



Trusted Postgres Architect

Version 23

1	Trusted Postgres Architect	5
2	Trusted Postgres Architect release notes	8
2.1	Trusted Postgres Architect 23.31 release notes	9
2.2	Trusted Postgres Architect 23.30 release notes	9
2.3	Trusted Postgres Architect 23.29 release notes	10
2.4	Trusted Postgres Architect 23.28 release notes	10
2.5	Trusted Postgres Architect 23.27 release notes	11
2.6	Trusted Postgres Architect 23.26 release notes	12
2.7	Trusted Postgres Architect 23.25 release notes	12
2.8	Trusted Postgres Architect 23.24 release notes	13
2.9	Trusted Postgres Architect 23.23 release notes	13
2.10	Trusted Postgres Architect 23.22 release notes	14
2.11	Trusted Postgres Architect 23.21 release notes	14
2.12	Trusted Postgres Architect 23.20 release notes	15
2.13	Trusted Postgres Architect 23.19 release notes	15
2.14	Trusted Postgres Architect 23.18 release notes	16
2.15	Trusted Postgres Architect 23.17 release notes	16
2.16	Trusted Postgres Architect 23.16 release notes	17
2.17	Trusted Postgres Architect 23.15 release notes	17
2.18	Trusted Postgres Architect 23.14 release notes	17
2.19	Trusted Postgres Architect 23.13 release notes	18
2.20	Trusted Postgres Architect 23.12 release notes	18
2.21	Trusted Postgres Architect 23.1 to 23.11 release notes	19
3	TPA installation	25
4	Open source TPA	28
5	Installing TPA from source	29
6	A first cluster deployment	31
7	Cluster configuration	35
8	tpaexec provision	44
9	tpaexec deploy	48
10	tpaexec test	49
11	PGD-Always-ON	50
12	BDR-Always-ON	52
13	M1	53
14	aws	55
15	bare(-metal servers)	61
16	Docker	63
17	Cluster configuration	66
18	Instance configuration	70
19	Building from source	73
20	TPA hooks	75
21	Upgrading your cluster	78
22	tpaexec switchover	81
23	BDR/HAProxy server pool management	82
24	tpaexec rehydrate	82
25	TPA and Ansible Tower/Ansible Automation Platform	84
26	TPA, Ansible, and sudo	88
27	TPA - PuTTY configuration guide	90

28	Troubleshooting	91
29	Running TPA in a Docker container	93
29.1	Managing clusters in a disconnected or air-gapped environment	94
29.2	Distribution support	96
29.3	TPA capabilities and supported software	97
29.4	Reconciling changes made outside of TPA	98
29.5	EDB Postgres Distributed configuration	101
29.6	Barman	104
29.7	Configuring EFM	106
29.8	Configuring haproxy	107
29.9	Configuring HARP	109
29.10	Configuring Postgres Enterprise Manager (PEM)	112
29.11	Configuring pgbouncer	114
29.12	Configuring pgd-proxy	115
29.13	pglogical configuration	117
29.14	Configuring repmgr	121
29.15	How TPA uses 2ndQuadrant and EDB repositories	122
29.16	Configuring EDB Repos 2.0 repositories	124
29.17	Configuring 2ndQuadrant repositories	124
29.18	Configuring APT repositories	125
29.19	Configuring YUM repositories	125
29.20	Creating and using a local repository	126
29.21	Installing from source	129
29.22	Git credentials	130
29.23	Environment	131
29.24	Python environment	131
29.25	Configuring /etc/hosts	132
29.26	Filesystem configuration	133
29.27	Uploading artifacts	138
29.28	ssh_key_file	138
29.29	Managing SSH host keys	139
29.30	Postgres source installation	139
29.31	Installing packages	141
29.32	Running initdb	142
29.33	Installing Postgres-related packages	142
29.34	SSL Certificates	143
29.35	Setting sysctl values	144
29.36	Creating Postgres databases	144
29.37	Creating Postgres tablespaces	145
29.38	postgresql.conf	145
29.39	pg_hba.conf	148
29.40	pg_ident.conf	149
29.41	Configuring .pgpass	149
29.42	The postgres Unix user	150
29.43	Creating Postgres users	151
29.44	tpaexec archive-logs	152
29.45	tpaexec download-packages	154
29.46	TPA custom commands	155

29.47	TPA custom tests	156
29.48	Locale	158
29.49	Patroni cluster management commands	158
29.50	Adding Postgres extensions	161
29.51	tpaexec deprovision	162
29.52	tpaexec info	162
29.53	tpaexec reconfigure	163
30	Selective task execution	164

1 Trusted Postgres Architect

© Copyright EnterpriseDB UK Limited 2015-2024 - All rights reserved.

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 apply to quick testbed setups as well as 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 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, such as the Postgres version to install. The default configuration is ready to use as is, but you can edit it to suit your needs. (The generated configuration is a text file, `config.yml`).

Using this configuration, TPA can:

1. Provision servers, for example, 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, for example, 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, such as PGD, Barman, pgbouncer, repmgr, and various Postgres extensions.
4. Run automated tests on the cluster after deployment.
5. Deploy future changes to your configuration, such as changing Postgres settings, installing and upgrading packages, adding new servers, and so on.

How do you 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 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
```

```
[tpa]$ tpaexec configure clustername --architecture M1 --platform aws
\
    --postgresql 14 \
    --failover-manager
repmgr

# 2. Provisioning: create the servers needed to host the
cluster
[tpa]$ tpaexec provision
clustername

# 3. Deployment: install and configure the necessary
software
[tpa]$ tpaexec deploy
clustername

# 4. Testing: make sure everything is working as
expected
[tpa]$ 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.

For more information, see [TPA 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, for example, 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 (for example, M1) and platform (for example, AWS) that you selected while configuring the cluster.

For example, given AWS access with the necessary privileges, TPA provisions EC2 instances, VPCs, subnets, routing tables, internet gateways, security groups, EBS volumes, elastic IPs, and so on.

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 as if they were created by TPA.

You aren't restricted to a single platform. You can spread your cluster out across some AWS instances (in multiple regions) and some on-premises servers or servers in other data centers as needed.

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

Deployment

The `tpaexec deploy` command installs and configures Postgres and other software on the provisioned servers. (These servers can be created by

TPA, but they don't have to be. It doesn't matter which application created them as 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 is up and running.

Testing

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

At the end of the testing stage, you 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 changed in the configuration or on the instances, then rerunning the entire process doesn't 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 rerunning 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. Create `commands/xyz.yml` in your cluster directory, and invoke it using `tpaexec xyz /path/to/cluster`. Custom commands are 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 don't miss out on anything important, either when dealing with a crisis or during routine cluster management.
- [Hook scripts](#), which are invoked during various stages of the deployment. For example, tasks in `hooks/pre-deploy.yml` are run before the main deployment. There are many other hooks, including `post-deploy`. Using hook scripts gives you easy access to the full range of Ansible functionality.

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's no hidden magic or anything standing in the way between you and the database. You can do everything on a TPA cluster that you can 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, we moved to a derivative of semantic versioning. For historical reasons, we aren't using the full three-part semantic version number. Instead TPA uses a two-part `major.minor` format. The minor version is incremented on every release. The major version is incremented only when required to comply with the backward compatibility principle that follows.

Backward compatibility

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

- A `config.yml` created with TPA X.a is valid with TPA X.b, where $b \geq a$.
- The cluster created from that `config.yml` can be maintained and redeployed 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 do so only 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.

Version	Release date
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

Version	Release date
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	-

2.1 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.

2.2 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 <code>ansible-builder</code> and a <code>python-alpine</code> 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 <code>extra_postgres_extensions</code> list or the <code>extensions</code> list of a database entry in <code>postgres_databases</code> .
Enhancement	The EDB Advanced Storage Pack package and shared preload library entry will automatically be added for <code>bluefin</code> when a user specifies it as an extension and the <code>postgres_version</code> is 15 or greater.
Enhancement	Added a new 'provision_only' option for instances. If an instance has <code>provision_only: true</code> in <code>config.yml</code> , it will be provisioned as normal but not added to the inventory which is seen by <code>tpaexec deploy</code> .
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 <code>bdr.join_node_group()</code> for witness nodes.

Type	Description
Change	Selective execution of tasks is now supported using custom selectors rather than Ansible tags. To run only tasks matching a certain selector: <code>tpaexec deploy . --included_tasks=barman</code> . To skip tasks matching a certain selector: <code>tpaexec deploy . --excluded_tasks=ssh</code> . Task selectors can also be used by specifying the <code>excluded_tasks</code> or <code>included_tasks</code> variables in <code>config.yml</code> .
Change	Ansible 2.9 is no longer supported, neither the community distribution nor the 2ndQuadrant fork. Users who have been using the <code>--skip-tags</code> option to <code>tpaexec deploy</code> should move to the new <code>--excluded_tasks</code> option.

2.3 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 <code>keyring_backend</code> is set to <code>system</code> (default for new clusters). This change does not impact existing clusters or any clusters that set <code>keyring_backend</code> to <code>legacy</code> in <code>config.yml</code> .
Enhancement	The <code>--ansible-version</code> argument to <code>tpaexec setup</code> now accepts <code>8</code> or <code>9</code> as valid ansible versions, as well as the existing <code>2q</code> or <code>community</code> , both of which imply ansible 2.9. The default is now <code>8</code> . Support for ansible 9 is experimental and requires python 3.10 or above.
Bug Fix	Fixed an issue whereby <code>edb_repositories</code> already defined in <code>config.yml</code> are not kept during reconfigure. Fixes bdr4 to pgd5 upgrade scenario in air gapped environment.
Bug Fix	TPA's <code>postgres-monitor</code> will now recognize the message "the database system is not yet accepting connections" as a recoverable error.
Bug Fix	TPA now correctly skips the <code>postgres/config/final</code> 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 <code>apt-get update</code> 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.

2.4 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 <code>postgres_log_file</code> . This option sets the Postgres log file, whether logging through stderr or syslog. The default is <code>'/var/log/postgres/postgres.log'</code> , the previously hard-coded value.
Enhancement	Added a new hook <code>barman-pre-config</code> . 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.

Type	Description
Enhancement	The key <code>elastic_ip</code> on an AWS instance in <code>config.yml</code> 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 <code>harp_manager_user</code> (by default <code>harpmanager</code>), belonging to the <code>bdr_superuser</code> role, and set HARP Manager to operate as this user instead of <code>postgres</code> superuser. This does not affect the existing clusters where TPA will keep using <code>postgres</code> as the HARP Manager user, unless the user overrides this behavior by explicitly setting <code>harp_manager_user</code> to a different value in <code>config.yml</code> .
Bug Fix	Fixed an issue whereby TPA would erroneously attempt to install <code>repmgr</code> 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 <code>repmgr</code> when installing from PGDG repositories.
Bug Fix	Fixed barman connection failure when using <code>selinux</code> and a custom barman home directory.
Bug Fix	TPA will now use the correct cluster name in <code>show-password</code> and <code>store-password</code> 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 <code>--cluster-prefixed-hostnames</code> 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.

2.5 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 <code>edb-python-39</code> 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 <code>--replace-2q-repositories</code> argument to <code>tpaexec reconfigure</code> that will remove 2q repositories from an existing <code>config.yml</code> 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.

Type	Description
Change	A new variable <code>disable_repository_checks</code> 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 <code>primary_slot_name</code> 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 <code>commit_scope</code> for CAMO enabled partners is generated using existing config options from older BDR versions when running <code>tpaexec reconfigure</code> 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 <code>barman_client_dsn_attributes</code> with <code>sslmode=verify-full</code> .
Bug Fix	TPA will now assign a lower default <code>maintenance_work_mem</code> to avoid out-of-memory errors.

2.6 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 <code>--cluster-prefixed-hostnames</code> option to <code>tpaexec configure</code> . 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 <code>require_write_lead</code> 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 <code>hostname</code> rather than <code>bdr_node_name</code> was used when fencing or unfencing a HARP node.
Bug Fix	Fixed an issue whereby <code>provision</code> would be automatically run when <code>deploy</code> was invoked with options that suppress deployment.

2.7 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. <i>Note, upgrading clusters with CAMO is not yet supported.</i>
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.

Type	Description
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 <code>-f</code> switch.
Change	TPA now calls the <code>alter_node_kind</code> 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 <code>tpaexec setup</code> command for <code>tpaexec-deps</code> users.
Bug Fix	Fixed an issue whereby <code>pem_server_group</code> was not correctly applied when <code>pemworker</code> was invoked meaning servers were not grouped as expected in PEM.
Bug Fix	Fixed an issue with the <code>sys/sysstat</code> role whereby <code>sar</code> was not scheduled to run on instances other than the Barman instance.

2.8 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	<code>tpaexec setup</code> now defaults to using community ansible rather than 2ndQuadrant ansible. The option <code>--use-2q-ansible</code> 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 <code>--skip-tags</code> , see the install documentation .
Change	When a repository has been removed from <code>edb_repositories</code> in <code>config.yml</code> , <code>tpaexec deploy</code> now removes it from the nodes.
Change	TPA will now detect when <code>harp-proxy</code> and <code>harp-manager</code> are running on the same node and use a different config file for <code>harp-proxy</code> .
Change	The <code>upgrade</code> command will now update local repositories on target instances.
Bug Fix	Fixed an issue whereby TPA did not respect <code>postgres_wal_dir</code> in <code>pg_basebackup</code> invocation
Bug Fix	TPA will now accept <code>repmgr</code> 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.

