
    --platform docker                                           \
    --install-from-source 2ndqpostgres:2QREL_13_STABLE_dev      \
      pglogical3 bdr3                                           \
    --local-source-directories                                  \
      pglogical3:~/src/pglogical bdr3:~/src/bdr
```

After deploying your cluster, you can use `tpaexec deploy … --skip-tags build-clean` on subsequent runs to reuse build directories. (Otherwise the build directory is emptied before starting the build.)

Read on for a detailed explanation of how to build Postgres, pglogical, BDR, and other components with custom locations and build parameters.

Configuration

There are two aspects to configuring source builds.

If you just want a cluster running a particular combination of sources, run `tpaexec configure` to generate a configuration with sensible defaults to download, compile, and install the components you select. You can build Postgres or Postgres Extended, pglogical, and BDR, and specify branch names to build from, as shown in the examples above.

The underlying mechanism is capable of much more than the command-line options allow. By editing config.yml, you can clone different source repositories, change the build location, specify different configure or build parameters, redefine the build commands entirely, and so on. You can, therefore, build things other than Postgres, pglogical, and BDR.

The available options are documented here:

- Building Postgres from source
- Building extensions with `install_from_source`

Local source directories

You can use TPA to provision Docker containers that build Postgres and/or extensions from your local source directories instead of from a Git repository.

Suppose you're using `--install-from-source` to declare what you want to build:

```
tpaexec configure ~/clusters/speedy                        \
    --architecture BDR-Always-ON --layout bronze             \
    --harp-consensus-protocol etcd                           \
    --platform docker                                        \
    --install-from-source 2ndqpostgres:2QREL_13_STABLE_dev   \
      pglogical3:REL3_7_STABLE bdr3:REL3_7_STABLE            \
      …
```

By default, this will clone the known repositories for Postgres Extended, pglogical3, and bdr3, check out the given branches, and build them. But you can add `--local-source-directories` to specify that you want the sources to be taken directly from your host machine instead:

```
tpaexec configure ~/clusters/speedy                        \
    --architecture BDR-Always-ON --layout bronze             \
    --harp-consensus-protocol etcd                           \
    --platform docker                                        \
    --install-from-source 2ndqpostgres:2QREL_13_STABLE_dev   \
      pglogical3 bdr3                                        \
    --local-source-directories                               \
      pglogical3:~/src/pglogical bdr3:~/src/bdr              \
    …
```

This configuration will still install Postgres Extended from the repository, but it obtains pglogical3 and bdr3 sources from the given directories on the host. These directories are bind-mounted read-only into the Docker containers at the same locations where the git repository would have been cloned to, and the default (out-of-tree) build proceeds as usual.

If you specify a local source directory for a component, you cannot specify a branch to build (cf. `pglogical3:REL3_7_STABLE` vs. `pglogical3` for `--install-from-source` in the examples above). The source directory is mounted read-only in the containers, so TPA cannot do anything to change it—neither `git pull`, nor `git checkout`. You get whichever branch you have checked out locally, uncommitted changes and all.

Using `--local-source-directories` includes a list of Docker volume definitions in config.yml:

```
local_source_directories:
  - /home/ams/src/pglogical:/opt/postgres/src/pglogical:ro
  -
/home/ams/src/bdr:/opt/postgres/src/bdr:ro
  - ccache-bdr_src_36-20200828200021:/root/.ccache:rw
```

ccache

TPA installs ccache by default for source builds of all kinds. When you are using a Docker cluster with local source directories, by default a new Docker volume is attached to the cluster's containers to serve as a shared ccache directory. This volume is completely isolated from the host, and is removed when the cluster is deprovisioned.

Use the `--shared-ccache /path/to/host/ccache` configure option to specify a longer-lived shared ccache directory. This directory will be bind-mounted r/w into the containers, and its contents will be shared between the host and the containers.

(By design, there is no way to install binaries compiled on the host directly into the containers.)

Rebuilding

After deploying a cluster with components built from source, you can rebuild those components quickly without having to rerun `tpaexec deploy` by using the `tpaexec rebuild-sources` command. This will run `git pull` for any components built from git repositories on the containers, and rebuild all components.

24 TPA hooks

TPA can set up fully-functional clusters with no user intervention, and already provides a broad variety of settings to control your cluster configuration, including custom repositories and packages, custom Postgres configuration (both pg_hba.conf and postgresql.conf), and so on.

You can write hook scripts to address specific needs that are not met by the available configuration settings. Hooks allow you to execute arbitrary Ansible tasks during the deployment.

Hooks are the ultimate extension mechanism for TPA, and there is no limit to what you can do with them. Please use them with caution, and keep in mind the additional maintenance burden you are taking on. The TPA developers have no insight into your hook code, and cannot guarantee compatibility between releases beyond invoking hooks at the expected stage.

Summary

If you create files with specific names under the `hooks` subdirectory of your cluster directory, TPA will invoke them at various stages of the deployment process, as described below.

```
mkdir ~/clusters/speedy/hooks
cat > ~/clusters/speedy/hooks/pre-deploy.yml
```

```
output
---
- debug: msg="hello world!"
```

Hook scripts are invoked with `include_tasks`, so they are expected to be YAML files containing a list of Ansible tasks (not a playbook, which contains a list of plays). Unless otherwise documented below, hooks are unconditionally executed for all hosts in the deployment.

General-purpose hooks

pre-deploy

TPA invokes `hooks/pre-deploy.yml` after bootstrapping Python and configuring locale, but before configuring repositories and installing packages. This is the earliest stage at which you can execute your own code.

You can use this hook to set up custom repository configuration, beyond what you can do with `apt_repositories` or `yum_repositories`.

post-repo

TPA invokes `hooks/post-repo.yml` after configuring package repositories. You can use it to make corrections to the repository configuration before beginning to install packages.

pre-initdb

TPA invokes `hooks/pre-initdb.yml` before deciding whether or not to run initdb to create PGDATA if it does not exist. You should not ordinarily need to use this hook (but if you use it to create `PGDATA` yourself, then TPA will skip `initdb`).

postgres-config

TPA invokes `hooks/postgres-config.yml` after generating Postgres configuration files, including pg_hba.conf and the files in conf.d, but before the server has been started.

You can use this hook, for example, to create additional configuration files under `conf.d`.

postgres-config-final

TPA invokes `hooks/postgres-config-final.yml` after starting Postgres and creating users, databases, and extensions. You can use this hook to execute SQL commands, for example, to perform custom extension configuration or create database objects.

barman-pre-config

TPA invokes `hooks/barman-pre-config.yml` after installing Barman and setting up Barman users, but before generating any Barman configuration.

You can use this hook, for example, to perform any tasks related with Barman certificate files or mount points.

efm-pre-config

TPA invokes `hooks/efm-pre-config.yml` after installing efm, creating its configuration directory, and setting up the efm user, but before generating any efm configuration.

An example use of this hook is to install efm helper scripts.

efm-post-config

TPA invokes `hooks/efm-post-config.yml` after finishing the efm configuration step.

An example use of this hook is for additional steps that need to be taken into account before exiting the efm configuration process.

harp-config

TPA invokes `hooks/harp-config.yml` after generating HARP configuration files, but before the HARP service has been started.

You can use this hook, for example, to perform any customizations to the HARP proxy that are not provided by the built-in interface of TPA.

Please note that this hook will be run in any node that installs HARP packages, including PGD nodes.

pgd-proxy-config

TPA invokes `hooks/pgd-proxy-config.yml` after generating PGD PROXY configuration files, but before the PGD PROXY service has been started.

You can use this hook, for example, to perform any customizations to the PGD Proxy that are not provided by the built-in interface of TPA.

Note that this hook will run in any node that installs PGD PROXY packages.

post-deploy

TPA invokes `hooks/post-deploy.yml` at the end of the deployment.

You can go on to do whatever you want after this stage.

If you use this hook to make changes to any configuration files that were generated or altered during the TPA deployment, you run the risk that the next `tpaexec deploy` will overwrite your changes (since TPA doesn't know what your hook might have done).

PGD hooks

These hooks are specific to PGD deployments.

bdr-pre-node-creation

TPA invokes `hooks/bdr-pre-node-creation.yml` on all instances before creating a PGD node on any instance for the first time. The hook will not be invoked if all required PGD nodes already exist.

bdr-post-group-creation

TPA invokes `hooks/bdr-post-group-creation.yml` on all instances after creating any PGD node group on the `first_bdr_primary` instance. The hook will not be invoked if the required PGD groups already exist.

bdr-pre-group-join

TPA invokes `hooks/bdr-pre-group-join.yml` on all instances after creating, changing or removing the replication sets and configuring the required subscriptions, before the node join.

You can use this hook to execute SQL commands and perform other adjustments to the replication set configuration and subscriptions that might be required before the node join starts.

For example, you can adjust the PGD witness replication set to automatically add new tables and create DDL filters in general.

Other hooks

postgres-pre-update, postgres-post-update

The `upgrade` command invokes `hooks/postgres-pre-update.yml` on a particular instance before it installs any packages, and invokes `hooks/postgres-post-update.yml` after the package installation is complete. Both hooks are invoked only on the instance being updated.

You can use these hooks to customise the update process for your environment (e.g., to install other packages and stop and restart services that TPA does not manage).

New hooks

EDB adds new hooks to TPA as the need arises. If your use case is not covered by the existing hooks, please contact us to discuss the matter.

25 Upgrading your cluster

The `tpaexec upgrade` command is used to upgrade the software running on your TPA cluster (`tpaexec deploy` will not perform upgrades).

(This command replaces the earlier `tpaexec update-postgres` command.)

> **Note**
>
> TPA does not yet support using the `tpaexec upgrade` command for clusters that have shared Barman and/or shared PEM configurations, and will add this functionality in a future release.

Introduction

If you make any changes to config.yml, the way to apply those changes is to run `tpaexec provision` followed by `tpaexec deploy` .

The exception to this rule is that `tpaexec deploy` will refuse to install a different version of a package that is already installed. Instead, you must use `tpaexec upgrade` to perform software upgrades.

The following components are able to be upgraded for **any** architecture:

- Postgres
- PgBouncer
- Barman
- PG Backup API
- PEM (both server and agent)

The following components are able to be upgraded on **M1** architectures and depend on the failover manager used:

- EFM
- Patroni
- etcd (for Patroni)
- repmgr

The following components are able to be upgraded on **BDR-Always-ON/PGD-Always-ON** architectures, depending on the BDR version used:

- pgdcli (v1 for BDR 4, v5 for PGD5)
- pgd-proxy (only for PGD5)

> **Minor version upgrades only**
>
> `tpaexec upgrade` **does NOT support MAJOR version upgrades of Postgres and most cluster components** What TPA can upgrade is dependent on architecture:
>
> - The M1 architecture and all applicable failover managers for M1, `upgrade` can perform minor version upgrades of `Postgres` , the corresponding failover manager (`EFM` , `Patroni` or `repmgr`) and any non-architecture specific components that are selected.
> - With PGD architectures, `upgrade` will perform minor version upgrades of Postgres and the BDR extension as well as `pgd-cli` and `pgd-proxy` if they are explicitly opted-in.
> - With PGD architectures, and only in combination with the `reconfigure` command, `upgrade` can perform major-version upgrades of the BDR extension.
>
> Support for upgrading other cluster components was added in TPA `v23.41.0` *Certain components (such as `EFM`) are an exception. Consult the `upgrade` section of each component's documentation for further information.*

This command will try to perform the upgrade with minimal disruption to cluster operations. The exact details of the specialised upgrade process depend on the architecture of the cluster, as documented below.

When upgrading, you should always use barman to take a backup before beginning the upgrade and disable any scheduled backups which would take place during the time set aside for the upgrade.

In general, TPA will proceed instance-by-instance, stopping any affected services, installing new packages, updating the configuration if needed, restarting services, and performing any runtime configuration changes, before moving on to do the same thing on the next instance. At any time during the process, only one of the cluster's nodes will be unavailable.

When upgrading a cluster to PGD-Always-ON, upgrading an existing PGD-Always-ON cluster, or performing a minor upgrade of a PGD-S or PGD-X cluster, you can enable monitoring of the status of your proxy nodes during the upgrade by adding the option `-e enable_proxy_monitoring=true` to your `tpaexec upgrade` command line. If enabled, this will create an extra table in the bdr database and write monitoring data to it while the upgrade takes place. The performance impact of enabling monitoring is very small and it is recommended that it is enabled.

Before you upgrade

This section describes the checks you should perform on your cluster before running `tpaexec upgrade`. Working through these in order will catch most common problems before they affect a live cluster.

Verify cluster health

The upgrade is easiest to perform on a cluster that is in a known-good state. Run `tpaexec test <cluster>` to exercise the standard health checks. The upgrade process itself runs pre-upgrade health checks, but discovering a problem ahead of time gives you longer to address it.

For any BDR enabled architecture, also confirm that all nodes appear in its catalogue:

```
SELECT node_name, peer_state_name,
peer_target_state_name
  FROM bdr.node_summary;
```

Every entry should report `ACTIVE` (or `STANDBY` for read-only standbys). Investigate any other state before proceeding.

For M1 clusters using repmgr, run `repmgr cluster show` on the primary and confirm every node is shown as running and connected.

Verify backups

Before any upgrade, confirm that a recent Barman backup exists and is in a state from which you could restore. On the Barman host:

```
sudo -u barman barman list-backup <server>
sudo -u barman barman check <server>
```

Any failed status here should be investigated before starting. Once the upgrade is underway, the Barman WAL receiver is stopped, so the backup landscape during the upgrade window is fixed at whatever existed beforehand.

Verify replication lag

Whilst the cluster may be healthy at a high level, individual replicas may be lagging. The upgrade fences each instance in turn; if a replica is significantly behind, it will take longer to catch up after being unfenced, and the overall upgrade duration will grow.

For PGD clusters:


```
SELECT slot_name, active, restart_lsn, write_lag, flush_lag, replay_lag
  FROM bdr.node_slots;
```

For M1 clusters, on the primary:

```
SELECT application_name, state, write_lag, flush_lag,
replay_lag
  FROM pg_stat_replication;
```

Aim for lag measured in seconds, not minutes. Investigate any replica that is seriously behind before starting.

Pin and review package versions

For control over what is installed during the upgrade, it is recommended that you explicitly set package versions in `config.yml` before running `tpaexec upgrade`. The relevant variables are `postgres_package_version`, `bdr_package_version`, `pgd_proxy_package_version`, `pgdcli_package_version`, and the component-specific equivalents.

Without an explicit pin, TPA installs the latest available package for each component, which can result in an unintended major-version upgrade. See Package version selection for details.

> **Note**
>
> After updating versions in `config.yml`, run `tpaexec provision` to regenerate the inventory.

Read the release notes

Check the release notes for the target versions of every component you are upgrading but also the ones between current and target versions. EDB's PGD and Postgres release notes occasionally call out version-specific upgrade considerations: settings that must be changed before or after the upgrade, deprecations, and behaviour changes. The same applies to TPA's own release notes for any newer version of TPA you may be using to perform the upgrade.

Test the upgrade in a staging environment

`tpaexec upgrade` processes the cluster instance-by-instance. The upgrade is rolling: at any one time only a single instance is fenced off. However, the total wall-clock time depends on the cluster size, the number of components being upgraded, and the storage and network characteristics of each instance but also that the upgrade scenario goes through without issues.

We strongly recommend reproducing the upgrade in a non-production environment that matches your production cluster's architecture, Postgres flavour and version, package pinnings, and any custom hooks. This catches surprises (missing packages in your configured repository, unexpected configuration drift) before they affect live traffic.

Disable scheduled jobs

Anything scheduled to run against the cluster during the upgrade window should be paused. Typical items:

- Cron jobs that hit the database (analytics, reporting).
- External backup tools that are not aware of the upgrade.
- Long-running migrations or batch jobs.

TPA itself stops the Barman WAL receiver before starting and restarts it afterwards; you do not need to disable Barman manually.

During the upgrade

This section describes what to expect whilst `tpaexec upgrade` is running and what to watch for from outside TPA.

What to expect

TPA processes the cluster one instance at a time. For each instance, in turn, the upgrade:

1. Fences the instance off so the proxy or failover manager stops sending it new connections.
2. Stops the affected services on that instance.
3. Installs the new packages.
4. Updates configuration files where required.
5. Restarts services and waits for them to come up.
6. Unfences the instance.

You should expect brief connection interruptions whilst each instance is fenced and again when it returns to service. Applications using connection pooling and retry logic should typically not see client-visible errors during these transitions.

Whilst each non-leader node is being upgraded, write traffic continues through the current write leader. Each time the write leader itself is upgraded, a brief leader-election interruption occurs. PGD-S and PGD-X arrange for the write leader to be upgraded last; PGD-Always-ON upgrades nodes in inventory order, so a leader election occurs whenever the current leader's turn comes round.

What to watch from the application side

Whilst the upgrade is running, a brief pause at each fence/unfence transition is normal. Sustained connection failures across multiple instances are not — if you see those, stop and investigate before TPA proceeds to the next instance.

If an application is running against the cluster during the upgrade, monitor:

- Application error rates: transient retries are expected; sustained errors are not.
- Connection pool metrics: brief drops in available connections are expected.
- Replication slot lag: may grow transiently as each instance catches up post-upgrade.

After the upgrade

This section describes the verification you should perform once `tpaexec upgrade` has completed.

Verify installed package versions

Confirm that the new packages are installed on every instance. On RHEL-family hosts:

```
tpaexec cmd <cluster> all -m shell -a 'rpm -qa | grep -E "^(postgres|edb-)"'
```

On Debian / Ubuntu hosts:

```
tpaexec cmd <cluster> all -m shell -a 'dpkg -l | grep -E "(postgres|edb-)"'
```

The same package versions should be reported on every instance of the same role.

Verify services and replication

Confirm that all services restarted cleanly:

```
tpaexec cmd <cluster> all -m shell -a 'systemctl is-active postgres'
```

For PGD clusters, confirm that BDR has fully reconverged:

```
SELECT node_name, peer_state_name FROM bdr.node_summary;
```

For M1 clusters with repmgr:

```
tpaexec cmd <cluster> role_primary -m shell -a 'repmgr -f /etc/repmgr.conf cluster show'
```

Take a fresh backup

After any non-trivial upgrade, take a fresh Barman backup of the cluster:

```
tpaexec cmd <cluster> role_barman -m shell -a 'sudo -u barman barman backup <server>'
```

This gives you a known-good restore point that reflects the post-upgrade state.

Re-enable scheduled jobs

If you disabled cron jobs, batch jobs, or external scheduled tasks before the upgrade, re-enable them now. TPA's own Barman WAL receiver is restarted automatically.

Run tpaexec test

As a final smoke check, run the standard test suite:

```
tpaexec test <cluster>
```

This exercises connectivity, replication, role mapping, and the component-specific checks that ran before the upgrade. A clean test run is the simplest end-to-end confirmation that the upgrade landed.

Recovering from a failed upgrade

If `tpaexec upgrade` reports a failure, the cluster is left in a known state at whichever instance was being processed when the failure occurred. Subsequent instances are not touched.

When to roll back, when to call EDB Support

A roll-back is only safe before TPA has applied any non-reversible catalogue change — typically the BDR-side configuration updates that occur in major-version upgrades. Once those changes are in place, the correct path is forward, not backward.

If you are unsure whether to proceed or roll back, stop and contact EDB Support with:

- The `ansible.log` from the failed run.
- The output of `tpaexec test <cluster>` .
- A description of which instances completed the upgrade and which did not.
- The config.yml used.
- Optionally the EDB lasso report or any other meaningful logs.

Common pitfalls

This section consolidates a few specific situations that have caught users out in the past.

Pinned package versions cause no-op upgrades

If `postgres_package_version` (or any other `xxx_package_version`) already matches the installed package, `tpaexec upgrade` reports the components as "already at the desired version" and does nothing. The symptom is a run that completes quickly with no changes, and the desired version not installed.

Fix: update `config.yml` to set the new desired version, then re-run `tpaexec provision` followed by `tpaexec upgrade` . See Package version selection for details.

Shared Barman or shared PEM clusters

TPA does not currently support running `tpaexec upgrade` against clusters with shared Barman or shared PEM configurations. The restriction is enforced by the upgrade's preconditions, which abort early. Until support is added, upgrades for these cluster shapes need to be performed manually outside TPA — see After a manual major-version Postgres upgrade for how to reconcile the cluster afterwards.

Component selection

> **Note**
>
> Upgrading components is strictly opt-in

By default, `tpaexec upgrade` will update Postgres alone if the `--components` flag is not passed

```
tpaexec upgrade ~/clusters/speedy
```

To select specific components to update, the `--components` flag takes a comma-separated list

```
tpaexec upgrade ~/clusters/speedy \
   --components=postgres,pgd-proxy,pgdcli,pgbouncer,pg-backup-api,barman
```

If all applicable components in the cluster should be updated, `all` can be passed to the flag

```
tpaexec upgrade --components=all
```

component	value	architecture
Barman	barman	all
PEM	pem-server,pem-agent	all
PgBackupAPI	pg-backup-api	all

component	value	architecture
PgBouncer	pgbouncer	all
EFM	efm	M1 with `failover_manager=efm`
etcd	etcd	M1 with `failover_manager=patroni`
Patroni	patroni	M1 with `failover_manager=patroni`
RepMgr	repmgr	M1 with `failover_manager=repmgr`
PGD Cli	pgdcli	BDR-Always-ON, PGD-Always-ON
PGD-Proxy	pgd-proxy	PGD-Always-ON
Postgres,EPAS,PGE	postgres	All
All	all	All

Package version selection

By default, `tpaexec upgrade` will update to the latest available versions of the installed packages if you did not explicitly specify any package versions (e.g., Postgres, PGD, or pglogical) when you created the cluster.

> **Minor upgrade is not strictly enforced**
>
> If a desired package version is NOT provided when upgrading, TPA will install the latest available package. The minor version restriction is NOT strictly enforced during tpaexec upgrade. This can result in unwillingly attempting an unsupported major upgrade of a component. Thus, it is recommended to explicitly select versions for upgrade to ensure compatibility in the existing cluster. Postgres does not pose this issue since major versions are different packages altogether which stops this from happening.

If you did select specific versions, for example by using any of the `--xxx-package-version` options (e.g., postgres, bdr, pglogical) to `tpaexec configure`, or by defining `xxx_package_version` variables in config.yml, the upgrade will do nothing because the installed packages already satisfies the requested versions.

In this case, you must edit config.yml, update the version settings, and re-run `tpaexec provision`. The update will then install the selected version of the packages. You can also update to a specific version by specifying versions on the command line as shown below:

```
tpaexec upgrade ~/clusters/speedy -vv       \
  --components=postgres,pgbouncer           \
  -e postgres_package_version="16.10*"      \
  -e pgbouncer_package_version="1.24*"      \
  -e bdr_package_version="5.9.0*"
```

Please note that version syntax here depends on your OS distribution and package manager. In particular, yum accepts `*xyz*` wildcards, while apt only understands `xyz*` (as in the example above).

> **: see limitations of using wildcards in package_version in**
>
> tpaexec-configure.

It is your responsibility to ensure that the combination of Postgres, PGD, and pglogical package versions that you request are sensible. That is, they should work together, and there should be an upgrade path from what you have installed to the new versions.

For PGD clusters, it is a good idea to explicitly specify exact versions for all three components (Postgres, PGD, pglogical) rather than rely on the package manager's dependency resolution to select the correct dependencies.

We strongly recommend testing the upgrade in a QA environment before running it in production.

Configuration

In certain cases, minor-version upgrades do not need changes to config.yml. If no `postgres_package_version` is defined in `config.yml`, when `tpaexec upgrade` is run, it will upgrade Postgres to the latest available minor-version in a graceful way (what exactly that means depends on the details of the cluster).

For control over minor-version upgrades of other components, it is recommended to ensure a specific `xxx_package_version` is specified in `config.yml` before running `tpaexec upgrade` and explicitly opting-in to upgrade specific components using the `--components=x,y,z` flag (or `--components=all` to upgrade all, as applicable to the cluster). Running `tpaexec upgrade` and opting in to upgrade some or all components WITHOUT pinning their `xxx_package_version` in `config.yml` could result in a major version upgrade of installed component packages, which TPA does not support as it may break compatibility.

Sometimes an upgrade involves additional steps beyond installing new packages and restarting services. For example, in order to upgrade from BDR4 to PGD5, one must set up new package repositories and make certain changes to the BDR node and group configuration during the process.

In such cases, where there are complex steps required as part of the process of effecting a software upgrade, `tpaexec upgrade` will perform those steps. For example, in the above scenario, it will configure the new PGD5 package repositories (which deploy would also normally do).

However, it will make only those changes that are directly required by the upgrade process itself. For example, if you edit config.yml to add a new Postgres user or database, those changes will not be done during the upgrade. To avoid confusion, we recommend that you `tpaexec deploy` any unrelated pending changes before you begin the software upgrade process.

Upgrading from BDR-Always-ON to PGD-Always-ON

To upgrade from BDR-Always-ON to PGD-Always-ON (that is, from BDR3/4 to PGD5), first run `tpaexec reconfigure`:

```
tpaexec reconfigure ~/clusters/speedy\
  --architecture PGD-Always-ON\
  --pgd-proxy-routing local
```

This command will read config.yml, work out the changes necessary to upgrade the cluster, and write a new config.yml. For details of its invocation, see the command's own documentation. After reviewing the changes, run `tpaexec upgrade` to perform the upgrade:

```
tpaexec upgrade ~/clusters/speedy\
```

Or to run the upgrade with proxy monitoring enabled,

```
tpaexec upgrade ~/clusters/speedy\
  -e enable_proxy_monitoring=true
```

`tpaexec upgrade` will automatically run `tpaexec provision`, to update the ansible inventory. The upgrade process does the following:

1. Checks that all preconditions for upgrading the cluster are met.
2. For each instance in the cluster, checks that it has the correct repositories configured and that the required postgres packages are available in them.
3. For each BDR node in the cluster, one at a time:
    - Fences the node off to ensure that harp-proxy doesn't send any connections to it.
    - Stops, updates, and restarts postgres, including replacing BDR4 with PGD5.
    - Unfences the node so it can receive connections again.
    - Updates pgbouncer and pgd-cli, as applicable for this node.

4. For each instance in the cluster, updates its BDR configuration specifically for BDR v5

5. For each proxy node in the cluster, one at a time:
   - Sets up pgd-proxy.
   - Stops harp-proxy.
   - Starts pgd-proxy.

6. Removes harp-proxy and its support files.

Upgrading from PGD-Always-ON to PGD-X

Upgrading a `PGD-Always-ON` cluster to `PGD-X` is a **significant architectural evolution**, involving changes beyond a simple **software update**. It is a *carefully orchestrated, multi-stage process* that requires reconfiguring your cluster in distinct phases before the final software upgrade can take place. The procedure first modernizes your `PGD 5` cluster's connection handling by replacing `pgd-proxy` with the built-in `Connection Manager` –a step that currently requires manual operations on the live cluster but is planned for automation in a future TPA release–and then transitions the cluster to the new `PGD-X` architecture.

The upgrade process transitions the cluster through three distinct states:

1. **Start:** `PGD` 5.9+ (`PGD-Always-ON`) using `PGD-Proxy`
2. **Intermediate:** `PGD` 5.9+ (`PGD-Always-ON`) now using the built-in `Connection Manager`
3. **Final:** `PGD` 6 (`PGD-X Architecture`)

Prerequisites

Before you begin, ensure you have met the following requirements:

- **Cluster Version:** Your cluster must be running `PGD` version 5.9 or later. If you are on an earlier 5.x version, use `tpaexec upgrade` to upgrade to the latest minor version first. See the section (#pgd-always-on) for details on minor version upgrade of a PGD-Always-ON cluster.

- **Backup:** You have a current, tested backup of your cluster.

- **Review Overrides:** You have reviewed your `config.yml` for any instance-level proxy overrides (e.g., `pgd_proxy_options`). These cannot be migrated automatically and will require manual intervention.

- **Co-hosted Proxies:** Your `PGD 5` cluster must be configured with co-hosted proxies (where the `pgd-proxy` role is on the same instance as the `bdr` role). The presence of standalone proxy instances will cause the `switch2cm` command to abort. You must remove standalone proxy instances from your cluster before proceeding with the migration.

Stage 1: Migrating to the Built-in Connection Manager

The first stage is to reconfigure your `PGD 5.9+` cluster to switch from using the external `pgd-proxy` to the modern, built-in `Connection Manager` . TPA provides the `tpaexec switch2cm` command to automate this migration with minimal downtime.

> **Transitional State Only**
>
> This process creates a transitional `PGD 5.9+` cluster state that is intended only as an intermediate step before upgrading to `PGD 6` . TPA does not currently support staying in `PGD5.9+` with `Connection Manager` enabled or moving to a newer minor version of `PGD5.9+` with this configuration. A future TPA release will fully support lifecycle management of `PGD 5` with `Connection Manager` .

Step 1.1: Reconfigure for Connection Manager

Run the following command to update your `config.yml` file. This adds the settings required to enable the built-in `Connection Manager`.

This action only modifies the configuration file; it does not change the running state of your database cluster yet.

Before writing the new version, `reconfigure` automatically saves a backup of the current file (e.g., `config.yml.~1~`), providing a safe restore point.

For details of its invocation, see the command's own documentation.

```
tpaexec reconfigure ~/clusters/speedy --enable-connection-manager
```

Step 1.2: Switch to Connection Manager

Run the `tpaexec switch2cm` command to perform the migration from `pgd-proxy` to the built-in `Connection Manager`. This command automatically runs `tpaexec provision` to update the Ansible inventory, then switches all nodes with minimal downtime:

```
tpaexec switch2cm ~/clusters/speedy
```

The `switch2cm` command performs the following operations:

1. Updates the Ansible inventory with Connection Manager settings
2. For each node:
    - Fences the node to prevent new connections
    - Restarts PostgreSQL to load the Connection Manager configuration
    - Stops the `pgd-proxy` service
    - Restarts PostgreSQL again to allow Connection Manager to bind ports
    - Waits for Connection Manager to start listening
    - Unfences the node and verifies connectivity

This process follows the official EDB Connection Manager Migration procedure.

Stage 1 Complete

At the end of this stage, you will have a `PGD` cluster running with the built-in `Connection Manager`. This is an intermediate state, and you should proceed directly to Stage 2. While `tpaexec upgrade` for minor version upgrades is **not supported** in this intermediate state, we also advise agaist running `tpaexec deploy` until the upgrade to PGD 6 is complete.

Stage 2: Upgrading the Architecture to PGD-X

Once your cluster is running with the `Connection Manager`, you can proceed with the final configuration step to prepare for the `PGD 6` upgrade.

> **Note**
>
> You **must** start this process from a cluster that has successfully completed `Stage 1` and is running with the built-in `Connection Manager`.

Step 2.1: Reconfigure for the PGD-X Architecture

Run the following command to update your `config.yml` for the new architecture. This changes the cluster architecture type, sets the `BDR` version to 6, and removes any obsolete legacy settings.

This action only modifies the configuration file; it does not change the running state of your database cluster yet.

```
tpaexec reconfigure ~/clusters/speedy --architecture PGD-X
```

Step 2.2: Perform the Software Upgrade

After reviewing the final changes in `config.yml`, you can now run the standard `tpaexec upgrade` command. This will perform the software upgrade on all nodes, bringing your cluster to `PGD 6`.