2.9 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 <code>tpa_2q_repositories: []</code> 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 <code>pg_backup_api</code> tests were run with incorrect permissions causing them to fail.
Bug Fix	Fixed an issue whereby Apache HTTPD service for <code>pg_backup_api</code> would not start on boot.
Bug Fix	Fixed an issue whereby <code>bdr.standby_slot_names</code> and <code>bdr.standby_slots_min_confirmed</code> 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 <code>docker_images</code> whereby the resulting image name was incorrect.

2.10 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 .

2.11 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 <code>tpaexec configure</code> 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.

Type	Description
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 <code>/etc/edb</code> whereby if you added the <code>pgd-proxy</code> role to a data node in a deployed PGD5 cluster, <code>pgd-proxy</code> would fail to start because it did not have permissions to open <code>pgd-proxy-config.yml</code> .
Bug fix	Fixed an issue whereby <code>/var/log/postgres</code> could end up with inappropriate permissions (0600) if a strict <code>umask</code> was set
Bug fix	Fixed an issue whereby repeating <code>tpaexec deploy</code> on a Barman instance correctly registered with PEM would lose the PEM Agent Barman configuration.

2.12 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 <code>tpaexec reconfigure</code> to generate a revised <code>config.yml</code> and then <code>tpaexec upgrade</code> to perform the upgrade.
Enhancement	Added a new subcommand <code>tpaexec info validate</code> that runs a checksum over the TPA installation and confirms that it matches the one distributed with the package.
Change	The <code>update-postgres</code> command has been replaced with the more general <code>upgrade</code> command.
Change	TPA now explicitly adds <code>tzdata-java</code> 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 <code>tpaexec provision</code> automatically as part of <code>tpaexec deploy</code> or <code>tpaexec upgrade</code> if <code>config.yml</code> has changed.
Bug fix	Fixed a bug whereby TPA could attempt to use a non-existent user when running <code>pgd-cli</code> on <code>pgd-proxy</code> nodes.
Bug fix	Fixed a bug whereby changes made by <code>tpaexec relink</code> were not committed to the Git repository correctly.

2.13 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 <code>edb-pgd-proxy</code> and <code>edb-bdr-utilities</code> rather than always using the latest version.
Change	On Debian-like systems, the package selection code now uses <code>-dbg</code> rather than <code>-dbgsym</code> for certain packages where applicable.
Change	When configuring replication slots, TPA will now ensure that only valid characters are used in the <code>primary_slot_name</code> . Previously TPA would use the <code>inventory_hostname</code> as a default, which could contain hyphens; these are now replaced with underscores.

Type	Description
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 <code>Failed to commit files to git: b''</code> would be displayed during configure.
Bug fix	Fixed an issue whereby TPA would erroneously generate and overwrite Postgres user passwords when <code>generate_password: false</code> .
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.

2.14 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 <code>pg_basebackup</code> instead of <code>repmgr</code> 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 <code>tpaexec test</code> 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 <code>pg_receivewal</code> 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.

2.15 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 <code>--pgd-proxy-routing</code> parameter to the configure command. This can be set to <code>global</code> or <code>local</code> . 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 <code>--active-locations</code> 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.

Type	Description
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.

2.16 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, <code>--add-proxy-nodes-per-location N</code> , which creates separate proxy instances
Enhancement	TPA now adds a witness node automatically if <code>--data_nodes_per_location</code> is even and prints a warning if you specify a cluster with only two locations
Change	The parameter <code>--add-witness-only-location</code> has been renamed to <code>--witness-only-location</code> because we're NOT adding a location, but designating an already-named (in <code>--location-names</code>) location as witness-only.
Change	You must now specify Postgres flavour and version explicitly at <code>tpaexec configure</code> time
Enhancement	Added new CLI abbreviations for Postgres flavour and version, for example <code>--postgresql 14</code> or <code>--edbpgge 15</code>
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

2.17 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.

2.18 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)

2.19 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 <code>--epas</code> .
Bug fix	Fix error with PGD-Always-ON and <code>--postgres-version 15</code> .

2.20 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 <code>--active-locations</code> . 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. Ensure that nodes subscribe to bdr_child_group, if available.
Bug fix	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.

Type	Description
	Fail reliably with a useful error if Postgres doesn't start.
Bug fix	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 <code>cluster_facts</code> for the failed hosts later.
	Don't call <code>bdr.alter_node_replication_sets()</code> on witnesses for BDR 4.3 and later.
Bug fix	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 <code>barman_{user,group}</code> settings to customize the barman user and home directory.
Bug fix	Add <code>shared_preload_libraries</code> entries, where appropriate, for extensions mentioned under <code>postgres_databases[*].extensions</code> .
Bug fix	Ensure that <code>pgaudit</code> does not appear before <code>bdr</code> in <code>shared_preload_libraries</code> (to avoid a known crash).
Bug fix	Fix syntax error (DSN quoting) in <code>pgd-cli</code> config file.
	Sort endpoints in <code>pgd-proxy</code> config to avoid file rewrites.
Bug fix	This will likely require a <code>pgd-proxy</code> restart on the next deploy (but it will avoid unnecessary future rewrites/restarts on subsequent deploys).
Bug fix	Fix an error while installing <code>rsync</code> 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 <code>bdr</code> extension if it has been created on purpose, even if it is unused.

2.21 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 `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. If you have an EDB subscription plan, and therefore have access to the EDB repositories, follow these instructions to install TPA packages.

To install TPA from source, see [Installing from source](#).

See [Distribution support](#) for information about the platforms that are supported.

Info

Make absolutely sure that your system has the correct date and time set. Various operations will fail otherwise. We recommend you use a network time, for example, `sudo ntpdate pool.ntp.org`.

Quick start

To obtain your token, log in to [EDB Repos 2.0](#). Then execute the following command, substituting your token for `<your-token>`. Replace `<your-plan>` with one of the following according to the EDB plan you're subscribed to: `enterprise`, `standard`, `community360`, or `postgres_distributed`.

Add repository and install TPA on Debian or Ubuntu

```
curl -1sLf 'https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.deb.sh' | sudo -E bash
sudo apt-get install tpaexec
```

Add repository and install TPA on RHEL, Rocky, AlmaLinux, or Oracle Linux

```
curl -1sLf 'https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.rpm.sh' | sudo -E bash
sudo yum 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 follow.

Where to install TPA

As long as you're using a supported platform, you can install and run TPA from your workstation. This approach is fine for learning, local testing, or demonstration purposes. If you want to perform a complete deployment on your own workstation, TPA supports [deploying to Docker containers](#).

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 have to update only one copy of TPA and you need to provide network access only from a single TPA host to the target instances.

Installing TPA packages

To install TPA, you must first subscribe to an EDB repository that provides it. The preferred source for repositories is EDB Repos 2.0.

To obtain your token, log in to [EDB Repos 2.0](#). Then execute the following command, substituting your token for `<your-token>`. Replace `<your-plan>` with one of the following according to the EDB plan you're subscribed to: `enterprise`, `standard`, `community360`, or `postgres_distributed`.

Add repository on Debian or Ubuntu

```
curl -1sLf 'https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.deb.sh' | sudo -E bash
```

Add repository on RHEL, Rocky, AlmaLinux or Oracle Linux

```
curl -1sLf 'https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.rpm.sh' | sudo -E bash
```

Alternatively, you can obtain TPA from the legacy 2ndQuadrant repository. To do so, log in to the EDB Customer Support Portal and subscribe to the [products/tpa/release repository](#) by adding a subscription under **Support/Software/Subscriptions**. Then follow the instructions to enable the repository on your system.

Once you have enabled one of these repositories, you can install TPA as follows.

Install on Debian or Ubuntu

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

Install on RHEL, Rocky, AlmaLinux, or Oracle Linux

```
sudo yum install
tpaexec
```

This command installs TPA into `/opt/EDB/TPA`. It also ensures that other required packages (such as Python 3.9 or later) are installed.

We mention `sudo` here only to indicate the commands that need root privileges. You can 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, use `sudo -H tpaexec setup` to avoid subsequent permission errors during `tpaexec configure`.

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

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

Add `/opt/EDB/TPA/bin` to the `PATH` of the user who normally runs `tpaexec` commands. For example, you can 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)

You can install TPA onto a server that can't access either the EDB repositories, a Python package index, or both. For information on how to use TPA in such an environment, see [Managing clusters in a disconnected or air-gapped environment](#).

Downloading TPA packages

If you can't 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.

If this approach isn't possible, contact EDB Support, which can provide you with a download link or instructions appropriate to your subscription.

Installing without access to a Python package index

When you run `tpaexec setup`, it ordinarily downloads 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, install the `tpaexec-deps` package in the same way you installed `tpaexec`. The `tpaexec-deps` package (available from the same repository as `tpaexec`) bundles everything that you would have downloaded, so that they can be installed without network access. Install the package before you run `tpaexec setup`, and the bundled copies are used automatically.

Verifying your TPA installation

After completing the installation, verify your local installation:

```
tpaexec
selftest
```

If this 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 [Installing without access to a Python package index](#)).
3. Run `tpaexec setup` again.

If you subscribed to the TPA package repository, running `apt-get update && apt-get upgrade` or `yum update` installs 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 might not strictly require this, but others can't work without it.

Ansible versions

TPA uses Ansible version 8 by default (ansible-core 2.15).

TPA has experimental support for Ansible 9 (ansible-core 2.16), which can be specified using the `--ansible-version` argument to `tpaexec setup`. It requires Python 3.10 or greater, so if you have `edb-python 3.9` installed, you must explicitly set your python version when running `tpaexec setup`:

```
PYTHON=/usr/bin/python3.10 tpaexec setup --ansible-version
9
```

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 apply to quick testbed setups and 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, EDB's proprietary products, or combinations.

Can I report an issue?

Yes. If you're an EDB customer then contact Support. Otherwise open a GitHub Issue.

Can I contribute?

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

5 Installing TPA from source

You can use TPA from a copy of the source code repository.

Note

To receive EDB support for the software, EDB customers must [install TPA from packages](#).

To run TPA from source, you must install all of the dependencies (for example, Python 3.9+) that the packages would handle for you. Or, download the source and [run TPA in a Docker container](#). (Either way works fine on Linux and macOS.)

Quickstart

First, you must install the various dependencies: Python 3, Python venv, git, openvpn, and patch. Installing from EDB repositories installs these for you along with the TPA packages.

Before you install TPA, you must install the required packages:

- Debian/Ubuntu

```
sudo apt-get install python3 python3-pip python3-venv git openvpn patch
```
- Redhat, Rocky or AlmaLinux (RHEL7)

```
sudo yum install python3 python3-pip epel-release git openvpn patch
```
- Redhat, Rocky or AlmaLinux (RHEL8)

```
sudo yum install python36 python3-pip epel-release git openvpn patch
```

Clone and setup

After the prerequisites are installed, you can clone the repository:

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

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

If you prefer to check out with SSH, use:

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

Add the bin directory to your path. You can find the bin directory in your newly created clone.

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

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

You can now create a working TPA environment by running:

```
tpaexec setup
```

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

```
tpaexec selftest
```

tpaexec is now installed.

Dependencies

Python 3.9+

TPA requires Python 3.9 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. (If you weren't already using pyenv, add `pyenv` to your PATH in `.bashrc`, and call `eval "$(pyenv init -)"` as described in the [pyenv documentation](#).)

```
# 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 3.9.0
Downloading Python-3.9.0.tar.xz...
-> https://www.python.org/ftp/python/3.9.0/Python-
3.9.0.tar.xz
Installing Python-3.9.0...
Installed Python-3.9.0 to
/home/ams/.pyenv/versions/3.9.0

$ pyenv local 3.9.0
$ pyenv version
3.9.0 (set by /home/ams/pyenv/.python-
version)

$ pyenv which python3
/home/ams/.pyenv/versions/3.9.0/bin/python3
$ python3 --version
3.9.0
```

Virtual environment options

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

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

However, we strongly suggest leaving the default venv location. If you use a different location, you must also set the environment variable `TPA_VENV` to that location. For example, add the following line to your `.bashrc` or other shell startup scripts:

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

6 A first cluster deployment

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

This example runs TPA on an Ubuntu system, but the considerations are similar for most Linux systems.

Installing TPA

If you're an EDB customer, first follow the [EDB repo instructions](#), which install the TPA packages straight from EDB's repositories.

If you're an open source user of TPA, first see the [instructions on how to build from the source](#), which explains how to download TPA from Github.com.

Installing Docker

This tutorial deploys the example deployment onto Docker. If you don't already have Docker installed, you need to set it up.

To install Docker on Debian or Ubuntu:

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

For other Linux distributions, see [Install Docker Engine](#).

Add your user to the docker group:

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

Cgroups version

Currently, TPA requires Cgroups Version 1 to be configured on your system.

Run:

```
mount | grep cgroup | head -1
```

If you don't see a reference to `tmpfs` in the output, you need to disable cgroups v2.

To make the appropriate changes, run:

```
echo 'GRUB_CMDLINE_LINUX=systemd.unified_cgroup_hierarchy=false' | sudo tee
/etc/default/grub.d/cgroup.cfg
```

Then update Grub and reboot your system:

```
sudo update-grub
sudo reboot
```

Warning

Giving a user the ability to speak to the Docker daemon lets them trivially gain root on the Docker host. Give only trusted users access to the Docker daemon.

Creating a configuration with TPA

The next step is to create a configuration. TPA does most of the work for you by way of 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 the `tpaexec configure` command:

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

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

The `--platform docker` tells TPA to create this configuration 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.

The `--postgresql 15` argument instructs TPA to use community Postgres, version 15. There are several options for selecting software, but this is the most straightforward default for open-source users.

Adding `--enable-repmgr` tells TPA to configure the deployment to use [Replication Manager](#) to handle replication and failover.

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

Run this command, which doesn't return anything at the command line when it's complete. However, a directory called `demo` is created that contains some files. These files include `config.yml`, which is a blueprint for the new deployment.

Provisioning the deployment

Now you're ready to create the containers (or virtual machines) on which to run the new deployment. Use the `provision` command to achieve this:

```
tpaexec provision demo
```

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

Deploying

Once the containers are provisioned, you can move on to deployment. Deploying 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. This command runs `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. To ping all the connectable hosts in the deployment, run:

```
tpaexec ping demo
```

The output looks something like:

```
$ tpaexec ping demo
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 that responded with `SUCCESS`. This tutorial uses `uptake`.

If you're only planning on using Docker, use the command `docker exec -it uptake /bin/bash`, substituting 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, the tutorial configuration is called `demo`. Go to that directory and run `ssh -F ssh_config ourhostname` to connect:

```
cd demo
ssh -F ssh_config uptake
Last login: Wed Sep  6 10:08:01 2023 from 172.17.0.1
[root@uptake ~]#
```

In both cases, you're logged in as a root user on the container.

You can now change user to the postgres user using `sudo -iu postgres`. As postgres, you can run `psql`. TPA has already configured that user with a `.pgpass` file, so you don't need to enter a password.

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

postgres=#
```

You're connected to the database.

You can connect from the host system without using SSH to get 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
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

```

You're going to connect to `uptake`, so the IP address is 172.17.0.11.

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

```

tpaexec show-password demo postgres
a9LmI1X^uMOpPoEnLuRdL%L$orQak3om

```

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

```

psql --host 172.17.0.11 -U postgres
Password for user postgres:

```

Enter the password you previously retrieved:

```

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're now connected from the Docker host to Postgres running in one of the TPA-deployed Docker containers.

7 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

```

[tpa]$ 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`. 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 `--num-cascaded-replicas 3` to create a cluster with three cascaded replicas. 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

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.

Note: 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.

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.

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

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`.

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.

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

2ndQuadrant and EDB repositories

TPA can enable any 2ndQuadrant or EDB software repository that you have access to through a subscription.

By default, it will install the 2ndQuadrant public repository (which does not need a subscription) and add on any product repositories that the architecture may require (e.g., the PGD repository).

More detailed explanation of how TPA uses 2ndQuadrant and EDB repositories is available [here](#)

Specify `--2q-repositories source/name/maturity ...` or `--edb-repositories repository ...` to specify the complete list of 2ndQuadrant or EDB repositories to install on each instance in addition to the 2ndQuadrant public repository.

If any EDB repositories are specified, any 2ndQuadrant ones will be ignored.

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

To use these options, you must `export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` or `export EDB_SUBSCRIPTION_TOKEN=xxx` before you run `tpaexec`. You can get a 2ndQuadrant token from the 2ndQuadrant Portal under "Company info" in the left menu, then "Company". 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 11 through 16.

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*'`

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. Use `2ndqpostgres` instead of `postgres` to build and install 2ndQPostgres. By default, this will build the appropriate `REL_nnn_STABLE` branch.

You may use `--install-from-source 2ndqpostgres pglogical3 bdr3` to build and install all three components from source, or just use `--install-from-source pglogical3 bdr3` to use packages for 2ndQPostgres, but build and install pglogical v3 and PGD v3 from source. By default, this will build the `master` branch of pglogical and 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.

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 supported backend by python keyring module including gnome-keyring and secret-tool.

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 generate an UUID by default but there is no naming scheme requirements.

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: 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>`.

Examples

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

```
[tpa]$ tpaexec configure ~/clusters/speedy --architecture M1 \
  --num-cascaded-replicas 2 --distribution Debian \
  --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
[tpa]$

```

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

```

$ ls ~/clusters/speedy
total
8
drwxr-xr-x 2 ams ams 4096 Aug  4 16:23
commands
-rw-r--r-- 1 ams ams 1374 Aug  4 16:23
config.yml
lrwxrwxrwx 1 ams ams   51 Aug  4 16:23 deploy.yml
->
                                     /home/ams/work/2ndq/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:

```

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

cluster_rules:
- cidr_ip: 0.0.0.0/0
  from_port: 22
  proto:
tcp
  to_port: 22
- cidr_ip: 10.33.76.176/28
  from_port: 0
  proto:
tcp
  to_port: 65535
- cidr_ip:
10.33.148.240/28
  from_port: 0
  proto:
tcp
  to_port: 65535
ec2_ami:
  Name: debian-10-amd64-20210721-710
  Owner: '136693071363'
ec2_instance_reachability:
public
ec2_vpc:
  us-east-1:
    Name: Test
    cidr: 10.33.0.0/16

cluster_vars:
  enable_pg_backup_api: false
  failover_manager:
repmgr
  postgres_flavour: postgresql
  postgres_version: '14'
  preferred_python_version: python3

```

```

use_volatile_subscriptions: false

locations:
- Name: main
  az: us-east-1a
  region: us-east-1
  subnet: 10.33.76.176/28
- Name: dr
  az: us-east-1b
  region: us-east-1
  subnet: 10.33.148.240/28

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: upsets
  backup: kayak
  location: main
  node: 1
  role:
  - primary
- Name: zebra
  location: main
  node: 2
  role:
  - replica
  upstream: upsets
- Name: kayak
  location: main
  node: 3
  role:
  -
barman
  - log-server
  - monitoring-server
  - witness

```

```

upstream:
upsets
volumes:
- device_name:
/dev/sdf
  encrypted: true
  vars:
    volume_for:
barman_data
  volume_size: 128
  volume_type:
gp2
- Name: queen
  location: dr
  node: 4
  role:
- replica
  upstream: zebra
- Name: knock
  location: dr
  node: 5
  role:
- replica
  upstream: zebra

```

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](#).

8 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 LXN or Docker to provision containers. Consult the platform documentation for details.

Quickstart

```
[tpa]$ tpaexec provision ~/clusters/speedy
```

```
PLAY [Provision cluster]
```

```
*****
```

```

...

TASK [Set up EC2 instances]
*****
changed: [localhost] => (item=us-east-1:quirk)
changed: [localhost] => (item=us-east-1:keeper)
changed: [localhost] => (item=us-east-1:zealot)
changed: [localhost] => (item=us-east-1:quaver)
changed: [localhost] => (item=us-east-1:quavery)
...

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

PLAY RECAP *****
localhost                : ok=128  changed=20  unreachable=0    failed=0

real    2m19.386s
user    0m51.819s
sys     0m27.852s

```

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 overridden 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.

```

[tpa]$ cd ~/clusters/speedy
[tpa]$ cat
ssh_config
Host *
    Port 22

```

```

IdentitiesOnly
yes
IdentityFile "id_speedy"
UserKnownHostsFile "known_hosts
tpa_known_hosts"
ServerAliveInterval 60

Host quirk
    User admin
    HostName
54.227.207.189
Host
keeper
    User admin
    HostName
34.229.111.196
Host
zealot
    User admin
    HostName
18.207.108.211
Host
quaver
    User admin
    HostName 54.236.36.251
Host quavery
    User admin
    HostName
34.200.214.150
[tpa]$ ssh -F ssh_config
quirk
Linux quirk 4.9.0-6-amd64 #1 SMP Debian 4.9.82-1+deb9u3 (2018-03-02)
x86_64

The programs included with the Debian GNU/Linux system are free
software;
the exact distribution terms for each program are described in
the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
extent
permitted by applicable law.
Last login: Sat Aug  4 12:31:28 2018 from
136.243.148.74
admin@quirk:~$ sudo -i
root@quirk:~#

```

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:

```

[tpa]$ ls ~/clusters/speedy
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 `sshconfig` 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`.

```

[tpa]$ cat inventory/00-
speedy
[tag_Cluster_speedy]
quirk ansible_host=54.227.207.189 node=1
platform=aws
keeper ansible_host=34.229.111.196 node=2
platform=aws
zealot ansible_host=18.207.108.211 node=3
platform=aws
quaver ansible_host=54.236.36.251 node=4
platform=aws
quavery ansible_host=34.200.214.150 node=5
platform=aws

[tpa]$ cat inventory/group_vars/tag_Cluster_speedy/01-speedy.yml
cluster_name:
speedy
cluster_tag: tag_Cluster_speedy
postgres_version: 15
tpa_version: v23.10-22-g30c1d5ea
tpa_2q_repositories: []
vpn_network: 192.168.33.0/24

[tpa]$ cat inventory/host_vars/zealot/02-
topology.yml
role:
-
barman
- log-server
- openvpn-
server
- monitoring-
server

```

```
- witness
upstream: quirk
```

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.

9 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 TPA_2Q_SUBSCRIPTION_TOKEN=xxx` to enable any 2ndQuadrant repositories that require subscription. You can use the subscription token that you used to [install TPA](#) itself. If you forget to do this, an error message will soon remind you.

Quickstart

```
[tpa]$ tpaexec deploy ~/clusters/speedy -
v
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:

```
[tpa]$ 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.

10 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

```
[tpa]$ 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.

11 PGD-Always-ON

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

This architecture is valid for use with EDB Postgres Distributed version 5 only and requires a subscription to [EDB Repos 2.0](#).

Cluster configuration

Overview of configuration options

This example shows an invocation of `tpaexec configure` for this architecture:

```
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` option.

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

Mandatory options

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

Additional options

Options	Description	Behavior if omitted
<code>--platform</code>	One of <code>aws</code> , <code>docker</code> , <code>bare</code> .	Defaults to <code>aws</code> .
<code>--location-names</code>	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.
<code>--witness-only-location</code>	A location name, must be a member of <code>location-names</code> .	No witness-only location is added.
<code>--data-nodes-per-location</code>	The number of data nodes in each location, must be at least 2.	Defaults to 3.
<code>--add-proxy-nodes-per-location</code>	The number of proxy nodes in each location.	PGD-proxy will be installed on each data node.
<code>--enable-camo</code>	Sets two data nodes in each location as CAMO partners.	CAMO will not be enabled.
<code>--bdr-database</code>	The name of the database to be used for replication.	Defaults to <code>bdrdb</code> .
<code>--enable-pgd-probes</code>	Enable http(s) API endpoints like <code>health/is-ready</code> to probe pgd-proxy's health.	Disabled by default.

More detail about PGD-Always-ON configuration

A PGD-Always-ON cluster comprises a number of locations, preferably odd. Each location has the same number of data nodes, also preferably odd. If you don't specify any `--location-names`, the default is to use a single location with three data nodes.

You can specify location names for the cluster with `--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. Examples include AWS regions, your own data centers, or any other designation to identify where your servers are hosted.

for AWS users

If you're using TPA to provision an AWS cluster, the locations will be mapped to separate availability zones in the `--region` you specify. You can specify multiple `--regions`, but TPA doesn't currently set up VPC peering to allow instances in different regions to communicate with each other. For a multi-region cluster, you must 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 adds an extra witness node to each location. The witness preserves the ability to reliably establish consensus without the hardware requirements and extra cost of another data node.

A cluster with only two locations would entirely lose the ability to establish global consensus if one of the locations were to fail. You can retain the ability to establish consensus despite a single-location failure. To do so, we recommend adding a third witness-only location that contains only a witness node. Use `--witness-only-location loc` to designate one of your locations as a witness.

By default, every data node in every location also runs 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 routes connections to a write leader. Local routing makes every PGD Proxy route to a write leader within its own location (suitable for geosharding applications). Global routing makes every proxy route to a single write leader, elected among all available data nodes across all locations.

You can optionally specify:

- `--bdr-database dbname` to set the name of the database with BDR enabled (default: `bdrdb`)

- `--enable-camo` to set two data nodes in each region as CAMO partners
- `--enable-pgd-probes [{http, https}]` to enable http(s) api endpoints that allow you to easily probe proxy's health

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

12 BDR-Always-ON

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

This architecture requires a subscription to the legacy 2ndQuadrant repositories, and some options require a subscription to EDB Repos 1.0. See [How TPA uses 2ndQuadrant and EDB repositories](#) for more detail.

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

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

See 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

This example shows an invocation of `tpaexec configure` for this architecture:

```
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
```

Mandatory options

Option	Description
<code>--architecture (-a)</code>	Must be set to <code>BDR-Always-ON</code> .
<code>--postgresql 14</code> (for example)	Any valid flavour and version specifier .
<code>--layout</code>	One of <code>bronze</code> , <code>silver</code> , <code>gold</code> , <code>platinum</code> .
<code>--harp-consensus-protocol</code>	One of <code>bdr</code> , <code>etcd</code> .

Additional options

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

More detail about BDR-Always-ON configuration

Specify `--layout layoutname` to choose a layout: `bronze`, `silver`, `gold`, or `platinum`. The bronze, gold, and platinum layouts have a PGD witness node to ensure an odd number of nodes for Raft consensus majority. Witness nodes don't 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 can optionally specify `--bdr-database dbname` to set the name of the database with PGD enabled (default: `bdrdb`).

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

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

HARP2 is enabled by default in the BDR-Always-ON architecture.

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

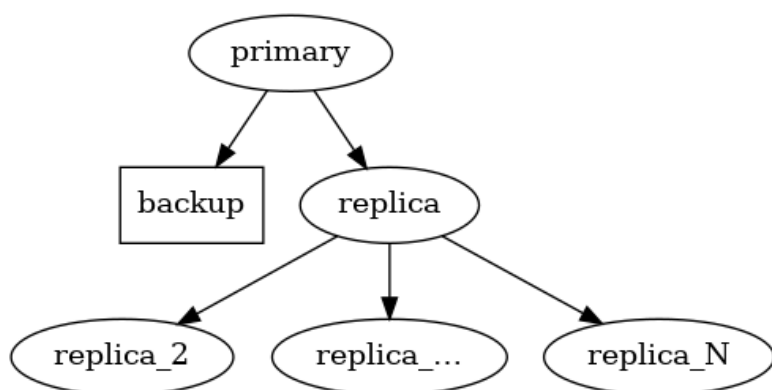
13 M1

The M1 architecture is a Postgres cluster with a primary and a streaming replica, one Barman server, and any number of additional replicas cascaded from the first one. This architecture is suitable for production. It's 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's sourced from EDB Repos 2.0. See [How TPA uses 2ndQuadrant and EDB repositories](#) for more detail.