```
tpaexec upgrade ~/clusters/speedy
```

Or to run the upgrade with proxy monitoring enabled,

```
tpaexec upgrade ~/clusters/speedy\
  -e enable_proxy_monitoring=true
```

`tpaexec upgrade` will automatically run `tpaexec provision`, to update the ansible inventory. The upgrade process does the following:

1. Checks that all preconditions for upgrading the cluster are met.
2. For each instance in the cluster, checks that it has the correct repositories configured and that the required postgres packages are available in them.
3. For each BDR node in the cluster, one at a time:
   - Fences the node off so there are no connections to it.
   - Stops, updates, and restarts postgres, including replacing PGD5 with PGD6.
   - Unfences the node so it can receive connections again.
   - Updates pgbouncer and pgd-cli, as applicable for this node.

4. Applies BDR configuration specifically for BDR v6

Upgrade Complete

Your cluster is now running `PGD 6` with the `PGD-X` architecture and is fully manageable with both `tpaexec deploy` and `tpaexec upgrade` as usual.

PGD-S or PGD-X

When upgrading an existing PGD6 (PGD-S or PGD-X) cluster to the latest available software versions, the upgrade process does the following:

1. Checks that the cluster is healthy and that the nodes are listening on the configured ports.

2. Checks that the nodes to be upgraded have their repositories configured and updated, including local repositories.

3. Checks that updated packages can be installed

4. Upgrade each BDR node in the cluster one at a time:

   Important: To ensure high availability, if the write leader is among the nodes being upgraded, it will be the very last node to be upgraded.

   - Fences the node off so it doesn't accept connections
   - Stops postgres
   - Updates postgres and PGD packages
   - Unfences the node so it can receive connections again
   - Checks that the BDR cluster has re-established Raft consensus
   - Checks that the upgraded node is listening on the configured ports

5. Re-runs the cluster health checks

6. Outputs information about the upgraded packages

PGD-Always-ON

When upgrading an existing PGD-Always-ON (PGD5) cluster to the latest available software versions, the upgrade process does the following:

1. Checks that all preconditions for upgrading the cluster are met, including that it is not a shared PEM or shared Barman cluster.
2. Runs pre-upgrade health checks for all components, as applicable to the cluster, including that no Barman backup is underway (this stops the WAL-receiver)
3. For each instance in the cluster, checks that it has the correct repositories configured and that the required postgres packages are available in them.
4. Checks that all selected components are able to be updated to the desired version (if a package version is provided)
5. For each BDR node in the cluster, one at a time:
   - Fences the node off to ensure that pgd-proxy doesn't send any connections to it.
   - Stops, updates, and restarts postgres.
   - Unfences the node so it can receive connections again.
   - Updates pgd-proxy and pgd-cli software (if explicitly opted-in)

6. For the applicable nodes in the cluster, updates pgbouncer, barman, pg-backup-api, and PEM agents/PEM server (according to the node's roles)
7. Starts the Barman WAL-receiver if required and runs post-upgrade health checks for all components (as applicable to the cluster)

BDR-Always-ON

For BDR-Always-ON clusters, the upgrade process goes through the cluster instances one by one and does the following:

1. Checks that all preconditions for upgrading the cluster are met, including that it is not a shared PEM or shared Barman cluster.
2. Runs pre-upgrade health checks for all components, as applicable to the cluster, including that no Barman backup is underway (this stops the WAL-receiver)
3. For each instance in the cluster, checks that it has the correct repositories configured and that the required postgres packages are available in them.
4. Tell haproxy the server is under maintenance.
5. If the instance was the active server, request pgbouncer to reconnect and wait for active sessions to be closed.
6. Stop Postgres, update Postgres, etcd, and pgdcli (if applicable and opted-in) packages and restart Postgres.
7. Finally, mark the server as "ready" again to receive requests through haproxy.
8. For the applicable nodes in the cluster, updates pgbouncer, barman, pg-backup-api, and PEM agents/PEM server (according to the node's roles)
9. Starts the Barman WAL-receiver if required and runs post-upgrade health checks for all components (as applicable to the cluster)

PGD logical standby or physical replica instances are updated without any haproxy or pgbouncer interaction. Non-Postgres instances in the cluster are left alone.

M1

For M1 clusters, `upgrade` will first update the streaming replicas and witness nodes when applicable, then perform a switchover from the primary to one of the upgraded replicas, update the primary, and switchover back to the initial primary node.

1. Checks that all preconditions for upgrading the cluster are met, including that it is not a shared PEM or shared Barman cluster.
2. Runs pre-upgrade health checks for all components, as applicable to the cluster, including that no Barman backup is underway (this stops the WAL-receiver)
3. For each instance in the cluster, checks that it has the correct repositories configured and that the required postgres packages are available in them.
4. Update Postgres on the streaming replicas and witness. nodes (when applicable)
5. Perform a switchover from the primary to one of the upgraded replicas
6. Update Postgres on the primary
7. Switchover back to the initial primary node.
8. For the applicable nodes in the cluster, updates pgbouncer, barman, pg-backup-api, and PEM agents/PEM server (according to the node's roles)
9. Starts the Barman WAL-receiver if required and runs post-upgrade health checks for all components (as applicable to the cluster)

Controlling the upgrade process

You can control the order in which the cluster's instances are upgraded by defining the `update_hosts` variable:

```
tpaexec upgrade ~/clusters/speedy \
  -e update_hosts=quirk,keeper,quaver
```

This may be useful to minimise lead/shadow switchovers during the upgrade by listing the active PGD primary instances last, so that the shadow servers are upgraded first.

If your environment requires additional actions, the postgres-pre-update and postgres-post-update hooks allow you to execute custom Ansible tasks before and after the package installation step.

Upgrading a Subset of Nodes

You can perform a rolling upgrade on a subset of instances by setting the `update_hosts` variable. However, support for this feature varies by architecture.

- For the **M1** architecture, this feature is fully supported for `repmgr` and `Patroni` managed clusters. `EFM` managed clusters respect the `update_hosts` list for all components *except* EFM. All data nodes will upgrade their EFM version regardless of the nodes specified in `update_hosts` as EFM does not support clusters running different versions across data nodes.

- For **PGD-Always-ON/BDR-Always-ON**, this is supported **only** during minor version upgrades.

Best Practice for PGD-Always-ON/BDR-Always-ON

When performing a minor upgrade on a subset of PGD nodes, it is highly recommended to update the **RAFT leader nodes last**. This strategy avoids potential issues with post-upgrade checks while the cluster is running mixed versions of BDR.

PGD-S and PGD-X

`tpaexec upgrade` arranges for the current write leader to be upgraded last in PGD-S and PGD-X clusters automatically: the Postgres/PGD rolling phase is split into two `serial: 1` plays, one for every BDR data node except the current write leader and one for the write leader itself. No additional steps are required to achieve the leader-last sequence on these architectures.

26 tpaexec switchover

The `tpaexec switchover` command performs a controlled switchover between a primary and a replica in a cluster that uses streaming replication. After you run this command, the selected replica is promoted to be the new primary, the former primary becomes a new replica, and any other replicas in the cluster will be reconfigured to follow the new primary.

The command checks that the cluster is healthy before switching roles, and is designed to be run without having to shut down any repmgr services beforehand.

(This is equivalent to running `repmgr standby switchover` with the `--siblings-follow` option.)

Example

This command will make `replicaname` be the new primary in `~/clusters/speedy`:

```
tpaexec switchover ~/clusters/speedy replicaname
```

Architecture options

This command is applicable only to M1 clusters that have a single writable primary instance and one or more read-only replicas.

For BDR-Always-ON clusters, use the HAProxy server pool management commands to perform maintenance on PGD instances.

Repmgr redirect pgbouncer

When using repmgr as failover manager, pgbouncer as connection pooler and setting `repmgr_redirect_pgbouncer: true`, switchover command ensures that the pgbouncer instances are redirected to the new primary node.

> **Revert to initial primary**
>
> In case you already switched over to a different primary, you can specify `revert_redirect: true` on the command that will switch back to the initial primary to make use of the initial pgbouncer config file instead of regenerating it. TPA saves the initial state of this config file as `pgbouncer.databases.ini.orig` during a switchover and can revert to it when going back to the initial primary
>
> ```
> # switchover to a replica
> tpaexec switchover <cluster_name> <replica_name>
> # revert to initial primary
> tpaexec switchover <cluster_name> <initial_primary_name> -e"revert_redirect=true"
> ```

27 BDR/HAProxy server pool management

The `tpaexec pool-disable-server` and `pool-enable-server` commands allow a PGD instance in a BDR-Always-ON cluster to be temporarily removed from the HAProxy active server pool for maintenance, and then added back afterwards.

These commands follow the same process as rolling updates by default, so `pool-disable-server` will wait for active transactions against a PGD instance to complete and for pgbouncer to direct new connections to another instance before completing. Use the `--nowait` option if you don't want to wait for active sessions to end.

Running `pool-disable-server` immediately followed by `pool-enable-server` on an instance will have the effect of moving all active traffic to a different instance (in essence, a switchover). This allows you to run online maintenace tasks such as long-running VACUUM commands, while maintaining instance availability.

If there are multiple HAProxy servers configured with the same set of `haproxy_backend_servers`, this command will remove or add the given server to the pool of every relevant proxy in parallel.

Example

```
tpaexec pool-disable-server ~/clusters/clockwork orange # --nowait

# HAProxy will no longer direct any traffic to the PGD instance named
# 'orange', so you can perform maintenance on it (e.g., run `tpaexec
# rehydrate`).

tpaexec pool-enable-server ~/clusters/clockwork orange
```

When you remove an instance from the server pool, HAProxy will not direct any traffic to it, even if the other instance(s) in the pool fail. You must remember to add the server back to the active pool once the maintenance activity is concluded.

28 tpaexec rehydrate

The `tpaexec rehydrate` command rebuilds AWS EC2 instances with an updated machine image (AMI), and allows for the rapid deployment of security patches and OS upgrades to a cluster managed by TPA.

Given a new AMI with all the required changes, this command terminates an instance, replaces it with a newly-provisioned instance that uses the new image, and attaches the data volumes from the old instance before recreating the configuration of the server exactly (based on `config.yml`).

Publishing up-to-date images and requiring servers to be rebuilt from scratch on a regular schedule is an alternative to allowing a fleet of servers to download and install individual security updates themselves. It makes it simpler to track the state of each server at a glance, and discourages any manual changes to individual servers (they would be wiped out during the instance replacement).

TPA makes it simple to minimise disruption to the cluster as a whole during the rehydration, even though the process must necessarily involve downtime for individual servers as they are terminated and replaced. On a streaming replication cluster, you can rehydrate the replicas first, then use `tpaexec switchover` to convert the primary to a replica before rehydrating it. On BDR-Always-ON clusters, you can remove each server from the haproxy server pool before rehydrating it, then add it back afterwards.

If you just want to install minor-version updates to Postgres and associated components, you can use the `tpaexec upgrade` command instead.

Prerequisites

To be able to rehydrate an instance, you must specify `delete_on_termination: no` and `attach_existing: yes` for each of its data volumes in `config.yml`. (The new instance will necessarily have a new EBS root volume.)

By default, when you terminate an EC2 instance, the EBS volumes attached to it are also terminated. In this case, since we want to reattach them to a new instance, we must disable `delete_on_termination`. Setting `attach_existing` makes TPA search for old volumes when provisioning a new instance and, if found, attach them to the instance after it's running.

Do not stop or terminate the old instance manually; the `tpaexec rehydrate` command will do this after verifying that the instance can be safely rehydrated.

Example

Let's assume you have an AWS cluster configuration in `~/clusters/night`.

Change the configuration

First, you must edit `config.yml` and specify the new AMI. For example:

```
ec2_ami:
  Name: RHEL-8.3_HVM-20210209-x86_64-0-Hourly2-GP2
  Owner: '309956199498'
```

Check that `delete_on_termination` is disabled for each data volume. If the parameter is not present, you can check its value through the AWS EC2 management console. Click on 'Instances', select an instance, then open the 'Description' tab and scroll down to 'Block devices', and click on an EBS volume. If the "Delete on termination" flag is set to true, you can change it using `awscli`. Also check `attach_existing` and set it to `yes` if it isn't set already.

Here's an example with both attributes correctly set:


```
instances:
- node: 1
  Name: vlad
  subnet: 10.33.14.0/24
  role: primary
  volumes:
  - device_name: /dev/xvdf
    volume_type:
gp2
    volume_size: 16
    attach_existing: yes
    delete_on_termination: false
    vars:
      volume_for: postgres_data
      mountpoint:
/var/lib/pgsql
```

(Note that volume parameters may be set in `instance_defaults` as well as under specific instances. Search for `volumes:` and make sure all of the relevant volumes have these two attributes set.)

Start the rehydration

Here's the syntax for the rehydrate command:

```
tpaexec rehydrate ~/clusters/night instancename
```

You can specify a single instance name or a comma-separated list of instance names (but you cannot rehydrate all of the instances in the cluster at once).

The command will first check that every non-root EBS volume attached to the instance (or instances) being rehydrated has the `delete_on_termination` flag set to false. If this is not the case, it will stop with an error before any instance is terminated.

If the volume attributes are set correctly, the command will first terminate each of the instances, then run provision and deploy to replace them with new instances using the new AMI.

Rehydrate in phases

In order to maintain cluster continuity, we recommend rehydrating the cluster in phases.

For example, in a cluster that uses streaming replication with a primary instance, two replicas, and a Barman backup server, you could rehydrate the Barman instance and one replica first, then another replica, then switchover from the primary to one of the rehydrated replicas, rehydrate the former primary, and (optionally), switchover back to the original primary. This sequence ensures that one primary and one replica are always available.

Appendix

Using awscli to change volume attributes

First, find the instance and EBS volume in the AWS management console. Click on 'Instances', select an instance, open the 'Description' tab and scroll down to 'Block devices', and select an EBS volume. To disable `delete_on_termination`, run the following command after substituting the correct values for the `--region`, `--instance-id`, and block device name:

```
aws ec2 modify-instance-attribute \
    --region eu-west-1 --instance-id i-XXXXXXXXXXXXXXXXX \
    --block-device-mappings \
      '[{"DeviceName": "/dev/xvdf", "Ebs": {"DeleteOnTermination": false}}]'
```

Do this for each of the data volumes for the instance, and after a brief delay, you should be able to see the changes in the management console, and `tpaexec rehydrate` will also detect that the instance can be safely rehydrated.

29 TPA and Ansible Tower/Ansible Automation Platform

TPA supports deployments via RedHat Ansible Automation Platform (AAP). The support, as detailed below, works by allowing you to run `deploy` and `upgrade` steps on AAP. Before you can run deploy or upgrades (later), you will need to run configuration (`configure` command) and provisioning (`provision` command) on a separate standalone machine that has tpa packages installed. Once you have run `configure` and `provision` on this standalone machine with suitable options, you can then import the resulting cluster directory on AAP. Support is limited to bare-metal platforms.

TPA is tested with AAP 2.4 and AAP 2.6.

AAP initial setup

Before TPA can use AAP to deploy clusters, you need to perform this initial setup.

Add TPA Execution Environment image (admin)

Starting with version 2.4, AAP uses container images to run Ansible playbooks. These containers, called Execution Environments (EE), bundle dependencies required by playbooks to run successfully. As a consequence, this means that in order to, resolve and use all required TPA dependencies, you will need an EE that includes TPA so your AAP can use it when running deployments and upgrades.

> **Get an EE**
>
> See Build an EE for TPA for instructions on building your own image.
>
> EDB customers can reach out to EDB Support for help with EE.

As an AAP admin, create an entry in your available EE list that points to your TPA enabled EE image.

Create the EDB_SUBSCRIPTION_TOKEN credential type (admin)

As an AAP admin, create the custom credential type `EDB_SUBSCRIPTION_TOKEN` to hold your EDB subscription access token:

1. Go to the Credential Types page in the AAP UI.

2. Set the **Name** field to `EDB_SUBSCRIPTION_TOKEN`.

3. Paste the following into the **Input Configuration** field:

```
fields:
- id: tpa_edb_sub_token
  type:
string
  label: EDB_SUBSCRIPTION_TOKEN
  secret: true
required:
- tpa_edb_sub_token
```

4. Paste the following into the **Injector Configuration** field:

```
env:
  EDB_SUBSCRIPTION_TOKEN: '{{ tpa_edb_sub_token
}}'
```


5. Save the changes.

6. Create a credential using the newly added type `EDB_SUBSCRIPTION_TOKEN`.

Setting up a cluster

Perform the initial steps on a workstation with the tpaexec packages installed.

On the TPA workstation

Configure

Run the `tpaexec configure` command, including these options: `--platform bare`, `--use-ansible-tower`, `--tower-git-repository`

```
tpaexec configure <clustername> \
         --platform bare \
         --use-ansible-tower https://aac.example.com \
         --tower-git-repository ssh://git@git.example.com/example \
         --hostnames-from <hostnamefile> \
         --architecture PGD-Always-ON \
         --pgd-proxy-routing local \
         --postgresql 16
```

`--use-ansible-tower` expects the AAP address as a parameter even if it isn't used at the time. `--tower-git-repository` is used to import the cluster data into AAP. TPA creates its own branch using `cluster_name` as the branch name, which allows you to use the same repository for all of your clusters. All other options to `tpaexec configure`, as described in Configuration, are still valid.

config.yml modification

`config.yml` includes the top-level dictionary `ansible_tower`, which causes `tpaexec provision` to treat the cluster as an AAP-enabled cluster.

Edit `config.yml` to ensure that `ansible_host` and `{private,public}_ip` are defined for each node and `ansible_host` is set to a value that AAP can resolve. Make any further changes or additions that you may need. See Cluster configuration for more details.

To generate inventory and other related files, run `tpaexec provision`.

On the AAP UI

Project

Add a project in AAP using the git repository as the source. Set the default EE of the project to use the TPA EE image.

> **Project options**
>
> To ensure changes are correctly synced before running a job, we strongly recommend using**Update Revision on Launch**.
>
> **Allow Branch Override** is required when trying to use multiple inventories with a single project.

Inventory

Add an empty inventory. Use the project as an external source to populate it using `inventory/00-cluster_name` as the inventory file.

> **Inventory options**
>
> To ensure changes are correctly synced, we strongly recommend using **Overwrite local groups and hosts from remote inventory source** in AAP 2.4, or the equivalent, **Overwrite** in AAP 2.6 .
>
> We also recommend using **Overwrite local variables from remote inventory source** (in AAP 2.4) or **Overwrite variables** (in AAP 2.6) when not setting additional variables outside TPA's control in AAP.

Credentials

Create a `vault` credential. You can retrieve the vault password using `tpaexec show-vault <cluster_dir>` on the TPA workstation.

Ensure that you have a `machine` credential available which will enable the AAP server to connect to your inventory nodes.

Template creation

To create a template:

1. Create a template that uses your project and your inventory.

2. Include these required credentials:

   - Vault credential
   - `EDB_SUBSCRIPTION_TOKEN` credential
   - Machine credential

3. Set two additional variables:

   ```
   tpa_dir: /opt/EDB/TPA
   cluster_dir: /runner/project
   ```

4. Select `deploy.yml` as the playbook.

5. To deploy your cluster, run a job based on the new template.

Running custom playbooks

In addition to the initial deploy, you will often want to run small ad-hoc playbooks against an existing cluster: to check service status, inspect logs, collect diagnostic output, and so on. AAP's ad-hoc command facility is not a good fit for this, because it runs directly against inventory hosts without reference to a Project and so has no access to the cluster directory's `ssh_config`, `ansible.cfg`, vault, or other TPA context. The right approach is to add your playbook to the cluster's git repository and run it from a second Template.

We recommend putting the playbook in the `commands/` subdirectory of the cluster, which has two benefits: it keeps custom playbooks separate from TPA's own files, and anything placed there is also picked up by `tpaexec` as a subcommand (the same mechanism works for shell scripts placed in `commands/`).

For example, save the following as `<cluster_dir>/commands/check-disk-space.yml` :


```
---
# Sample playbook: check disk space on every cluster
node.
- hosts:
all
  gather_facts: false
  tasks:
    - name: Show disk usage
      ansible.builtin.command: df -
h
      changed_when: false
```

Commit and push the file to the cluster's branch in the remote repository so AAP can see it:

```
git add commands/check-disk-space.yml
git commit -m "Add check-disk-space playbook"
git push
```

In the AAP UI, use the **Duplicate template** action on your deploy template to create a copy with the same project, inventory, credentials, and extra variables. Edit the new template and change the playbook to `commands/check-disk-space.yml`. Launch a job based on the new template. The per-host output of `df -h` is available by clicking through to each host in the job details and selecting the **Output** tab.

Use one project for multiple inventory

TPA uses a different branch name for each of your clusters in the associated git repository. This approach allows the use of a single project for multiple clusters.

Set Allow branch override option

In the AAP project, enable the **Allow branch override** option.

Define multiple inventories

TPA uses a different branch name for each of your clusters in the git repository. You can generate multiple inventories using the same project as the source by overriding the branch for each inventory.

Define credentials per inventory

Ensure vault passwords are set accordingly per inventory since these differ on each TPA cluster.

Update TPA on AAP

Updating TPA on AAP involves some extra steps.

Update TPA workstation package

Update your TPA workstation package as any OS package depending on your OS. See Installation.

Use EE image with same version tag

Modify the EE image in AAP to use the same version tag as the workstation package version used.

Run tpaexec relink on your cluster directory

Ensure that any cluster using AAP is up to date by running `tpaexec relink <cluster_dir> --force`. An example of when you need to do this is after you have upgraded your TPA installation to a new version. Be sure to push any change committed by the `relink` command:

```
git status
```

output

```
On branch cluster_name
Your branch is ahead of 'tower/cluster_name' by 1 commit.
  (use "git push" to publish your local commits)
...
```

```
git push tower
```

Sync project and inventories

If they aren't set to use **Update revision on job launch** and **Update on launch**, sync the project in the AAP UI and related inventories, respectively.

Build an EE for TPA

Prerequisites

In order to build your own EE image, we recommend using `ansible-builder`.

You need:

1. `docker` or `podman`
2. `ansible-builder` and `ansible-navigator` python toolkits
3. TPA source code checked out at tag `vA.B.C` from TPA repo where `vA.B.C` is the TPA version you want to use.

Environment file

`ansible-builder` uses an environment file to help generate a working EE image.

Here is a template example of such an environment file for TPA:

execution-environment.yml

```
version: 3
images:
  base_image:
    name: 'registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel9:latest'
dependencies:
  python: << TPA_REPO_CLONE_FOLDER >>/requirements-
aap.txt
  galaxy: << TPA_REPO_CLONE_FOLDER
>>/collections/requirements.yml
options:
  package_manager_path: /usr/bin/microdnf

additional_build_steps:
  append_final:
   - RUN mkdir -p
/opt/EDB/TPA
   - COPY << TPA_REPO_CLONE_FOLDER >>
/opt/EDB/TPA
   - ENV
PYTHONPATH="${PYTHONPATH:+${PYTHONPATH}:}/opt/EDB/TPA/lib"
```

> **Base image**
>
> Base image used here requires access to registry.redhat.io (should be provided alongside AAP license). This image already comes with most of the requirements for AAP 2.4 such as `python 3.12.*`, `ansible-core==2.16.*`, and `ansible-runner` which simplify the task.
>
> Different base image may require more `additional_build_steps`. See ansible-builder for advanced usage.

EE build command

The following command should build the EE image for you:

```
ansible-builder build \
  --file=execution-environment.yml \
  --container-runtime=<docker/podman> \
  --tag=<your-registry>/<your-namespace>/tpa-ee:vA.B.C \
  --verbosity 2
```

30 TPA, Ansible, and sudo

TPA uses Ansible with sudo to execute tasks with elevated privileges on target instances. This page explains how Ansible uses sudo (which is in no way TPA-specific), and the consequences to systems managed with TPA.

TPA needs root privileges;

- to install packages (required packages using the operating system's native package manager, and optional packages using pip)
- to stop, reload and restart services (i.e Postgres, repmgr, efm, etcd, haproxy, pgbouncer etc.)
- to perform a variety of other tasks (e.g., gathering cluster facts, performing switchover, setting up cluster nodes)

TPA also needs to be able to use sudo. You can make it ssh in as root directly by setting `ansible_user: root`, but it will still use sudo to execute tasks as other users (e.g., postgres).

Ansible sudo invocations

When Ansible runs a task using sudo, you will see a process on the target instance that looks something like this:

```
/bin/bash -c 'sudo -H -S -n  -u root /bin/bash -c \
  '"'"'echo BECOME-SUCCESS-kfoodiiprztsyerriqbjuqhhbemejgpc ; \
  /usr/bin/python2'"'"' && sleep 0'
```

People who were expecting something like `sudo yum install -y xyzpkg` are often surprised by this. By and large, most tasks in Ansible will invoke a Python interpreter to execute Python code, rather than executing recognisable shell commands. (Playbooks may execute `raw` shell commands, but TPA uses such tasks only to bootstrap a Python interpreter.)

Ansible modules contain Python code of varying complexity, and an Ansible playbook is not just a shell script written in YAML format. There is no way to “extract” shell commands that would do the same thing as executing an arbitrary Ansible playbook.

There is one significant consequence of how Ansible uses sudo: privilege escalation must be general. That, it is not possible to limit sudo invocations to specific commands in sudoers.conf, as some administrators are used to doing. Most tasks will just invoke python. You could have restricted sudo access to python if it were not for the random string in every command—but once Python is running as root, there's no effective limit on what it can do anyway.

Executing Python modules on target hosts is just the way Ansible works. None of this is specific to TPA in any way, and these considerations would apply equally to any other Ansible playbook.

Recommendations

- Use SSH public key-based authentication to access target instances.

- Allow the SSH user to execute sudo commands without a password.

- Restrict access by time, rather than by command.

TPA needs access only when you are first setting up your cluster or running `tpaexec deploy` again to make configuration changes, e.g., during a maintenance window. Until then, you can disable its access entirely (a one-line change for both ssh and sudo).

During deployment, everything Ansible does is generally predictable based on what the playbooks are doing and what parameters you provide, and each action is visible in the system logs on the target instances, as well as the Ansible log on the machine where tpaexec itself runs.

Ansible's focus is less to impose fine-grained restrictions on what actions may be executed and more to provide visibility into what it does as it executes, so elevated privileges are better assigned and managed by time rather than by scope.

SSH and sudo passwords

We *strongly* recommend setting up password-less SSH key authentication and password-less sudo access, but it is possible to use passwords too.

If you set `ANSIBLE_ASK_PASS=yes` and `ANSIBLE_BECOME_ASK_PASS=yes` in your environment before running tpaexec, Ansible will prompt you to enter a login password and a sudo password for the remote servers. It will then negotiate the login/sudo password prompt on the remote server and send the password you specify (which will make your playbooks take noticeably longer to run).

We do not recommend this mode of operation because we feel it is a more effective security control to completely disable access through a particular account when not needed than to use a combination of passwords to restrict access. Using public key authentication for ssh provides an effective control over who can access the server, and it's easier to protect a single private key per authorised user than it is to protect a shared password or multiple shared passwords. Also, if you limit access at the ssh/sudo level to when it is required, the passwords do not add any extra security during your maintenance window.

sudo options

To use Ansible with sudo, you must not set `requiretty` in sudoers.conf.

If needed, you can change the sudo options that Ansible uses (`-H -S -n`) by setting `become_flags` in the `[privilege_escalation]` section of ansible.cfg, or `ANSIBLE_BECOME_FLAGS` in the environment, or `ansible_become_flags` in the inventory. All three methods are equivalent, but please change the sudo options only if there is a specific need to do so. The defaults were chosen for good reasons. For example, removing `-S -n` will cause tasks to timeout if password-less sudo is incorrectly configured.

Managing privilege escalation configuration

Default sudo configuration

By default, TPA automatically manages sudo-related configuration on target instances, including installing the sudo package if not present and configuring sudoers files for various components.

The default value of `privilege_escalation_command` is `"sudo"`, which enables TPA to manage sudo installation and configuration.

Using an alternative privilege escalation command

If your environment uses a different privilege escalation command, you can configure TPA to use an alternative by setting `privilege_escalation_command` in `cluster_vars`:

```
cluster_vars:
  privilege_escalation_command: other_tool  # replaces sudo; may include
arguments
```

The value is used as a direct in-place replacement for `sudo` in each privilege escalation command TPA generates for managed applications. It may include additional arguments if needed (e.g., `other_tool --flag`).

You can set it to any privilege escalation command supported by Ansible's become mechanism. Refer to the Ansible privilege escalation documentation for the complete list of supported methods.

Important: `sudo` is the only privilege escalation command officially supported by EDB. When using alternative commands, you are responsible for ensuring compatibility and proper configuration. EDB Support may have limited ability to assist with issues related to alternative privilege escalation mechanisms.

Ansible's become method vs. `privilege_escalation_command`

There are two separate privilege escalation settings to be aware of:

- `ansible_become_method` controls how Ansible itself escalates privileges when running deployment tasks on target instances. This is an Ansible variable set in your inventory or `ansible.cfg`.
- `privilege_escalation_command` controls the command that the managed applications (EFM, repmgr, HARP) invoke at runtime to escalate privileges for service management operations, independently of Ansible.

Because they serve different purposes, both must be configured consistently. If your environment uses an alternative tool instead of sudo, you must set `privilege_escalation_command` in `cluster_vars` *and* configure `ansible_become_method` in your Ansible inventory or `ansible.cfg` to match.

Recommended approach: Using hooks

If you need to use an alternative privilege escalation command, we recommend using TPA hooks to configure your privilege escalation mechanism. Hooks allow you to run custom tasks at specific points during deployment, giving you full control over how privilege escalation is configured whilst keeping your customisations separate from TPA's core deployment logic.

For example, you can use a `post-repo` hook to install and configure your privilege escalation command after repositories are configured, or a `pre-deploy` hook to set up the necessary permissions before the main deployment begins. This approach provides better maintainability and makes it easier to manage environment-specific requirements.

Manual configuration requirements

When using an alternative privilege escalation command (anything other than `"sudo"`), TPA will skip sudo package installation and sudoers configuration. You must install and configure the chosen privilege escalation command on all target systems, either manually or with hooks, before running `tpaexec deploy`.

1. Service management permissions for the postgres user

Your privilege escalation mechanism must allow the postgres system user to execute systemctl commands for starting, stopping, restarting, and reloading PostgreSQL and related services. This is required for failover managers (repmgr, HARP, EFM) to function correctly during automatic failover operations.

For example, with sudo, TPA would configure the following permissions:

```
postgres ALL=(ALL) NOPASSWD: /bin/systemctl start postgresql
postgres ALL=(ALL) NOPASSWD: /bin/systemctl stop postgresql
postgres ALL=(ALL) NOPASSWD: /bin/systemctl restart postgresql
postgres ALL=(ALL) NOPASSWD: /bin/systemctl reload postgresql
```

You must configure equivalent permissions in your chosen privilege escalation system.

2. EFM database function permissions (EFM clusters only)

If your cluster uses EFM as the failover manager, your privilege escalation mechanism must allow the EFM system user to execute the `efm_db_functions` script as the postgres user. This is required for EFM to perform health checks and failover operations.

For example, with sudo, TPA would configure:

```
efm ALL=(postgres) NOPASSWD: /usr/edb/efm-X.Y/bin/efm_db_functions
```

Configure equivalent permissions in your privilege escalation system to allow the efm user to run this script as the postgres user.

Logging

For playbook executions, the sudo logs will show mostly invocations of Python (just as it will show only an invocation of bash when someone uses `sudo -i`).

For more detail, the syslog will show the exact arguments to each module invocation on the target instance. For a higher-level view of why that module was invoked, the ansible.log on the controller shows what that task was trying to do, and the result.

If you want even more detail, or an independent source of audit data, you can run auditd on the server and use the SELinux log files. You can get still more fine-grained syscall-level information from bpftrace/bcc (e.g., opensnoop shows every file opened on the system, and execsnoop shows every process executed on the system). You can do any or all of these things, depending on your needs, with the obvious caveat of increasing overhead with increased logging.

Local privileges

The installation instructions for TPA mention sudo only as shorthand for “run these commands as root somehow”. Once TPA is installed and you have run `tpaexec setup` , TPA itself does not require elevated privileges on the local machine. (But if you use Docker, you must run tpaexec as a user that belongs to a group that is permitted to connect to the Docker daemon.)

31 TPA - PuTTY Configuration guide

In order to use PuTTY under Windows to connect via ssh to the AWS instances that were created by the TPA utility ***tpaexec provision***, the keys will need to be converted from the private key format (.pem) generated by Amazon EC2 to the PuTTY format (.ppk).