Default layout

By default, the primary has one read-only replica attached in the same location. The replica, in turn, has one cascaded replica attached in a different location, where the Barman server is also configured to take backups from the primary.



If the number of Postgres nodes is even, the Barman node is also configured as a witness. Having an odd number of nodes in total helps to establish consensus in case of automatic failover.

Application and backup failover

The M1 architecture implements failover management in that it ensures that a replica is promoted to take the place of the primary if the primary become unavailable. However it doesn't provide any automatic facility to reroute application traffic to the primary. If you require automatic failover of application traffic, you must configure this at the application (for example using multi-host connections) or by using an appropriate proxy or load balancer and the facilities offered by your selected failover manager.

This is also true of the connection between the backup node and the primary created by TPA. The backup isn't automatically adjusted to target the new primary in the event of failover. Instead, it remains connected to the original primary. If you're performing a manual failover and want to connect the backup to the new primary, you can rerun `tpaexec deploy`. If you want to automatically change the backup source, implement this using your selected failover manager.

Cluster configuration

Overview of configuration options

This example shows an invocation of `tpaexec configure` for this architecture:

```
tpaexec configure ~/clusters/m1 \
  --architecture M1 \
  --platform aws --region eu-west-1 --instance-type t3.micro \
```

```
--distribution Debian \
--postgresql 14 \
--failover-manager repmgr
```

You can list all available options using the `help` command:

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

Mandatory options

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

Additional options

Parameter	Description	Behavior if omitted
<code>--platform</code>	One of <code>aws</code> , <code>docker</code> , <code>bare</code> .	Defaults to <code>aws</code> .
<code>--num-cascaded-replicas</code>	The number of cascaded replicas from the first replica.	Defaults to 1.
<code>--enable-haproxy</code>	2 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.
<code>--patroni-dcs</code>	Select the Distributed Configuration Store backend for patroni. Only option is <code>etcd</code> at this time. Only supported with Patroni as the failover manager.	Defaults to <code>etcd</code> .

More detail about M1 configuration

You can optionally specify `--num-cascaded-replicas N` to request N cascaded replicas (including 0 for none; default: 1).

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

14 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,

```
[tpa]$ cat >
~/aws/credentials
[default]
aws_access_key_id =
AKIAIOSFODNN7EXAMPLE
aws_secret_access_key =
wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
```

The AMI 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 different 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-overlapping 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 recommend 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-xxx
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.)

This platform supports Debian 9 (stretch), RedHat Enterprise Linux 7, Rocky 8, Ubuntu 16.04 (Xenial), and SUSE Linux Enterprise Server 15.

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
```

15 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:

```
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 `xyzyy@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 `~xyzyy/.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 xyzyy@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 (8/9/10; or jessie/stretch/buster), Ubuntu (16.04/18.04/20.04/22.04; or xenial/bionic/focal/jammy), and RHEL/CentOS/Rocky/AlmaLinux (7.x/8.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.

16 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:

```
[tpa]$ tpaexec configure clustername --platform docker
[...]
[tpa]$ tpaexec provision
clustername
[tpa]$ 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, the `centos/systemd` image (based on CentOS 7) can be used in this way.

TPA does not support Debian 8 (jessie) or Ubuntu 16.04 (xenial) for Docker containers, because of bugs in the old version of systemd shipped on those distributions.

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.

CgroupVersion

Support for CgroupVersion 2 is not fully baked yet for docker sdk in ansible and related tooling. So while we recommend using a recent version of docker; we rely on CgroupVersion 1 until version 2 is fully supported. Instructions below suggest the changes to switch to CgroupVersion 1 if your platform uses CgroupVersion 2 by default.

On Linux:

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

```
[tpa]$ docker version --format
'{{.Server.Version}}'
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 privileged 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

Caution: 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:

```
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 container management

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

```
[tpa]$ tpaexec start-containers
clustername
[tpa]$ 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:

```
[tpa]$ tpaexec list-containers
clustername
```

17 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 on the instances.

The `tpaexec configure` command generates 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 need to edit `config.yml`.

An overview of the configuration mechanisms available follows. For more details about the specific variables you can set to customize the deployment process, see [Instance configuration](#).

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 might 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 can affect several variables. (For example, `--bdr-version` might set `postgres_version`, `tpa_2q_repositories`, `edb_repositories`, `extra_postgres_extensions`, and so on.) But you can always accomplish with an editor the same things you can 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`.

Note

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 first, 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 replica role, you must also declare the name of its upstream instance, and that instance must exist. However, TPA doesn't prevent you from setting variables on the instances. 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 translates `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. Deployment then uses the inventory values.

Note

After you change `config.yml`, remember to run `tpaexec provision` before `tpaexec deploy`.

You can set any variable for the entire cluster, an individual host, or both. Host variables override group variables. In practice, setting `x: 42` in `cluster_vars` is the same as setting it in every host's `vars`. A host that needs `x` during deployment sees the value 42 either way. A host always sees the most specific value, so it's 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. because there's less repetition, it's easier to review and change. 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` can map to any valid YAML type and are translated directly into group variables in the Ansible inventory:

```
cluster_vars:
  postgres_version: 14
  tpa_2q_repositories:
  -
products/bdr3/release
  - products/pglogical3/release
  postgres_conf_settings:
    bdr.trace_replay: true
```

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

Instance variables

We use the term “instance variables” to refer to any variables that are defined for a specific instance in `config.yml`. This example shows 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 don't come from `vars` alone. Some other instance settings, including `platform`, `location`, `volumes`, and `role`, are also copied to the inventory as host `vars`. However, you can't define these settings under `vars` or `cluster_vars` instead.

The settings outside `vars` can describe the properties of the instance (such as `Name` and `node`) or its place in the topology of the cluster (such as `role` and `backup`). Or they can be platform-specific attributes (such as instance `type` and `volumes`). Other than knowing that they can't be defined under `vars`, it's rarely necessary to distinguish between these instance settings and instance variables.

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

instance_defaults

This setting is a mechanism to further reduce repetition in `config.yml`. It's most useful for instance settings that can't be defined as `cluster_vars`. For example, you can 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, that value takes precedence over the default.

It might help to think of `instance_defaults` as being a macro facility to use in defining `instances`. What's 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 can't be defined as a variable (for example, `platform` or `type`). This 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 isn't a 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. Just like `instance_defaults`, an instance can override the defaults that it inherits from its location. Similarly, you can use this feature to fill in `vars` for an instance. This approach 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.

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

18 Instance configuration

This is an overview of the TPA settings you can use to customize the deployment process on cluster instances.

There's also [an overview of configuring a cluster](#), which explains how to use cluster and instance variables together to write a concise, easy-to-review `config.yml`.

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 it [configures Postgres](#).

- [Distribution support](#)
- [Python environment](#) (`preferred_python_version`)
- [Environment variables](#) (for example, `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 2ndQuadrant and 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 repository configuration. For example, you can use it 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](#) (for example, `acl`, `openssl`, `sysstat`)

Postgres and other components (for example, Barman, `repmgr`, `pgbouncer`) are installed separately according to the cluster configuration. See [Other components](#).

Other system-level tasks

- [Create and mount file systems](#) (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's interleaved with the configuration of other components like `repmgr` and `pgbouncer`. The first step is to install Postgres.

Version selection

Use the [configure options](#) to select a Postgres flavor and version, or set `postgres_version` in `config.yml` to specify the Postgres major version you want to install.

That's all you need to do to set up a working cluster. Everything else described here is optional. You can control every aspect of the deployment if you want to, but the defaults are carefully selected 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](#) (for example, `pglogical`, `BDR`, and so on)
- [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 are installed (for example, to install additional configuration files).

Once the Postgres configuration is in place, TPA installs and configures other components, such as Barman, repmgr, pgbouncer, and haproxy, according to the details of the architecture.

Other components

- [Configure Barman](#)
- [Configure pgbouncer](#)
- [Configure haproxy](#)
- [Configure HARP](#)
- [Configure EFM](#)

Configuring and starting services

TPA installs systemd service unit files for each service. The service for Postgres is named `postgres.service`. You can use `systemctl start postgres` to start it and `systemctl stop postgres` to stop it.

If you're deploying a cluster for the first time, TPA starts the Postgres service at this point. On an existing cluster, if there are any relevant configuration changes, TPA reloads or restarts the Postgres service as appropriate. If there are no changes and Postgres is already running, it leaves the service alone. (If Postgres isn't running on an existing cluster, TPA starts it.)

In any case, Postgres is 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 is complete (for example, 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 is complete.

19 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 ability 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. However, package dependencies prevent installing pglogical from source and PGD from packages.

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

Quick start

Set 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
```

On Socker clusters, you can also build components from local work trees instead of a remote git repository:

```
$ 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.

BDR, and other components with custom locations and build parameters.

Configuration

There are two aspects to configuring source builds.

If you 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 in [Quick start](#).

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.

For the available options, see:

- [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 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 command results in a cluster configuration that cases `tpaexec deploy` to clone the known repositories for Postgres Extended, pglogical3, and bdr3, checks out the given branches, and builds 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 installs Postgres Extended from the repository, but 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 can't specify a branch to build (see `pglogical3:REL3_7_STABLE` versus `pglogical3` for `--install-from-source` in the previous examples). The source directory is mounted read-only in the containers, so TPA can't use `git pull` or `git checkout` to update it. You get whichever branch is checked out locally, including any uncommitted changes.

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're 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 is bind-mounted read-write into the containers, and its contents are shared between the host and the containers.

(By design, there's no way to install binaries compiled on the host directly into the containers.)

Rebuilding

After deploying a cluster with components built from source, run `tpaexec rebuild-sources` to quickly rebuild and redeploy just those components. This command is faster than running `tpaexec deploy` but doesn't apply any configuration changes.

20 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
---
- 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` immediately after bootstrapping Python—but before doing anything else like 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.

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.

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.

21 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.)

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.

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 or upgrading an existing PGD-Always-ON 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.

Configuration

In many cases, minor-version upgrades do not need changes to config.yml. Just run `tpaexec upgrade`, and it will upgrade to the latest available versions of the installed packages in a graceful way (what exactly that means depends on the details of the cluster).

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

Note: The upgrade procedure from BDR-Always-ON to PGD-Always-ON for camo enabled clusters using BDR version 3.7 is not yet supported. This support will come in a later release.

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\  
--enable-proxy-monitoring true
```

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:
 - o Fences the node off to ensure that harp-proxy doesn't send any connections to it.
 - o Stops, updates, and restarts postgres, including replacing BDR4 with PGD5.
 - o Unfences the node so it can receive connections again.
 - o 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:
 - o Sets up pgd-proxy.
 - o Stops harp-proxy.
 - o Starts pgd-proxy.
6. Removes harp-proxy and its support files.

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.
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:
 - o Fences the node off to ensure that pgd-proxy doesn't send any connections to it.
 - o Stops, updates, and restarts postgres.
 - o Unfences the node so it can receive connections again.
 - o Updates pgbouncer, pgd-proxy, and pgd-cli, as applicable for this node.

BDR-Always-ON

For BDR-Always-ON clusters, the upgrade process goes through the cluster instances one by one and does the following:

1. Tell haproxy the server is under maintenance.
2. If the instance was the active server, request pgbouncer to reconnect and wait for active sessions to be closed.
3. Stop Postgres, update packages, and restart Postgres.
4. Finally, mark the server as "ready" again to receive requests through haproxy.

PGD logical standby or physical replica instances are updated without any haproxy or pgbouncer interaction. Non-Postgres instances in the cluster are left alone.

You can control the order in which the cluster's instances are updated 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 update by listing the active PGD primary instances last, so that the shadow servers are updated 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.

M1

For M1 clusters, `upgrade` will first update the streaming replicas one by one, then perform a [switchover](#) from the primary to one of the replicas, update the primary, and switchover back to it again.

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.

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 satisfy the requested versions.

In this case, you must edit `config.yml`, remove the version settings, and re-run `tpaexec provision`. The update will then install the latest available packages. You can still update to a specific version by specifying versions on the command line as shown below:

```
$ tpaexec upgrade ~/clusters/speedy -vv \
-e postgres_package_version="2:11.6r2ndq1.6.13*" \
-e pglogical_package_version="2:3.6.11*" \
-e bdr_package_version="2:3.6.11"
```

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).

Note: 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.

22 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.

23 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 maintenance 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.

24 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.

25 TPA and Ansible Tower/Ansible Automation Platform

TPA has support for RedHat Ansible Automation Platform (AAP) an automation controller. You run only deploy and upgrade steps on AAP. You run configuration and provisioning on a standalone machine with the tpa package installed. You can then import the resulting cluster directory on AAP. Support is limited to bare-metal platforms.

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.

Note

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 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 Credentials Type 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 package installed.

On the TPA workstation

Configure

Run the `tpaexec configure` command, including these options: `--platform bare`, `--use-ansible-tower`, `--tower-git-repository`

```
[tpa]$ 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 change or addition needed. See [Cluster configuration](#).

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 to use the image provided by TPA.

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**.

We also recommend using **Overwrite local variables from remote inventory source** 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.

To connect to your inventory nodes by way of SSH during deployment, make sure the machine credential is available in AAP.

Template creation

To create a template:

1. Create a template that uses your project and your inventory.
2. Include these required credentials:
 - o Vault credential
 - o `EDB_SUBSCRIPTION_TOKEN` credential
 - o 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.

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>`. Be sure to push any change committed by the `relink` command:

```
$ git status
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.

26 TPA, Ansible, and sudo

TPA uses Ansible with sudo to execute tasks with elevated privileges on target instances. It's important to understand how Ansible uses sudo (which isn't specific to TPA) 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 (that is, Postgres, repmgr, efm, etcd, haproxy, pgbouncer, and so on)
- To perform a variety of other tasks (such as gathering cluster facts, performing switchover, and setting up cluster nodes)

TPA also must be able to use sudo. You can make it ssh in as root directly by setting `ansible_user: root`, but it still uses sudo to execute tasks as other users (for example, postgres).

Ansible sudo invocations

When Ansible runs a task using sudo, you 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-kfoodiiprzttsyerriqbjquqhhbemejgpc ; \
  /usr/bin/python2'''' && sleep 0'
```

Users who were expecting something like `sudo yum install -y xyzpkg` are often surprised by this. By and large, most tasks in Ansible invoke a Python interpreter to execute Python code rather than executing recognizable shell commands. (Playbooks can 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 isn't just a shell script written in YAML format. There's no way to "extract" shell commands that do the same thing as executing an arbitrary Ansible playbook.

One significant consequence of how Ansible uses sudo is that [privilege escalation must be general](#). It isn't possible to limit sudo invocations to specific commands in `sudoers.conf`, as some administrators are used to doing. Most tasks just invoke Python. You could have restricted sudo access to Python if it weren't for the random string in every command. However, once Python is running as root, there's no effective limit on what it can do anyway.

Executing Python modules on target hosts is how Ansible works. None of this is specific to TPA, and these considerations 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're first setting up your cluster or running `tpaexec deploy` again to make configuration changes, for example, during a maintenance window. Until then, you can disable its access entirely, which is 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 the parameters you provide. Each action is visible in the system logs on the target instances as well as in the Ansible log on the machine where tpaexec runs.

Ansible's focus is less to impose fine-grained restrictions on the actions you can execute and more to provide visibility into what it does as it executes. Thus elevated privileges are better assigned and managed by time rather than by scope.

SSH and sudo passwords

We strongly recommend setting up passwordless SSH key authentication and passwordless sudo access. However, it's 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 prompts you to enter a login password and a sudo password for the remote servers. It then negotiates the login/sudo password prompt on the remote server and sends the password you specify, which makes your playbooks take noticeably longer to run.

We don't recommend this mode of operation because it's 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 authorized 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's required, the passwords don't add any extra security during your maintenance window.

sudo options

To use Ansible with sudo, don't set `requiretty` in `sudoers.conf`.

If needed, you can change the sudo options that Ansible uses (`-H -S -n`) by setting either:

- `become_flags` in the `[privilege_escalation]` section of `ansible.cfg`
- `ANSIBLE_BECOME_FLAGS` in the environment
- `ansible_become_flags` in the inventory

All three methods are equivalent, but change the sudo options only if there's a specific need to do so. The defaults were chosen for good reasons. For example, removing `-S -n` will cause tasks to time out if passwordless sudo is incorrectly configured.

Logging

For playbook executions, the sudo logs show mostly invocations of Python, just as it shows only an invocation of bash when `sudo -i` is used.

For more detail, the syslog shows 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. (For example, 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've run `tpaexec setup`, TPA doesn't require elevated privileges on the local machine. (But if you use Docker, you must run tpaexec as a user that belongs to a Unix group that has permission to connect to the Docker daemon.)

27 TPA - PuTTY configuration guide

You can use PuTTY under Windows to connect by way of SSH to the AWS instances that were created by the TPA utility tpaexec provision. You need to convert the keys from the private key format generated by Amazon EC2 (`.pem`) to the PuTTY format (`.ppk`).

```
# Provision the cluster
[tpa]$ tpaexec provision <clustername>
```

PuTTY has a tool named PuTTYgen that can convert keys to the required format.

Key conversion

Follow these steps to perform the key conversion.

Locate private key

Locate the private key in the cluster directory `<clustername>`. The name uses the `cluster_name` variable, which is set in `config.yml`, prefixed by `id_`. For example, if `cluster_name` is set to `testenv1`, then the private key is `id_testenv1`.

Save key as .pem

Copy the private key file into your Windows file system and save it as a `.pem` file. Using the same example, the file name is `id_testenv1.pem`. To create the file, it's okay to paste the content into a text file.

Key conversions

1. Start PuTTYgen. Under **Parameters**, select the appropriate type of key to generate:

- For older versions of PuTTYgen, select **SSH-2 RSA**.
- For recent versions, select **RSA**.

Note

Don't select **SSH-1 (RSA)**.

2. Select **Load**. In the box labeled **PuTTY Private Key Files (*.ppk)**, select **All Files (*.*)**.

3. Select your `.pem` file and select **Open**. Then select **OK**.

4. Select **Save private key**, and select **Yes** to ignore the warning about saving the key without a passphrase. Make sure that the file suffix is `.ppk`, and select the same name as you did for the `.pem` file. Using the same example, the filename is `id_testenv1.ppk`.

Configure PuTTY

1. Start PuTTY. In the Category window, select **Session**.

2. In the **Host Name** panel, enter `<user>@<IP address>`. In the **Port** panel, enter `22`.

You can find the `<user>` and `<IP address>` values in the `<clustername>/ssh_config` file that's created by the `tpaexec` provision utility.

3. In the PuTTY Category window, select **Connection**. Expand **SSH** and select **Auth**.

4. For the panel labeled **Private key file for authentication**, select **Browse**. Select the `.ppk` file that you saved earlier, and then select **Open**.

5. In the PuTTY Category** window, select **Session**. In **Saved Sessions**, enter a session name and select ****Save****.

You can now connect to the AWS host by way of PuTTY by selecting this saved session.

28 Troubleshooting

Re-create Python virtual environment

Occasionally the Python venv can get in an inconsistent state. In this case, the easiest solution is to delete the environment and create it again. Symptoms of a broken venv can include errors during provisioning like:

```
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!"}
```

With tpaexec installed in the default location, to create a virtual environment, run:

```
[tpa]$ sudo rm -rf /opt/EDB/TPA/tpa-venv
[tpa]$ sudo /opt/EDB/TPA/bin/tpaexec setup
```

Strange AWS errors regarding credentials

If the time and date of the TPA server isn't correct, during provisioning, you can get AWS errors similar to this:

```
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.

```
[tpa]$ sudo ntpdate pool.ntp.org
```

Logging

By default, all tpaexec logging is saved in the log file `<clusterdir>/ansible.log`.

To change the log file location, set the environment variable `ANSIBLE_LOG_PATH` to the desired location. For example:

```
export ANSIBLE_LOG_PATH=~/.ansible.log
```

To increase the verbosity of logging, add `-v` / `-vv` / `-vvv` / `-vvvv` / `-vvvvv` to the tpaexec command line:

```
[tpa]$ tpaexec deploy <clustername> -v
```

```
-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 smoke test an existing cluster is to run:

```
[tpa]$ tpaexec test <clustername>
```

This command does 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, you can omit benchmarking by running:

```
[tpa]$ tpaexec test <clustername> --excluded_tasks pgbench
```

TPA server test

To check the installation of the TPA server, run:

```
[tpa]$ 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's [task selection mechanism](#).

29 Running TPA in a Docker container

If you're 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 3.9+ interpreter), your last resort might be to build a Docker image and run TPA inside a Docker container.

You don't need to run TPA in a Docker container to [deploy to Docker containers](#). It's always preferable to run TPA directly if you can, even on MacOS X.

Quick start

Make sure you have Docker installed and working on your system.

To clone the tpaexec source repository from Github and build a new Docker image named `tpa/tpaexec`, run the following commands:

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

```
$ docker build -t tpa/tpaexec
```

```
.
```

Double-check the created image:

```
$ docker image ls
tpa/tpaexec
REPOSITORY    TAG       IMAGE ID       CREATED
SIZE
tpa/tpaexec   latest   e145cf8276fb   8 seconds ago
1.73GB
$ docker run --platform=linux/amd64 --rm tpa/tpaexec tpaexec
info
# TPAexec v20.11-59-g85a62fe3 (branch:
master)
tpaexec=/usr/local/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.7.3, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.8.15)
```

Create a TPA container and make your cluster configuration directories available inside the container:

```
$ docker run --platform=linux/amd64 --rm -v ~/clusters:/clusters
\
-it tpa/tpaexec:latest
```

You can now run commands like `tpaexec provision /clusters/speedy` at the container prompt. (When you exit the shell, the container is removed.)

If you want to provision Docker containers using TPA, you must also allow the container to access the Docker control socket on the host:

```
$ docker run --platform=linux/amd64 --rm -v ~/clusters:/clusters \
-v /var/run/docker.sock:/var/run/docker.sock \
-it tpa/tpaexec:latest
```

Run `docker ps` in the container to make sure that your connection to the host Docker daemon is working.

Installing Docker

See the [Docker documentation](#) if you need help [installing Docker](#) and [getting started](#) with it.

On MacOS X, you can [install Docker Desktop for Mac](#) and launch Docker from the application menu.

29.1 Managing clusters in a disconnected or air-gapped environment

In a security-controlled environment where no direct connection to the Internet is allowed, you must provide all packages needed by TPA to complete the deployment on each node of the cluster. You can supply those packages using whatever method you choose, for example, by way of shared network repos or local repos preconfigured on each node.

You can alternatively use the local-repo approach that TPA provides, as described in [Creating and using a local repository](#). If you use this approach, you still need to make sure that TPA's local repo has all the required packages needed during the deployment of each node.

To help with this requirement when using the local-repo approach, TPA provides the `download-packages` command. This command can populate a local repository created using the local-repo approach. Use this command to ensure that you download all the required packages needed for the deployment to succeed.

Preparation

Choose an internet-connected machine where you can install TPA. Follow these instructions to either copy an existing cluster configuration or create a new cluster.

Note

If TPA isn't already installed on the air-gapped server, follow [these instructions](#) 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` file. Create a directory and copy that file into it. Then run `tpaexec relink` on that directory to generate the remaining files that are normally created by `tpaexec configure`.

Alternatively, to create a new configuration for an environment where the target instances doesn't have network access, configure a new cluster using this option:

```
tpaexec configure --use-local-repo-only ...
```

This command does everything that `--enable-local-repo` does and disables the configuration for all other package repositories. On RedHat instances, it also disables 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
```

You don't need separate cluster configurations for internet-connected and disconnected environments. The options that follow work in both.

See [Creating and using a local repository](#) for more information.

Downloading packages

On the internet-connected machine with [docker installed](#), run:

```
tpaexec download-packages cluster-dir --os <OS> --os-version <version>
```

See the detailed description for the [package downloader](#).

Copying packages to the target environment

The resulting repository is 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 then copies packages to the instances when you run `deploy`.

Deploying in a disconnected environment

Make sure that the cluster `config.yml` is configured as described in [Preparation](#). Run `tpaexec provision` and `deploy` as you do normally.

Updating in a disconnected environment

You can use the `upgrade` command to perform updates in an air-gapped environment. Before running this command, you must run `download-packages` on the connected controller and copy the updated repository to the disconnected controller.

29.2 Distribution support

TPA detects and adapts to the distribution running on each target instance. Listed here are the platforms that are actively supported and legacy distributions that were previously supported.

Deploying to a legacy platform is likely to work as long as you have access to the necessary packages. However, this isn't a supported use of TPA and isn't 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 x86

- Debian 11/bullseye is fully supported.
- Debian 10/buster is fully supported.
- Debian 9/stretch is a legacy distribution.
- Debian 8/jessie is a legacy distribution.

Ubuntu x86

- Ubuntu 22.04/jammy is fully supported.
- Ubuntu 20.04/focal is fully supported.
- Ubuntu 18.04/bionic is a legacy distribution.
- Ubuntu 16.04/xenial is a legacy distribution.

Oracle Linux

- Oracle Linux 9.x is fully supported (Docker only).
- Oracle Linux 8.x is fully supported (Docker only).
- Oracle Linux 7.x is fully supported (Docker only).

RedHat x86

- RHEL/Rocky/AlmaLinux/Oracle Linux 9.x is fully supported (Python 3 only).
- RHEL/CentOS/Rocky/AlmaLinux 8.x is fully supported (Python 3 only).
- RHEL/CentOS 7.x is fully supported (Python 2 only).

SLES

- SLES 15.x is fully supported.

Platform-specific considerations

Some platforms might not work with the legacy distributions mentioned here. For example, Debian 8 and Ubuntu 16.04 aren't available in [Docker containers](#).

29.3 TPA capabilities and supported software

- [Python requirements](#)
- [Supported distributions](#)

Supported software

TPA can install and configure the following major components.

- Postgres 16, 15, 14, 13, 12, 11
- EPAS (EDB Postgres Advanced Server) 16, 15, 14, 13, 12
- 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)

29.4 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 [bdr_pg_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 \  
--edbpgc 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.yml`.

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`.

```
---
architecture: M1
```

```

cluster_name: singlenode

cluster_vars:
  postgres_flavour: postgresql
  postgres_version: '15'
  preferred_python_version: python3
  tpa_2q_repositories: []

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.

```

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.

29.5 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, and 5.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 for an authoritative description of the details, see the [PGD documentation](#).

How TPA approaches PGD installation

TPA installs PGD and any dependencies on all PGD instances. It also installs Postgres.