```
# Provision the cluster
[tpa]$ tpaexec provision <clustername>
```

PuTTY has a tool named **PuTTYgen**, which can convert keys to the required format.

Key conversion

Locate private key

Locate the private key in the cluster directory `<clustername>` - it will be named according to the **cluster_name** variable set in **config.yml** prefixed by **id_** - e.g. if the cluster_name is set to **testenv1**, then the private key will be called **id_testenv1**.

Save key as .pem

Copy this file into your Windows filesystem & save it as a .pem file - in this example **id_testenv1.pem** - cut and pasting into a text file will work fine for this.

Key conversion

Start **PuTTYgen** and under Parameters, select appropriate Type of key to generate:

For older versions of **PuTTYgen**, select **SSH-2 RSA**; for recent versions select **RSA**

Do not select SSH-1 (RSA)

Now choose **Load** - in the box that says **PuTTY Private Key Files (*.ppk)** you will need to select **All Files (*.*)**

Select your **.pem** file and choose **Open**, then click **OK**.

Select **Save private key** and click **Yes** to ignore the warning about saving the key without a passphrase. Make sure that the file suffix is **.ppk** and choose the same name as for the **.pem** file; in this example the filename might be **id_testenv1.ppk**

Configure PuTTY

Start **PuTTY** and select **Session** from the **Category** window. In the **Host Name** panel, enter `<user>@<IP address>` and in the Port Panel, enter **22**

The `<user>` and `<IP address>` can be found in the `<clustername>/ssh_config` file which gets created by the `tpaexec provision` utility.

In the Putty **Category** window, Select **Connection**, expand **SSH** and select **Auth**

For the panel marked ***Private key file for authentication***, click **Browse** and select the .ppk file that was saved above, then select **Open**

In the Putty **Category** window, select **Session** again, enter a session name in **Saved Sessions**, and **Save**

You should now be able to connect to the AWS host via PuTTY by selecting this saved session.

32 Troubleshooting

Recreate python virtual environment

Occasionally the python venv can get in an inconsistent state, in which case the easiest solution is to delete and recreate it. Symptoms of a broken venv can include errors during provisioning like:

```output
TASK [Write Vagrantfile and firstboot.sh]
************************************************************************************************************
****************
failed: [localhost] (item=Vagrantfile) => {"changed": false, "checksum":
"bf1403a17d897b68fa8137784d298d4da36fb7f9", "item": "Vagrantfile", "msg": "Aborting, target uses selinux
but python bindings (libselinux-python) aren't installed!"}
```

To create a new virtual environment (assuming tpaexec was installed into the default location):

```
sudo rm -rf /opt/EDB/TPA/tpa-venv
sudo /opt/EDB/TPA/bin/tpaexec setup
```

Strange AWS errors regarding credentials

If the time & date of the TPA server isn't correct, you can get AWS errors similar to this during provisioning:

```output
TASK [Register key tpa_cluster in each region] **********************************************
An exception occurred during task execution. To see the full traceback, use -vvv. The error was:
ClientError: An error occurred (AuthFailure) when calling the DescribeKeyPairs operation: AWS was not able
to validate the provided access credentials
failed: [localhost] (item=eu-central-1) => {"boto3_version": "1.8.8", "botocore_version": "1.11.8",
"changed": false, "error": {"code": "AuthFailure", "message": "AWS was not able to validate the provided
access credentials"}, "item": "eu-central-1", "msg": "error finding keypair: An error occurred
(AuthFailure) when calling the DescribeKeyPairs operation: AWS was not able to validate the provided
access credentials", "response_metadata": {"http_headers": {"date": "Thu, 27 Sep 2018 12:49:41 GMT",
"server": "AmazonEC2", "transfer-encoding": "chunked"}, "http_status_code": 401, "request_id": "a0d905ba-
188f-48fe-8e5a-c8d8799e3232", "retry_attempts": 0}}
```

Solution - set the time and date correctly.

```
sudo ntpdate pool.ntp.org
```

Logging

By default, all tpaexec logging will be saved in logfile `<clusterdir>/ansible.log`

To change the logfile location, set environment variable `ANSIBLE_LOG_PATH` to the desired location - e.g.

```
export ANSIBLE_LOG_PATH=~/ansible.log
```

To increase the verbosity of logging, just add `-v` / `-vv` / `-vvv` / `-vvvv` / `-vvvvv` to tpaexec command line:

```
tpaexec deploy <clustername> -v
```

Where:

```
-v     shows the results of modules
-vv    shows the files from which tasks come
-vvv   shows what commands are being executed on the target machines
-vvvv  enables connection debugging, what callbacks have been loaded
-vvvvv shows some additional ssh configuration, filepath information
```

Cluster test

An easy way to smoketest an existing cluster is to run:

```
tpaexec test <clustername>
```

This will do a functional test of the cluster components, followed by a performance test of the cluster, using pgbench. As pgbench can take a while to complete, benchmarking can be omitted by running:

```
tpaexec test <clustername> --excluded_tasks=pgbench
```

TPA server test

To check the installation of the TPA server itself, run:

```
tpaexec selftest
```

Including or excluding specific tasks

When re-running a tpaexec provision or deploy after a failure or when running tests, it can sometimes be useful to miss out tasks using TPA'stask selection mechanism.

33 Selective task execution

Using task selectors

You can tell TPA to run only a subset of the tasks that constitute a full deployment using the `--excluded_tasks` and `--included_tasks` options to `tpaexec deploy`. Each of these arguments is a string treated as a comma-separated list of selectors. Equivalently, you can set the `excluded_tasks` and `included_tasks` variables in `config.yml`, either for the whole cluster or for the separate instances. In `config.yml`, you can use either a comma-separated string or a yaml list.

Tasks matched by `excluded_tasks` are always excluded. If you specify `included_tasks`, then non-matching tasks are implicitly excluded.

Some selectors may be used in either list, and some only in the `excluded_tasks` list, as detailed below. A separate set of selectors is available for `tpaexec test`.

Examples

To deploy without running barman-related tasks:

```
tpaexec deploy <clustername> --excluded_tasks=barman
```

To deploy running only repmgr-related tasks:

```
tpaexec deploy <clustername> --included_tasks=repmgr
```

To deploy without trying to set hostnames on the instances:

```
tpaexec deploy <clustername> --excluded_tasks=hostname
```

To prevent bootstrap and ssh tasks from ever running, put the following into `config.yml`:

```
cluster_vars:
    excluded_tasks:
    - bootstrap
    -
ssh
```

Supported selectors for `tpaexec deploy`

The following selectors are supported for either inclusion or exclusion:

- barman

  Tasks related to Barman.

- bdr

  Tasks related to setting up BDR, including when it is as used within a PGD cluster. If this selector is excluded, TPA will still install and configure the extension as specified in config.yml, but won't create the node groups or try to join the nodes.

- create_pgd_proxy_system_user

  Tasks which creates the system user for pgd_proxy

- create_postgres_system_user

  Tasks which creates the system user for postgres

- efm

  Tasks related to EFM.

- etcd

  Tasks related to etcd.

- first-backup

  Tasks which ensure the minimum number of barman backups exist.

- haproxy

  Tasks related to haproxy.

- harp

  Tasks related to harp.

- patroni

  Tasks related to patroni.

- pem-agent

  Tasks related to the PEM agent.

- pem-server

  Tasks related to the PEM server.

- pem-webserver

  Tasks related to configuring the web server on a PEM server.

- pg-backup-api

  Tasks related to Barman's Postgres backup API.

- pgbouncer

  Tasks related to PgBouncer.

- pgd-proxy

  Tasks related to PGD Proxy.

- pglogical

  Tasks related to pglogical.

- pkg

  Tasks which install packages using the system package manager.

- post-deploy

  The post-deploy hook, if one is defined.

- postgres

  Tasks related to postgres.

- replica

  Tasks which are run and instances acting as postgres replicas.

- repmgr

  Tasks related to repmgr.

- restart

  Tasks which restart services

- sys

  Tasks related to system setup before any tasks specific to postgres or related software.

- zabbix-agent

  Tasks related to the zabbix agent.

The following selectors are supported only for exclusion:

- artifacts

  Tasks related to artifacts.

- barman-clean

  Tasks which clean up the Barman build directory if Barman is being built from source.

- barman-pre-config

  The barman-pre-config hook, if one is defined.

- bdr-pre-node-creation

  The bdr-pre-node-creation hook, if one is defined.

- bdr-post-group-creation

  The bdr-post-group-creation hook, if one is defined.

- bdr-pre-group-join

  The bdr-pre-group-join hook, if one is defined.

- bootstrap

  Tasks which ensure that python and other minimal dependencies are present before the rest of the deploy runs. Exclude this only if you are sure you have manually installed the relevant requirements.

- build-clean

  Tasks which clean up build directories for any software that is being built from source.

- build-configure

  Tasks which configure any software that is being built from source.

- cloudinit

  Tasks which are run only on hosts managed by cloud-init.

- commit-scopes

  Tasks related to configuration of BDR commit scopes.

- config

  Tasks which create config files.

- efm-pre-config

  The efm-pre-config hook, if one is defined.

- efm-post-config

  The efm-post-config hook, if one is defined.

- fs

  Tasks related to setting up additional volumes on instances.

- hostkeys

  Tasks which set up ssh host keys.

- hostname

  Tasks which set the hostname.

- hosts

  Tasks which add entries to /etc/hosts

- initdb

  Tasks which run initdb.

- local-repo

  Tasks which set up local package repositories.

- locale

  Tasks which install locale support.

- openvpn

  Tasks which set up OpenVPN.

- pg-backup-api-clean

  Tasks which clean up the build directory if the Postgres backup API is being built from source.

- pgbouncer-config

  Tasks which create configuration files for pgbouncer.

- pgpass

  Tasks which create the .pgpass file.

- post-repo

  The post-repo hook, if one is defined.

- postgres-clean

  Tasks which clean up the build directory if postgres is being built from source.

- postgres-config

  The postgres-config hook, if one is defined.

- postgres-config-final

  The postgres-config-final hook, if one is defined.

- pre-deploy

  The pre-deploy hook, if one is defined.

- pre-initdb

  The pre-initdb hook, if one is defined.

- replication-sets

  Tasks related to witness-only replication sets on a BDR3 cluster.

- repmgr-clean

  Tasks which clean up the build directory if repmgr is being built from source.

- repmgr-configure

  Tasks which configure repmgr if it is being built from source.

- repo

  Tasks which set up package repositories.

- rsyslog

  Tasks related to rsyslog.

- service

  Tasks related to system services, including configuration and restarting.

- src

  Tasks which build and install packages from source.

- ssh

  Tasks related to setting up ssh between instances.

- ssh-cluster-key-config

  Tasks which add the cluster SSH public key to the `ansible_user` 's (default: root) authorized_keys file. This can be excluded on platforms where authorized_keys is managed externally or is read-only.

- sysctl

  Tasks which set and reload sysctl settings.

- sysstat

  Tasks releated to the sysstat service.

- tpa

  Tasks related to TPA's own files installed on instances.

- user

  Tasks related to setting up system users.

- watchdog

  Tasks related to the kernel watchdog on a patroni cluster.

Supported selectors for `tpaexec test`

The following selectors apply only for execution of `tpaexec test`:

- camo

  Tasks related to testing CAMO in a BDR or PGD cluster.

- ddl

  Tasks related to testing DDL in a BDR or PGD cluster.

- fail

  Tasks which abort tests if a problem is detected. Exclude this selector to run tests regardless of failures.

- pgbench

  Tasks which run pgbench.

- sys

  Tasks which run system-level tests.

- barman, bdr, haproxy, pg-backup-api, pgbouncer, pgd-proxy, postgres, repmgr,

  Tasks which test the various software components.

34 Running TPA in a Docker container

If you are using a system for which there are no TPA packages available, and it's difficult to run TPA after installing from source (for example, because it's not easy to obtain a working Python 3.12+ interpreter), your last resort may be to build a Docker image and run TPA inside a Docker container.

Please note that you do not need to run TPA in a Docker container in order to deploy to Docker containers. It's always preferable to run TPA directly if you can (even on MacOS X).

Quickstart

You must have Docker installed and working on your system already.

Run the following commands to clone the tpaexec source repository from Github and build a new Docker image named `tpa/tpaexec`:

```
git clone ssh://git@github.com/EnterpriseDB/tpa.git
cd tpa
docker build -f docker/Dockerfile --build-arg TPA_VER=$(git describe) -t tpaexec:latest .
```

Double-check the created image:

```
docker image ls tpaexec
```

```
output
REPOSITORY   TAG      IMAGE ID       CREATED          SIZE
tpaexec      latest   3943dec4d660   20 minutes ago   658MB
```

```
docker run --rm tpaexec info
```

```
output
# TPAexec v23.38.0-38-g4dc030dc1
tpaexec=/usr/local/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.13.5, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.16.14)
```

Then you need to setup an alias for `tpaexec` on the shell session you are running:

```
alias tpaexec="docker run --rm -v $PWD:/work -v /var/run/docker.sock:/var/run/docker.sock tpaexec"
```

Now you can run commands like:

```
tpaexec configure cluster -a M1 --postgresql 15 --failover-manager patroni --platform docker
tpaexec deploy cluster
```


Installing Docker

Please consult the Docker documentation if you need help to install Docker and get started with it.

On MacOS X, you can install "Docker Desktop for Mac" and launch Docker from the application menu.

34.1 Managing clusters in a disconnected or air-gapped environment

In a security controlled environment where no direct connection to the Internet is allowed, it is necessary to provide all packages needed by TPA to complete the deployment. This can be done via a local-repo on each node in the cluster. TPA supports the addition of custom repositories on each node via a local-repo and the required packages can be downloaded using the download-packages command.

Preparation

Choose an internet connected machine where you can install TPA and follow the instructions below to either copy an existing cluster configuration or create a new cluster.

> **Note**
>
> If the air-gapped server does not already have TPA installed, follow the instructions here to install it.

If you have an existing cluster in a disconnected environment, all you need on the internet connected host is the config.yml. Create a directory and copy that file into it then run `tpaexec relink` on that directory to generate the remaining files that would normally be created by `tpaexec configure`.

Alternatively, to create a new configuration for an environment where the target instances will not have network access, configure a new cluster with this option:

```
tpaexec configure --use-local-repo-only …
```

This will do everything that `--enable-local-repo` does, and disable the configuration for all other package repositories. On RedHat instances, this also includes disabling access to subscription-based services.

In an existing cluster, you can set `use_local_repo_only: yes` in `config.yml`:

```
cluster_vars:
    use_local_repo_only: yes
```

Note: that you do not need separate cluster configurations for internet connected and disconnected environments, the options below work in both.

More info on using local-repo for distributing packages

Downloading packages

On the internet connected machine, ensure that you have docker installed and run:

```
tpaexec download-packages cluster-dir --os <OS> --os-version <version>
```

See detailed description for the package downloader.

Copying packages to the target environment

The resulting repository will be contained in the `cluster-dir/local-repo` directory. This is a complete package repo for the target OS. Copy this directory, from the connected controller to the disconnected controller that will be used to deploy the cluster. Place the directory in the same place, beneath the cluster directory. TPA will then copy packages to the instances automatically when `deploy` is run.

Deploying in a disconnected environment

Ensure that the cluster config.yml has been configured as above in Preparation. Run `tpaexec provision` and `deploy` as you would normally.

Updating in a disconnected environment

You can use the upgrade command to perform updates in an air-gapped environment. Prior to running this command you must follow the steps below.

Updating local repository content with direct access to internet on TPA node

If the air-gapped environment has internet access, we can safely rely on `tpaexec download-packages` to download the new packages, check the download section to know its usage.

Updating local repository content without direct access to internet on TPA node

There are two ways of performing the updating an internet-disconnected local repository: Recreate the local repository manually following the correct steps depending on the distribution of your choice; or use `--refresh-repository` option in `tpaexec download-packages` after placing the new packages on the local repository. Check the documentation for more information (Docker must be available, alongside the corresponding Docker image to the TPA nodes cluster).

Once the changes are saved, we can execute `tpaexec upgrade cluster-dir` and the packages will be updated.

Cross-version Shared Barman

When a single Barman host backs up Postgres instances running different major version from the initial cluster managing the shared barman node, the additional major client packages must be present in the Barman cluster's `local-repo`. The default `tpaexec download-packages` run is only aware of the cluster's version and won't pre-fetch additional version from other clusters using the shared barman instance. See Backing up nodes at different PostgreSQL versions for the recommended workflow.

34.2 Distribution support

TPA detects and adapts to the distribution running on each target instance. This page lists platforms which are actively supported and 'legacy distributions' which have previously been supported. Deploying to a legacy platform is likely to work as long as you have access to the necessary packages, but this is not considered a supported use of TPA and is not suitable for production use.

Fully supported platforms are supported both as host systems for running TPA and target systems on which TPA deploys the Postgres cluster.

Debian ARM64

- Debian 12/bookworm is fully supported

Debian x86

- Debian 12/bookworm is fully supported
- Debian 11/bullseye is fully supported
- Debian 10/buster is a legacy distribution
- Debian 9/stretch is a legacy distribution
- Debian 8/jessie is a legacy distribution

Ubuntu x86

- Ubuntu 24.04/noble is fully supported
- Ubuntu 22.04/jammy is fully supported
- Ubuntu 20.04/focal is a legacy distribution
- Ubuntu 18.04/bionic is a legacy distribution
- Ubuntu 16.04/xenial is a legacy distribution

Oracle Linux x86

- Oracle Linux 9.x is fully supported
- Oracle Linux 8.x is fully supported
- Oracle Linux 7.x is a legacy distribution

RedHat x86

- RHEL/Rocky/AlmaLinux/Oracle Linux 10.x is fully supported (python3 only)
- RHEL/Rocky/AlmaLinux/Oracle Linux 9.x is fully supported (python3 only)
- RHEL/CentOS/Rocky/AlmaLinux 8.x is fully supported (python3 only)
- RHEL/CentOS 7.x is a legacy distribution (python2 only)

> **Warning**
>
> To deploy to instances running RHEL 8.x or equivalents, the TPA controller must also be running RHEL 8.x . This is because a TPA controller running another OS may be running a newer ansible version than ansible-core 2.16, and therefore be incompatible with RHEL 8.x on the target instances.

RedHat ppc64le

- RHEL/Rocky/AlmaLinux 10.x is fully supported (python3 only)
- RHEL/Rocky/AlmaLinux 9.x is fully supported (python3 only)
- RHEL/AlmaLinux 8.x is fully supported (python3 only)

SLES x86

- SLES 15 SP7 is supported.

> **Note**
>
> M1 architecture using Repmgr in combination with EDBPGE or EPAS is not supported for SLES. Use community PostgreSQL in this situation.

Platform-specific considerations

Some platforms may not work with the legacy distributions mentioned here. For example, Debian 8 and Ubuntu 16.04 are not available in Docker containers.

34.3 TPA capabilities and supported software

- Python requirements
- Supported distributions

Supported software

TPA can install and configure the following major components.

- Postgres
- EPAS (EDB Postgres Advanced Server)
- PGD 5, 4, 3.7
- pglogical 3, 2 (open source)
- pgd-cli and pgd-proxy
- HARP 2
- repmgr
- Barman
- pgbouncer
- haproxy (supported only for PGD 3.7)
- Failover Manager (EFM)
- Postgres Enterprise Manager (PEM)

34.4 Reconciling changes made outside of TPA

Any changes made to a TPA created cluster that are not performed by changing the TPA configuration will not be saved in `config.yml`. This means that your cluster will have changes that the TPA configuration won't be able to recreate.

This page shows how configuration is managed with TPA and the preferred ways to make configuration changes. We then look at strategies to make, and reconcile, the results of making manual changes to the cluster.

Why might I need to make manual configuration changes?

The most common scenario in which you may need to make configuration changes outside of TPA is if the operation you are performing is not supported by TPA. The two most common such operations are destructive changes, such as removing a node, and upgrading the major version of Postgres.

Destructive changes

In general TPA will not remove previously deployed elements of a cluster, even if these are removed from `config.yml`. This sometimes surprises people because a strictly declarative system should always mutate the deployed artifacts to match the declaration. However, making destructive changes to production database can have serious consequences so it is something we have chosen not to support.

Major-version Postgres upgrades

TPA does not yet provide an automated mechanism for performing major version upgrades of Postgres. Therefore if you need to perform an in-place upgrade on an existing cluster this must be performed using other tools such as pg_upgrade or pgd node upgrade.

What can happen if changes are not reconciled?

A general issue with unreconciled changes is that if you deploy a new cluster using your existing `config.yml`, or provide your `config.yml` to EDB Support in order to reproduce a problem, it will not match the original cluster. In addition, there is potential for operational problems should you wish to use TPA to manage that cluster in future.

The operational impact of unreconciled changes varies depending on the nature of the changes. In particular whether the change is destructive, and whether the change blocks TPA from running by causing an error or invalidating the data in `config.yml`.

Non-destructive, non-blocking changes

Additive changes are often accommodated with no immediate operational issues. Consider manually adding a user. The new user will continue to exist and cause no issues with TPA at all. You may prefer to manage the user through TPA in which case you can declare it in `config.yml` but the existence of a manually-added user will cause no operational issues.

Some manual additions can have more nuanced effects. Take the example of an extension which has been manually added. Because TPA does not make destructive changes, the extension will not be removed when `tpaexec deploy` is next run. **However**, if you made any changes to the Postgres configuration to accommodate the new extension these may be overwritten if you did not make them using one of TPA's supported mechanisms (see below).

Furthermore, TPA will not make any attempt to modify the `config.yml` file to reflect manual changes and the new extension will be omitted from `tpaexec upgrade` which could lead to incompatible software versions existing on the cluster.

Destructive, non-blocking changes

Destructive changes that are easily detected and do not block TPA's operation will simply be undone when `tpaexec deploy` is next run. Consider manually removing an extension. From the perspective of TPA, this situation is indistinguishable from the user adding an extension to the `config.yml` file and running deploy. As such, TPA will add the extension such that the cluster and the `config.yml` are reconciled, albeit in the opposite way to that the user intended.

Similarly, changes made manually to configuration parameters will be undone unless they are:

1. Made in the `conf.d/9999-override.conf` file reserved for manual edits;
2. Made using `ALTER SYSTEM` SQL; or
3. Made natively in TPA by adding `postgres_conf_settings`.

Other than the fact that option 3 is self-documenting and portable, there is no pressing operational reason to reconcile changes made by method 1 or 2.

Destructive, blocking changes

Changes which create a more fundamental mismatch between `config.yml` can block TPA from performing operations. For example if you physically remove a node in a bare metal cluster, attempts by TPA to connect to that node will fail, meaning most TPA operations will exit with an error and you will be unable to manage the cluster with TPA until you reconcile this difference.

How to reconcile configuration changes

In general, the reconciliation process involves modifying `config.yml` such that it describes the current state of the cluster and then running `tpaexec deploy`.

Example: parting a PGD node

Deploy a minimal PGD cluster using the bare architecture and a configure command such as:

```
tpaexec configure mycluster \
  -a PGD-Always-ON \
  --platform bare \
  --edbpge 15 \
  --location-names a \
  --pgd-proxy-routing local
```

Part a node using this SQL, which can be executed from any node:

select * from bdr.part_node('node-2');

Rerun `deploy`. Note that, whilst no errors occur, the node is still parted. This can be verified using the command `pgd show-nodes` on any of the nodes. This is because TPA will not overwrite the metadata which tells PGD the node is parted.

> **Note**
>
> It is not possible to reconcile the `config.yml` with this cluster state because TPA, and indeed PGD itself, has no mechanism to initiate a node in the 'parted' state. In principle you could continue to use TPA to continue this parted cluster, but this is not advisable. In most cases you will wish to continue to fully remove the node and reconcile `config.yaml`.

Example: removing a PGD node completely

The previous example parted a node from the PGD cluster, but left the node itself intact and still managed by TPA in a viable but unreconcilable state.

To completely decommission the node, it is safe to simply turn off the server corresponding to `node-2`. If you attempt to run `deploy` at this stage, it will fail early when it cannot reach the server.

To reconcile this change in `config.yml` simply delete the entry under `instances` corresponding to `node-2`. It will look something like this:

```
- Name: node-2
  public_ip: 44.201.93.236
  private_ip: 172.31.71.186
  location:
a
  node: 2
  role:
  -
bdr
  - pgd-proxy
  vars:
    bdr_child_group: a_subgroup
    bdr_node_options:
      route_priority: 100
```

You can now manage this node as usual using TPA. The original cluster still has metadata that refers to `node-2` as a node whose state is `PARTED`, which is not removed by default as it does not affect cluster functionality.

> **Note**
>
> If you wish to join the original `node-2` back to the cluster after having removed it from `config.yml`, you can do so by restoring the deleted lines of `config.yml`, stopping Postgres, deleting the `PGDATA` directory on that node, and then repeating `tpaexec deploy`. As noted above, TPA will not remove an existing database, even if the corresponding entry is deleted from `config.yml`, so you need to perform this action manually.

Example: changing the superuser password

TPA automatically generates a password for the superuser which you may view using `tpaexec show-password <cluster> <superuser-name>`. If you change the password manually (for example using the `/password` command in psql) you will find that after `tpaexec deploy` is next run, the password has reverted to the one set by TPA. To make the change through TPA, and therefore make it persist across runs of `tpaexec deploy`, you must use the command `tpaexec store-password <cluster> <superuser-name>` to specify the password, then run `tpaexec deploy`. This also applies to any other user created through TPA.

Example: adding or removing an extension

A simple single-node cluster can be deployed with the following `config.yml`.

```yaml
---
architecture: M1
cluster_name: singlenode

cluster_vars:
  postgres_flavour: postgresql
  postgres_version: '15'
  preferred_python_version: python3

instance_defaults:
  image: tpa/debian:11
  platform:
docker
  vars:
    ansible_user: root

instances:
- Name: nodeone
  node: 1
  role:
  - primary
```

You may manually add the pgvector extension by connecting to the node and running `apt install postgresql-15-pgvector` then executing the following SQL command: `CREATE EXTENSION vector;`. This will not cause any operational issues, beyond the fact that `config.yml` no longer describes the cluster as fully as it did previously. However, it is advisable to reconcile `config.yml` (or indeed simply use TPA to add the extension in the first place) by adding the following cluster variables.

```yaml
cluster_vars:
  ...
  extra_postgres_packages:
   common:
   - postgresql-15-
pgvector
  extra_postgres_extensions:
  -
vector
```

After adding this configuration, you may manually remove the extension by executing the SQL command `DROP EXTENSION vector;` and then `apt remove postgresql-15-pgvector`. However if you run `tpaexec deploy` again without reconciling `config.yml`, the extension will be reinstalled. To reconcile `config.yml`, simply remove the lines added previously.

> **Note**
>
> As noted previously, TPA will not honour destructive changes. So simply removing the lines from `config.yml` will not remove the extension. It is necessary to perform this operation manually then reconcile the change.

After a manual major-version Postgres upgrade

When you have used `pg_upgrade`, `pgd node upgrade`, or another out-of-band procedure to upgrade Postgres on the cluster, TPA does not automatically update `config.yml` to reflect the new version. Subsequent `tpaexec deploy` runs will continue to behave as if the cluster were still on the previous version, and any deploy-time settings that depend on the Postgres version (such as `shared_preload_libraries` choices or version-conditional Ansible tasks) will not be re-evaluated.

To reconcile `config.yml` with the cluster's new state:

1. **Update the version variables in `config.yml`**. At minimum, set:

   ```
   cluster_vars:
     postgres_version: '<new-major>'
     postgres_package_version: '<new-package-version>'
   ```

   For PGD clusters, also update `bdr_version` and `bdr_package_version` if BDR was upgraded as part of the same procedure.

2. Re-run `tpaexec provision` so the new versions are written into the inventory:

   ```
   tpaexec provision <cluster>
   ```

3. Re-run `tpaexec deploy` so any version-conditional configuration is regenerated:

   ```
   tpaexec deploy <cluster>
   ```

   `tpaexec deploy` does not perform any package install or upgrade itself — it refuses to install a different version of a package that is already installed. It does, however, refresh configuration files, `shared_preload_libraries` entries, and version-conditional settings that depend on `postgres_version`.

4. Verify TPA's view of the cluster matches reality:

   ```
   tpaexec cmd <cluster> all -m shell -a 'psql -tAc "SELECT version()"'
   tpaexec test <cluster>
   ```

   Every instance should report the new Postgres version, and `tpaexec test` should pass.

> **Note**
>
> This reconciliation procedure assumes the manual upgrade left every instance in a consistent, running state. If only some instances were upgraded, or if any instance is in a partial state, address that first before running `tpaexec deploy`.

34.5 EDB Postgres Distributed configuration

TPA can install and configure EDB Postgres Distributed (PGD), formerly known as BDR (Bi-directional replication) versions 3.7, 4.x, 5.x. and 6.x.

Access to PGD packages is through EDB's package repositories only. You must have a valid EDB subscription token to download the packages.

This documentation touches on several aspects of PGD configuration, but we refer you to the PGD documentation for an authoritative description of the details.

Introduction

TPA will install PGD and any dependencies on all PGD instances along with Postgres itself.

After completing the basic Postgres setup and starting Postgres, TPA will then create the `bdr_database` and proceed to set up a PGD cluster through the various steps described below.

Installation

TPA will install the correct PGD packages, depending on the version and flavour of Postgres in use (e.g., Postgres, Postgres Extended, or EPAS).

Set `bdr_version` to determine which major version of PGD to install (i.e., 3, 4, 5, 6). Set `bdr_package_version` to determine which exact package to install (e.g., '5.0*' to install the latest 5.0.x).

Overview of cluster setup

After installing the required packages, configuring Postgres to load PGD, and starting the server, TPA will go on to set up PGD nodes, groups, replication sets, and other resources.

Here's a summary of the steps TPA performs:

- Create a PGD node (using bdr.create_node()) for each participating instance
- Create one or more PGD node groups (using bdr.create_node_group()) depending on `bdr_node_groups`
- Create replication sets, if required, to control exactly which changes are replicated (depending on node group type and memberships, e.g., subscriber-only and witness nodes may need special handling)
- Join the relevant node groups on the individual instances
- Perform additional configuration, such as enabling subgroup RAFT or proxy routing.

(This process involves executing a complex sequence of queries, some on each instance in turn, and others in parallel. To make the steps easier to follow, TPA designates an arbitrary PGD primary instance as the "first_bdr_primary" for the cluster, and uses this instance to execute most of these queries. The instance is otherwise not special, and its identity is not significant to the PGD configuration itself.)

Instance roles

Every instance with `bdr` in its `role` is a PGD instance, and implicitly also a `postgres` server instance.

A PGD instance with `readonly` in its role is a logical standby node (which joins the PGD node group with `pause_in_standby` set), eligible for promotion.

A PGD instance with `subscriber-only` in its role is a subscriber-only node, which receives replicated changes but does not publish them.

A PGD instance with `witness` in its role is a witness node.

Every PGD instance described above is implicitly also a `primary` instance. The exception is an instance with `replica` in its role; that indicates a physical streaming replica of an upstream PGD instance. Such instances are not included in any recommended PGD architecture, and not currently supported by TPA.

Configuration settings

The settings mentioned below should ordinarily be set in `cluster_vars`, so that they are set uniformly for all the PGD instances in the cluster. You can set different values on different instances in some cases (e.g., `bdr_database`), but in other cases, the result is undefined (e.g., all instances must have exactly the same value of `bdr_node_groups`).

We strongly recommend defining your PGD configuration by setting uniform values for the whole cluster under `cluster_vars`.

bdr_database

The `bdr_database` (default: bdrdb) will be initialised with PGD.

bdr_client_dsn_attributes

Any *additional* parameter keywords supported by libpq can be included in `bdr_client_dsn_attributes`.

Do not include `host`, `port`, `dbname` and `user`, as these will already be included in the connection string.

> **When pgd-proxy and pgd-cli are installed**
>
> Since pgd-proxy and pgd-cli are written in Go, they use Go drivers for connecting to Postgres.
>
> These drivers do **not** support the full set of DSN attributes provided by the libpq C library.
>
> If pgd-proxy and/or pgd-cli are installed and `bdr_client_dsn_attributes` includes parameters that are *unsupported* by the Go driver (such as timeout), two new variables must be included in the cluster configuration:
>
> - `pgd_proxy_dsn_attributes`, which is used to create the connection strings in `pgd-proxy-conf`
> - `pgd_cli_dsn_attributes`, which is used to create the connection strings in `pgd-cli-conf`
>
> These two strings must ONLY contain parameter keywords compatible with the Go driver.
>
> If the `bdr_client_dsn_attributes` does not include any unsupported parameters, this can be ignored and the `bdr_client_dsn_attributes` will be included in the connection strings for `pgd-proxy-conf` and `pgd-cli-conf`.

bdr_node_group

The setting of `bdr_node_group` (default: based on the cluster name) identifies which PGD cluster an instance should be a part of. It is also used to identify a particular cluster for external components (e.g., pgd-proxy or harp-proxy).

bdr_node_groups

This is a list of PGD node groups that must be created before the group join stage (if the cluster requires additional subgroups).

In general, `tpaexec configure` will generate an appropriate value based on the selected architecture.