After completing the basic Postgres setup and starting Postgres, TPA then creates `bdr_database` and proceeds to set up a PGD cluster using the steps that follow.

Installation

TPA installs the correct PGD packages based on the version and flavor of Postgres in use (for example, Postgres, Postgres Extended, or EDB Postgres Advanced Server).

Set `bdr_version` to determine the major version of PGD to install (that is, 3, 4, or 5). Set `bdr_package_version` to determine the package to install. For example, use `5.0*` to install the latest 5.0.x package.

Overview of cluster setup

After installing the required packages, configuring Postgres to load PGD, and starting the server, TPA sets 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()`) based on `bdr_node_groups`.
- Create replication sets, if required, to control the changes that are replicated. This decision is based on node group type and memberships. For example, subscriber-only and witness nodes might 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 isn't significant to the PGD configuration.

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 doesn't publish them.

A PGD instance with `witness` in its role is a witness node.

Each of these PGD instances is implicitly also a `primary` instance. The exception is an instance with `replica` in its role. That setting indicates a physical streaming replica of an upstream PGD instance. Such instances aren't included in any recommended PGD architecture and aren't currently

supported by TPA.

Configuration settings

The settings that follow are ordinarily set in `cluster_vars` so that they're set uniformly for all the PGD instances in the cluster. You can set different values on different instances in some cases, for example, `bdr_database`. In other cases, though, the result is undefined. For example, 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`) is initialized with PGD.

`bdr_node_group`

The setting of `bdr_node_group` (default: based on the cluster name) identifies the PGD cluster for an instance to be a part of. It's also used to identify a particular cluster for external components, such as `pgd-proxy` or `harp-proxy`.

`bdr_node_groups`

This setting 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` generates 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 can optionally specify the `node_group_type` optional.

Each entry can 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 joins that group instead of

`bdr_node_group`.

`bdr_commit_scopes`

This setting is an optional list of `commit scopes` that must exist in the PGD database (available for PGD 4.1 and later).

```
cluster_vars:
  bdr_commit_scopes:
    - name: somescope
      origin: somegroup
      rule: 'ALL (somegroup) ON received ...'
    - name: otherscope
      origin: othergroup
      rule:
'...'
...
```

Each entry must specify a `name` value for the commit scope, the name of the `origin` group, and a commit scope `rule` value. The groups must correspond to entries in `bdr_node_groups`.

If you set `bdr_commit_scopes` explicitly, TPA creates, alters, or drops commit scopes as needed to ensure that the database matches the configuration. If you don't set it, TPA leaves existing commit scopes alone.

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 risks data loss.

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 before v3.7.

29.6 Barman

When an instance has the `barman` role in `config.yml`, TPA configures the instance 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`. Any `postgres` instance can have a list of Barman servers named as its `backup`. In this case, backups are taken to all of the named servers.

The default Barman configuration connects to PostgreSQL using `pg_receivewal` to take continuous backups of WAL. It takes 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 previous 4 weeks.

Barman configuration

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 `/var/lib/barman/one`. The file and directory names are taken from the backed-up instance's `backup_name` setting. The default for this setting is the instance name.

You can set the following variables on the backed-up instance. They are passed through into Barman's configuration with the prefix `barman_` removed.

Variable	Default
<code>barman_archiver</code>	<code>false</code>
<code>barman_log_file</code>	<code>/var/log/barman.log</code>
<code>barman_backup_method</code>	<code>rsync</code>
<code>barman_compression</code>	<code>pigz</code>
<code>barman_reuse_backup</code>	<code>link</code>
<code>barman_parallel_jobs</code>	<code>1</code>
<code>barman_backup_options</code>	<code>concurrent_backup</code>
<code>barman_immediate_checkpoint</code>	<code>false</code>
<code>barman_network_compression</code>	<code>false</code>
<code>barman_basebackup_retry_times</code>	<code>3</code>
<code>barman_basebackup_retry_sleep</code>	<code>30</code>
<code>barman_minimum_redundancy</code>	<code>3</code>
<code>barman_retention_policy</code>	<code>RECOVERY WINDOW OF 4 WEEKS</code>
<code>barman_last_backup_maximum_age</code>	<code>1 WEEK</code>
<code>barman_pre_archive_retry_script</code>	
<code>barman_post_backup_retry_script</code>	

Variable	Default
barman_post_backup_script	
barman_streaming_wals_directory	

Backup scheduling

TPA installs a cron job in `/etc/cron.d/barman` that runs every minute and invokes `barman cron` to perform maintenance tasks.

For each instance being backed up, TPA installs another cron job in `/etc/cron.d/<backup_name>` that takes the backups of that instance. This job runs as determined by the `barman_backup_interval` variable for the instance. The default is to take backups at 04:00 every Wednesday and Saturday.

SSH keys

TPA generates ssh key pairs for the postgres and barman users and installs them into the respective `~/.ssh` directories. Keys for the postgres user are added to the barman `authorized_keys` file, and keys for the barman user are added to the postgres `authorized_keys` file. The postgres user must be able to ssh to the Barman server to archive WAL segments (if configured), and the barman user must be able to ssh to the Postgres instance to take or restore backups.

29.7 Configuring EFM

TPA installs and configures EFM when `failover_manager` is set to `efm`.

You need a valid subscription to EDB's package repositories to obtain EFM packages.

EFM configuration

TPA generates `efm.nodes` and `efm.properties` with the appropriate instance-specific settings or default values. TPA also installs an `efm.notification.sh` script, which does nothing by default. You can fill it in however you want.

See the [EFM documentation](#) for more details on configuring EFM.

efm_conf_settings

You can use `efm_conf_settings` to set any parameters, whether or not TPA recognizes them. Where needed, you need to quote the value exactly as you would if you were editing `efm.properties` manually:

```
cluster_vars:
  efm_conf_settings:
    standby.restart.delay: 1
    application.name:
quarry
```

```
reconfigure.num.sync: true
reconfigure.num.sync.max: 1
reconfigure.sync.primary: true
```

If you change `efm_conf_settings`, TPA always restarts EFM to activate the changes.

EFM witness

TPA installs and configures EFM as a witness on instances whose `role` contains `efm-witness`.

Repmgr

EFM doesn't provide a way to create new replicas. TPA uses repmgr to create replicas for Postgres versions 11 and earlier. Although repmgr packages are installed in this case, the repmgrd daemon remains disabled when EFM is in use.

For Postgres versions 12 and later, any cluster that uses EFM uses `pg_basebackup` to create standby nodes and doesn't use repmgr in any form.

29.8 Configuring haproxy

TPA installs and configures 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 installs the latest available version of haproxy by default. Set `haproxy_package_version: 1.9.15*` or any valid version specifier to install a different version.

Note

See limitations of using wildcards in `package_version` in `tpaexec-configure`.

Haproxy packages are selected according to the type of architecture. You can use an EDB-managed haproxy package, but it 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
<code>haproxy_bind_address</code>	127.0.0.1	The address for haproxy to bind to.
<code>haproxy_port</code>	5432 (5444 for EPAS)	The TCP port for haproxy to listen on.
<code>haproxy_read_only_port</code>	5433 (5445 for EPAS)	TCP port for read-only load balancer.
<code>haproxy_backend_servers</code>	None	A list of Postgres instance names.
<code>haproxy_maxconn</code>	<code>max_connections</code> × 0.9	The maximum number of connections allowed per backend server. The default is derived from the backend's <code>max_connections</code> setting.

Variable	Default value	Description
<code>haproxy_peer_enabled</code>	True*	Add known haproxy hosts as <code>peer</code> list. * <code>False</code> if <code>failover_manager</code> 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. Currently, this is supported only with the Patroni failover manager. The backend health check determines which Postgres instances are currently acting as replicas and uses round-robin load balancing to distribute traffic to them.

The read-only load balancer is disabled by default. You can turn it on by setting `haproxy_read_only_load_balancer_enabled: true`.

Server options

TPA generates `/etc/haproxy/haproxy.cfg` with a backend that has a `default-server` line and one line per backend server. All but the first one are marked as "backup" servers.

To add options to the `default-server` line, set `haproxy_default_server_extra_options` to a list of options on the haproxy instance.

To add options (which override the defaults) to the individual server lines for each backend, set `haproxy_server_options` to a list of options on the backend server.

Example

```
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
```

29.9 Configuring HARP

TPA installs and configures HARP when `failover_manager` is set to `harp`. This value is the default for BDR-Always-ON clusters.

Installing HARP

You must provide the `harp-manager` and `harp-proxy` packages. Contact EDB to obtain access to these packages.

Configuring HARP

See the [HARP documentation](#) for more details on HARP configuration.

Variable	Default value	Description
<code>cluster_name</code>	<code>''</code>	The name of the cluster.
<code>harp_consensus_protocol</code>		The consensus layer to use (<code>etcd</code> or <code>bdr</code>).
<code>harp_location</code>	<code>location</code>	The location of this instance (defaults to the <code>location</code> parameter).
<code>harp_ready_status_duration</code>	<code>10</code>	Amount of time in seconds the node's readiness status persists if not refreshed.
<code>harp_leader_lease_duration</code>	<code>6</code>	Amount of time in seconds the Lead Master lease persists if not refreshed.
<code>harp_lease_refresh_interval</code>	<code>2000</code>	Amount of time in milliseconds between refreshes of the Lead Master lease.
<code>harp_dcs_reconnect_interval</code>	<code>1000</code>	The interval, measured in ms, between attempts that a disconnected node tries to reconnect to the DCS.
<code>harp_dcs_priority</code>	<code>500</code>	In the case in which two nodes have an equal amount of lag and other qualified criteria to take the Lead Master lease, acts as an additional ranking value to prioritize one node over another.
<code>harp_stop_database_when_fenced</code>	<code>false</code>	Rather than removing a node from all possible routing, stop the database on a node when it's fenced.
<code>harp_fenced_node_on_dcs_failure</code>	<code>false</code>	If HARP is unable to reach the DCS, then fence the node.
<code>harp_maximum_lag</code>	<code>1048576</code>	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.
<code>harp_maximum_camo_lag</code>	<code>1048576</code>	Highest allowable variance (in bytes) between last received LSN and applied LSN between this node and its CAMO partners.
<code>harp_camo_enforcement</code>	<code>lag_only</code>	Whether to strictly enforce CAMO queue state.
<code>harp_use_unix_socket</code>	<code>false</code>	Use Unix domain socket for manager database access.
<code>harp_request_timeout</code>	<code>250</code>	Time in milliseconds to allow a query to the DCS to succeed.

Variable	Default value	Description
<code>harp_watch_poll_interval</code>	500	Milliseconds to sleep between polling DCS. Applies only when <code>harp_consensus_protocol</code> is <code>bdr</code> .
<code>harp_proxy_timeout</code>	1	Builtin proxy connection timeout, in seconds, to Lead Master.
<code>harp_proxy_keepalive</code>	5	Amount of time builtin proxy waits on an idle connection to the Lead Master before sending a keepalive ping.
<code>harp_proxy_max_client_conn</code>	75	Maximum number of client connections accepted by harp-proxy (<code>max_client_conn</code>).
<code>harp_ssl_password_command</code>	None	A custom command to receive the obfuscated sslpassword in the stdin and provide the handled sslpassword via stdout.
<code>harp_db_request_timeout</code>	10s	Similar to <code>dcg -> request_timeout</code> but for connection to the database.

You can use the [harp-config hook](#) to execute tasks after the HARP configuration files are installed, for example, 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 sets `harp_consensus_protocol` to `etcd` in `config.yml`. It gives the `etcd` role to a suitable subset of the instances, depending on your chosen layout.

HARP v2 requires etcd v3.5.0 or later, 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
<code>etcd_peer_port</code>	2380	The port used by etcd for peer communication
<code>etcd_client_port</code>	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 sets `harp_consensus_protocol` to `bdr` in `config.yml`. In this case, the existing PGD instances are 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 read-only user, you can specify that by setting `harp_dcs_user: username` under `cluster_vars`. TPA uses the `harp_dcs_user` setting to create a read-only 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`. TPA uses 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` is used by HARP to de-obfuscate the `sslpassword` given in the connection string. If `sslpassword` isn't present, then `harp_ssl_password_command` is ignored. If `sslpassword` isn't obfuscated, then `harp_ssl_password_command` isn't required and should not be specified.

Configuring the HARP service

You can configure the following parameters for the HARP service.

Variable	Default value	Description
<code>harp_manager_restart_on_failure</code>	<code>false</code>	If <code>true</code> , the <code>harp-manager</code> service is overridden so it's restarted on failure. The default is <code>false</code> to comply with the service installed by the <code>harp-manager</code> package.

Configuring HARP http(s) health probes

You can enable and configure the http(s) service for HARP that provides API endpoints to monitor the service's health.

Variable	Default value	Description
<code>harp_http_options</code>	<code>enable: false`secure: false`host: <inventory_hostname>`port: 8080`probes: timeout: 10s`endpoint: "host=<proxy_name> port=<6432> dbname=<bdrdb> user=<username>"</code>	Configure the http section of HARP <code>config.yml</code> 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 isn't "well-known." You need to add this certificate to the trust store of each machine that probes 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 the [HARP documentation](#) for more information about the available API endpoints.

29.10 Configuring Postgres Enterprise Manager (PEM)

TPA installs and configures PEM when you run the `tpaexec configure` command with the `--enable-pem` command line option.

The default behavior with `--enable-pem` is to enable the pem-agent role for all `postgres` instances in the cluster. The pem-agent role is also added to Barman nodes when the `--enable-pg-backup-api` command line option is used along with `--enable-pem`.

A dedicated instance named `pemserver` is also added to the cluster.

Since PEM server uses a Postgres backend, a `pemserver` instance implicitly uses the postgres role as well, which ensures that `pemserver` gets a valid Postgres cluster configured for use as a PEM backend. All configuration options available for a normal Postgres instance are valid for PEM's backend Postgres instance as well. For details, see:

- [Configure pg_hba.conf](#)
- [Configure postgresql.conf](#)

PEM is available only by way of EDB's package repositories and therefore requires a valid subscription.

Supported architectures

PEM is supported with all architectures by way of the `--enable-pem` configuration command-line option. An exception is the BDR-Always-ON architecture when used with EDB Postgres Extended. You can optionally edit the generated cluster config file (`config.yml`) and assign or remove the pem-agent role from any Postgres instance in the cluster to enable or disable PEM there.

PEM configuration

TPA configures PEM agents and PEM server with the appropriate instance-specific settings, with the remaining settings set to their default values. Some of the configuration options might be exposed for user configuration in the future.

PEM server's web interface is configured to run on https and uses port 443 for that configuration. PEM's web server configuration uses self-signed certificates.

The default login credentials for the PEM server web interface use the Postgres backend database user, which is set to postgres for PostgreSQL and enterisedb for EDB Postgres Advanced Server clusters by default. You can get the login password for the web interface by running `tpaexec show-password $clusterdir $user`.

Shared PEM server

Some deployments might want to use a single PEM server for monitoring and managing multiple clusters in the organization. Shared PEM server deployment in tpaexec is supported by way of the `pem_shared` variable. You can set this variable by way of `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 PEM server instance as shared, we tell the given cluster config that the PEM server instance is in fact managed by a separate cluster config that provisioned and deployed the PEM server in the first place. So any changes you want to make to the PEM server instance, including the Postgres backend for PEM, are managed by the cluster where the PEM server instance isn't declared as a shared PEM instance.

A typical workflow for using a shared PEM server across multiple clusters looks something like this:

1. Create a tpaexec cluster with a single instance that has the pem-server role. (Call it `pem-cluster` for this example.) You can 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 a PEM server. We use a single-node cluster because it's 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 PEM server from `$clusters/pem-cluster` as a shared PEM server instance. Use `bare` as the platform so you aren't trying to create a new PEM server instance. Also, specify the IP address of the PEM server that this cluster can use to access the PEM server instance.

```
- Name: pemserv
  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 the PEM server instance by way of SSH. You can allow this access by copying the `pg-cluster` public key to the PEM server instance by way of `ssh-copy-id`. Then do an SSH to make sure you can log in 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
pemserv
```

4. Update the PostgreSQL config file on the PEM server node so it allows connections from the new `pg-cluster`. You can modify the existing `pg_hba.conf` file on the PEM server by adding new entries to `pem_postgres_extra_hba_settings` under `vars:` in the `pem-cluster config.yml` file. For example:

```
instances:
- Name: pemserv
  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

```

Then run `tpaexec provision $clusters/pem-cluster` followed by `tpaexec deploy $clusters/pem-cluster`. When complete, nodes from your new `pg-cluster` can speak with the PEM server backend.

- To ensure that PEM agents from the nodes in `pg-cluster` can connect and register with the PEM server backend, you must run `export EDB_PEM_CREDENTIALS_FILE=/path/to/pem/credentials/file` before you run `tpaexec deploy`. The credentials file is a text file that contains your access credentials to the PEM server's backend Postgres instance in the `username:password` format.

```

$ cat
pem_creds
postgres:f1I%fw!QmWevdzw#EL#$Ulu1cWhg7&RT

```

If you don't know the backend password, you can get that by using the `tpaexec show-password` command.

```
tpaexec show-password $pem-clusterdir $user
```

- Run `tpaexec deploy $clusters/pg-cluster` so PEM is deployed on the new `pg-cluster` while using the shared PEM server instance.

Connecting to the PEM UI

The PEM UI runs on an https interface, so you can connect with a running instance of a PEM server by way of `https://$pem-server-ip/pem`. Login credentials for the PEM UI are set to the Postgres backend user that uses `postgres` for `postgresql` or `enterprisedb` for `epas` flavors. The `tpaexec show-password` command shows the password for the backend user. For example:

```
tpaexec show-password $clusterdir $user
```

See the [PEM documentation](#) for more details on using and configuring PEM.

29.11 Configuring pgbouncer

TPA will install and configure pgbouncer on instances whose `role` contains `pgbouncer`.

By default, pgbouncer listens for connections on port 6432 and forwards connections to `127.0.0.1:5432` (which may be either Postgres or `haproxy`, depending on the architecture).

You can set the following variables on any `pgbouncer` instance.

Variable	Default value	Description
<code>pgbouncer_port</code>	6432	The TCP port pgbouncer should listen on
<code>pgbouncer_backend</code>	127.0.0.1	A Postgres server to connect to

Variable	Default value	Description
<code>pgbouncer_backend_port</code>	5432	The port that the <code>pgbouncer_backend</code> listens on
<code>pgbouncer_max_client_conn</code>	<code>max_connections</code> × 0.7	The maximum number of connections allowed; the default is derived from the backend's <code>max_connections</code> setting if possible
<code>pgbouncer_auth_user</code>	<code>pgbouncer_auth_user</code>	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.

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
```

29.12 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
<code>pgd_proxy_options</code>	<code>bdr.alter_proxy_option()</code>	pgd-proxy configuration, e.g. port
<code>bdr_node_groups</code>	<code>bdr.alter_node_group_option()</code>	configuration for the proxy's node group, e.g. <code>enable_proxy_routing</code>
<code>bdr_node_options</code>	<code>bdr.alter_node_option()</code>	routing configuration for individual PGD nodes

See the PGD documentation for more details.

`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

instances:
- Name: someproxy
  vars:
    pgd_proxy_options:
      listen_port: 9000

```

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.

29.13 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.otherable'
    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.

```
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.

```

- 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.

29.14 Configuring 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 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 on PGD instances

On PGD instances, `repmgr_failover` will be set to `manual` by default.

29.15 How TPA uses 2ndQuadrant and EDB repositories

TPA can download EDB software (including 2ndQuadrant) from several package sources, depending on the selected software.

Only the special configuration options and logic for EDB and 2ndQuadrant sources are described here. You can add arbitrary `yum` or `apt` repositories independently of this logic. Likewise, you can [download packages in advance](#) and add them to a [local repository](#) if you prefer.

Package sources used by TPA

TPA downloads software from three package sources. Each of these sources provides multiple repositories. In some cases, the same software is available from more than one source.

- [EDB Repos 2.0](#)
- [EDB Repos 1.0](#)
- [2ndQuadrant Repos](#)

By default, TPA [selects sources and repositories](#) based on the architecture and other options you specify. It's generally not necessary to change these. However, before running `tpaexec deploy`, you must ensure that you have a valid subscription for all the sources used and that you [exported the token](#). Otherwise, the operation fails.

Note

EDB is in the process of publishing all software through Repos 2.0, and will eventually remove the older repositories.

Authenticating with package sources

To use [EDB Repos 2.0](#), before you run `tpaexec`, you must run `export EDB_SUBSCRIPTION_TOKEN=xxx`. You can get your subscription token from [the web interface](#).

To use [2ndQuadrant repositories](#), before you run `tpaexec`, you must run `export TPA_2Q_SUBSCRIPTION_TOKEN=xxx`. You can get your subscription token from the 2ndQuadrant Portal. In the left menu, under **Company info**, select **Company**. Some repositories are available only by prior arrangement.

To use [EDB Repos 1.0](#), you must create a text file that contains your access credentials in the `username:password` format. Before you run `tpaexec`, run:

```
export EDB_REPO_CREDENTIALS_FILE=/path/to/credentials/file
```

If you don't have an account for any of the sites listed, you can register for access at the [Account Registration page](#).

How sources are selected by default

If you select the PGD-Always-ON architecture, repositories are selected from EDB Repos 2.0, and all software is sourced from these repositories.

If you select the M1 architecture and don't select any proprietary EDB software, all packages are sourced from PGDG. If you select any proprietary EDB software, all packages are sourced from EDB Repos 2.0.

For the BDR-Always-ON architecture, the default source is EDB Repos 2.0 for new clusters, and the necessary repositories are added from this source. (Existing clusters use 2ndQuadrant repositories until you reconfigure them.) In addition, the PGDG repositories are used for community packages, such as PostgreSQL and `etcd`, as required. If EDB software that isn't available in the 2ndQuadrant repos is required (such as EDB Postgres Advanced Server), TPA selects the repositories from EDB Repos 1.0.

Specifying EDB 2.0 repositories

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 configures the `enterprise` and `postgres_distributed` repositories, giving access to EDB Postgres Advanced Server and PGD version 5 products. On Debian or Ubuntu systems, it uses the apt repository. RedHat systems use the rpm repositories through the yum front end. SLES systems use the rpm repositories through the zypper front end.

If you specify any EDB repositories, any 2ndQuadrant repositories specified are ignored and no EDB Repos 1.0 are installed.

Specifying 2ndQuadrant repositories

To specify the complete list of 2ndQuadrant repositories to install on each instance in addition to the 2ndQuadrant public repository, set `tpa_2q_repositories` to a list of 2ndQuadrant repository names:

```
cluster_vars:
  tpa_2q_repositories:
    - products/pglogical3/release
    -
  products/bdr3/release
```

This example installs the `pglogical3` and `bdr3` release repositories. On Debian and Ubuntu systems, it uses the apt repository, and on RedHat systems, it uses the yum repository.

The `dl/default/release` repository is always installed by default, unless you either:

- Explicitly set `tpa_2q_repositories: []`
- Have at least one entry in `edb_repositories`

Either of these action results in no 2ndQuadrant repositories being installed.

29.16 Configuring EDB Repos 2.0 repositories

You can configure EDB Repos 2.0 package repositories using cluster variables.

For more details on the EDB and 2ndQuadrant package sources used by TPA, see [How TPA uses 2ndQuadrant and EDB repositories](#).

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 installs the enterprise subscription repository as well as `postgres_distributed`, giving access to EDB Postgres Advanced Server and PGD version 5 products. On Debian or Ubuntu systems, it uses the apt repository. On RedHat or SLES systems, it uses the rpm repositories, through the yum or zypper front ends, respectively.

If you specify any EDB repositories, any 2ndQuadrant repositories specified are ignored and no EDB Repos 1.0 are installed.

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](#).

29.17 Configuring 2ndQuadrant repositories

This page explains how to configure 2ndQuadrant package repositories on any system.

For more details on the EDB and 2ndQuadrant package sources used by TPA see [this page](#).

To specify the complete list of 2ndQuadrant repositories to install on each instance in addition to the 2ndQuadrant public repository, set `tpa_2q_repositories` to a list of 2ndQuadrant repository names:

```
cluster_vars:
  tpa_2q_repositories:
    - products/pglogical3/release
    - products/bdr3/release
```

This example will install the `pglogical3` and `bdr3` release repositories. On Debian and Ubuntu systems, it will use the APT repository, and on RedHat systems, it will use the YUM repository. The 2ndQuadrant repositories are not available for SLES systems.

To use [2ndQuadrant repositories](#), you must `export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` before you run `tpaexec`. You can get your subscription token from the 2ndQuadrant Portal, under "Company info" in the left menu, then "Company". Some repositories are available only by prior arrangement.

The `dl/default/release` repository is always installed by default, unless you

- explicitly set `tpa_2q_repositories: []`, or
- have at least one entry in `edb_repositories`.

Either or the above will result in no 2ndQuadrant repositories being installed.

29.18 Configuring APT repositories

You can 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:
XXXXXXXXXX
      key_url: https://repo.example.com/path/to/XXXXXXXXXX.asc
      repo: >-
          deb https://repo.example.com/repos/Example/ xxx-Example
main

  apt_repository_list:
    - PGDG
    - Example
```

This configuration installs:

- The GPG key (with id `key_id`, obtained from `key_url`)
- 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)
- 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.

29.19 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.XXXXXXXXXXXXXXXXXX.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 [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`.

29.20 Creating and using a local repository

If you create a local repository in your cluster directory, TPA makes any packages in the repository available to cluster instances. This provides 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, disabling 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.

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 you can use the `--enable-local-repo` switch with `tpaexec configure` to create an empty directory structure to use as a local repository. Use `tpaexec download-packages` to populate 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 command generates your cluster configuration and creates a `local-repo` directory and OS-specific subdirectories. See [Local repo layout](#) for details.

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 contains the full dependency tree of all packages so the entire cluster can be installed from this repository. Metadata for the repository is also created, which means it's 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 creates a `local-repo` directory and OS-specific subdirectories within it (for example, `local-repo/Debian/10`), based on the OS you select for the cluster. We recommend that you also use this structure for repositories you create manually.

For example, a cluster running RedHat 8 might have the following layout:

```
local-repo/
|-- RedHat
    |-- 8.5 -> 8
    |-- 8
        |-- repodata
```

For each instance, TPA looks for the following subdirectories of `local-repo` in order and uses 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, TPA doesn't try to set up any local repository on target instances.

Populating the repository and generating metadata

You must complete the steps that follow before running `tpaexec deploy`.

To populate the repository, copy the packages you want to include into the appropriate directory. Then generate metadata using the correct tool for your system, as follows.

Note

You must generate the metadata on the control node, that is, the machine where you run `tpaexec`. TPA copies the metadata and packages to target instances.

Note

You must generate the metadata in the subdirectory that the instance will use. That is, if you copy packages into `local-repo/Debian/10`, 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
```

Use `dpkg-scanpackages` to generate the metadata:

```
$ cd local-repo/Debian/buster
# download/copy .deb package files
$ dpkg-scanpackages . | gzip > Packages.gz
```

RedHat/SLES repository metadata

First, install the `createrepo` package:

```
$ sudo yum install -y createrepo
```

Use `createrepo` to generate the metadata:

```
$ cd local-repo/RedHat/8
# download/copy .rpm package files
$ createrepo .
```


How TPA uses the local repository

Copying the repository

TPA uses rsync to copy the contents of the repository directory to a directory on target instances. The contents include the generated metadata.