```
cluster_vars:
  bdr_node_groups:
  - name:
topgroup
  - name: abc_subgroup
    node_group_type: data
    parent_group_name:
topgroup
    options:
      location:
abc

…
```

The first entry must be for the cluster's `bdr_node_group`.

Each subsequent entry in the list must specify a `parent_group_name`, and may specify the `node_group_type` (optional).

Each entry may also have an optional key/value mapping of group options. The available options vary by PGD version.

bdr_child_group

If `bdr_child_group` is set for an instance (to the name of a group that is mentioned in `bdr_node_groups`), it will join that group instead of `bdr_node_group`.

bdr_commit_scopes

This is an optional list of commit scopes that must exist in the PGD database (available for PGD 4.1 and above).

```
cluster_vars:
  bdr_commit_scopes:
  - name: somescope
    origin: somegroup
    rule: 'ALL (somegroup) ON received …
`
  - name:
otherscope
    origin: othergroup
    rule:
'…'

…
```

Each entry must specify the `name` of the commit scope, the name of the `origin` group, and the commit scope `rule`. The groups must correspond to entries in `bdr_node_groups`.

If you set `bdr_commit_scopes` explicitly, TPA will create, alter, or drop commit scopes as needed to ensure that the database matches the configuration. If you do not set it, it will leave existing commit scopes alone.

preferred_first_bdr_primary

This is the name of the instance that TPA will use for PGD-related tasks that only run on a single node. This includes executing DDL/DML, which is then propagated to the other PGD nodes by replication. It also includes reading data for initializing a new PGD node. If not set, or set to an instance which is not usable, TPA will choose an instance arbitrarily from the suitable candidates if any.

The instance selected by TPA is stored in the fact `first_bdr_primary`. The list of viable instances is stored in the fact `first_bdr_primary_candidates`. Neither of these facts can be set from `config.yml`, they are noted here for use in hooks or debugging.

> **Note**
>
> The choice of `preferred_first_bdr_primary` does not influence which nodes are elected as write leaders in the deployed cluster.

Advanced use of preferred_first_bdr_primary

If you wish to set `preferred_first_bdr_primary` for a single operation, you can pass it using the Ansible `-e` option rather than specifying it in `config.yml`. For example:

```
tpaexec deploy . -e preferred_first_bdr_primary=instance_name
```

This can be useful if you are adding or rebuilding a PGD node and wish to ensure that data is clone from a node in the same data centre for example.

Similarly, you can set `preferred_first_bdr_primary` as an instance variable, with different instances on each node. For example, this configuration would mean that if `node1` was rebuilt, the `tpaexec deploy` command would prefer `node2` as the source of data with which to rebuild it.

```
instances:
  - Name: node1

...
    instance_vars:
      preferred_first_bdr_primary: node2
```

If you use this configuration, it should be applied after the node is initially deployed; having multiple values of `preferred_first_bdr_primary` during a first deploy is not recommended.

Miscellaneous notes

Hooks

TPA invokes the bdr-node-pre-creation, bdr-post-group-creation, and bdr-pre-group-join hooks during the PGD cluster setup process.

Database collations

TPA checks that the PGD database on every instance in a cluster has the same collation (LC_COLLATE) setting. Having different collations in databases in the same PGD cluster is a data loss risk.

Older versions of PGD

TPA no longer actively supports or tests the deployment of BDR v1 (with a patched version of Postgres 9.4), v2 (with Postgres 9.6), or any PGD versions below v3.7.

34.6 Barman

When an instance is given the `barman` role in config.yml, TPA will configure it as a Barman server to take backups of any other instances that name it in their `backup` setting.

```
instances:
- Name:
one
  backup:
two

…

- Name:
two
  role:
  -
barman

…
```

Multiple `postgres` instances can have the same Barman server named as their `backup`; equally, one `postgres` instance can have a list of Barman servers named as its `backup` and backups will be taken to all of the named servers.

The default Barman configuration will connect to PostgreSQL using pg_receivewal to take continuous backups of WAL, and will take a full backup of the instance using rsync over ssh twice weekly. Full backups and WAL are retained for long enough to enable recovery to any point in the last 4 weeks.

Barman package version

By default, TPA installs the latest available version of Barman.

The version of the Barman package that is installed can be specified by including `barman_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    barman_package_version: '1.56.2-1'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

WAL restore compression flag

TPA automatically detects the installed Barman version and selects the correct compression flag for `barman-wal-restore` in the `restore_command` setting:

- Barman >= **3.12**: uses `--keep-compression` (the `-z` option was deprecated in this release)
- Barman < **3.12**: uses `-z`

On first deploy, detection may not be possible because Barman is installed after the `restore_command` is configured. In this case TPA defaults to `--keep-compression`, which matches the latest Barman version installed by default. If you are deploying an older Barman version, set `barman_package_version` explicitly to ensure the correct flag is selected.

Barman configuration

The Barman home directory on the Barman server can be set using the cluster variable `barman_home`; its default value is `/var/lib/barman`.

On each Barman server, a global configuration file is created as `/etc/barman.conf`. This file contains default values for many Barman configuration variables. For each Postgres server being backed up, an additional Barman configuration file is created. For example, to back up the server `one`, the file is `/etc/barman.d/one.conf`, and the backups are stored in the subdirectory `one` in the Barman home directory. The configuration file and directory names can be changed from the backed-up instance's `backup_name` setting defined on the `vars` section before the provisioning step.

```
- Name: myPrimary
  backup:
myBarman
  platform: bare
  ip_address: x.x.x.x
  node: 1
  role:
  - primary
  vars:
    backup_name: my_backup
```

The following variables can be set on the backed-up instance and are passed through into Barman's configuration with the prefix `barman_` removed:

variable	default
barman_archiver	false
barman_log_file	/var/log/barman.log
barman_backup_method	rsync
barman_compression	pigz
barman_reuse_backup	link
barman_parallel_jobs	1
barman_backup_options	concurrent_backup
barman_immediate_checkpoint	false
barman_network_compression	false
barman_basebackup_retry_times	3
barman_basebackup_retry_sleep	30
barman_minimum_redundancy	3
barman_retention_policy	RECOVERY WINDOW OF 4 WEEKS
barman_last_backup_maximum_age	1 WEEK
barman_pre_archive_retry_script	
barman_post_backup_retry_script	
barman_post_backup_script	
barman_streaming_wals_directory	
barman_path_prefix	*backed up node's postgres bin dir*
backup_name	*backed up instance's name*

Backup scheduling

TPA installs a cron job in `/etc/cron.d/barman` which will run every minute and invoke `barman cron` to perform maintenance tasks.

For each instance being backed up, it installs another cron job in `/etc/cron.d/<backup_name>` which takes the backups of that instance. This job runs as determined by the `barman_backup_interval` variable for the instance, with the default being to take backups at 04:00 every Wednesday and Saturday.

SSH keys

TPA will generate ssh key pairs for the `postgres` and `barman` users and install them into the respective ~/.ssh directories, and add them to each other's authorized_keys file. The postgres user must be able to ssh to the barman server in order to archive WAL segments (if configured), and the barman user must be able to ssh to the Postgres instance to take or restore backups.

`barman` and `barman_role` Postgres users

TPA will create two Postgres users, `barman` and `barman_role`.

TPA versions `<23.35` created the `barman` Postgres user as a `superuser`.

Beginning with `23.35` the `barman` user is created with `NOSUPERUSER`, so any re-deploys on existing clusters will remove the `superuser` attribute from the `barman` Postgres user. Instead, the `barman_role` is granted the required set of privileges and the `barman` user is granted `barman_role` membership.

This avoids granting the `superuser` attribute to the `barman` user, using the set of privileges provided in the Barman Manual.

Shared Barman server

> **Note**
>
> To use the shared Barman functionality with clusters created using a TPA version earlier than 23.35, you must: a) upgrade to a version of TPA that supports creating shared Barman instances. b) after upgrading TPA, run deploy on $first-cluster so TPA can make necessary config changes for subsequent clusters to run smoothly against the shared Barman node.

Some deployments may want to share a single Barman server for multiple clusters. Shared Barman server deployment within tpaexec is supported via the `barman_shared` setting that can be set via `vars:` under the Barman server instance for the given cluster config that plans to use an existing Barman server. `barman_shared` is a boolean variable so possible values are true and false(default). When making any changes to the Barman config in a shared scenario, you must ensure that configurations across multiple clusters remain in sync so as to avoid a scenario where one cluster adds a specific configuration and a second cluster overrides it.

A typical workflow for using a shared Barman server across multiple clusters is described below.

1. Create a TPA cluster with an instance that has `barman` role (call it 'first-cluster' for this example).

2. In the second cluster (second-cluster for example), reference this particular Barman instance from $clusters/first-cluster as a shared Barman server instance and use `bare` as platform so we are not trying to create a new Barman instance when running provision. Also specify the IP address of the Barman instance that this cluster can use to access it.

```
- Name:
myBarman
  node: 5
  role:
      -
barman
  platform: bare
  ip_address: x.x.x.x
  vars:
      barman_shared: true
```

3. Once the second-cluster is provisioned but before running deploy, make sure that it can access the Barman server instance via ssh. You can allow this access by copying second-cluster's public key to Barman server instance via `ssh-copy-id` and then do an ssh to make sure you can login without having to specify the password.

```
# add first-cluster's key to the ssh-agent
cd $clusters/first-cluster
ssh-add id_first-clutser
cd $clusters/second-cluster
ssh-keyscan -t rsa,ecdsa -4 $barman-server-ip >> tpa_known_hosts
ssh-copy-id -i id_second-cluster.pub -o 'UserKnownHostsFile=tpa_known_hosts' $user@$barman-server-ip
ssh -F ssh_config $barman-server
```

4. Copy the Barman user's keys from first-cluster to second-cluster

```
mkdir $clusters/second-cluster/keys
cp $clusters/first-cluster/keys/id_barman* clusters/second-cluster/keys
```

5. Run `tpaexec deploy $clusters/second-cluster`

> **Mixed-platform clusters**
>
> By declaring the shared Barman instance as `platform: bare` you might have changed your cluster to a mixed-platform cluster. This may require you to adjust other parts of `config.yml` to accommodate this change. Specifically, the `instance_defaults` section must only contain settings which are applicable to all instances in the cluster. If, for example, your `instance_defaults` contains a setting such as `type` which is only valid for `platform: aws` you must move that setting out of `instance_defaults` and into only the instances which use the AWS platform.

Backing up nodes at different PostgreSQL versions

A single Barman server can back up Postgres instances that run different PostgreSQL major versions from itself.

The per-server configuration file in `/etc/barman.d/<backup>.conf` sets `path_prefix` from the backed up node's own `postgres_bin_dir`, so Barman uses the correct client binaries (such as `pg_basebackup` and `pg_receivewal`) for each node it backs up. The global `path_prefix` in `/etc/barman.conf` continues to point at the Barman host's own bin directory and serves as the default for same-version scenarios.

When the backed up node's `postgres_version` differs from the Barman host's, TPA installs the matching Postgres client packages on the Barman host automatically (via `delegate_to`), so the per-server `path_prefix` resolves to real binaries.

For this to work the Barman host must have package repository access for the additional PostgreSQL version. EDB enterprise repositories (`edb_repositories: ["enterprise"]`) carry all supported major versions and satisfy this requirement; in PGDG-only deployments you must enable the corresponding `pgdgNN` repository entry on the Barman host yourself, because TPA's `sys/repositories` role disables PGDG entries for every major version other than the host's own `postgres_version` .

To override the resolved path on a particular node, set `barman_path_prefix` in that node's `vars` (see the variables table above).

Air-gapped clusters

In an air-gapped cluster (`use_local_repo_only: true`), the cross-version client install on the Barman host requires the matching Postgres client packages to already exist in the Barman cluster's `local-repo` . `tpaexec download-packages` fetches packages for a single Postgres major per cluster, so by default the additional-major client packages are not pre-fetched.

If you intend to back up nodes at additional Postgres majors in an air-gapped Shared Barman setup, copy the additional-major client packages into the Barman cluster's `local-repo` before deploying the cluster that introduces the additional version.

See Managing clusters in a disconnected or air-gapped environment for the directory layout and the `tpaexec download-packages` reference for how to populate a `local-repo` .

Special considerations for shared Barman servers

You must use caution when setting up clusters that share a Barman server instance. There are a number of important aspects you must consider before attempting such a setup.

1. Make sure that no two instances in any of the clusters sharing a Barman server use the same name. The `--cluster-prefixed-hostnames` option of `tpaexec configure` may be helpful in this respect.

2. Barman configuration and settings otherwise should remain in sync in all the clusters using a common Barman server to avoid a scenario where one cluster sets up a specific configuration and the others do not either because the configuration is missing or uses a different value.

3. The version of Postgres on instances being backed up across different clusters needs to be the same.

4. Different clusters using a common Barman server cannot specify different versions of Barman packages when attempting to override the default.

Some of these may be addressed in a future release as we continue to improve the shared Barman server support.

> **Warning**
>
> Be extremely careful when deprovisioning clusters sharing a common Barman node. Especially where the first cluster that deployed Barman uses non-bare platform. Deprovisioning the first cluster that originally provisioned and deployed Barman will effectively leave other clusters sharing the Barman node in an inconsistent state because the Barman node will already have been deprovisioned by the first cluster and it won't exist anymore.

Minor update using `tpaexec upgrade`

When trying to upgrade to a specific package version, ensure the `barman_package_version` in `config.yml` is updated to reflect the desired version. The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
-e barman_package_version="<desired version>"
```

Refer to the section on package version selection and upgrade for more information.

To select Barman for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `barman` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.7 Configuring EFM

TPA will install and configure EFM when `failover_manager` is set to `efm`.

The version of EFM to be used can be configured by changing the `efm_version`:

```
cluster_vars:
…
  failover_manager:
efm
  efm_version: 5.2
…
```

It can also be defined when executing `tpaexec configure`:

```
tpaexec configure my-cluster-dir -a M1 --failover_manager efm --efm-version 5.2
```

Note that EFM is only available via EDB's package repositories and requires a valid subscription.

EFM configuration

TPA will generate `efm.nodes` and `efm.properties` with the appropriate instance-specific settings, with remaining settings set to the respective default values. TPA will also place an `efm.notification.sh` script which basically contains nothing by default and leaves it up to the user to fill it in however they want. TPA will override the default settings for `auto.allow.hosts` and `stable.nodes.file` to simplify adding agents to the cluster.

See the EFM documentation for more details on EFM configuration.

efm_user_password_encryption

Must be either `scram-sha-256` or `md5`

Set `efm_user_password_encryption` to control the `auth-method` for the `efm` Postgres user's `auth-method` in `pg_hba.conf` as well as the algorithm used when generating its encrypted password.

> **Note**
>
> The default value for `efm_user_password_encryption` is `scram-sha-256` for new clusters.

```
efm_user_password_encryption: 'scram-sha-256' # or can be set to
`md5`
```

efm_conf_settings

You can use `efm_conf_settings` to set specific parameters. These must be written as entries in an Ansible dictionary, in `key: value` form

See the documentation on the `efm.properties` file for details on which settings can be configured.

```
cluster_vars:
  efm_conf_settings:
     notification.level: WARNING
     ping.server.ip: <well known address in
network>
```

If you make changes to values under `efm_conf_settings`, TPA will always restart EFM to activate the changes.

efm_user_is_superuser

By default, when deploying on a cluster, TPA creates the `efm` user with superuser privileges. You can disable this behaviour by adding the `efm_user_is_superuser` option to `cluster_vars` in `config.yml`:

```
cluster_vars:
  efm_user_is_superuser: false
```

This way, EFM will create the `efm` user with the minimum privileges required for EFM to operate (see the EFM documentation). You can change EFM's user privileges by switching this variable from `true` to `false` and viceversa via `tpaexec deploy`.

EFM witness

TPA will install and configure EFM as witness on instances whose `role` contains `efm-witness` (when using EFM, you can also use the role `witness`, as it gets implicitly converted to `efm-witness`). For such instances the `upstream` property must be specified to point to the designated primary database instance.

Repmgr

EFM works as a failover manager and therefore TPA will still install repmgr for setting up postgresql replicas on postgres versions 11 and below. `repmgrd` i.e. repmgr's daemon remains disabled in this case and repmgr's only job is to provided replication setup functionality.

For postgres versions 12 and above, any cluster that uses EFM will use `pg_basebackup` to create standby nodes and not use repmgr in any form.

Node Promotability

TPA determines whether a node is eligible for promotion by EFM during failover based on the node's role and replication topology. The following rules are applied when generating the EFM configuration:

- **Witness nodes** (`efm-witness` role) are never promotable.
- **Nodes with the `efm-not-promotable` role** are not eligible for promotion. This can be used to prevent specific standbys, such as DR or reporting nodes, from being promoted to primary.
- **Cascading standbys** (nodes that are not directly replicating from the primary) are also not promotable.
- All other nodes are considered promotable by default.

To explicitly prevent a standby from being promoted, add `efm-not-promotable` to the node’s `roles` list in your cluster configuration. This ensures that EFM will not attempt to promote this node during failover events.

Minor update for EFM using `tpaexec upgrade`

> **EFM is *not* a package update**
>
> Due to how EFM versions are released, each EFM minor version has its own package (`edb-efm49` for v4.9, `edb-efm50` for v5.0, etc.) which installs its own distinct binary and configuration directories.

When upgrading EFM, set the `efm_version` in `config.yml` to reflect the desired version.

TPA installs the package for the new version and copies over the configuration files from the existing EFM version's configuration directory into the new EFM version's configuration directory, and removes the existing EFM version's package, service file and binary directory for cleanup.

Because EFM upgrade depends on a previous version being installed and configured (the source), TPA first checks to ensure this is true. In the event a source EFM version is not installed (it's binary directory does not exist) or not configured (it's configuration directory does not exist), the upgrade will exit with an error. When running the upgrade, if the target EFM version (specified by `efm_version` in the `config.yml`) already has a binary directory and configuration directory, upgrade skips over EFM as it is already installed and configured to the desired version.

> **Note**
>
> EFM upgrade is supported from EFM 4.9 to 5.0 even if not a minor upgrade per say, no additional steps are required for this upgrade to happen.

To select EFM for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `efm` (or `all`)

Refer to the section on component selection for upgrade for more information.

> **Note**
>
> EFM can't work in a mixed version environment. Due to this limitation, there is no good reason to support a partial upgrade of the EFM component. TPA will **always** upgrade the EFM software on the complete set of EFM nodes when `efm` is part of the list of components to upgrade. TPA will ignore `update_hosts` when deciding which nodes should have EFM upgraded.

34.8 Configuring haproxy

TPA will install and configure haproxy on instances whose `role` contains `haproxy`.

By default, haproxy listens on `127.0.0.1:5432` for requests forwarded by `pgbouncer` running on the same instance. You must specify a list of `haproxy_backend_servers` to forward requests to.

TPA will install the latest available version of haproxy by default. You can install a specific version instead by setting `haproxy_package_version: 1.9.15*` (for example).

Note: see limitations of using wildcards in package_version in tpaexec-configure.

Haproxy packages are selected according to the type of architecture. An EDB managed haproxy package may be used but requires a subscription. Packages from PGDG extras repo can be installed if required.

You can set the following variables on any `haproxy` instance.

Variable	Default value	Description
`haproxy_bind_address`	127.0.0.1*	The address to which haproxy should bind *0.0.0.0 if `failover_manager` is patroni. **Users should change this value to something more restrictive and appropriate for their cluster networking**
`haproxy_port`	5432 (5444 for EPAS)	The TCP port haproxy should listen on
`haproxy_read_only_port`	5433 (5445 for EPAS)	TCP port for read-only load-balancer
`haproxy_backend_servers`	None	A list of Postgres instance names
`haproxy_maxconn`	`max_connections` ×0.9	The maximum number of connections allowed per backend server; the default is derived from the backend's `max_connections` setting
`haproxy_peer_enabled`	True*	Add known haproxy hosts as `peer` list. * `False` if `failover_manager` is harp or patroni.

Read-Only load-balancer

Haproxy can be configured to listen on an additional port for read-only access to the database. At the moment this is only supported with the Patroni failover manager. The backend health check determines which postgres instances are currently acting as replicas and will send traffic using a roundrobin load balancing algorithm.

The read-only load balancer is disabled by default but can be turned on using the cluster_vars variable `haproxy_read_only_load_balancer_enabled`.

Patroni Health Checks

When Patroni is used as the failover manager, HAProxy performs health checks against the Patroni REST API. TPA automatically configures these checks based on Patroni's security settings:

- **HTTP:** If `patroni_ssl_enabled` is `false`, standard HTTP checks (`option httpchk`) are used against the Patroni API port (default 8008).
- **HTTPS (Basic Auth or no Auth):** If `patroni_ssl_enabled` is `true` and `patroni_authentication_mode` is `basic`, HAProxy uses `check-ssl` and is configured to trust the cluster's CA certificate to validate the Patroni API server certificate (using `verify required ca-file <path>`).

- **HTTPS (mTLS):** If `patroni_authentication_mode` is `mtls`, HAProxy uses `check-ssl` and is configured with the cluster's CA certificate *and* a unique client certificate/key pair for authentication (using `verify required ca-file <path> crt <path>`). TPA automatically generates and deploys these client certificates for HAProxy.

Server options

TPA will generate `/etc/haproxy/haproxy.cfg` with a backend that has a `default-server` line and one line per backend server. All but the first one will be marked as "backup" servers.

Set `haproxy_default_server_extra_options` to a list of options on the haproxy instance to add options to the `default-server` line; and set `haproxy_server_options` to a list of options on the backend server to add options (which will override the defaults) to the individual server lines for each backend. TPA automatically adds necessary `check` and `check-ssl` options based on the failover manager configuration.

The size of the RSA key size used for ssl-dh-param-file can be modified by adding the variable `ha_proxy_dhparams_key_size` in the `cluster_vars` section.

Example

```
cluster_vars:
  ha_proxy_dhparams_key_size: 4096
instances:
- Name:
one
  vars:
    haproxy_server_options:
    - maxconn 33
- Name:
two
…
- Name: proxy
  role:
  - haproxy
  vars:
    haproxy_backend_servers:
    -
one
    -
two
    haproxy_default_server_extra_options:
    - on-error mark-
down
    - on-marked-down shutdown-
sessions
```


34.9 Configuring HARP

TPA will install and configure HARP when `failover_manager` is set to `harp`, which is the default for BDR-Always-ON clusters.

Installing HARP

You must provide the `harp-manager` and `harp-proxy` packages. Please contact EDB to obtain access to these packages.

Variables for HARP configuration

See the HARP documentation for more details on HARP configuration.

Variable	Default value	Description
`cluster_name`	``	The name of the cluster.
`harp_consensus_protocol`	``	The consensus layer to use (`etcd` or `bdr`)
`harp_location`	`location`	The location of this instance (defaults to the `location` parameter)
`harp_ready_status_duration`	`10`	Amount of time in seconds the node's readiness status will persist if not refreshed.
`harp_leader_lease_duration`	`6`	Amount of time in seconds the Lead Master lease will persist if not refreshed.
`harp_lease_refresh_interval`	`2000`	Amount of time in milliseconds between refreshes of the Lead Master lease.
`harp_dcs_reconnect_interval`	`1000`	The interval, measured in ms, between attempts that a disconnected node tries to reconnect to the DCS.
`harp_dcs_priority`	`500`	In the case two nodes have an equal amount of lag and other qualified criteria to take the Lead Master lease, this acts as an additional ranking value to prioritize one node over another.
`harp_stop_database_when_fenced`	`false`	Rather than simply removing a node from all possible routing, stop the database on a node when it is fenced.
`harp_fenced_node_on_dcs_failure`	`false`	If HARP is unable to reach the DCS then fence the node.
`harp_maximum_lag`	`1048576`	Highest allowable variance (in bytes) between last recorded LSN of previous Lead Master and this node before being allowed to take the Lead Master lock.
`harp_maximum_camo_lag`	`1048576`	Highest allowable variance (in bytes) between last received LSN and applied LSN between this node and its CAMO partner(s).
`harp_camo_enforcement`	`lag_only`	Whether CAMO queue state should be strictly enforced.
`harp_use_unix_sock`	`false`	Use unix domain socket for manager database access.
`harp_request_timeout`	`250`	Time in milliseconds to allow a query to the DCS to succeed.
`harp_watch_poll_interval`	`500`	Milliseconds to sleep between polling DCS. Only applies when `harp_consensus_protocol` is `bdr`.
`harp_proxy_timeout`	`1`	Builtin proxy connection timeout, in seconds, to Lead Master.

Variable	Default value	Description
`harp_proxy_keepalive`	`5`	Amount of time builtin proxy will wait on an idle connection to the Lead Master before sending a keepalive ping.
`harp_proxy_max_client_conn`	`75`	Maximum number of client connections accepted by harp-proxy (`max_client_conn`)
`harp_ssl_password_command`	None	a custom command that should receive the obfuscated sslpassword in the stdin and provide the handled sslpassword via stdout.
`harp_db_request_timeout`	`10s`	similar to dcs -> request_timeout, but for connection to the database itself.
`harp_local_etcd_only`	None	limit harp manager endpoints list to only contain the local etcd node instead of all etcd nodes

You can use the harp-config hook to execute tasks after the HARP configuration files have been installed (e.g., to install additional configuration files).

Consensus layer

The `--harp-consensus-protocol` argument to `tpaexec configure` is mandatory for the BDR-Always-ON architecture.

etcd

If the `--harp-consensus-protocol etcd` option is given to `tpaexec configure` , then TPA will set `harp_consensus_protocol` to `etcd` in config.yml and give the `etcd` role to a suitable subset of the instances, depending on your chosen layout.

When using BDR 4 with HARP and `harp_consensus_protocol: etcd` , at least one instance must have the `etcd` role defined in config.yml. Typically, the `etcd` role is added to BDR primary instances. TPA will validate this requirement during provision and fail with a clear error message if no etcd roles are found.

HARP v2 requires etcd v3.5.0 or above, which is available in the products/harp/release package repositories provided by EDB.

You can configure the following parameters for etcd:

Variable	Default value	Description
etcd_peer_port	2380	The port used by etcd for peer communication
etcd_client_port	2379	The port used by clients to connect to etcd

bdr

If the `--harp-consensus-protocol bdr` option is given to `tpaexec configure` , then TPA will set `harp_consensus_protocol` to `bdr` in config.yml. In this case the existing PGD instances will be used for consensus, and no further configuration is required.

Configuring a separate user for harp proxy

If you want harp proxy to use a separate readonly user, you can specify that by setting `harp_dcs_user: username` under cluster_vars. TPA will use `harp_dcs_user` setting to create a readonly user and set it up in the DCS configuration.

Configuring a separate user for harp manager

If you want harp manager to use a separate user, you can specify that by setting `harp_manager_user: username` under `cluster_vars`. TPAexec will use that setting to create a new user and grant it the `bdr_superuser` role.

Custom SSL password command

The command provided by `harp_ssl_password_command` will be used by HARP to de-obfuscate the `sslpassword` given in connection string. If `sslpassword` is not present then `harp_ssl_password_command` is ignored. If `sslpassword` is not obfuscated then `harp_ssl_password_command` is not required and should not be specified.

Configuring the harp service

You can configure the following parameters for the harp service:

Variable	Default value	Description
`harp_manager_restart_on_failure`	`false`	If `true`, the `harp-manager` service is overridden so it's restarted on failure. The default is `false` to comply with the service installed by the `harp-manager` package.

Configuring harp http(s) health probes

You can enable and configure the http(s) service for harp that will provide api endpoints to monitor service's health.

Variable	Default value	Description

`harp_http_options`	`enable: false`
	`secure: false`
	`host: <inventory_hostname>`
	`port: 8080`
	`probes:`
	` timeout: 10s`
	`endpoint: "host=<proxy_name> port=<6432> dbname=<bdrdb> user=<username>"`
		Configure the http section of harp config.yml that defines the http(s) api settings.

The variable can contain these keys:

```
enable: false
secure: false
cert_file: "/etc/tpa/harp_proxy/harp_proxy.crt"
key_file: "/etc/tpa/harp_proxy/harp_proxy.key"
host: <inventory_hostname>
port: 8080
probes:
  timeout:
10s
endpoint: "<valid dsn>"
```

The `cert_file` and `key_file` keys are both required if you use `secure: true` and are willing to use your own certificate and key.

You must ensure that both certificate and key are available at the given location on the target node before running `deploy`.

Leave both `cert_file` and `key_file` empty if you want TPA to generate a certificate and key for you using a cluster specific CA certificate. TPA CA certificate won't be 'well known', you will need to add this certificate to the trust store of each machine that will probe the endpoints. The CA certificate can be found on the cluster directory on the TPA node at: `<cluster_dir>/ssl/CA.crt` after `deploy`.

see harp documentation for more information on the available api endpoints.

34.10 Configuring Postgres Enterprise Manager (PEM)

TPA will install and configure PEM when `tpaexec configure` command is run with `--enable-pem` command line option.

The default behavior with `--enable-pem` is to enable `pem-agent` role for all `postgres` instances in the cluster. `pem-agent` role will also be added to barman nodes when `--enable-pg-backup-api` command line option is used alongside `--enable-pem`.

A dedicated instance named `pemserver` will also be added to the cluster.

Since PEM server uses postgres backend; pemserver instance implicitly uses `postgres` role as well which ensures that pemserver gets a valid postgres cluster configured for use as PEM backend. All configuration options available for a normal postgres instance are valid for PEM's backend postgres instance as well. See following for details:

- Configure pg_hba.conf
- Configure postgresql.conf

Note that PEM is only available via EDB's package repositories and therefore requires a valid subscription.

Supported architectures

PEM is supported with all architectures via the `--enable-pem` configuration command line option, with the exception of the BDR-Always-ON architecture when used with EDB Postgres Extended. You can optionally edit the generated cluster config (config.yml) and assign or remove `pem-agent` role from any postgres instance in the cluster in order to enable or disable PEM there.

PEM component package versions

By default, TPA installs the latest available version of PEM agent and PEM server.

The version of the PEM agent and PEM server packages that are installed can be specified by including `pem_agent_package_version: xxx` and `pem_server_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    pem_agent_package_version: '9.7.0-1.el9'
    pem_server_package_version: '9.7.0-1.el9'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

PEM configuration

TPA will configure pem agents and pem server with the appropriate instance-specific settings, with remaining settings set to the respective default values. Some of the configuration options may be exposed for user configuration at some point in future.

> **PEM components are upgraded independently.**
>
> Passing `pem-server` to `--components` upgrades ONLY the PEM server, whereas passing `pem-agent` upgrades ONLY the PEM agents. If upgrading separately, it is recommended to upgrade the agents before the server. When both are upgraded together, TPA upgrades the agents before the server.

Obtain the password for the web interface by running `tpaexec show-password $clusterdir $user`.

Passing additional options when registering PEM agents

TPA registers each PEM agent in the cluster using the `pemworker` utility's `--register agent` command.

A list of additional registration options can be passed by including `pemagent_registration_opts` in the cluster config.

For example:

```
  pemagent_registration_opts:
  - --enable-smtp
true
  - --enable-heartbeat-
connection
  - --allow-batch-probes
true
  - -l
DEBUG1
```

The PEM documentation lists more information about registration options.

Useful extensions for the nodes with pem agent

By default, TPA will add `sql_profiler`, `edb_wait_states` and `query_advisor` extensions to any instances that have `pem-agent` role.

This list of default extensions for pem-agent nodes can be overriden by setting `pemagent_extensions` in config.yml.

If this list is empty, no extensions will be automatically included.

Providing an external certificate for PEM server SSL authentication

By default, the PEM server creates a self-signed certificate pair, `server-pem.crt` and `server-pem.key` and configures the webserver to use them for HTTPS access.

The size of `server-pem.key` can be modified adding the variable `pem_rsa_key_size` to the `cluster_vars` section:

```
(...)
cluster_vars:
  pem_rsa_key_size: 4096
```

The size of the CA certificate expedited by the PEM database can also be modified adding the variable `pem_db_ca_certificate_key_size` to the `cluster_vars` section:

```
(...)
cluster_vars:
  pem_db_ca_certificate_key_size: 4096
```

By default, the self-signed TLS certificate has a period of 10 years. The period can be adjusted by defining `openssl_certificate_validity` in the `cluster_vars` section. If the certificate needs to be replaced, use the `pem_web_server_renew_tls_certificates` variable on `tpaexec deploy`: `tpaexec deploy (...) -e pem_web_server_renew_tls_certificates=true`.

To provide your own certificate pair, create a directory under the root of the cluster directory named `ssl/pemserver` and place the certificate pair inside.

```
cluster directory
├──
ssl
│   └──
pemserver
│       ├── externally-
provided.crt
│       └── externally-
provided.key
```

Next, set the variables `pem_server_ssl_certificate` and `pem_server_ssl_key` with the respective file names as values for the `vars:` under the pem server instance or `cluster_vars` in the cluster config file.

TPA will handle copying these files over to the pem server instance and configure the webserver accordingly.

```
- Name: pemserver
  location: main
  node: 4
  role:
  - pem-server
  vars:
    pem_server_ssl_certificate: externally-
provided.crt
    pem_server_ssl_key: externally-
provided.key
```

Organising instances in PEM

PEM supports collating servers and/or agents into **groups** and **clusters**. You can specify this grouping from TPA by setting the following instance variables.

On instances with the `pem-agent` role, but without `pem-server` role:

- `monitoring_group` specifies the name of the group to which the Postgres server on this instance will be assigned. If not specified it defaults to the value of `pem_server_group`.
- `monitoring_agent_group` specifies the name of the group to which the PEM agent on this instance will be assigned. If not specified it defaults to the value of `pem_agent_group`.

- `monitoring_cluster` specifies the name of the cluster to which the Postgres server on this instance will be assigned. If not specified, no cluster will be assigned/created.
- `monitoring_agent_cluster` specifies the name of the cluster to which the PEM agent on this instance will be assigned. If not specified, no cluster will be assigned/created.

> **Note**
>
> PEM only permits a given cluster name to exist in a single group, so some combinations of these values are not viable. For example if you specify the same `monitoring_cluster` and `monitoring_agent_cluster`, but different `monitoring_group` and `monitoring_agent_group`, this implies the same cluster name would appear in two different groups. TPA allows PEM to handle such inconsistencies rather than attempting to prevent them. This generally means `deploy` will succeed but you may not get exactly the organisation of servers and agents you expected.

On instances with the `pem-server` role:

- `pem_server_group` specifies the name of the default group to which Postgres servers registered with this PEM Server will be assigned. If not specified it defaults to `PEM Server Directory`.
- `pem_agent_group` specifies the name of the default group to which PEM Agents registered with this PEM Server will be assigned. If not specified it defaults to `PEM Agents`

Note that the Postgres server and PEM Agent on the instance with `pem-server` role are always assigned to `pem_server_group` and `pem_agent_group` respectively. TPA does not support adding this server or agent to a cluster.

Shared PEM server

Some deployments may want to use a single PEM server for monitoring and managing multiple clusters in the organization. Shared pem server deployment within tpaexec is supported via the `pem_shared` variable that you could set via `vars:` under the pem server instance for the given cluster config that plans to use an existing pem server. `pem_shared` is a boolean variable so possible values are true and false(default). When declaring a pemserver instance as shared, we tell the given cluster config that pemserver instance is in fact managed by a separate cluster config that provisioned and deployed the pem server in the first place. So any changes we wanted to make to the pem server instance including postgres backend for pem would be managed by the cluster where pemserver instance is NOT declared as a shared pem instance.

A typical workflow for using a shared pem server across multiple clusters would look something like this:

1. Create a tpaexec cluster with a single instance that has `pem-server` role (call it 'pem-cluster' for this example). We could as easily use the same workflow in a scenario where pem is provisioned as part of a larger cluster and not just a single instance that runs as pemserver but we use a single node cluster because it is easier to use that as an example and arguably easy to maintain as well.

2. In the other cluster (pg-cluster for example), reference this particular pemserver from $clusters/pem-cluster as a shared pem server instance and use `bare` as platform so we are not trying to create a new pemserver instance. Also specify the IP address of the pemserver that this cluster can use to access pemserver instance.

   ```
   - Name: pemserver
     node: 5
     role:
     - pem-server
     platform: bare
     public_ip: 13.213.53.205
     private_ip: 10.33.15.102
     vars:
       pem_shared: true
   ```

3. Before running deploy in the postgres cluster, make sure that pg-cluster can access pem server instance via ssh. You can allow this access by copying pg-cluster's public key to pem server instance via `ssh-copy-id` and then do an ssh to make sure you can login without having to specify the password.

```
# add pem-clusters key to the ssh-agent (handy for `aws`
platform)
$ cd $clusters/pem-cluster
$ ssh-add id_pem-clutser
$ cd $clusters/pg-cluster
$ ssh-keyscan -4 $pem-server-ip >>
known_hosts
$ ssh-copy-id -i id_pg-cluster.pub -o 'UserKnownHostsFile=tpa_known_hosts' $user@$pem-server-
ip
$ ssh -F ssh_config
pemserver
```

4. Update postgresql config on pem server node so it allows connections from the new pg-cluster. You can modify existing pg_hba.conf on pem server by adding new entries to `pem_postgres_extra_hba_settings` under `vars:` in pem-cluster's config.yml. For example:

```
instances:
- Name: pemserver
  location: main
  node: 1
  role:
  - pem-server
  vars:
    pem_postgres_extra_hba_settings:
      - "# Allow pem connections from pg-
cluster1.quire"
      - hostssl pem +pem_agent 10.33.15.108/32
cert
      - "# Allow pem connections from pg-
cluster1.upside"
      - hostssl pem +pem_agent 10.33.15.104/32
cert
      - "# Allow pem connections from pg-
cluster2.zippy"
      - hostssl pem +pem_agent 10.33.15.110/32
cert
      - "# Allow pem connections from pg-
cluster2.utopic"
      - hostssl pem +pem_agent 10.33.15.109/32
cert
```

and then run `tpaexec provision $clusters/pem-cluster` followed by `tpaexec deploy $clusters/pem-cluster`. When complete, nodes from your new pg-cluster should be able to speak with pem server backend.

5. In order to make sure pem agents from the nodes in pg-cluster can connect and register with the pem server backend, you must first `export EDB_PEM_CREDENTIALS_FILE=/path/to/pem/credentials/file` before you run `tpaexec deploy`. Credentials file is a text file that contains your access credentials to the pemserver's backend postgres instance in the `username:password` format.

```
cat pem_creds
```

```
output
postgres:f1I%fw!QmWevdzw#EL#$Ulu1cWhg7&RT
```

If you don't know the backend password, you can get that by using `show-password` tpaexec command.

```
tpaexec show-password $pem-clusterdir $user
```

6. Run `tpaexec deploy $clusters/pg-cluster` so pem is deployed on the new pg-cluster while using shared pem server instance.

> **Mixed-platform clusters**
>
> By declaring the shared PEM instance as `platform: bare` you might have changed your cluster to a mixed-platform cluster. This may require you to adjust other parts of `config.yml` to accommodate this change. Specifically, the `instance_defaults` section must only contain settings which are applicable to all instances in the cluster. If, for example, your `instance_defaults` contains a setting such as `type` which is only valid for `platform: aws` you must move that setting out of `instance_defaults` and into only the instances which use the AWS platform.

Connecting to the PEM UI

PEM UI runs on https interface so you can connect with a running instance of PEM server via https://$pem-server-ip/pem. Login credentials for PEM UI are set to the postgres backend user which uses `postgres` or `enterprisedb` for `postgresql` and `epas` flavours respectively. tpaexec's show-password command will show the password for the backend user. For example:

```
tpaexec show-password $clusterdir $user
```

See PEM documentation for more details on PEM configuration and usage.

Minor update for PEM using `tpaexec upgrade`

> **PEM major version upgrades from version 9 to version 10.**
>
> ```
> The PEM server package includes an SQL upgrade script which handles upgrading the database
> according to the PEM schema version.
> Upgrading from PEM `v9.8.0` to PEM `10.1.1` using `tpaexec upgrade` has been tested and works, but
> is a major version upgrade.
> ```

> **PEM components are upgraded independently.**
>
> Passing `pem-server` to `--components` upgrades ONLY the PEM server, whereas passing `pem-agent` upgrades ONLY the PEM agents. If upgrading separately, it is recommended to upgrade the agents before the server. When both are upgraded together, TPA upgrades the agents before the server.

When trying to upgrade the PEM agents to a specific package version, ensure the `pem_agent_package_version` in `config.yml` is updated to reflect the desired version.

When trying to upgrade the PEM server to a specific package version, ensure the `pem_server_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> \
  -e pem_agent_package_version="<desired version>" \
  -e pem_server_package_version="<desired version>" \
  --components=pem-agent,pem-server
```

Refer to the section on package version selection and upgrade for more information.

To select PEM agents for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pem-agent` (or `all`)

To select the PEM server for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pem-server` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.11 PgBouncer

PgBouncer package version

By default, TPA installs the latest available version of PgBouncer.

The version of the PgBouncer package that is installed can be specified by including `pgbouncer_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:
…
    pgbouncer_package_version: '1.8*'
…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

Configuring PgBouncer

TPA will install and configure PgBouncer on instances whose `role` contains `pgbouncer`.

By default, PgBouncer listens for connections on port 6432 and, if no `pgbouncer_backend` is specified, forwards connections to `127.0.0.1:5432` (which may be either Postgres or haproxy, depending on the architecture).

> **Using PgBouncer to route traffic to the primary**
>
> If you are using the M1 architecture with repmgr you can set `repmgr_redirect_pgbouncer: true` hash under `cluster_vars` to have PgBouncer connections directed to the primary. The PgBouncer will be automatically updated on failover to route to the new primary. You should use this option in combination with setting `pgbouncer_backend` to the primary instance name to ensure that the cluster is initially deployed with PgBouncer configured to route to the primary.

You can set the following variables on any `pgbouncer` instance.

Variable	Default value	Description
`pgbouncer_port`	6432	The TCP port pgbouncer should listen on
`pgbouncer_backend`	127.0.0.1	A Postgres server to connect to
`pgbouncer_backend_port`	5432	The port that the `pgbouncer_backend` listens on
`pgbouncer_max_client_conn`	`max_connections` ×0.	The maximum number of connections allowed; the default is derived from the backend's `max_connections` setting if possible
`pgbouncer_auth_user`	pgbouncer_auth_user	Postgres user to use for authentication

Databases

By default, TPA will generate `/etc/pgbouncer/pgbouncer.databases.ini` with a single wildcard `*` entry under `[databases]` to forward all connections to the backend server. You can set `pgbouncer_databases` as shown in the example below to change the database configuration.

Authentication

PgBouncer will connect to Postgres as the `pgbouncer_auth_user` and execute the (already configured) `auth_query` to authenticate users.

The `pgbouncer_get_auth()` function used as the `auth_query` by PgBouncer is created in a single database, the `pgbouncer_auth_database`. Execute permissions are granted on this function to the `pgbouncer_auth_user`.

Example

```
instances:
- Name:
one
  vars:
    max_connections: 300
- Name:
two
- Name: proxy
  role:
  - pgbouncer
  vars:
    pgbouncer_backend:
one
    pgbouncer_databases:
    - name:
dbname
      options:
        pool_mode:
transaction
        dbname: otherdb
    - name: bdrdb
      options:
        host:
two
        port: 6543
```

Minor update for PgBouncer using `tpaexec upgrade`

When trying to upgrade to a specific package version, ensure the `pgbouncer_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> \
  -e pgbouncer_package_version="<desired version>" \
  --components=pgbouncer
```

Refer to the section on package version selection and upgrade for more information.

To select PgBouncer for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pgbouncer` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.12 pgd-proxy

pgd-proxy package version

By default, TPA installs the latest available version of `pgd-proxy`.

The version of the `pgd-proxy` package that is installed can be specified by including `pgd_proxy_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:
…
    pgd_proxy_package_version: '5.0.0-1'
…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

Configuring pgd-proxy

TPA will install and configure pgd-proxy for the PGD-Always-ON architecture with PGD 5 on any instance with `pgd-proxy` in its `role`.

(By default, the PGD-Always-ON architecture will run `pgd-proxy` on all the data nodes in every location, but you can instead create any number of additional proxy instances with `--add-proxy-nodes-per-location 3`.)

Configuration

`pgd-proxy` is configured at PGD level via SQL functions.

Hash	Function	Description
`pgd_proxy_options`	`bdr.alter_proxy_option()`	pgd-proxy configuration, e.g. port
`bdr_node_groups`	`bdr.alter_node_group_option()`	configuration for the proxy's node group, e.g. `enable_proxy_routing`
`bdr_node_options`	`bdr.alter_node_option()`	routing configuration for individual PGD nodes

See the PGD documentation for more details.

You can use the pgd-proxy-config hook to execute tasks after the PGD PROXY configuration files have been installed (e.g., to install additional configuration files).

bdr_node_groups

Group-level options related to pgd-proxy can be set under `bdr_node_groups` along with other node group options:

```
cluster_vars:
  bdr_node_groups:
  - name:
group1
    options:
      enable_proxy_routing: true
```

Note that `enable_proxy_routing` must be explicitly set to `true` for pgd-proxy to be enabled for the group.

bdr_node_options

Node-level options related to pgd-proxy can be set under `bdr_node_options` on any PGD instance:

```
instances:
- Name: first
  vars:
    bdr_node_options:
      route_priority: 42
```

pgd_proxy_options

Options for a pgd-proxy instance itself, rather than the group or nodes it is attached to, can be set under `default_pgd_proxy_options` under `cluster_vars` (which applies to all proxies), or under `pgd_proxy_options` on any pgd-proxy instance:

```
cluster_vars:
  default_pgd_proxy_options:
    listen_port: 6432
    read_listen_port: 6433

instances:
- Name: someproxy
  vars:
    pgd_proxy_options:
      listen_port: 9000
      read_listen_port: 9001
```

In this case, while other instances will get their `listen_port` setting from `cluster_vars`, `someproxy` overrides that default setting and configures its own `listen_port` in the instances' `vars` section.

PGD proxy http(s) health probes

You can enable and configure the http(s) service for PGD proxy that will provide api endpoints to monitor the proxy's health.

`pgd_http_options` under `cluster_vars` or instance `vars` will store all the settings that defines the http(s) api which live under the `http` subsection of the `proxy` top section of `pgd-proxy-config.yml`.

The variable can contain these keys:

```
enable: false
secure: false
cert_file: "/etc/tpa/harp_proxy/harp_proxy.crt"
key_file: "/etc/tpa/harp_proxy/harp_proxy.key"
host: <inventory_hostname>
port: 8080
probes:
  timeout:
10s
endpoint: "<valid dsn>"
```

The `cert_file` and `key_file` keys are both required if you use `secure: true` and are willing to use your own certificate and key.

You must ensure that both certificate and key are available at the given location on the target node before running `deploy`.

Leave both `cert_file` and `key_file` empty if you want TPA to generate a certificate and key for you using a cluster specific CA certificate. TPA CA certificate won't be 'well known', you will need to add this certificate to the trust store of each machine that will probe the endpoints. The CA certificate can be found on the cluster directory on the TPA node at: `<cluster_dir>/ssl/CA.crt` after `deploy`.

see pgd-proxy documentation for more information on the available api endpoints.

Updating pgd-proxy using `tpaexec upgrade`

When trying to upgrade to a specific package version, ensure the `pgd_proxy_package_version` in `config.yml` is updated to reflect the desired version. This version should be kept in line with the `bdr_package_version` and `pgdcli_package_version` versions specified for the cluster. Defining `bdr_package_version` and omitting `pgd_proxy_package_version` and `pgdcli_package_version` from `config.yml` will ensure the same value is used across all three components.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> \
  -e pgd_proxy_package_version="<desired version>" \
  --components=pgd-proxy
```

Refer to the section on package version selection and upgrade for more information.

To select pgd-proxy for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pgd-proxy` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.13 pglogical

pglogical package version

By default, TPA installs the latest available version of `pglogical`.

The version of the `pglogical` package that is installed can be specified by including `pglogical_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    pglogical_package_version: '2.2.0*'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

pglogical configuration

TPA can configure pglogical replication sets (publications) and subscriptions with pglogical v2 and pglogical v3.

```
instances:
- node: 1
  Name: kazoo

…
  vars:
    publications:
    - type: pglogical
      database: example
      name:
some_publication_name
      replication_sets:
      - name: custom_replication_set

…

- node: 2
  Name:
keeper
  vars:
    subscriptions:
    - type: pglogical
      database: example
      name: some_subscription_name
      publication:
        name:
some_publication_name
      replication_sets:
        - default
        -
default_insert_only
        - custom_replication_set

…
```

The pglogical extension will be created by default if you define publications or subscriptions with `type: pglogical`, but it is up to you to determine which version will be installed (e.g., subscribe to the `products/pglogical3/release` repository for pglogical3).

Introduction

TPA can configure everything needed to replicate changes between instances using pglogical, and can also alter the replication setup based on config.yml changes.

To publish changes, you define an entry with `type: pglogical` in `publications`. To subscribe to these changes, you define an entry with `type: pglogical` in `subscriptions`, as shown above.

Pglogical does not have a named publication entity (in the sense that built-in logical replication has `CREATE PUBLICATION`). A publication in config.yml just assigns a name to a collection of replication sets, and subscriptions can use this name to refer to the desired provider.

To use pglogical replication, both publishers and subscribers need a named local pglogical node. TPA will create this node with `pglogical.create_node()` if it does not exist. For publications, the publication name is used as the pglogical node name. There can be only one pglogical node in any given database, so you can have only one entry in `publications` per database.

However, pglogical subscriptions *do* have a name of their own. TPA will create subscriptions with the given `name`, and use a default value for the pglogical node name based on the instance's name and the name of the database in which the subscription is created. You can specify a different `node_name` if required—for example, when you have configured a publication in the same database, so that all subscriptions in that database must share the same pglogical node.

TPA does some basic validation of the configuration—it will point out the error if you spell `replication_sets` as `replciation_sets`, or try to subscribe to a publication that is not defined, but it is your responsibility to specify a meaningful set of publications and subscriptions.

TPA will configure pglogical after creating users, extensions, and databases, but before any PGD configuration. You can set `postgres_users` and `postgres_databases` to create databases for replication, and use the `postgres-config-final` hook to populate the databases before pglogical is configured.

Publications

An entry in `publications` must specify a `name` and `database`, and may specify a list of named `replication_sets` with optional attributes, as well as a list of table or sequence names.

```
publications:
- type: pglogical
  database: example
  name:
some_publication_name
  replication_sets:
  - name: default
    replicate_insert: true
    replicate_update: true
    replicate_delete: true
    replicate_truncate: true
    autoadd_tables: false
    autoadd_sequences: false
    autoadd_existing: true
  - name: custom_replication_set
    tables:
    - name: sometable
    - name: '"some-schema".othertable'
      columns: [a, b,
c]
      row_filter: 'a >
42'
      synchronize_data: true
    sequences:
    - name: someseq
      synchronize_data: true
    - name: '"some-schema".otherseq'
```

Each replication set may specify optional attributes such as `replicate_insert` and `autoadd_existing`. If specified, they will be included as named parameters in the call to `pglogical.create_replication_set()`, otherwise they will be left out and the replication set will be created with pglogical's defaults instead.

Apart from manipulating the list of relations belonging to the replication set using the `autoadd_*` parameters in pglogical3, you can also explicitly specify a list of tables or sequences. The name of each relation may be schema-qualified (unqualified names are assumed to be in `public`), and the entry may include optional attributes such as `row_filter` (for tables only) or `synchronize_data`, as shown above.

Subscriptions

An entry in `subscriptions` must specify a `name` and `database`, define a publication to subscribe to, and may specify other optional attributes of the subscription.

```yaml
subscriptions:
- type: pglogical
  database: example
  name: some_subscription_name
  node_name: optional_pglogical_node_name
  publication:
    name:
some_publication_name
  # Optional
attributes:
  synchronize_structure: true
  synchronize_data: true
  forward_origins: ['all']
  strip_origins: false
  apply_delay: '1 second'
  writer: 'heap'
  writer_options:
    - 'magic'
    - 'key=value'
    - 'just-a-string'
  # Optional attributes that can be changed for an
existing
  #
subscription:
  replication_sets:
    - default
    -
default_insert_only
    - custom_replication_set
  enabled: true
```

A subscription can set `publication.name` (as shown above) to define which publication to subscribe to. If there is more than one publication with that name (across the entire cluster), you may specify the name of an instance to disambiguate. If you want to refer to publications by name, don't create multiple publications with the same name on the same instance.

```yaml
- type: pglogical

…
  publication:
    name:
some_publication_name
    instance: kazoo

  #
OR

  provider_dsn: "host=… dbname=…"
```

Instead of referring to publications by name, you may explicitly specify a `provider_dsn` instead. In this case, the given DSN is passed to `pglogical.create_subscription()` directly (and `publication` is ignored). You can use this mechanism to subscribe to instances outside the TPA cluster.

The other attributes in the example above are optional. If defined, they will be included as named parameters in the call to `pglogical.create_subscription()`, otherwise they will be left out. (Some attributes shown are specific to pglogical3.)

Configuration changes

For publications, you can add or remove replication sets, change the attributes of a replication set, or change its membership (the tables and sequences it contains).

If you change `replicate_*` or `autoadd_*` , TPA will call `pglogical.alter_replication_set()` accordingly (but note that you cannot change `autoadd_existing` for existing replication sets, and the `autoadd_*` parameters are all pglogical3-specific).

If you change the list of `tables` or `sequences` for a replication set, TPA will reconcile these changes by calling `pglogical.alter_replication_set_{add,remove}_{table,sequence}()` as needed.

However, if you change `synchronize_data` or other attributes for a relation (table or sequence) that is already a member of a replication set, TPA will not propagate the changes (e.g., by dropping the table and re-adding it with a different configuration).

For subscriptions, you can only change the list of `replication_sets` and enable or disable the subscription (`enabled: false`).

In both cases, any replication sets that exist but are not mentioned in the configuration will be removed (with `pglogical.alter_subscription_remove_replication_set()` on the subscriber, or `pglogical.drop_replication_set()` on the publisher—but the default replication sets named `default` , `default_insert_only` , and `ddl_sql` will not be dropped.)

If you edit config.yml, remember to run `tpaexec provision` before running `tpaexec deploy` .

Interaction with PGD

It is possible to use PGD and pglogical together in the same database if you exercise caution.

PGD v3 uses pglogical3 internally, and will create a pglogical node if one does not exist. There can be only one pglogical node per database, so if you configure a pglogical publication in `bdr_database` , the instance's `bdr_node_name` must be the same as the publication's `name` . Otherwise, the node will be created for the publication first, and `bdr.create_node()` will fail later with an error about a node name conflict. Any `subscriptions` in `bdr_database` must use the same `node_name` too.

Limitations

- There is currently no support for `pglogical.replication_set_{add,remove}_ddl()`
- There is currently no support for `pglogical.replication_set_add_all_{tables,sequences}()`
- There is currently no support for `pglogical.alter_subscription_{interface,writer_options}()` or `pglogical.alter_subscription_{add,remove}_log()`
- pglogical v1 support is not presently tested.

34.14 repmgr

TPA will install repmgr on all postgres instances that have the `failover_manager` instance variable set to `repmgr`; this is the default setting.

The directory of the `repmgr` configuration file defaults to `/etc/repmgr/<version>`, where `<version>` is the major version of postgres being installed on this instance, but can be changed by setting the `repmgr_conf_dir` variable for the instance. The configuration file itself is always called `repmgr.conf`.

The default repmgr configuration will set up automatic failover between instances configured with the role `primary` and the role `replica`.

repmgr package version

By default, TPA installs the latest available version of repmgr.

The version of the repmgr package that is installed can be specified by including `repmgr_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    repmgr_package_version: '4.0.5-1.pgdg90+1'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

repmgr configuration

The following instance variables can be set:

`repmgr_priority`: sets `priority` in the config file `repmgr_location`: sets `location` in the config file `repmgr_reconnect_attempts`: sets `reconnect_attempts` in the config file, default `6` `repmgr_reconnect_interval`: sets `reconnect_interval` in the config file, default `10` `repmgr_use_slots`: sets `use_replication_slots` in the config file, default `1` `repmgr_failover`: sets `failover` in the config file, default `automatic`

Any extra settings in `repmgr_conf_settings` will also be passed through into the repmgr config file.

repmgr service configuration

`repmgr_service_environment`: sets environment variables to the repmgr service unit file. The environment variables must be defined under the `repmgr_service_environment` variable, in a 'key-value' fashion (see example below). You can use `repmgr_service_environment` to set any parameters, whether recognised by TPA or not. You need to quote the value exactly as it would appear in the `repmgrd.service` file.

```
cluster_vars:
  repmgr_service_environment:
    LD_PRELOAD: '/usr/edb/pge15/lib/libpq.so.5:/usr/edb/pge15/lib/libpqagent86.so'
    AGENT86_SHARD_FILE: '/var/lib/pgsql/shard.dat'
```


On the repmgr service file:

```
root@kaput:~# cat /etc/systemd/system/repmgr.service
[Unit]
Description=Postgres replication manager
After=postgres-monitor.service
Wants=postgres-monitor.service

[Service]
Type=simple
User=postgres
Group=postgres
Environment=AGENT86_SHARD_FILE=/var/lib/pgsql/shard.dat
Environment=LD_PRELOAD=/usr/edb/pge15/lib/libpq.so.5:/usr/edb/pge15/lib/libpqagent86.so
StandardOutput=syslog
ExecStart=/usr/lib/postgresql/17/bin/repmgrd -f /etc/repmgr/17/repmgr.conf --verbose --daemonize=false
ExecStop=/bin/kill -TERM $MAINPID
ExecReload=/bin/kill -HUP $MAINPID
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

repmgr on PGD instances

On PGD instances, `repmgr_failover` will be set to `manual` by default.

Minor update for repmgr using `tpaexec upgrade`

> **`repmgr` minor-versioning is DIFFERENT: the THIRD digit (X.Y.Z) is the minor version**
>
> A minor release upgrade involves updating repmgr from one minor release to another minor release within the same major release (e.g. 5.3.1 to 5.3.2). An upgrade between minor releases of differing major releases (e.g. 5.2.1 to 5.3.2) is a MAJOR upgrade. TPA does NOT support major version upgrades of `repmgr`

When trying to upgrade to a specific package version, ensure the `repmgr_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> \
    -e repmgr_package_version="<desired version>"\
    --components=repmgr
```

Refer to the section on package version selection and upgrade for more information.

To select repmgr for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `repmgr` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.15 Configuring EDB Repos 2.0 repositories

TPA sources EDB software from EDB Repos 2.0. To use EDB Repos 2.0 you must `export EDB_SUBSCRIPTION_TOKEN=xxx` before you run tpaexec. You can get your subscription token from the web interface.

> **Note**
>
> If you create your `config.yml` file using the `tpaexec configure` command, the `edb_repositories` key will be automatically populated with the necessary repositories for your selected configuration, so you shouldn't need to edit it.

To specify the complete list of repositories from EDB Repos 2.0 to install on each instance, set `edb_repositories` to a list of EDB repository names:

```
cluster_vars:
  edb_repositories:
    - enterprise
    - postgres_distributed
```

This example will install the 'enterprise' subscription repository as well as 'postgres_distributed' giving access to EPAS and PGD products. On Debian or Ubuntu systems, it will use the APT repository and on RedHat or SLES systems, it will use the rpm repositories, through the yum or zypper frontends respectively.

34.16 Configuring APT repositories

This page explains how to configure APT package repositories on Debian and Ubuntu systems.

You can define named repositories in `apt_repositories`, and decide which ones to use by listing the names in `apt_repository_list`:

```
cluster_vars:
  apt_repositories:
    Example:
      key_id:
XXXXXXXX
      key_url: https://repo.example.com/path/to/XXXXXXXX.asc
      repo: >-
        deb https://repo.example.com/repos/Example/ xxx-Example
main

  apt_repository_list:
    - PGDG
    - Example
```

This configuration would install the GPG key (with id `key_id`, obtained from `key_url`) and a new entry under `/etc/apt/sources.list.d` with the given `repo` line (or lines) for the PGDG repository (which is already defined by default) and the new Example repository.

When you configure additional repositories, remember to include PGDG in `apt_repository_list` if you still want to install PGDG packages.

You can set `apt_repository_list: []` to not install any repositories.

34.17 Configuring YUM repositories

This page explains how to configure YUM package repositories on RedHat systems.

You can define named repositories in `yum_repositories`, and decide which ones to use by listing the names in `yum_repository_list`:

```
cluster_vars:
  yum_repositories:
    Example:
      rpm_url: >-
        https://repo.example.com/repos/Example/example-
repo.rpm

    Other:
      description: "Optional repository description"
      baseurl:
https://other.example.com/repos/Other/$basearch
      gpgkey:

https://other.example.com/repos/Other/gpg.XXXXXXXXXXXXXXXX.key

  yum_repository_list:
    - EPEL
    - PGDG
    - Example
    - Other
```

This example shows two ways to define a YUM repository.

If the repository has a “repo RPM” (a package that customarily installs the necessary `/etc/yum.repos.d/*.repo` file and any GPG keys needed to verify signed packages from the repository), you can just point to it.

Otherwise, you can specify a description, a `baseurl`, and a `gpgkey` URL, and TPA will create a `/etc/yum.repos.d/Other.repo` file for you based on this information.

The EPEL and PGDG repositories are defined by default. The EPEL repository is required for correct operation, so you must always include EPEL in `yum_repository_list`. You should also include PGDG if you want to install PGDG packages.

You can set `yum_repository_list: []` to not install any repositories (but things will break without an alternative source of EPEL packages).

If you need to perform any special steps to configure repository access, you can use a pre-deploy hook to create the .repo file yourself:

```
- name: Define Example
repository
  copy:
    dest:
/etc/yum.repos.d/example.repo
    owner: root
    group: root
    mode: "0644"
    content: |
      [example]
      name=Example repo

baseurl=https://repo.example.com/repos/Example/
      enabled=1
      gpgkey=https://repo.example.com/repokey.asc
      gpgcheck=1
```

In this case, you do not need to list the repository in `yum_repository_list`.

34.18 Creating and using a local repository

If you create a local repository within your cluster directory, TPA will make any packages in the repository available to cluster instances. This is an easy way to ship extra packages to your cluster.

Optionally, you can also instruct TPA to configure the instances to use *only* this repository, i.e., disable all others. In this case, you must provide *all* packages required during the deployment, starting from basic dependencies like rsync, Python, and so on.

You can create a local repository manually, or have TPA create one for you. Instructions for both are included below.

> **Note**
>
> Specific instructions are available for managing clusters in an air-gapped environment.

Creating a local repository with TPA

TPA includes tools to help create such a local repository. Specifically the `--enable-local-repo` switch can be used with `tpaexec configure` to create an empty directory structure to be used as a local repository, and `tpaexec download-packages` populates that structure with the necessary packages.

Creating the directory structure

To configure a cluster with a local repository, run:

```
tpaexec configure --enable-local-repo …
```

This will generate your cluster configuration and create a `local-repo` directory and OS-specific subdirectories. See below for details of the layout.

Populate the repository and generate metadata

Run `tpaexec download-packages` to download all the packages required by a cluster into the local-repo. The resulting repository will contain the full dependency tree of all packages so the entire cluster can be installed from this repository. Metadata for the repository will also be created automatically meaning it is ready to use immediately.

Creating a local repository manually

Local repo layout

To create a local repository manually, you must first create an appropriate directory structure. When using `--enable-local-repo`, TPA will create a `local-repo` directory and OS-specific subdirectories within it (e.g., `local-repo/Debian/12`), based on the OS you select for the cluster. We recommend that this structure is also used for manually created repositories.

For example, a cluster running RedHat 8 might have the following layout:

```
local-repo/
`-- RedHat
    |-- 8.5 -> 8
    `-- 8
        `-- repodata
```

For each instance, TPA will look for the following subdirectories of `local-repo` in order and use the first one it finds:

- `<distribution>/<version>`, e.g., `RedHat/8.5`
- `<distribution>/<major version>`, e.g., `RedHat/8`
- `<distribution>/<release name>`, e.g., `Ubuntu/focal`
- `<distribution>`, e.g., `Debian`
- The `local-repo` directory itself.

If none of these directories exists, of course, TPA will not try to set up any local repository on target instances.

Populating the repository and generating metadata

The steps detailed below must be completed before running `tpaexec deploy`.

To populate the repository, copy the packages you wish to include into the appropriate directory. Then generate metadata using the correct tool for your system as detailed below.

> **Note**
>
> You must generate the metadata on the control node, i.e., the machine where you run tpaexec. TPA will copy the metadata and packages to target instances.

> **Note**
>
> You must generate the metadata in the subdirectory that the instance will use, i.e., if you copy packages into `local-repo/Debian/12`, you must create the metadata in that directory, not in `local-repo/Debian`.

Debian/Ubuntu repository metadata

For Debian-based distributions, install the `dpkg-dev` package:

```
sudo apt-get update && sudo apt-get install -y dpkg-dev
```

Now you can use `dpkg-scanpackages` to generate the metadata:

```
cd local-repo/Debian/bookworm
# download/copy .deb package files
dpkg-scanpackages . | gzip > Packages.gz
```

RedHat/SLES repository metadata

First, install the `createrepo` package:

```
sudo yum install -y createrepo
```

Now you can use `createrepo` to generate the metadata:

```
cd local-repo/RedHat/8
# download/copy .rpm package files
createrepo .
```

In RHEL 8+ repositories, we also need to run these additional steps:

```
repo2module --module-name tpa --module-stream local --module-version 1 --module-context f32 . modules.yaml
modifyrepo_c --mdtype=modules modules.yaml repodata/
dnf clean all
```

How TPA uses the local repository

Copying the repository

TPA will use rsync to copy the contents of the repository directory, including the generated metadata, to a directory on target instances.

If rsync is not already available on an instance, TPA can install it (i.e., `apt-get install rsync` or `yum install rsync`). However, if you have set `use_local_repo_only`, the rsync package must be included in the local repo. If required, TPA will copy just the rsync package using scp and install it before copying the rest.

Repository configuration

After copying the contents of the local repo to target instances, TPA will configure the destination directory as a local (i.e., path-based, rather than URL-based) repository.

If you provide, say, `example.deb` in the repository directory, running `apt-get install example` will suffice to install it, just like any package in any other repository.

Package installation

TPA configures a repository with the contents that you provide, but if the same package is available from different repositories, it is up to the package manager to decide which one to install (usually the latest, unless you specify a particular version).

(However, if you set `use_local_repo_only: yes`, TPA will disable all other package repositories, so that instances can only use the packages that you provide in `local-repo`.)

34.19 Installing from source

You can define a list of extensions to build and install from their Git repositories by setting `install_from_source` in config.yml:

```
cluster_vars:
  install_from_source:
    - name:
ext
      git_repository_url: https://repo.example.com/ext.git
      git_repository_ref:
dev/example

    - name:
otherext
      git_repository_url: ssh://repo.example.com/otherext.git
      git_repository_ref:
master
      source_directory:
/opt/postgres/src/otherext
      build_directory: /opt/postgres/build/otherext
      build_commands:
        - "make -f /opt/postgres/src/otherext/Makefile
install"
      build_environment:
        VAR: value
```

TPA will build and install extensions one by one in the order listed, so you can build extensions that depend on another (such as pglogical and BDR) by mentioning them in the correct order.

Each entry must specify a `name`, `git_repository_url`, and `git_repository_ref` (default: `master`) to build. You can use SSH agent forwarding or an HTTPS username/password to authenticate to the Git repository; and also set `source_directory`, `build_directory`, `build_environment`, and `build_commands` as shown above.

Run `tpaexec deploy … --skip-tags build-clean` in order to reuse the build directory when doing repeated deploys. (Otherwise the old build directory is emptied before starting the build.) You can also configure local source directories to speed up your development builds.

Whenever you run a source build, Postgres will be restarted.

Build dependencies

If you're building from source, TPA will ensure that the basic Postgres build dependencies are installed. If you need any additional packages, mention them in `packages`. For example

```
cluster_vars:
  packages:
    common:
    - golang-1.16
```

34.20 Git credentials

This page explains how to clone Git repositories that require authentication.

This may be required when you change `postgres_git_url` to install Postgres from source or use `install_from_source` to compile and install extensions.

You have two options to authenticate without writing the credentials to disk on the target instance:

- For an `ssh://` repository, you can add an SSH key to your local ssh-agent. Agent forwarding is enabled by default if you use `--install-from-source` (`forward_ssh_agent: yes` in config.yml).
- For an `https://` repository, you can `export TPA_GIT_CREDENTIALS=username:token` in your environment before running `tpaexec deploy` .

> **Note**
>
> When deploying to Docker on macOS, you should use only `https://` repository URLs because Docker containers cannot be accessed by ssh from the host in this environment.

SSH key authentication

If you are cloning an SSH repository and have an SSH keypair (`id_example` and `id_example.pub`), use SSH agent forwarding to authenticate on the target instances:

- **You need to run `ssh-agent` locally.** If your desktop environment does not already set this up for you (as most do— `pgrep ssh-agent` to check if it's running), run `ssh-agent bash` to temporarily start a new shell with the agent enabled, and run `tpaexec deploy` from that shell.
- **Add the required key(s) to the agent** with `ssh-add /path/to/id_example` (the private key file)
- **Enable SSH agent forwarding** by setting `forward_ssh_agent: yes` at the top level in config.yml before `tpaexec provision` . (This is done by default if you use `--install-from-source` .)

During deployment, any keys you add to your agent will be made available for authentication to remote servers through the forwarded agent connection.

Use SSH agent forwarding with caution, preferably with a disposable keypair generated specifically for this purpose. Users with the privileges to access the agent's Unix domain socket on the target server can co-opt the agent into impersonating you while authenticating to other servers.

HTTPS username/password authentication

If you are cloning an HTTPS repository with a username and authentication token or password, just `export TPA_GIT_CREDENTIALS=username:token` in your environment before `tpaexec deploy` . During deployment, these credentials will be made available to any `git clone` or `git pull` tasks (only). They will not be written to disk on the target instances.

34.21 Environment

You can set `target_environment` to specify environment variables that TPA should set on the target instances during deployment (e.g., to specify an HTTPS proxy, as shown below).

```
cluster_vars:
    target_environment:
        https_proxy:
https://proxy.example:8080
```

TPA will ensure these settings are present in the environment (along with any others it needs) during deployment and the later execution of any cluster management commands.

These environment settings are not persistent, but you can instead use `extra_bashrc_lines` to set environment variables for the postgres user.

34.22 Python environment

TPA decides which Python interpreter to use based on the distribution it detects on a target instance. It will use Python 3 wherever possible, and fall back to Python 2 only when unavoidable.

The `tpaexec configure` command will set `preferred_python_version` according to the distribution.

Distribution	Python 2	Python 3
Debian 12/bookworm	✓	✓ (3.11)
Debian 11/bullseye	✓	✓ (3.9)
Debian 10/buster	✓	✓ (3.7)
Ubuntu 24.04/noble	✗	✓ (3.12)
Ubuntu 22.04/jammy	✗	✓ (3.10)
Ubuntu 20.04/focal	✗	✓ (3.8)
RHEL 10.x	✗	✓ (3.12)
RHEL 9.x	✗	✓ (3.9)
RHEL 8.x	✗	✓ (3.6)
RHEL 7.x	✓	✗ (3.6)
SLES 15	✗	✓ (3.6)

Ubuntu 20.04, 22.04, 24.04, RHEL 8.x, 9.x, 10.x, and SLES 15 can be used only with Python 3.

RHEL 7.x ships with Python 3.6, but the librpm bindings for system Python 3 are not available.

You can decide for other distributions whether you prefer `python2` or `python3`, but the default for new clusters is `python3`.

Backwards compatibility

For compatibility with existing clusters, the default value of `preferred_python_version` is `python2`, but you can explicitly choose `python3` even on systems that were already deployed with `python2`.

```
cluster_vars:
  preferred_python_version: python3
```

TPA will ignore this setting on distributions where it cannot use Python 3.

Python interpreter for installed scripts

TPA installs a small number of helper Python scripts on cluster nodes (for example, `/etc/tpa/postgres-monitor`, which the deploy uses to wait for Postgres to become available). The shebang line of each installed script is templated from the `script_python_interpreter` fact.

By default, `script_python_interpreter` is set to `/usr/bin/env {{ python }}`, which means the script will run under whichever `python3` binary `env` finds first on the node's `PATH` at execution time. This matches TPA's historic behaviour and is the right answer on most hosts: it picks up the system Python that TPA itself targets for package installation.

If you need TPA's installed scripts to run under a specific Python interpreter — for example, because you have multiple Python versions on the host and want to remove ambiguity, or because you want to use a non-system Python — set `script_python_interpreter` in `cluster_vars` to the absolute path of the interpreter you want:

```
cluster_vars:
  script_python_interpreter:
/usr/bin/python3
```

Or, to point at a non-system Python:

```
cluster_vars:
  script_python_interpreter: /opt/python3.11/bin/python3
```

The value is used verbatim after `#!` in the shebang line, so it must be a path that the kernel can `exec` directly. If you set this variable, ensure that the OS packages TPA installs to support its scripts (for example `python3-psycopg2`) are available to that interpreter — see the next section for how to influence which packages TPA installs.

Python version for installed packages

TPA installs several OS packages (RPMs or DEBs, depending on the distribution) that provide Python modules its scripts depend on — for example, `python3-psycopg2`. The name prefix used to construct these package names is taken from the `python_pkg_prefix` fact, which defaults to the distribution-detected Python (`python3` on every currently-supported distribution other than RHEL 7).

If you want TPA to install packages for a non-default Python — for example, because you have installed Python 3.11 alongside the system Python and want TPA to depend on the matching `python311-psycopg2` package rather than the default `python3-psycopg2` — set `python_pkg_prefix` in `cluster_vars`:

```
cluster_vars:
  python_pkg_prefix: python311
```

TPA will then construct package names using your prefix, installing e.g. `python311-psycopg2`, `python311-barman`, and so on.

Important: `python_pkg_prefix` only controls *the names of the OS packages TPA installs*. It does not by itself change which Python interpreter TPA-installed scripts (such as `/etc/tpa/postgres-monitor`) execute under. If you set `python_pkg_prefix` on a host that has more than one Python interpreter installed, you almost certainly also want to set `script_python_interpreter` (see the previous section) to the absolute path of the matching interpreter. Otherwise the default `/usr/bin/env`-based shebang on TPA's installed scripts may resolve via `PATH` to a different Python than the one the `python_pkg_prefix`-named packages were installed for, and imports will fail at runtime.

34.23 Configuring /etc/hosts

By default, TPA will add lines to /etc/hosts on the target instances with the IP address and hostname(s) of every instance in the cluster, so that they can use each other's names for communication within the cluster (e.g., in `primary_conninfo` for Postgres).

You can specify a list of `extra_etc_hosts_lines` too:

```
instances:
- Name:
one

…
  vars:
    extra_etc_hosts_lines:
    - 192.0.2.1
acid.example.com
    - 192.0.2.2 water.example.com
```

If you don't want the default entries at all, you can specify the complete list of `etc_hosts_lines` for an instance instead, and only those lines will be added to /etc/hosts:

```
instances:
- Name:
one

…
  vars:
    etc_hosts_lines:
    - 192.0.2.1
acid.example.com
    - 192.0.2.2 water.example.com
    - 192.0.2.3
base.example.com
```

If your /etc/hosts doesn't contain the default entries for instances in the cluster, you'll need to ensure the names can be resolved in some other way.

34.24 Filesystem configuration

TPA allows you to define a list of `volumes` attached to each instance.

This list comprises both platform-specific settings that are used during provisioning and filesystem-level settings used during deployment.

First, `tpaexec provision` will use the information to create and attach volumes to the instance (if applicable; see platform-specific sections below for details). Then it will write a simplified list of volumes (containing only non-platform-specific settings) as a host var for the instance. Finally, `tpaexec deploy` will act on the simplified list to set up and mount filesystems, if required.

Here's a moderately complex example from an AWS cluster:

```
instances:
- Name:
one

…
  volumes:
  - device_name: root
    volume_type:
gp2
    volume_size: 32
  - device_name: /dev/xvdf
    volume_type:
io2
    volume_size: 64
    iops: 5000
    vars:
      volume_for: postgres_data
      encryption: luks
  - device_name: /dev/xvdb
    ephemeral: ephemeral0
    vars:
      mountpoint: /mnt/scratch
```

In this example, the EC2 instance will end up with a 32GB EBS root volume, a 64GB io2 volume (provisioned-iops EBS volumes) mounted as /opt/postgres/data, and a /tmp/scratch filesystem provided by an instance-store (“ephemeral”) volume, whose number and size are determined by the instance type.

The details are documented in the section on AWS below, but settings like `volume_type` and `volume_size` are used during provisioning, while settings under `vars` like `volume_for` or `mountpoint` are written to the inventory for use during deployment.

> **ephemeral0 instance store**
>
> nowadays most of the internal storage is NVMe in which volumes are automatically enumerated and assigned a device name by AWS, hence you might need to modify `device_name` in your config.yml to whatever was given after the provision phase.

default_volumes

Volumes are properties of an instance. You cannot set them in `cluster_vars`, because they contain platform-specific settings.

The `instance_defaults` mechanism makes special allowances for volume definitions. Since volume definitions in a large cluster may be quite repetitive (especially since we recommend that instances in a cluster be configured as close to each other as possible, you can specify `default_volumes` as shown here:

```
instance_defaults:
  default_volumes:
  - device_name: root
    volume_type:
gp2
    volume_size: 32
  - device_name: /dev/xvdf
    volume_size: 100

instances:
- Name:
one

…
- Name:
two
  volumes:
  - device_name: /dev/xvdf
    volume_size: 64
  - device_name: /dev/xvdg
    volume_size: 64

…
- Name: three
  volumes:
  - device_name: /dev/xvdf
    volume_type: none
- Name: four
  volumes: []
```

Here every instance will have a 32GB root volume and a 100GB additional volume by default (as is the case for instance `one`, which does not specify anything different). Instance `two` will have the same root volume, but it overrides `/dev/xvdf` to be 64GB instead, and has another 64GB volume in addition. Instance `three` will have the same root volume, but no additional volume because it sets `volume_type: none` for the default `/dev/xvdf`. Instance `four` will have no volumes at all.

An instance starts off with whatever is specified in `default_volumes`, and its `volumes` entries can override a default entry with the same `device_name`, remove a volume by setting `volume_type` to `none`, add new volumes with different names, or reject the defaults altogether.

(This behaviour of merging two lists is specific to `default_volumes`. If you set any other list in both `instance_defaults` and `instances`, the latter will override the former completely.)

Platform AWS

On AWS EC2 instances, you can attach EBS volumes.

```
instances:
- Name:
one

…
  volumes:
  - device_name: root
    volume_type:
gp2
    volume_size: 32
    encrypted: yes

…
  - device_name: /dev/xvdf
    volume_type:
io1
    volume_size: 32
    iops: 10000
    delete_on_termination: false

…
  - device_name: /dev/xvdg
    ephemeral: ephemeral0

…
```

TPA translates a `device_name` of `root` to `/dev/sda` or `/dev/xvda` based on the instance type, so that you don't need to remember (or change) which one to use.

The `volume_type` specifies the EBS volume type, e.g., `gp2` (for "general-purpose" EBS volumes), `io1` for provisioned-IOPS volumes (in which case you must also set `iops: 5000`), etc.

The `volume_size` specifies the size of the volume in gigabytes.

Set `encrypted: yes` to enable EBS encryption at rest. (This is an AWS feature, enabled by default in newly-generated TPA configurations, and is different from LUKS encryption, explained below.)

Set `delete_on_termination` to `false` to prevent the volume from being destroyed when the attached instance is terminated (which is the default behaviour).

Set `ephemeral: ephemeralN` to use a physically-attached instance store volume, formerly known as an ephemeral volume. The number, type, and size of available instance store volumes depends on the instance type. Not all instances have instance store volumes. Use instance store volumes only for testing or temporary data, and EBS volumes for any data that you care about.

For an EBS volume, you can also set `snapshot: snap-xxxxxxxx` to attach a volume from an existing snapshot. Volumes restored from snapshots may be extraordinarily slow until enough data has been read from S3 and cached locally. (In particular, you can spin up a new instance with `PGDATA` from a snapshot, but expect it to take several hours before it is ready to handle your full load.)

If you set `attach_existing: yes` for a volume, and there is an existing unattached EBS volume with matching Name/type/size/iops, a new volume will not be created when launching the instance, but instead the existing one will be attached to the instance the first time it starts. Reattached EBS volumes do not suffer from the performance limitations of volumes created from snapshots.

Platform bare

TPA has no control over what volumes may be attached to pre-provisioned `bare` instances, but if you define `volumes` with the appropriate `device_name`, it will handle `mkfs` and `mount` for the devices if required.

Platform Docker

Docker containers can have attached volumes, but they are bind-mounted directories, not regular block devices. They do not need to be separately initialised or mounted. As such, the configuration looks quite different.

```
instances:
- Name:
one
  platform:
docker

…
  volumes:
  - /host/path/to/dir:/tmp/container/path:ro
  - named_volume:/mnt/somevol:rw
```

You may recognise these volume specifications as arguments to `docker run -v`.

The volumes are attached when the container is created, and there are no further actions during deployment.

RAID arrays

On AWS EC2 instances, only RAID 0 is recommended by Amazon. You can create RAID volumes with a similar command:

```
sudo mdadm --create --verbose /dev/md0 --level=0 --name=MY_RAID --raid-devices=number_of_volumes
device_name1 device_name2
```

This example will attach the block device named `/dev/md0`. The handling of `volume_for` or `mountpoint` during deployment happens as the same as with any other volume. TPA will handle `mkfs` and `mount` for it.

```
- Name:
one

…
  volumes:
  - device_name:
/dev/md0
    vars:
      volume_for: postgres_data
```


LUKS encryption

TPA can set up a LUKS-encrypted device:

```
instances:
- Name:
one

…
  volumes:
  - device_name:
/dev/xyz
    vars:
      encryption: luks
      luks_volume: mappedname
      volume_for:
…
```

If a volume with `encryption: luks` set is not already initialised, TPA will use `cryptsetup` to first `luksFormat` and then `luksOpen` it to map it under `/dev/mapper/mappedname` before handling filesystem creation as with any other device.

(To avoid any possibility of data loss, TPA will refuse to set up LUKS encryption on a device that contains a valid filesystem already.)

If you create a LUKS-encrypted `volume_for: postgres_data` , TPA will configure Postgres to not start automatically at boot. You can use `tpaexec start-postgres clustername` to mount the volume and start Postgres (and `stop-postgres` to stop Postgres and unmap the volume).

The LUKS passphrase is generated locally and stored in the vault.

Filesystem creation and mounting

If any `device` does not contain a valid filesystem, it will be initialised with `mkfs` .

```
instances:
- Name:
one

…
  volumes:
  - device_name:
/dev/xyz
    vars:
      volume_for:
…
      fstype: ext4
      fsopts:
        - -
cc
        - -m
2
      mountopts: 'defaults,relatime,nosuid'
      readahead: 65536
      owner: root
      group: root
      mode: "0755"
```

You can specify the `fstype` (default: ext4), `fsopts` to be passed to mkfs (default: none), and `mountopts` to be passed to mount and written to fstab (see below).

TPA will set the readahead for the device to 16MB by default (and make the value persist across reboots), but you can specify a different value for the volume as shown above.

There are two ways to determine where a volume is mounted. You can either specify a `mountpoint` explicitly, or you can set `volume_for` to `postgres_data`, `postgres_wal`, `postgres_tablespace` or `barman_data`, and TPA will translate the setting into an appropriate mountpoint for the system.

Once the `mountpoint` is determined, the `device` will be mounted there with the given `mountopts` (default: `defaults,noatime`). An entry will also be created for the filesystem in `/etc/fstab`.

You may optionally specify `owner`, `group`, or `mode` for the volume, and these attributes will be set on the `mountpoint`. Remember that at this very early stage of deployment, you cannot count on the `postgres` user to exist. In any case, TPA will (separately) ensure that any directories needed by Postgres have the right ownership and permissions, so you don't have to do it yourself.

34.25 Uploading artifacts

You can define `artifacts` to create or copy files to target instances:

```
cluster_vars:
  artifacts:
  - type: path
    path: /some/target/path
    state: directory
    owner: root
    group: root
    mode: "0755"
  - type: file
    src: /host/path/to/file
    dest: /target/path/to/file
    owner: root
    group: root
    mode: "0644"
  - type: archive
    src:
example.tar.gz
    dest: /some/target/path
  - type: directory
    src: /host/path/a/
    dest: /target/path/b/
```

The following types are supported:

- Use `path` to create or remove and change the ownership or mode of files and directories (takes the same parameters as Ansible's `file` module, which it uses internally)
- Use `file` to copy a file from the controller and set the ownership and mode (uses `copy`)
- Use `archive` to extract files from an archive to a specified location (uses `unarchive`)
- Use `directory` to rsync a directory from the controller to target instances (uses `synchronize`)

The example shows one entry for each of the above artifact types, but you can use these or any other parameters that the corresponding Ansible module accepts.

Copying files and directories to target instances is a common-enough need that this feature provides a convenient shortcut you can use instead of writing a custom hook.

34.26 ssh_key_file

By default, `tpaexec provision` will use `ssh-keygen` to generate a new SSH keypair for the cluster (into files named `id_cluster_name` and `id_cluster_name.pub` inside the cluster directory).

If you want to use an existing key instead, you can set `ssh_key_file` at the top level of config.yml to the location of an SSH private key file. The corresponding public key must be available with an extension of `.pub` at the same location:

```
ssh_key_file: ~/.ssh/id_rsa
```

(If this file does not already exist, it will be created by `ssh-keygen` during provisioning.)

Skipping authorized_keys management

During deployment, TPA adds the cluster's public key to the admin user's `authorized_keys` file on each instance. In some environments, like Oracle Cloud Infrastructure, this file may be read-only or managed externally.

To skip this step, use the `ssh-cluster-key-config` task selector:

```
tpaexec deploy <clustername> --excluded_tasks=ssh-cluster-key-config
```

Or in `config.yml`:

```
cluster_vars:
  excluded_tasks:
  - ssh-cluster-key-config
```

When this task is excluded, you must ensure that SSH access to the instances is already configured through other means before running `tpaexec deploy`.

34.27 Managing SSH host keys

TPA generates a set of SSH host keys while provisioning a cluster. These keys are stored in the cluster directory, under the `hostkeys` subdirectory. These host keys are automatically installed into `/etc/ssh` on AWS EC2 instances and Docker containers.

By default, these host keys are not installed on bare instances, but you can set `manage_ssh_hostkeys` to enable it:

```
instances:
- Name:
one

…
  platform: bare
  vars:
    manage_ssh_hostkeys: yes
```

You must initially set up `known_hosts` in your cluster directory with correct entries, as described in the docs for bare instances. TPA will replace the host keys during deployment.

The `manage_ssh_hostkeys` setting is meaningful only for bare instances. The generated host keys will be installed on all other instances.

known_hosts

TPA will add entries for every host and its public host keys to the global `ssh_known_hosts` file on every instance in the cluster, so that they can ssh to each other without host key verification prompts, regardless of whether they have `manage_ssh_hostkeys` set or not.

34.28 Postgres source installation

TPA will compile and install Postgres from source if you set `postgres_installation_method` to `src`. This feature is meant for use in development and testing, and allows you to switch between packaged and source builds within an identically-configured cluster.

Even here, you do not need to change the defaults, which will give you a working cluster with debugging enabled.

Git repository

The default settings will build and install Postgres from the community Git repository, using the `REL_xx_STABLE` branch corresponding to your `postgres_version`. You can specify a different repository or branch (any valid git reference) as follows:

```
cluster_vars:
  postgres_git_url: git://git.postgresql.org/git/postgresql.git
  postgres_git_ref: REL_12_STABLE
```

The default git.postgresql.org repository does not require authentication, but if necessary, you can useSSH agent forwarding or an HTTPS username/password to authenticate to other repositories.

The repository will be cloned into `postgres_src_dir` (default: `/opt/postgres/src/postgres`), or updated with `git pull` if the directory already exists (e.g., if you are re-deploying).

Build customisation

By default, TPA will configure and build Postgres with debugging enabled and sensible defaults in `postgres_build_dir` (default: `/opt/postgres/build/postgres`). You can change various settings to customise the build:

```
cluster_vars:
  postgres_extra_configure_env:
    CFLAGS: "-O3"
  postgres_extra_configure_opts:
    - --with-
llvm
    - --disable-tap-
tests
```

This will run `./configure` with the options in `postgres_extra_configure_opts` and the settings from `postgres_extra_configure_env` defined in the environment. Some options are specified by default (e.g., `--with-debug`), but can be negated by the corresponding `--disable-xxx` or `--without-xxx` options. Building `--without-openssl` is not supported.

If required, you can also change the following default build commands:

```
cluster_vars:
  postgres_make_command: "make -
s"
  postgres_build_targets:
    - "all"
    - "-C contrib all"
  postgres_install_targets:
    - "install"
    - "-C contrib
install"
```

Run `tpaexec deploy … --skip-tags build-clean` in order to reuse the build directory when doing repeated deploys. (Otherwise the old build directory is emptied before starting the build.) You can also configure local source directories to speed up your development builds.

Whenever you run a source build, Postgres will be restarted.

Additional components

Even if you install Postgres from packages, you can compile and install extensions from source. There's a separate page about how to configure `install_from_source`.

If you install Postgres from source, however, you will need to install extensions from source as well, because the extension packages typically depend on the Postgres package(s) being installed.

Package installation

There's a separate page about installing Postgres and Postgres-related packages with `postgres_installation_method: pkg` (the default).

34.29 Installing packages

TPA installs a batch of non-Postgres-related packages early during the deployment, then all Postgres-related packages together, and then packages for optional components separately. This page is about installing packages like sysstat or strace, which have no dependency on Postgres packages.

You can add entries to `packages` under `cluster_vars` or a particular instance's `vars` in config.yml:

```
cluster_vars:
  packages:
    common:
      - pkg1
      - pkg2
    Debian:
      - debpkg1
    RedHat:
      -
rhpkg1
      -
rhpkg2
    Ubuntu:
      -
ubpkg1
    SLES:
      -
slespkg1
```

In the example above, TPA will install its own list of `default_packages` and the packages listed under `packages.common` on every instance, and the remaining distribution-specific packages based on which distribution the instance is running. If any of these packages is not available, the deployment will fail.

Don't list any packages that depend on Postgres; use `extra_postgres_packages` instead.

Optional packages

You can specify a list of `optional_packages` to install. They will be installed if they are available, and ignored otherwise. As with the other settings, the `common` entries apply to every instance, whereas any other lists apply only to instances running the relevant distribution.

```
optional_packages:
  common:
    - pkg1
    - pkg2
  Debian:
    - debpkg4
```

Removing packages

You can specify a list of `unwanted_packages` that should be removed if they are installed.

```
unwanted_packages:
  common:
    - badpkg1
  Ubuntu:
    - badpkg2
```


34.30 Running initdb

TPA will first create `postgres_data_dir` if it does not exist, and ensure it has the correct ownership, permissions, and SELinux context. Then, unless the directory already contains a `VERSION` file, it will run `initdb` to initialise `postgres_data_dir`.

You can use the pre-initdb hook to execute tasks before `postgres_data_dir` is created and `initdb` is run. If the hook initialises `postgres_data_dir`, TPA will find the `VERSION` file and realise that it does not need to run `initdb` itself.

You can optionally set `postgres_initdb_opts` to a list of options to pass to `initdb`:

```
cluster_vars:
  postgres_locale: de_DE.UTF-8
  postgres_initdb_opts:
  - --data-
checksums
```

We recommend always including the `--data-checksums` option (which is included by default).

TPA will set `TZ=UTC` in the environment, and set `LC_ALL` to the `postgres_locale` you specify, when running `initdb`.

Separate configuration directory

By default, `postgres_conf_dir` is equal to `postgres_data_dir`, and the Postgres configuration files (postgresql.conf, pg_ident.conf, pg_hba.conf, and the include files in conf.d) are created within the data directory. If you change `postgres_conf_dir`, TPA will move the generated configuration files to the new location after running `initdb`.

34.31 Installing Postgres-related packages

TPA installs a batch of non-Postgres-related packages early during the deployment, then all Postgres-related packages together, and then packages for optional components separately. This page is about installing packages like pglogical that depend on Postgres itself.

To install extra packages that depend on Postgres (e.g., Postgis), list them under `extra_postgres_packages` in `cluster_vars` or a particular instance's `vars` in config.yml:

```
cluster_vars:
  extra_postgres_packages:
    common:
      - postgres-pkg1
      - postgres-pkg2
    Debian:
      - postgres-deb-pkg1
    RedHat:
      - postgres11-
rhpkg1
      - postgres11-
rhpkg2
    Ubuntu:
      -
ubpkg1
    SLES:
      -
slespkg1
```

The packages listed under `packages.common` will be installed on every instance, together with the default list of Postgres packages, and any distribution-specific packages you specify.

There's a separate page about compiling and installing Postgres from source.

34.32 PG Backup API

If you set `enable_pg_backup_api: true` in `config.yml` or use the `--enable-pg-backup-api` command line option during configure, instances with the `barman` role will install pg-backup-api and set up an apache proxy for client cert authentication. This apache proxy will use an SSL CA generated for the cluster to generate its server and client certificates.

```
cluster_vars:
  enable_pg_backup_api: true
```

PG Backup API package version

By default, TPA installs the latest available version of pg-backup-api.

The version of the pg-backup-api package that is installed can be specified by including `pg_backup_api_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    pg_backup_api_package_version: '2.0.0-1.el8'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

Installation Options

pg-backup-api will be installed via packages by default, but you can also install from a git branch or a local directory. Seeconfigure-source.md and install_from_source.md for more details.

Verify Setup

Run `pg-backup-api status` on the barman node running pg-backup-api - if you get "OK" back, the pg-backup-api service is running.

To test that the proxy is working, run

```
curl --cert /etc/tpa/pg-backup-api/pg-backup-user.crt \
     --key /etc/tpa/pg-backup-api/pg-backup-user.key \
     -X GET https://{hostname}/diagnose
```

If it's working, you'll get a large json output. You can compare this with the output of `barman diagnose`, they should match exactly.

SSL Certificates

The root certificate will be copied to `/etc/tpa/pg-backup-api/` by default.

A client certificate and key (`pg-backup-user.crt` and `pg-backup-user.key`) will be generated for testing (through `tpaexec test`) or command line from the barman host. See Testing.

An apache proxy server certificate and key (`pg-backup-api.crt` and `pg-backup-api.key`) will also be generated

Each service needing to query the api will need to generate its own client certificate separately. PEM agent role, for instance, generates a client certificate during it's setup when both `--enable-pem` and `--enable-pg-backup-api` (or config.yml equivalent) are used.

Minor update for pg backup api using `tpaexec upgrade`

When trying to upgrade to a specific package version, ensure the `pg_backup_api_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> -e pg_backup_api_package_version="<desired version>"
  --components=pg-backup-api
```

Refer to the section on package version selection and upgrade for more information.

To select PG Backup API for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pg-backup-api` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.33 Setting sysctl values

By default, TPA sets various sysctl values on target instances, and includes them in `/etc/sysctl.conf` so that they persist across reboots.

You can optionally specify your own values in `sysctl_values`:

```
cluster_vars:
  sysctl_values:
    kernel.core_pattern: core.%e.%p.%t
    vm.dirty_bytes: 4294967296
    vm.zone_reclaim_mode: 0
```

Any values you specify will take precedence over TPA's default values for that variable (if any). The settings will first be added to `sysctl.conf` line-by-line, and finally loaded with `sysctl -p`.

Docker and lxd instances do not support setting sysctls, so TPA will skip this step altogether for those platforms.

Hugepages

By default, TPA reserves hugepages of the architecture's default size (2MB on x86_64) for Postgres to use for `shared_buffers`. It does this by adding `hugepages=N` to the kernel command line and writing `vm.nr_hugepages = N` to `/etc/sysctl.conf`.

If you want Postgres to use a larger hugepage size (typically 1GB on x86_64), set `huge_page_size` to a Postgres-style memory string:

```
instances:
- Name: primary
  vars:
    huge_page_size:
1GB
```

When `huge_page_size` is set, TPA will:

- reserve pages of that size on the kernel command line, e.g. `hugepagesz=1G hugepages=N`;
- omit `vm.nr_hugepages` from `/etc/sysctl.conf` (that sysctl only ever applies to the architecture's default-size pool, and is silently ineffective for any other size); and
- set the `huge_page_size` GUC in `postgresql.conf` so Postgres uses pages from the chosen pool.

It is almost always right to set `huge_page_size` on individual Postgres instances rather than in `cluster_vars`: any reservation made on the kernel command line is taken out of normal memory at boot, so a Barman server or other non-Postgres instance in the same cluster does not want the same allocation as a Postgres node.

By default, TPA calculates the number of hugepages to reserve as `shared_buffers` plus 3% of total RAM, divided by the hugepage size. You can override this calculation by setting `nr_hugepages` directly:

```
instances:
- Name: primary
  vars:
    huge_page_size:
1GB
    nr_hugepages: 16
```

> **Note**
>
> For backwards compatibility, an existing `sysctl_values['vm.nr_hugepages']` is honoured as the page count when `nr_hugepages` is not set explicitly.

> **Note**
>
> Note that updating the kernel command line on an already-deployed instance is outside TPA's scope: changing `huge_page_size` or `nr_hugepages` after the first deploy will not rewrite the existing GRUB configuration. Edit it by hand and reboot to apply the change.

Transparent hugepages

By default, TPA will set transparent_hugepage to "never" on the kernel command line and also install a script, `/etc/tpa/rc.local`, to set it at boot-time via the `sys` filesystem. To use a different setting, set `transparent_hugepage` to `always` or `madvise` in an instances `vars` or in `cluster_vars` to control all instances.

34.34 Creating Postgres databases

To create Postgres databases during deployment, add entries to the list of `postgres_databases` under `cluster_vars` or a particular instance's `vars` in config.yml:

```
cluster_vars:
  postgres_databases:
  - name: exampledb

  - name: complexdb
    owner: example
    encoding: UTF8
    lc_collate: de_DE.UTF-8
    lc_ctype: de_DE.UTF-8
    template: template0
    extensions:
    - name:
hstore
    - name:
dblink
    languages:
    - name:
plperl
    - name:
plpython
    tablespace: exampletablespace
```

The example above would create two databases (apart from any databases that TPA itself decides to create, such as `bdr_database`).

Each entry must specify the `name` of the database to create. All other attributes are optional.

The `owner` is `postgres` by default, but you can set it to any valid username (the users in `postgres_users` will have been created by this time).

The `encoding`, `lc_collate`, and `lc_ctype` values default to the `postgres_locale` set at the time of running initdb (the default is to use the target system's LC_ALL or LANG setting). If you are creating a database with non-default locale settings, you will also need to specify `template: template0`.

You can optionally specify the default `tablespace` for a database; the tablespace must already exist (see `postgres_tablespaces`).

You can specify optional lists of `extensions` and `languages` to create within each database (in addition to any extensions or languages inherited from the template database). Any packages required must be installed already, for example by including them in `extra_postgres_packages`.

TPA will not drop existing databases that are not mentioned in `postgres_databases`, and it may create additional databases if required (e.g., for BDR).

34.35 Creating Postgres tablespaces

To create Postgres tablespaces during deployment, define their names and locations in `postgres_tablespaces` under `cluster_vars` or a particular instance's `vars` in config.yml.

If you define volumes with `volume_for: postgres_tablespace` set and a `tablespace_name` defined, they will be added as default entries to `postgres_tablespaces`.

```
cluster_vars:
  postgres_tablespaces:
    explicit:
      location: /some/path

instances:
- Name: example

…
  volumes:
  - device_name: /dev/xvdh

…
    vars:
      volume_for:
postgres_tablespace
      tablespace_name:
implicit
```

The example above would create two tablespaces: explicit (at /some/path) and implicit (at /opt/postgres/tablespaces/implicit/tablespace_data by default, unless you specify a different mountpoint for the volume).

Every `postgres_tablespace` volume must have `tablespace_name` defined; the tablespace location will be derived from the volume's mountpoint.

Every entry in `postgres_tablespaces` must specify a tablespace name (as the key) and its `location`. If you are specifying tablespace locations explicitly, do not put tablespaces inside PGDATA, and do not use any volume mountpoint directly as a tablespace location (`lost+found` will confuse some tools into thinking the directory is not empty).

By default, the tablespace `owner` is `postgres`, but you can set it to any valid username (the users in `postgres_users` will have been created by this time).

Streaming replicas must have the same `postgres_tablespace` volumes and `postgres_tablespaces` setting as their upstream instance

You can set the default tablespace for a database in `postgres_databases`.

34.36 Configuring .pgpass

TPA creates `~postgres/.pgpass` by default with the passwords for the `postgres_user` (`postgres` or `entreprisedb` by default depending on the `postgres_flavour`) in it, for use between cluster instances.

You can set `pgpass_users` to create entries for a different list of users. Note that the `pgpass_users` list overrides default values, so the `postgres_user` (`postgres` / `enterprisedb`) is NOT included unless you explicitly include it in the `pgpass_users` list.

You can also include the `postgres/pgpass` role from hook scripts to create your own `.pgpass` file:

```
- include_role: name=postgres/pgpass
  vars:
    pgpassfile: ~otheruser/.pgpass
    pgpass_owner: otheruser
    pgpass_group: somegroup
    pgpass_users:
    - xyzuser
    - pqruser
```

34.37 The postgres Unix user

This page documents how the postgres user and its home directory are configured.

There's a separate page about how to create Postgres users in the database.

Shell configuration

TPA will install a `.bashrc` file and ensure that it's also included by the `.profile` or `.bash_profile` files.

It will set a prompt that includes the username and hostname and working directory, and ensure that `postgres_bin_dir` in in the `PATH`, and set `PGDATA` to the location of `postgres_data_dir`.

You can optionally specify `extra_bashrc_lines` to append arbitrary lines to `.bashrc`. (Use the YAML multi-line string syntax `>-` to avoid having to worry about quoting and escaping shell metacharacters.)

```
cluster_vars:
  extra_bashrc_lines:
  - alias la=ls\ -
la
  -
>-
    export
PATH="$PATH":/some/other/dir
```

It will edit sudoers to allow `sudo systemctl start/stop/reload/restart/status postgres`, and also change `ulimits` to allow unlimited core dumps and raise the file descriptor limits.

SSH keys

TPA will use `ssh-keygen` to generate and install an SSH keypair for the postgres user, and edit `.ssh/authorized_keys` so that the instances in the cluster can ssh to each other as `postgres`.

TLS certificates

By default, TPA will generate a private key and a self-signed TLS certificate which are used by Postgres as the `ssl_key_file` and `ssl_cert_file` respectively. The files are named using the TPA cluster name (`cluster_name.key` and `cluster_name.crt`) and located in `/etc/tpa`. For more information, including how to provide your own key and certificate, see the documentation for postgresql.conf.

The size of self-signed TLS key can be modified adding the variable `postgres_rsa_key_size` to the `cluster_vars` section:

```
  (...)
  cluster_vars:
    postgres_rsa_key_size: 4096
```

By default, the self-signed TLS certificate has a period of 3650 days. The period can be adjusted by defining `openssl_certificate_validity` (in days) in the `cluster_vars` section. If the certificate needs to be replaced, use the `tpa_tls_renew_tls_certificates` variable on `tpaexec deploy`: `tpaexec deploy (...) -e tpa_tls_renew_tls_certificates=true`.

Username

The `postgres_user` and `postgres_group` settings (both `postgres` by default) are used consistently everywhere. You can change them if you need to run Postgres as a different user for some reason.

34.38 Creating Postgres users

To create Postgres users during deployment, add entries to the list of `postgres_users` under `cluster_vars` or a particular instance's `vars` in config.yml:

```
cluster_vars:
  postgres_users:
  - username: example

  - username: otheruser
    generate_password: true
    role_attrs:
    - superuser
    -
replication
    granted_roles:
    - r1
    - r2
```

The example above would create two users (apart from any users that TPA itself decides to create, such as repmgr or barman).

Each entry must specify the `username` to create.

Any roles in the `granted_roles` list will be granted to the newly-created user.

The `role_attrs` list may contain certain CREATE ROLE options such as `[NO]SUPERUSER`, `[NO]CREATEDB`, `[NO]LOGIN` (to create a user or a role) etc.

Password generation

By default, TPA will generate a random password for the user, and store it in a vault-encrypted variable named `<username>_password` in the cluster's inventory. You can retrieve the value later:

```
tpaexec show-password ~/clusters/speedy example
```

```
output
beePh~iez6lie4thi5KaiG%eghaeT]ai
```

You cannot explicitly specify a password in config.yml, but you can store a different `<username>_password` in the inventory instead:

```
tpaexec store-password ~/clusters/speedy example --random
tpaexec show-password ~/clusters/speedy example
```

```
output
)>tkc}}k1y4&epaJ?;NJ:l'uT{C7D*&lt;p
```

```
tpaexec store-password ~/clusters/speedy example
```

(enter password at the prompt)

```
output
Password:
```

```
tpaexec show-password ~/clusters/speedy example
```

```
output
terrible insecure password
```

If you don't want the user to have a password at all, you can set `generate_password: false`.

34.39 tpaexec archive-logs

To create a log directory and archive logs from instances, run

```
tpaexec archive-logs <cluster-dir>
```

This will create a logs/YYYYMMDDHHMMss/ directory in your cluster directory and download a tar.gz archive of all the files under /var/log on each instance in the cluster into a separate directory.

Prerequisites

If you have an existing cluster you can run `tpaexec archive-logs` immediately. But if you are configuring a new cluster, you must at least provision the cluster. You will get more logs if you also deploy the cluster.

Quickstart

```
tpaexec archive-logs ~/clusters/speedy
```

```
output
PLAY [Prepare local host archive] *******************************************

TASK [Collect facts] ********************************************************
ok: [localhost]

TASK [Set time stamp] *******************************************************
ok: [localhost]

TASK [Create local log archive directory] ***********************************
changed: [localhost]

PLAY [Archive log files from target instances] ******************************

...

TASK [Remove remote archives] ***********************************************
changed: [kinship]
changed: [khaki]
changed: [uncivil]
changed: [urchin]

PLAY RECAP ******************************************************************
khaki                      : ok=3    changed=3    unreachable=0    failed=0
kinship                    : ok=3    changed=3    unreachable=0    failed=0
localhost                  : ok=3    changed=1    unreachable=0    failed=0
uncivil                    : ok=3    changed=3    unreachable=0    failed=0
urchin                     : ok=3    changed=3    unreachable=0    failed=0
```

You can append `-v`, `-vv`, etc. to the command if you want more verbose output.

Generated files

You can find the logs for each instance under the cluster directory:

output

```
~/clusters/speedy/logs/
`-- 220220306T185049
    |-- khaki-logs-20220306T185049.tar.gz
    |-- kinship-logs-20220306T185049.tar.gz
    |-- uncivil-logs-20220306T185049.tar.gz
    `-- urchin-logs-20220306T185049.tar.gz
```

Archive contents example:

output

```
khaki-logs
|-- anaconda
|   |-- anaconda.log
|   |-- dbus.log
|   |-- dnf.librepo.log
|   |-- hawkey.log
|   |-- journal.log
|   |-- ks-script-ipdkisn0.log
|   |-- ks-script-jr03uzns.log
|   |-- ks-script-mh2iidvh.log
|   |-- lvm.log
|   |-- packaging.log
|   |-- program.log
|   |-- storage.log
|   |-- syslog
|   `-- X.log
|-- btmp
|-- dnf.librepo.log
|-- dnf.log
|-- dnf.rpm.log
|-- hawkey.log
|-- lastlog
|-- private
|-- tpaexec.log
`-- wtmp
```

34.40 tpaexec download-packages

The purpose of the downloader is to provide the packages required to do a full installation of a TPA cluster from an existing configuration. This is useful when you want to ship packages to secure clusters that do not have internet access, or avoid downloading packages repeatedly for test clusters.

The downloader will download the full dependency tree of packages required, and the resulting package repository will include metadata files for the target distribution package manager, so can be used exclusively to build clusters. At this time package managers Apt and YUM are supported.

> **Note**
>
> The download-packages feature requires Docker to be installed on the TPA host. This is because the downloader operates by creating a container of the target operating system and uses that system's package manager to resolve dependencies and download all necessary packages. The required Docker setup for download-packages is the same as that for using Docker as a deployment platform.

Usage

An existing cluster configuration needs to exist which can be achieved using the `tpaexec configure` command. No specific options are required to use the downloader. See configuring a cluster .

Execute the download-packages subcommand to start the download process. Provide the OS and OS version that should be used by the downloader.

```
tpaexec download-packages cluster-dir --os RedHat --os-version 8
```

This can also be expressed as a specific docker image. It is strongly recommended that you use one of the tpa images prefixed like the example below.

```
tpaexec download-packages cluster-dir --docker-image tpa/redhat:8
```

The downloader will place files downloaded in the directory `local-repo` by default. It is possible to download to alternative directory by using the option `--download-dir path`.

Using the result

The contents of the `local-repo` directory is populated with a structure determined by ansible according to the OS contained in the docker image. For example, the docker image `tpa/redhat:8` would have the following:

```
cluster-dir/
`-- local-repo
    `-- RedHat
        `-- 8
            |-- *.rpm
            `-- repodata
                `-- *repodata-files*
```

You can use this in the cluster as is or copy it to a target control node. See recommendations for installing to an air-gapped environment. A local-repo will be detected and used automatically by TPA.

Regenerate local repository metadata

You can use the `download-packages` command with the `--refresh-repository` option to regenerate the local repository metadata and acknowledge new packages or new version of existing packages, prior to an upgrade. After placing the new contents in the `local-repo` directory and updating the new version of the packages on its respective `config.yml` file, the command needs to be executed as it follows:

```
tpaexec download-packages cluster-dir --os <OS> --os-version <version> --refresh-repository
```

The `--refresh-repository` option will create a new docker container, regenerate the repository's metadata, and then remove itself.

> **Note**
>
> When executing the command above, the local repository's cache on each of the nodes is not automatically refreshed, this is performed on the upgrade stage automatically, or needs to be done manually using `tpaexec cmd`.

Cleaning up failed downloader container

If there is an error during the download process, the command will leave behind the downloader container running to help with debugging. For instance you may want to log in to the failed downloader container to inspect logs or networking. Downloader container is typically named $cluster_name-downloader unless it exceeds the allowed limit of 64 characters for the container name. You can check for the exact name by running `docker ps` to list the running containers and look for a container name that matches your cluster name. In most cases you can log in to the running container by executing `docker exec -it $cluster_name-downloader /bin/bash`. After the inspection, you can clean up the left over container by running the `download-packages` command with `--tags cleanup`. For example:

```
tpaexec download-packages cluster-dir --docker-image tpa/redhat:8 --tags cleanup
```

34.41 TPA custom commands

You can define custom commands that perform tasks specific to your environment on the instances in a TPA cluster.

You can use this mechanism to automate any processes that apply to your cluster. These commands can be invoked against your cluster directory, like any built-in cluster management command. Having a uniform way to define and run such processes reduces the likelihood of errors caused by misunderstandings and operator error, or process documentation that was correct in the past, but has drifted away from reality since then.

Writing Ansible playbooks means that you can implement arbitrarily complex tasks; following the custom command conventions means you can take advantage of various facts that are set based on your config.yml and the cluster discovery tasks that TPA performs, and not have to think about details like connections, authentication, and other basic features.

This makes it much easier to write resilient, idempotent commands in a way that ad-hoc shell scripts (could be, but) usually aren't.

Quickstart

- Create `commands/mycmd.yml` within your cluster directory
- Run `tpaexec mycmd /path/to/cluster`

Example

Here's an example of a command that runs a single command on all instances in the cluster. Depending on the use-case, you can write commands that target different hosts (e.g., `hosts: role_postgres` to run only on Postgres instances), or run additional tasks and evaluate conditions to determine exactly what to do.

```yaml
---
# Always start with this
- import_playbook: "{{ tpa_dir }}/architectures/lib/init.yml"
  tags: always

- name: Perform custom command tasks
  hosts: all
  tasks:
  - name: Display last five lines of syslog
    command: tail -5 /var/log/syslog
    become_user: root
    become: yes
```


34.42 TPA custom tests

You can easily define in-depth tests specific to your environment and application to augment TPA's builtin tests.

We strongly recommend writing tests for any tasks, no matter how simple, that you would run on your cluster to reassure yourself that everything is working as you expect. Having a uniform and repeatable way to run such tests ensures that you don't miss out on anything important, whether you're dealing with a crisis or just doing routine cluster management.

If you write tests that target cluster instances by their configured role (or other properties), you can be sure that all applicable tests will be run on the right instances. No need to look up or remember how many replicas to check the replication status on, nor which servers are running pgbouncer, or any other such details that are an invitation to making mistakes when you are checking things by hand.

Tests must not make any significant changes to the cluster. If it's not something you would think of doing on a production server, it probably shouldn't be in a test.

Quickstart

- Create `tests/mytest.yml` within your cluster directory
- Run `tpaexec test /path/to/cluster mytest`

You can also create tests in some other location and use them across clusters with the `--include-tests-from /other/path` option to `tpaexec test`.

(Run `tpaexec help test` for usage information.)

Example

Here's how to write a test that is executed on all Postgres instances (note `hosts: role_postgres` instead of `hosts: all`).

You can use arbitrary Ansible tasks to collect information from the cluster and perform tests. Just write tasks that will fail if some expectation is not met (`assert`, `fail … when`, etc.).


```yaml
---
- name: Perform my custom tests
  hosts: role_postgres
  tasks:

  # Always start with this
  - include_role:
      name: test
      tasks_from: prereqs.yml

  # Make sure that the PGDATA/PG_VERSION file exists. (This is just a
  # simplified example, not something that actually needs testing.)
  - name: Perform simple test
    command: "test -f {{ postgres_data_dir }}/PG_VERSION"
    become_user: "{{ postgres_user }}"
    become: yes

  - name: Run pg_controldata
    command: >
      {{ postgres_bin_dir }}/pg_controldata {{ postgres_data_dir }}
    register: controldata
    become_user: "{{ postgres_user }}"
    become: yes

  # Write output to clusterdir/$timestamp/$hostname/pg_controldata.txt
  - name: Record pg_controldata output
    include_role:
        name: test
        tasks_from: output.yml
    vars:
      output_file: pg_controldata.txt
      content: |
        {{ controldata.stdout }}
```

You can use the builtin `output.yml` as shown above to record arbitrary test output in a timestamped test directory in your cluster directory.

Each test must be a complete Ansible playbook (i.e., a list of plays, not just a list of tasks). It will be imported and executed after the basic TPA setup tasks.

Destructive tests

Tests should not, by default, make any significant changes to a cluster. (Even if they do something like creating a table to test replication, they must be careful to clean up after themselves.)

Any test that makes changes to a cluster that would be unacceptable on a production cluster MUST be marked as `destructive`. These may be tests that you run only in development, or during the initial cluster "burn in" process.

You can define "destructive" tests by setting `destructive: yes` when including `prereqs.yml` in your test:

```
- hosts:
…
  tasks:
  - include_role:
      name: test
      tasks_from:
prereqs.yml
    vars:
      destructive: yes
```

If someone then runs `tpaexec test /path/to/cluster mytest`, they will get an error asking them to confirm execution using the `--destroy-this-cluster` option.

(Note: using `--destroy-this-cluster` signifies an awareness of the risk of running the command. It does not guarantee that the test will actually destroy the cluster.)

34.43 Configuring the PGAI agent

> **Note**
>
> TPA currently only supports using the PGAI Agent to integrate clusters with EDB Postgres AI Cloud Service. Using the agent to integrate with EDB Postgres AI Hybrid Manager, or for usage reporting, will be supported in a future release of TPA.

TPA installs and configures the PGAI agent (packaged as `beacon-agent`) on nodes which have the role `beacon-agent` in `config.yml`. If `--enable-beacon-agent` is passed to `tpaexec configure`, then all of the postgres nodes in the cluster have this role.

PGAI agent package version

By default, TPA installs the latest available version of `beacon-agent`.

The version of the `beacon-agent` package that is installed can be specified by including `beacon_agent_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:
…
    beacon_agent_package_version: '1.56.2-1'
…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

PGAI agent configuration

The PGAI agent configuration contains two parameters which must be set per-cluster, the access key and the project id.

The access key is kept encrypted in the cluster directory and can be set or read using tpa's `store-password` and `show-password` commands:

```
tpaexec store-password . beacon_agent_access_key
```

output

```
Password:
```

```
tpaexec show-password . beacon_agent_access_key
```

If the environment variable BEACON_AGENT_ACCESS_KEY is set when `tpaexec provision` is run, the access key is set from its value.

The project id is stored in `config.yml` under the `beacon_agent_project_id` key in `cluster_vars`. If the `--beacon_agent_project_id` argument is passed to `tpaexec configure` then its value is written to `config.yml` appropriately.

Installing the PGAI agent

TPA installs the PGAI agent from EDB's repositories and creates an operating system user called `beacon` and a database user called `beacon`. A configuration file for the agent is written to `.beacon/beacon_agent.yaml` in the beacon user's home directory.

Running the PGAI agent

TPA installs a systemd service unit file to start the agent at boot-time, running as the beacon user.

34.44 Compliance

TPA can generate configurations designed to make it easy for a cluster to comply with the STIG or CIS standards. If you pass `--compliance stig` or `--compliance cis` to `tpaexec configure`, TPA will:

- Check that other options are compatible with the appropriate standard.
- Add various entries to the generated `config.yml`, including marking that this is a cluster meant to comply with a particular standard and setting Postgres configuration as required by the standard.
- Adjust some deployment tasks to enforce compliance.
- Run checks at the end of deployment.

The deploy-time checks can be skipped by giving the option `--excluded_tasks=compliance` to `tpaexec deploy`. This feature is intended for testing only, when using a test system on which full compliance is impossible (for example, because SSL certificates are not available).

There are some situations in which TPA will intentionally fail to comply with the selected standard; these are documented under Exceptions below.

STIG

STIG compliance is indicated by the `--compliance stig` option to `tpaexec configure`.

Option compatibility

STIG compliance requires the `bare` platform and the `epas` flavour. It requires the RedHat OS with version 8 or 9.

Settings in config.yml

The following entry is added to `cluster_vars` to use the SQL/Protect feature of EDB Postgres Advanced Server:

```
extra_postgres_extensions: [ 'sql_protect' ]
```

The following entries are added to `cluster_vars` to force clients to use SSL authentication:

```
hba_force_hostssl: True
hba_force_certificate_auth: True
hba_cert_authentication_map:
sslmap
```

The following entries are added to `cluster_vars` to set GUCs in postgresql.conf:

```
tcp_keepalives_idle: 10
tcp_keepalives_interval: 10
tcp_keepalives_count: 10
log_destination: "stderr"
postgres_log_file_mode: "0600"
```

The following entries are added to `postgres_conf_settings` in `cluster_vars` to set GUCs in postgresql.conf:

```
edb_audit: "xml"
edb_audit_statement: "all"
edb_audit_connect: "all"
edb_audit_disconnect: "all"
statement_timeout: 1000
client_min_messages: "ERROR"
```

Deployment differences

During deployment, TPA will set connection limits for the database users it creates, corresponding to the number of connections that are needed for normal operation. As each user is set up, it will also check that an SSL client certificate has been provided for it.

Providing client ssl certificates

STIG requires DOD-approved ssl certificates for client connections. These certificates can't be generated by TPA and therefore must be supplied. When setting up authentication for a user from a node in the cluster, TPA will look for a certificate/key pair on the node. The certificate and key should be in files called `<username>.crt` and `<username>.key` in the directory given by the `ssl_client_cert_dir` setting. The default for this setting is `/`, so the files would be, for example, `/barman.crt` and `/barman.key` when the `barman` user is being set up.

Final checks

At the end of deployment, TPA will check that the server has FIPS enabled.

Exceptions

If you select EFM as the failover manager, TPA will configure password authentication for the EFM user. This goes against the STIG requirement that all TCP connections use certificate authentication. The reason for this exception is that EFM does not support certificate authentication.

CIS

CIS compliance is indicated by the `--compliance cis` option to `tpaexec configure`.

Settings in config.yml

The following entries are added to `cluster_vars` to set GUCs in postgresql.conf:

```
log_connections: "on"
log_disconnections: "on"
```

The following entry is added to `cluster_vars` to enable required extensions:

```
extra_postgres_extensions: ["passwordcheck", "pgaudit"]
```

On EPAS clusters, `pgaudit` is omitted from this list, because EPAS has built-in audit logging.

The following entry is added to `cluster_vars` to set the umask for the postgres OS user:

```
extra_bash_rc_lines: "umask 0077"
```


The following entries are added to `postgres_conf_settings` in `cluster_vars` to set GUCs in postgresql.conf:

```
log_error_verbosity: "verbose"
log_line_prefix: "'%m [%p]: [%l-1] db=%d,user=%u,app=%a,client=%h '"
log_replication_commands: "on"
temp_file_limit: "1GB"
```

Final checks

At the end of deployment, TPA will check that the server has FIPS enabled.

Exceptions

TPA does not support pgBackRest as mentioned in the CIS specification. Instead TPA installs Barman.

TPA does not install and configure `set_user` as required by the CIS specification. This is because preventing logon by the Postgres user would leave TPA unable to connect to, and configure, the database.

34.45 etcd

`etcd` is a distributed, reliable key-value store commonly used for distributed coordination. TPA configures `etcd` primarily as the Distributed Configuration Store (DCS) when using Patroni as the failover manager for PostgreSQL clusters.

TPA automatically deploys and configures a 3-node `etcd` cluster by default when you enable Patroni in the M1 architecture.

Installation

TPA installs the `etcd` package available from the configured system repositories. On SLES and RHEL-based systems, TPA automatically enables the PGDG `extras` repository to provide the `etcd` package.

etcd package version

By default, TPA installs the latest available version of etcd.

The version of the etcd package that is installed can be specified by including `etcd_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:
…
    etcd_package_version: '3.6.2*'
…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try prepending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

Configuration

TPA generates the `/etc/etcd/etcd.conf` file based on your `config.yml` settings and cluster topology. It configures peer and client communication URLs, data directory, and SSL certificates.

Security

TPA provides robust options for securing communication with and within the `etcd` cluster. This is controlled primarily by the `etcd_ssl_enabled` and `etcd_authentication_mode` variables.

TLS Encryption

Setting `etcd_ssl_enabled: true` enables TLS encryption for all `etcd` traffic. TPA will automatically:

- Generate a Cluster CA and server certificates (including appropriate IP and DNS Subject Alternative Names) for each etcd node.
- Configure etcd to use HTTPS for both peer-to-peer and client-server communication.
- Configure clients (`etcdctl` , Patroni) to connect via HTTPS and validate the server certificate using the CA.

Client Authentication

Once TLS encryption is enabled (`etcd_ssl_enabled: true`), you can choose an authentication mode using the `etcd_authentication_mode` variable:

- `none` **(default):** No client authentication is performed. Communication relies solely on TLS encryption if `etcd_ssl_enabled` is `true` .
- `basic` : Requires clients to authenticate using a username and password. TPA automatically:
  - Creates an administrative `root` user with a generated password.
  - If using Patroni, creates a dedicated, least-privilege user for Patroni (configurable via `patroni_etcd_user`) with `readwrite` access only to its specific key prefix (e.g., `/tpa/cluster_name`).
  - Configures Patroni to use these dedicated credentials.
- `mtls` : Requires clients (including peer nodes) to present a valid TLS certificate signed by the trusted cluster CA. TPA automatically:
  - Configures `etcd` to require and validate client certificates for both peer (`ETCD_PEER_CLIENT_CERT_AUTH=true`) and client (`ETCD_CLIENT_CERT_AUTH=true`) connections.
  - Configures clients (`etcdctl` , Patroni) to present their client certificates for authentication.

> **Note**
>
> TPA includes robust logic to handle transitions between different authentication modes during deployment or reconfiguration, ensuring the cluster remains stable. Configuration validation checks are performed early to prevent invalid combinations (e.g., `mtls` requires `etcd_ssl_enabled: true`). However, it's currently not able to handle transition of values for `etcd_ssl_enabled` (i.e. from `false` to `true` or vice-versa). This limitation also makes it impossible to switch between any `etcd_authentication_mode` values that require different values of `etcd_ssl_enabled` . For example, `basic` to `mtls` , or `mtls` to `basic` . Clusters deployed using a version of TPA below 23.41.0 defaulted to `etcd_ssl_enabled: false` , and so it's not possible to modify `etcd_ssl_enabled` .

Configuration Variables

You can set the following variables for `etcd` .

Variable	Default value	Description
`etcd_peer_port`	`2380`	The TCP port `etcd` uses for peer-to-peer (server-to-server) communication.
`etcd_client_port`	`2379`	The TCP port `etcd` uses for client communication.
`etcd_ssl_enabled`	`false` *	Enable SSL/TLS encryption for all `etcd` communication. See Security. * `true` for new clusters via `tpaexec configure` .
`etcd_authentication_mode`	`none` *	Defines the client authentication mode (`none` , `basic` , `mtls`). Requires `etcd_ssl_enabled: true` for modes other than `none` . See Security. * `mtls` for new clusters via `tpaexec configure` .
`etcd_compaction_mode`	`revision`	The automatic compaction mode (`revision` or `periodic`).
`etcd_compaction_retention`	`10`	The retention value for automatic compaction. For `revision` mode, this is the number of revisions to keep. For `periodic` mode, this is the time interval (e.g., `1h`).

Minor update for etcd using `tpaexec upgrade`

> **ETCDCTL_API v2 is deprecated**
>
> etcd v3.4 introduced a breaking change, making ETCDCTL_API v3 the default version, but allowed the v2 API to be enabled manually.
>
> etcd v3.6 completely removed the `--enable-v2` flag, making it impossible to use the v2 API at all.
>
> TPA upgrade ONLY supports upgrade of `etcd` versions using the v3 API.
>
> Before using TPA to upgrade `etcd` in a cluster, ensure it has migrated from the v2 API to the v3 API

When trying to upgrade to a specific package version, ensure the `etcd_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade -e etcd_package_version="<desired version>" --components=etcd
```

Refer to the section on package version selection and upgrade for more information.

To select etcd for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `etcd` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.46 Locale

For some platform images and environments it might be desirable to set the region and language settings.

By default, TPAexec will install the `en_US.UTF-8` locale system files. You can set the desired locale in your `config.yml`:

```
user_locale: en_GB.UTF-8
```

To find supported locales consult the output of the following command:

```
localectl list-locales
```

Or the contents of the file /etc/locales.defs on Debian or Ubuntu.

34.47 Patroni cluster management commands

Patroni can be used as a single master failover manager with the M1 architecture using the following command options.

```
tpaexec configure cluster_name -a M1 --enable-patroni --postgresql 14
```

Or by setting the config.yml option

```
cluster_vars:
  failover_manager: patroni
```

TPA is able to deploy Patroni clusters using either `patroni` packages (from PGDG repositories) or `edb-patroni` packages (from EDB repositories). You can configure that through the `patroni_package_flavour` option under `cluster_vars` in the config.yml, which can also be set through the `--patroni-package-flavour` command-line argument. If no `patroni_package_flavour` is explicitly set, TPA will attempt to infere the flavour based on the configured repositories: if EDB repositories were configured, implicitly select `edb` flavour, otherwise implicitly select `community` flavour.

TPA `configure` will add 3 etcd nodes, and may add 2 haproxy nodes if you specify the option `--enable-haproxy`. Etcd is used for the Distributed Configuration Store (DCS). Patroni supports other DCS backends, but they are not currently supported by EDB or TPA.

As an alternative to HAProxy, you can use the `--enable-pgbouncer` option to configure PgBouncer in the Postgres nodes. PgBouncer will be configured to pool connections for the primary. Patroni will be configured to reconfigure PgBouncer upon failovers or switchovers in the cluster, so PgBouncer follows the new primary Postgres instance.

TPA uses Patroni's feature of converting an existing PostgreSQL cluster. This allows for TPA to initialise and manage configuration. Once the PostgreSQL cluster has been created, Patroni will take the management over. TPA will then remove any postgres configuration files used during setup.

Once set up, Postgres can continue to be managed using TPA and settings in `config.yml` for the cluster. You can also use Patroni interfaces, such as the command line `patronictl` and the REST API, but it is recommended to use TPA methods wherever possible.

Patroni package version

By default, TPA installs the latest available version of Patroni.

The version of the Patroni package that is installed can be specified by including `patroni_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:

…
    patroni_package_version: '4.0.0-1PGDG.rhel8`'

…
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try appending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

TPA provides robust options for securing the Patroni REST API. This is controlled primarily by the `patroni_ssl_enabled` and `patroni_authentication_mode` variables.

TLS Encryption

Setting `patroni_ssl_enabled: true` enables TLS encryption for the REST API. TPA will automatically:

- Generate a TPA cluster CA and server certificates for each Patroni node.
- Configure the Patroni REST API to serve traffic over HTTPS.
- Configure clients (like `patronictl` and HAProxy) to connect via HTTPS and validate the server certificate using the CA.

Client Authentication

Once TLS encryption is enabled (`patroni_ssl_enabled: true`), you can choose an authentication mode using the `patroni_authentication_mode` variable:

- `basic` **(default):** Requires clients to provide the configured Patroni REST API username and password. TPA generates these credentials automatically.

- `mtls` : Requires clients to present a valid TLS certificate signed by the trusted cluster CA. TPA automatically:

    - Configures Patroni REST API to require and validate client certificates (`verify_client: required`).
    - Configures clients (`patronictl` , HAProxy) with the necessary client certificates and keys.

API Access Control (Allowlist)

For enhanced security, Patroni's REST API is automatically configured with an `allowlist` . This list restricts access to potentially harmful API endpoints (those using `POST` , `PUT` , `PATCH` , `DELETE`) to only the other Patroni nodes within the cluster, based on the hostnames defined in the TPA inventory. Read-only endpoints (like health checks) remain accessible from other sources (e.g., HAProxy).

These configuration variables can be used to control certain behaviours in the deployment of Patroni in TPA.

Variable	Default value	Description
`patroni_super_user`	postgres	User to create in postgres for superuser role.
`patroni_replication_user`	replicator	Username to create in postgres for replication role.
`patroni_restapi_user`	patroni	Username to configure for the patroni REST API.
`patroni_rewind_user`	rewind	Username to create in postgres for pg_rewind function.
`patroni_installation_method`	pkg	Install patroni from packages or source (e.g. git repo or local source directory if docker).
`patroni_package_flavour`	community if no EDB repository is configured, else edb	Whether to install `edb-patroni` package (`edb` flavour, requires EDB repositories) or `patroni` package (`community` flavour, requires PGDG and EPEL (RedHat based only) repositories).
`patroni_ssl_enabled`	no	Whether to enable SSL for REST API and ctl connection. Will use the cluster SSL cert and CA if available.
`patroni_authentication_mode`	`basic`	Defines the client authentication mode (`basic` or `mtls`) for the REST API. Requires `patroni_ssl_enabled: true` .
`patroni_rewind_enabled`	yes	Whether to enable postgres rewind, creates a user defined by patroni_rewind_user and adds config section.

Variable	Default value	Description
`patroni_watchdog_enabled`	no	Whether to configure the kernel watchdog for additional split brain prevention.
`patroni_dcs`	etcd	What backend to use for the DCS. The only option is etcd at the moment.
`patroni_dcs_namespace`	`/tpa`	The namespace within the DCS under which Patroni stores its cluster state.
`patroni_listen_port`	8008	REST API TCP port number
`patroni_etcd_user`	etcd_patroni	The username for the dedicated etcd user Patroni uses when `etcd_authentication_mode` is `basic`.
`patroni_conf_settings`	{}	A structured data object with overrides for patroni configuration. Partial data can be provided and will be merged with the generated config. Be careful to not override values that are generated based on instance information known at runtime.
`patroni_dynamic_conf_settings`	{}	Optional structured data just for DCS settings. This will be merged onto `patroni_conf_settings`.
`patroni_repl_max_lag`	None	This is used in the haproxy backend health check only when `haproxy_read_only_load_balancer_enabled` is true. See REST API documentation for possible values for `/replica?lag`

> **Note**
>
> When deploying via Ansible Tower, changing `patroni_etcd_user` from its default value will cause the deployment to fail. This happens because TPA cannot handle dynamic secret names in its inventory. To avoid this issue, you must manually generate the password locally and push it to your configuration repository after provisioning and before deploying:
>
> ```
> tpaexec store-password <your_custom_username> .
> ```

Patroni configuration file settings

Configuration for patroni is built from three layers, starting with defaults set by the Patroni daemon, config loaded from the DCS, and finally from local configuration. The last can be controlled from either configuration file and overrides via the environment. TPA controls the configuration file and values are built up in this order.

DCS config to be sent to the API and stored in the bootstrap section of the config file:

- TPA vars for `postgres` are loaded into the DCS settings, see postgresql.conf.md. Some features are not supported, see notes below.
- Patroni defaults for DCS settings
- User supplied defaults in `patroni_dynamic_conf_settings`, if you want to override any DCS settings you can do that here.

Local config stored in the YAML configuration file:

- `bootstrap.dcs` loaded from previous steps above.
- configuration enabled by feature flags, such as `patroni_ssl_enabled`, see table above.
- then finally overloaded from user supplied settings, the `patroni_conf_settings` option. If you want to change or add configuration not controlled by a feature flag then this is the best place to do it.

Please note that configuration is *merged* on top of configuration generated by TPA from cluster information, such as IP addresses, port numbers, cluster roles, etc. Exercise caution in what you override as this might affect the stable operation of the cluster.

As Patroni stores all postgres configuration in the DCS and controls how and when this is distributed to postgres, some features of TPA are incompatible with patroni:

- It is not possible to change the template used to generate `postgresql.conf` with the setting `postgres_conf_template`.
- You cannot change the location of Postgres config files with the setting `postgres_conf_dir`.

Patroni configuration in TPA `config.yml`

You can override single values:

```
cluster_vars:
  patroni_conf_settings:
    bootstrap:
      dcs:
        ttl: 120
```

Or full blocks (with an example from Patroni docs):

```
cluster_vars:
  patroni_conf_settings:
    restapi:
      http_extra_headers:
        'X-Frame-Options': 'SAMEORIGIN'
        'X-XSS-Protection': '1;
mode=block'
        'X-Content-Type-Options': 'nosniff'
      https_extra_headers:
        'Strict-Transport-Security': 'max-age=31536000; includeSubDomains'
```

If you want to negate a value or section that is present in the default TPA config vars you can set the value to `null`. This will cause patroni to ignore this section when loading the config file.

For example the default TPA config for `log` is

```
log:
  dir:
/var/log/patroni
```

To turn off logging add this to `config.yml`:

```
cluster_vars:
  patroni_conf_settings:
    log: null
```

Etcd Integration

When `etcd` is used as the DCS, TPA automatically configures Patroni to connect to it based on the `etcd` security configuration:

- **Protocol:** Uses `http` or `https` based on `etcd_ssl_enabled`.
- **CA Certificate:** Provides the cluster CA certificate path when TLS is enabled.

- **Authentication:**
  - If `etcd_authentication_mode` is `basic`, Patroni is configured with the username (`patroni_etcd_user`) and its generated password.
  - If `etcd_authentication_mode` is `mtls`, Patroni is configured with the paths to its client certificate and key.

TPA provides these minimal set of tools for managing Patroni clusters.

Status

To see the current status of the TPA cluster according to Patroni run

```
tpaexec status cluster_name
```

Switchover

To perform a switchover to a replica node (e.g. to perform maintenance) run the command

```
tpaexec switchover cluster_name new_primary
```

The new_primary argument must be the name of an existing cluster node that is currently running as a healthy replica. Checks will be performed to ensure this is true before a switchover is performed.

Once a switchover has been performed it is recommended that you run `deploy` and `test` to ensure a healthy cluster.

```
tpaexec deploy cluster_name
tpaexec test cluster_name
```

TPA will detect the current role of nodes during deploy regardless of what config.yml contains, for example if a different node is the leader.

Minor update for patroni using `tpaexec upgrade`

When trying to upgrade to a specific package version, ensure the `patroni_package_version` in `config.yml` is updated to reflect the desired version.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> -e patroni_package_version="<desired version>" --components=patroni
```

Refer to the section on package version selection and upgrade for more information.

To select Patroni for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `patroni` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.48 pg_hba.conf

The Postgres documentation explains the various options available in `pg_hba.conf`.

By default, TPA will generate a sensible `pg_hba.conf` for your cluster, to allow replication between instances, and connections from authenticated clients.

You can add entries to the default configuration by providing a list of `postgres_hba_settings`:

```
cluster_vars:
  postgres_hba_settings:
  - "# let authenticated users connect from
anywhere"
  - hostssl all all 0.0.0.0/0 scram-sha-
256
```

You can override the default `local all all peer` line in pg_hba.conf by setting `postgres_hba_local_auth_method: md5`.

If you don't want any of the default entries, you can change `postgres_hba_template`:

```
cluster_vars:
  postgres_hba_template: pg_hba.lines.j2
  postgres_hba_settings:
  - "# my lines of text"
  - "# and nothing but my
lines"
  - "# …not even any
clients!"
  - hostssl all all 0.0.0.0/0
reject
```

You can even create `templates/my_hba.conf.j2` in your cluster directory and set:

```
cluster_vars:
  postgres_hba_template:
my_hba.conf.j2
```

If you put any template files outside the cluster directory's `templates` subdirectory, make sure to specify the absolute path to the file:

```
# in the root of the cluster
directory
cluster_vars:
  postgres_hba_template: "{{ cluster_dir
}}/my_hba.conf.j2"
```

```
# in a subdirectory of the cluster directory that is NOT
'templates'
cluster_vars:
  postgres_hba_template: "{{ cluster_dir
}}/subdirectory/my_hba.conf.j2"
```

```
# in a directory outside of the cluster
directory
cluster_vars:
  postgres_hba_template: /path/to/file/outside/cluster_dir/my_hba.conf.j2
```


If you just want to leave the existing `pg_hba.conf` alone, you can do that too:

```
cluster_vars:
  postgres_hba_template: ''
```

Although it is possible to configure `pg_hba.conf` to be different on different instances, we generally recommend a uniform configuration, so as to avoid problems with access and replication after any topology-changing events such as switchovers and failovers.

34.49 pg_ident.conf

You should not normally need to change `pg_ident.conf`, and by default, TPA will not modify it.

You can set `postgres_ident_template` to replace `pg_ident.conf` with whatever content you like.

```
cluster_vars:
    pg_ident_template:
ident.j2
```

You will also need to create `templates/ident.j2` in the cluster directory:

```
{% for u in ['unixuser1', 'unixuser2'] %}
mymap {{ u }} dbusername
{% endfor %}
```

34.50 pgd-cli

pgd-cli package version

By default, TPA installs the latest available version of `pgd-cli`.

> **Only pgd-cli v1 packages are supported for BDR-Always-ON architectures**
>
> TPA only installs `pgd-cli` in BDR-Always-ON clusters if `pgdcli` is explicitly added to the `role` list of a BDR node.

The version of the `pgd-cli` package that is installed can be specified by including `pgdcli_package_version: xxx` under the `cluster_vars` section of the `config.yml` file.

```
cluster_vars:
    pgdcli_package_version: '5.0.0-1'
```

You may use any version specifier that apt or yum would accept.

If your version does not match, try prepending a `*` wildcard. This is often necessary when the package version has an epoch qualifier like `2:...`.

Updating pgd-cli using `tpaexec upgrade`

> **Note**
>
> The documentation refers to this component as `pgd-cli` but the name to pass to the `--components` list is `pgdcli`.

When trying to upgrade to a specific package version, ensure the `pgdcli_package_version` in `config.yml` is updated to reflect the desired version. This version should be kept in line with the `bdr_package_version` and `pgd_proxy_package_version` versions specified for the cluster. Defining `bdr_package_version` and omitting `pgd_proxy_package_version` and `pgdcli_package_version` from `config.yml` will ensure the same value is used across all three components.

The desired version can also be passed as an extra argument to the `tpaexec upgrade` command with:

```
tpaexec upgrade <cluster_dir> -e pgdcli_package_version="<desired version>" \
    --components=pgdcli
```

Refer to the section on package version selection and upgrade for more information.

To select pgd-cli for upgrade, ensure the `--components` flag passed to the `tpaexec upgrade` command contains `pgdcli` (or `all`)

Refer to the section on component selection for upgrade for more information.

34.51 Adding Postgres extensions

Default Postgres extensions

By default, TPA adds the following extensions to every Postgres database (and if needed, automatically adds the corresponding entries into shared preload libraries)

- pg_stat_statements
- pg_freespacemap
- pg_visibility
- pageinspect
- pgstattuple

User defined extensions

Additional extensions can be configured within `config.yml`, by specifying the extension name, any required shared preload entries and the package containing the extension.

When adding extensions, be sure to include both the package name to `extra_postgres_packages` and the extension name under `extra_postgres_extensions` (or to the `extensions` list of a database defined under `postgres_databases`).

If the extension requires, add the shared preload entry name for the extension to the `preload_extensions` list. Note this name may differ from the extension name itself, so be sure to check the extension's own documentation.

Here is a quick example for an extension that requires to be added to the shared preload extension list with a different entry for extension and library name.

```
cluster_vars:
  [...]
  extra_postgres_packages:
    common:
      - postgresql-17-my-
extension
  extra_postgres_extensions:
    - my-
extension
  preload_extensions:
    -  my_extension
```

The following sections provide further information.

- Adding the *vector* extension through configuration
- Specifying extensions for configured databases
- Including shared preload entries for extensions
- Installing Postgres-related packages

TPA recognised extensions

The following list of extensions only require the extension name to be added in `config.yml` (either to `extra_postgres_extensions` OR to the `extensions` list of a database specified in `postgres_databases`) and TPA will automatically include the correct package and any required entries to shared_preload_libraries.

- edb_pg_tuner

- query_advisor
- edb_wait_states
- sql_profiler
- pg_failover_slots
- sql_protect
- edb_stat_monitor
- autocluster
- refdata
- bluefin
- postgis
- pgaudit
- passwordcheck

34.52 postgresql.conf

TPA creates a `conf.d` directory with various `.conf` files under it, and uses `include_dir` in the main `postgresql.conf` to use these additional configuration files.

The Postgres configuration files (postgresql.conf, pg_ident.conf, and pg_hba.conf) and the included files under `conf.d` are always stored in `postgres_conf_dir`. This is the same as `postgres_data_dir` by default, but you can set it to a different location if you wish to keep the configuration separate from the data directory.

The main configuration mechanism is to set variables directly:

```
cluster_vars:
  temp_buffers: 16MB
  log_connections: on
  autovacuum_vacuum_cost_limit: -1
  effective_cache_size:
4GB
  max_connections: 300
  max_wal_senders: 32
```

TPA splits the configuration up into multiple files. The two main files are `0000-tpa.conf` and `0001-tpa_restart.conf`. These contain settings that require a server reload or restart to change, respectively. During deployment, TPA will write any changes to the correct file and reload or restart Postgres as required.

TPA may use other files in certain circumstances (e.g., to configure optional extensions), but you do not ordinarily need to care where exactly a given parameter is set.

You should never edit any of the files under `conf.d`, because the changes may be overwritten when you next run `tpaexec deploy`.

postgres_port

The Postgres port can be overwritten from its default value to another valid port number.

```
cluster_vars:
  postgres_port: 5433
```

This method is preferred over setting the port via `postgres_conf_settings`, since `postgres_port` ensures that the value is correctly shared and used across all components relying on this value to interact with the postgres cluster.

postgres_conf_settings

TPA provides variables like `temp_buffers` and `maintenance_work_mem` that you can set directly for many, but not all, available postgresql.conf settings.

You can use `postgres_conf_settings` to set any parameters, whether recognised by TPA or not. You need to quote the value exactly as it would appear in `postgresql.conf`:

```
cluster_vars:
  effective_cache_size:
2GB
  postgres_conf_settings:
    effective_cache_size:
4GB
    authentication_timeout: 1min
    synchronous_standby_names: >-
      'any 2 ("first", "second",
"third")'
    bdr.global_lock_statement_timeout:
60s
```

This is most useful with settings that TPA does not recognise natively, but you can use it for any parameter (e.g., `effective_cache_size` can be set as a variable, but `authentication_timeout` cannot).

These settings will be written to `conf.d/9900-role-settings.conf`, and therefore take priority over variables set in any other way.

If you make changes to values under `postgres_conf_settings`, TPA has no way to know whether the a reload is sufficient to effect the changes, or if a restart is required. Therefore it will always restart the server to activate the changes. This is why it's always preferable to use variables directly whenever possible.

shared_buffers

By default, TPA will set `shared_buffers` to 25% of the available memory (this is just a rule of thumb, not a recommendation). You can override this default by setting `shared_buffers_ratio: 0.35` to use a different proportion, or by setting `shared_buffers_mb: 796` to a specific number of MB, or by specifying an exact value directly, e.g., `shared_buffers: "2GB"`.

huge_page_size

TPA reserves hugepages of the architecture's default size for Postgres to use. To use a different size, seehugepages: setting `huge_page_size` reserves pages of that size on the kernel command line and sets `huge_page_size` in `postgresql.conf` accordingly.

effective_cache_size

By default, TPA will set `effective_cache_size` to 50% of the available memory. You can override this default by setting `effective_cache_size_ratio: 0.35` to use a different proportion, or by setting `effective_cache_size_mb: 796` to a specific number of MB, or by specifying an exact value directly, e.g., `effective_cache_size: "8GB"`.

max_active_replication_origins

PostgreSQL 18 introduced the `max_active_replication_origins` GUC to control the number of simultaneously active replication origins. Before PostgreSQL 18, this was implicitly limited by `max_replication_slots`.

For PGD clusters on PostgreSQL 18 and above, TPA sets `max_active_replication_origins` to `3 * number_of_nodes + 3`. This stays comfortably above the "3 per peer node" minimum recommended by the EDB PGD documentation, leaving a safety margin for transient origins created during node join or sync operations.

For non-PGD clusters, TPA leaves the PostgreSQL default (10) in place.

You can override this by setting the variable directly:

```
cluster_vars:
  max_active_replication_origins: 50
```

This setting is ignored on PostgreSQL versions before 18.

shared_preload_libraries

TPA maintains an internal list of extensions that require entries in `shared_preload_libraries` to work, and if you include any such extensions in `postgres_extensions`, it will automatically update `shared_preload_libraries` for you.

If you are using unrecognised extensions that require preloading, you can add them to `preload_extensions`:

```
cluster_vars:
  preload_extensions:
  - myext
  -
otherext
```

Now if you add `myext` to `postgres_extensions`, `shared_preload_libraries` will include `myext`.

By default, `shared_preload_libraries` is set in `conf.d/8888-shared_preload_libraries.conf`.

Setting `shared_preload_libraries` directly as a variable is not supported. You should not normally need to set it, but if unavoidable, you can set a fully-quoted value under `postgres_conf_settings`. In this case, the value is set in `conf.d/9900-tpa_postgres_conf_settings.conf`.

Postgres logging

By default, TPA configures the Postgres `log_destination` GUC as `syslog` and configures rsyslog to write Postgres logs to `/var/log/postgres/postgres.log`.

You can change these defaults using the following cluster variables.

- `postgres_log_file` is the path to the log file. Defaults to `/var/log/postgres/postgres.log`.
- `postgres_log_file_mode` is the mode of the log file. Defaults to `0640`.
- `postgres_log_directory_mode` is the mode of the log directory. Defaults to `0700`.
- `log_destination` sets the Postgres GUC of the same name. Defaults to `syslog`.
- `logging_collector` sets the Postgres GUC of the same name. Defaults to `off` if `log_destination` is `syslog`, otherwise `on`.

If you select a `log_destination` other than `syslog` TPA will set up the Postgres logging collector to write the logs. In all cases, TPA will take care of creating the directories and configuring log rotation.

The following example tells TPA to log to the selected location using `rsyslog`.

```
cluster_vars:
  [...]
  postgres_log_file: '/srv/fantastic_logs/pg_server.log'
```

This example, logs to the same location, but using the Postgres logging collector to write JSON logs, and permitting read and execute on the log directory for group members.


```
cluster_vars:
  [...]
  postgres_log_file: '/srv/fantastic_logs/pg_server.log'
  postgres_log_directory_mode: '0750'
  log_destination: 'jsonlog'
```

> File extensions for log files
>
> When `log_destination` is `syslog` or `stderr`, the exact value of `postgres_log_file` will be used for the current log file, including any extension. When `log_destination` is `jsonlog` or `csvlog`, the specified `postgres_log_file` will have `.json` or `.csv` appended. If `.log` was included in the specified `postgres_log_file`, it will be removed. This behaviour comes from Postgres rather than from TPA.
>
> If you need to access the exact path of the log file, including the final extension—for example as part of a hook—this is stored in the Ansible fact `postgres_log_file_with_extension`. This cannot be set directly.

SSL configuration

By default, TPA will generate a private key and a self-signed TLS certificate which are used by Postgres as the `ssl_key_file` and `ssl_cert_file` respectively. The files are named using the TPA cluster name (`cluster_name.key` and `cluster_name.crt`) and located in `/etc/tpa`, resulting in the following default configuration in `0001-tpa_restart.conf`:

```
ssl_key_file=/etc/tpa/cluster_name.key
ssl_cert_file=/etc/tpa/cluster_name.crt
```

This is sufficient to ensure that traffic between clients and server is encrypted in transit.

To provide your own certificates, upload them to the target nodes as artifacts, then set the path by specifying the following cluster variables:

```
cluster_vars:

...
  artifacts:
  - type: file
    dest:
/path/to/your_key.key
    src: /local/path/to/your_key.key
    owner: root
    group: root
    mode: "0644"
  - type: file
    dest: /path/to/your_cert.crt
    src: /local/path/to/your_cert.crt
    owner: root
    group: root
    mode: "0600"
  ssl_key_file:
/path/to/your_key.key
  ssl_cert_file: /path/to/your_cert.crt
```

Alternatively, if you upload your key and certificate to the default location, TPA will use them instead of generating its own, and you do not need to specify `ssl_key_file` or `ssl_cert_file`. Note, however, that you must explicitly create `/etc/tpa` because it doesn't exist at the time artifacts are uploaded. The permissions and ownership of these files will be adjusted by TPA when the `postgres` user is created during deployment.

```
cluster_vars:

...
  artifacts:
  - type: path
    path:
/etc/tpa
    state: directory
    owner: root
    group: root
    mode: "0755"
  - type: file
    dest: /etc/tpa/cluster_name.key
    src: /local/path/to/your_key.key
    owner: root
    group: root
    mode: "0644"
  - type: file
    dest: /etc/tpa/cluster_name.crt
    src: /local/path/to/your_cert.crt
    owner: root
    group: root
    mode: "0600"
```

> **Other SSL settings**
>
> TPA does not specify `ssl_ca_file` or `ssl_crl_file` by default. To provide these files yourself you can do so using artifacts and by specifying the cluster variables of the same name.

Making changes by hand

There are two ways you can override anything in the TPA-generated configuration.

The first (and recommended) option is to use `ALTER SYSTEM`, which always takes precedence over anything in the configuration files:

```
ALTER SYSTEM SET bdr.global_lock_statement_timeout TO '60s';
```

You can also edit `conf.d/9999-override.conf`:

```
echo "bdr.global_lock_statement_timeout='60s'" >> conf.d/9999-override.conf
```

All other files under `conf.d` are subject to be overwritten during deployment if the configuration changes, but TPA will never change `9999-override.conf` after initially creating the empty file.

Depending on which settings you change, you may need to execute `SELECT pg_reload_conf()` or restart the server for the changes to take effect.

Generating postgresql.conf from scratch

By default, TPA will leave the default (i.e., `initdb`-generated) postgresql.conf file alone other than adding the `include_dir`. You should not ordinarily need to override this behaviour, but you can set `postgres_conf_template` to do so:

```
cluster_vars:
  postgres_conf_template: 'pgconf.j2'
```

Now the `templates/pgconf.j2` in your cluster directory will be used to generate postgresql.conf.

34.53 tpaexec deprovision

Deprovision destroys a cluster and associated resources.

For a cluster using the `aws` platform, it will remove the instances and all keypairs, policies, volumes, security groups, route tables, VPC subnets, internet gateways and VPCs which were set up for the cluster.

For a cluster using the `docker` platform, it will remove the containers, any ccache directories which were set up for source builds in the containers, and any docker networks which were set up for the cluster.

For all platforms, it will remove all the files created locally by `tpaexec provision`, including ssh keys, stored passwords, ansible inventory, and logs.

34.54 tpaexec info

You can use the info command to output information about the TPA installation. Providing this information is valuable for troubleshooting.

Usage

- Run `tpaexec info`

Subcommands

- `tpaexec info version`

  Displays current TPA version

- `tpaexec info platforms`

  Displays available deployment platforms

- `tpaexec info architectures`

  Displays available deployment architectures

- `tpaexec info platforms/<name>`

  Displays information about a particular platform

- `tpaexec info architectures/<name>`

  Displays information about a particular architecture

Example Output

The `tpaexec info` command outputs the following:

```
# TPAexec 23.29
tpaexec=/opt/EDB/TPA/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.12.18, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.16.3)
Validated: ea844d1b90295597d080bbf824dbbc6954886cb54ffdb265c7c71b99bedee67b [OK]
```


34.55 tpaexec reconfigure

The `tpaexec reconfigure` command reads config.yml and generates a revised version of it that changes the cluster in various ways according to its arguments.

Arguments

As with other tpaexec commands, the cluster directory must always be given.

Changing a cluster's architecture

The following arguments enable the cluster's architecture to be changed:

- `--architecture <architecture>` (required)
  The new architecture for the cluster. Accepts `PGD-Always-ON` and `PGD-X` as valid arguments

- `--pgd-proxy-routing <global|local>` (required for `PGD-Always-ON`)
  How PGD-Proxy is to route connections. See the PGD-Always-ON documentation for more information about the meaning of this argument.

- `--edb-repositories <repositories>` (optional)
  A space-separated list of EDB package repositories. It is usually unnecessary to specify this; `tpaexec configure` will choose a suitable repository based on the postgres flavour in use in the cluster.

After changing the architecture, run tpaexec upgrade to make the required changes to the cluster.

Changing a cluster from PGD-Proxy to Connection Manager in PGD-Always-ON

- `--enable-connection-manager(required)` The option to allow migration from PGD-Proxy on a PGD 5.9+ cluster to the new builtin Connection Manager.

Changing a cluster from 2q to EDB repositories

The `--replace-2q-repositories` argument removes any 2ndQuadrant repositories the cluster uses and adds EDB repositories as required to replace them.

After reconfiguring with this argument, run [tpaexec deploy)(tpaexec-deploy.md) to make the required changes to the cluster.

Output format

The following options control the form of the output:

- `--describe`
  Shows a description of what would be changed, without changing anything.

- `--check`
  Validates the changes that would be made and shows any errors any errors or warnings that result from validation, without changing anything.

- `--output <filename>`
  Writes the output to a file other than config.yml.

Sample invocation

```
tpaexec reconfigure ~/clusters/speedy\
  --architecture PGD-Always-ON\
  --pgd-proxy-routing local
```

35 tpaexec eval

You can use the eval command to evaluate an arbitrary Jinja/Ansible expression in the context of the cluster's inventory. The information can be related to variables, inventory, gathered OS facts, computed paths, package names, etc.

Usage

- Run `tpaexec eval ~/clusters/speedy '<expression>'`

Options

- `tpaexec eval ~/clusters/speedy '<expression>' --hosts <pattern>`

  Restrict evaluation to a host pattern. The pattern can be a single instance name (`unreal`), a colon-separated list (`unreal:ultra`), `localhost`, or a group name (`role_primary`, `role_etcd`).

- `tpaexec eval ~/clusters/speedy '<expression>' --no-init`

  Skip the cluster initialisation playbook. Only variables derived from `config.yml` and the generated inventory are available; `ansible_*` host facts will be undefined. Recommended for quick inventory inspection.

Example Output

Look up a value from `config.yml` for a single host:

```
tpaexec eval ~/clusters/speedy '{{ postgres_version }}' --no-init --hosts unreal
```

```
output
  unreal => {
    "msg": "17"
}
```

Print a value for every host in the cluster:

```
tpaexec eval ~/clusters/speedy '{{ ip_address }}' --no-init
```

```
output
  unreal => {
    "msg": "10.33.155.2"
}
  ultra => {
    "msg": "10.33.155.3"
}
  quondam => {
    "msg": "10.33.155.4"
}
```

Inspect inventory-wide data by targeting `localhost`:

```
tpaexec eval ~/clusters/speedy '{{ groups.keys() | list }}' --no-init --hosts localhost
```

```
output
  localhost => {
    "msg": [
        "all",
        "ungrouped",
        "tag_Cluster_speedy"
    ]
}
```

List every variable defined for a host (useful when you don't yet know the name of the variable you want):

```
tpaexec eval ~/clusters/speedy '{{ hostvars[inventory_hostname].keys() | list | sort }}' --no-init --hosts
unreal
```

```
output
  unreal => {
    "msg": [
        "ansible_user",
        "backup",
        "cluster_name",
        "failover_manager",
        "ip_address",
        "postgres_version",
        "role",
        "..."
    ]
}
```

Read a fact gathered from the host itself (omit `--no-init` so the initialisation playbook runs and populates `ansible_*` facts):

```
tpaexec eval ~/clusters/speedy '{{ ansible_distribution }}' --hosts unreal
```

```
output
  unreal => {
    "msg": "Debian"
}
```