If rsync isn't already available on an instance, TPA can install it (that is, `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 copies just the rsync package using scp and installs it before copying the rest.

Repository configuration

After copying the contents of the local repo to target instances, TPA configures the destination directory as a local repository, that is, path based, rather than URL based.

If you provide, say, `example.deb` in the repository directory, running `apt-get install example` is enough 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's up to the package manager to decide which one to install. Usually it installs the latest, unless you specify a particular version.

However, if you set `use_local_repo_only: yes`, TPA disables all other package repositories, so that instances can use only the packages that you provide in `local-repo`.

29.21 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 builds and installs 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. Also set `source_directory`, `build_directory`, `build_environment`, and `build_commands`, as shown in the example.

To reuse the build directory when doing repeated deploys, run `tpaexec deploy ... --skip-tags build-clean`. 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 is restarted.

Build dependencies

If you're building from source, TPA ensures 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
```

29.22 Git credentials

You can clone Git repositories that require authentication. If you're installing Postgres from source or using `install_from_source` to compile and install extensions, and the source repositories require authentication, you can use SSH key-based authentication or HTTPS username/password based authentication to access them with TPA.

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

Docker containers on macOS can't use `ssh://` URLs because SSH access from the host to containers doesn't work. `https://` repository URLs will work fine.

SSH key authentication

If you're cloning an SSH repository and have an SSH key pair (`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 doesn't 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. Then 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 are made available for authentication to remote servers through the forwarded agent connection.

Use SSH agent forwarding with caution, preferably with a disposable key pair 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're 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 are made available to any `git clone` or `git pull` tasks (only). They aren't written to disk on the target instances.

29.23 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.

29.24 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 10/buster	✓	✓ (3.7)
Debian 9/stretch	✓	✓ (3.5)
Debian 8/jessie	✓	✗ (3.4)
Ubuntu 16.04/xenial	✓	✓ (3.5)
Ubuntu 18.04/bionic	✓	✓ (3.6)

Distribution	Python 2	Python 3
Ubuntu 20.04/focal	✗	✓ (3.8)
Ubuntu 22.04/jammy	✗	✓ (3.10)
RHEL 7.x	✓	✗ (3.6)
RHEL 8.x	✗	✓ (3.6)

Ubuntu 20.04, 22.04 and RHEL 8.x can be used only with Python 3.

RHEL 7.x ships with Python 3.6, but the librpm bindings for Python 3 are not available, so TPA must use Python 2 instead. Debian 8 does not have the Python 3.5+ required to support Ansible.

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.

29.25 Configuring /etc/hosts

By default, TPA adds lines to `/etc/hosts` on the target instances with the IP address and hostnames of every instance in the cluster. This enables the instances to use each other's names for communication within the cluster (for example, 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 any of the default entries, you can specify the complete list of `etc_hosts_lines` for an instance instead. Add only those lines 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 need to ensure the names can be resolved in some other way.

29.26 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
- raid_device:
/dev/md0
  device_name: /dev/xvdf
  volume_type:
io2
  volume_size: 64
  raid_units: 2
  raid_level: 1
  iops: 5000
  vars:
    volume_for: postgres_data
    encryption: luks
- raid_device:
/dev/md1
  device_name: /dev/xvdh
  ephemeral: ephemeral0
  raid_units:
all
  vars:
    mountpoint: /mnt/scratch

```

In this example, the EC2 instance will end up with a 32GB EBS root volume, a 64GB RAID-1 volume comprising two provisioned-iops EBS volumes mounted as `/opt/postgres/data`, and a `/tmp/scratch` filesystem comprising all available instance-store ("ephemeral") volumes, 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.

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-xxxxxxx` 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, you can define RAID volumes:

```
instances:
- Name:
  one
...
volumes:
- raid_device:
  /dev/md0
  device_name: /dev/xvdf
  raid_units: 2
  raid_level: 1
  volume_type:
  gp2
  volume_size: 100
  vars:
    volume_for: postgres_data
```

This example will attach 4×100GB EBS gp2 volumes (`/dev/xvd[f-i]`) and assemble them into a RAID-1 volume named `/dev/md0` . The handling of `volume_for` or `mountpoint` during deployment happens as with any other volume.

TPA does not currently support the creation and assembly of RAID arrays on other platforms, but you can use an existing array by adding an entry to volumes with `device_name: /dev/md0` or `/dev/mapper/xyz` . TPA will handle `mkfs` and `mount` as with any other block device.

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.

29.27 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. (This type 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 these artifact types. You can use these or any other parameters that the corresponding Ansible module accepts.

Copying files and directories to target instances is a common need. Defining artifacts can be a convenient shortcut to writing a [custom hook](#).

29.28 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.)

29.29 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 aren't 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 documentation for `bare instances`. TPA replaces the host keys during deployment.

The `manage_ssh_hostkeys` setting is meaningful only for bare instances. The generated host keys are installed on all other instances.

known_hosts

TPA adds entries for every host and its public host keys to the global `ssh_known_hosts` file on every instance in the cluster. This way, they can ssh to each other without host key verification prompts, regardless of whether they have `manage_ssh_hostkeys` set.

29.30 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 use [SSH 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).

29.31 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. These instructions are for 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 this example, TPA installs its own list of `default_packages`, the packages listed under `packages.common` on every instance, and the remaining distribution-specific packages based on the distribution the instance is running. If any of these packages isn't available, the deployment fails.

Don't list any packages that depend on Postgres. Use `extra_postgres_packages` instead.

Optional packages

You can specify a list of `optional_packages` that can be installed. They will be installed if they're 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` to remove if they're installed.

```
unwanted_packages:
  common:
    - badpkg1
  Ubuntu:
    - badpkg2
```

29.32 Running initdb

TPA first creates `postgres_data_dir` if it doesn't exist and ensures it has the correct ownership, permissions, and SELinux context. Then, unless the directory already contains a `VERSION` file, it runs `initdb` to initialize `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 initializes `postgres_data_dir`, TPA finds the `VERSION` file and therefore doesn't run `initdb`.

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.

When running `initdb`, TPA sets `TZ=UTC` in the environment and sets `LC_ALL` to the `postgres_locale` you specify.

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 in the data directory. If you change `postgres_conf_dir`, after running `initdb`, TPA moves the generated configuration files to the new location.

29.33 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](#).

29.34 SSL Certificates

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` will be installed via packages by default, but you can also install from a git branch or a local directory. See [configure-source.md](#) and [install_from_source.md](#) for more details.

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.

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.

29.35 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.

29.36 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).

29.37 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`.

29.38 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_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"`.

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"`.

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 log

The default log file is defined as `/var/log/postgres/postgres.log`. If you need to change that, you can now set `postgres_log_file` in your `config.yml`:

```
cluster_vars:
  [...]
  postgres_log_file: '/srv/fantastic_logs/pg_server.log'
```

TPA will take care of creating the directories and rotate the log when needed.

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`.

29.39 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.j2` in your cluster directory and set:

```
cluster_vars:
  postgres_hba_template: my_hba.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.

29.40 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 %}
```

29.41 Configuring .pgpass

TPA will create `~postgres/.pgpass` by default with the passwords for `postgres` and `repmgr` in it, for use between cluster instances. You can set `pgpass_users` to create entries for a different list of users.

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
```

29.42 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` is 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 for use within the cluster. This is sufficient to ensure that traffic between clients and server is encrypted in transit. Should you wish to use your own certificate signing infrastructure you may replace these after deployment is complete, or replace them during deployment using a [hook](#).

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.

29.43 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
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
)>tkc}}kly4&epaJ?;NJ:l'uT{C7D*<p
$ tpaexec store-password ~/clusters/speedy
example
Password:
$ tpaexec show-password ~/clusters/speedy example
terrible insecure
password
$
```

If you don't want the user to have a password at all, you can set `generate_password: false`.

29.44 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

```
[tpa]$ tpaexec archive-logs
~/clusters/speedy

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:

```

~/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:

```

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
```

29.45 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.

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
```

29.46 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.

```
---
# 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

```

29.47 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.).

```

---
- 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.)

29.48 Locale

For some platform images and environments, you might want to set the region and language settings.

By default, tpaexec installs 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 see the supported locales, use the following command:

```
localectl list-locales
```

Alternatively, on Debian or Ubuntu, look at the contents of the file `/etc/locales.defs`.

29.49 Patroni cluster management commands

You can use Patroni 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
```

You can also use Patroni as a failover manager by setting the following `config.yml` option:

```
cluster_vars:
  failover_manager: patroni
```

If deploying to RedHat, you must also add the `PGDG` repository to your yum repository list in `config.yml`:

```
cluster_vars:
  yum_repository_list:
  - PGDG
```

TPA `configure` adds 3 etcd nodes and 2 haproxy nodes. Etcd is used for the Distributed Configuration Store (DCS). Patroni supports other DCS backends, but they aren't currently supported by EDB or TPA.

TPA uses Patroni's feature of converting an existing PostgreSQL standalone server. This mechanism allows for TPA to initialize and manage configuration. Once a single PostgreSQL server and database is created, Patroni creates replicas and configures replication. TPA then removes any

Postgres configuration files used during setup.

Once this is set up, you can continue to manage Postgres 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 we recommend using TPA methods wherever possible.

Configuration options

You can use these configuration variables to control certain behaviors when deploying Patroni in TPA.

Variable	Default value	Description
<code>patroni_super_user</code>	postgres	User to create in Postgres for superuser role.
<code>patroni_replication_user</code>	replicator	Username to create in Postgres for replication role.
<code>patroni_restapi_user</code>	patroni	Username to configure for the Patroni REST API.
<code>patroni_rewind_user</code>	rewind	Username to create in postgres for <code>pg_rewind</code> function.
<code>patroni_installation_method</code>	pkg	Install Patroni from packages or source (for example, Git repo or local source directory if Docker).
<code>patroni_ssl_enabled</code>	no	Whether to enable SSL for REST API and ctl connection. Uses the cluster SSL cert and CA if available.
<code>patroni_rewind_enabled</code>	yes	Whether to enable Postgres rewind; creates a user defined by <code>patroni_rewind_user</code> and adds config section.
<code>patroni_watchdog_enabled</code>	no	Whether to configure the kernel watchdog for additional split-brain prevention.
<code>patroni_dcs</code>	etcd	The backend to use for the DCS. Currently, the only option is etcd.
<code>patroni_listen_port</code>	8008	REST API TCP port number.
<code>patroni_conf_settings</code>	<code>{}</code>	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.
<code>patroni_dynamic_conf_settings</code>	<code>{}</code>	Optional structured data just for DCS settings. This will be merged onto <code>patroni_conf_settings</code> .
<code>patroni_repl_max_lag</code>	None	This is used in the haproxy backend health check only when <code>haproxy_read_only_load_balancer_enabled</code> is true. See REST API documentation for possible values for <code>/replica?lag</code> .

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 by way of 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 aren't supported. See notes that follow.
- 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.
- Configuration enabled by feature flags, such as `patroni_ssl_enabled`. See the table in [Configuration options](#).
- Then, finally, overrode from user-supplied settings, the `patroni_conf_settings` option. If you want to change or add a configuration not controlled by a feature flag, then this is the best place to do it.

Configuration is merged on top of the configuration generated by TPA from cluster information, such as IP addresses, port numbers, cluster roles, and so on. Use 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:

- You can't change the template used to generate `postgresql.conf` with the setting `postgres_conf_template`.
- You can't 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
```

You can also override full blocks (with an example from Patroni documentation):

```
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's present in the default TPA config vars, you can set the value to `null`. This causes 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
```

TPA provides this 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 (for example, to perform maintenance) run:

```
tpaexec switchover cluster_name <new_primary>
```

The `new_primary` argument must be the name of an existing cluster node that's currently running as a healthy replica. Checks are performed to ensure this is true before a switchover is performed.

Once a switchover has been performed, we recommend that you run `deploy` and `test` to ensure a healthy cluster:

```
tpaexec deploy cluster_name
tpaexec test cluster_name
```

TPA detects the current role of nodes during deploy regardless of what `config.yml` contains, for example, if a different node is the leader.

29.50 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.

- [Adding the `vector` extension through configuration](#)
- [Specifying extensions for configured databases](#)
- [Including shared preload entries for extensions](#)
- [Installing Postgres-related packages](#)

TPA recognized 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`
- `autocluster`
- `refdata`

29.51 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.

29.52 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 (v3.9.18, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.15.9)
Validated: ea844d1b90295597d080bbf824dbbc6954886cb54ffdb265c7c71b99bedee67b [OK]
```

29.53 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. At present the only supported architecture is `PGD-Always-ON`
- `--pgd-proxy-routing <global|local>` (required) 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 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 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
```

30 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:

```
[tpa]$ tpaexec deploy <clustername> --excluded_tasks barman
```

To deploy running only repmgr-related tasks:

```
[tpa]$ tpaexec deploy <clustername> --included_tasks repmgr
```

To deploy without trying to set hostnames on the instances:

```
[tpa]$ 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.

- 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.

- 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.

- 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.

- 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.

- postgres-clean

Tasks which clean up the build directory if postgres is being built from source.

- 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.

- sysctl

Tasks which set and reload sysctl settings.

- sysstat

Tasks related 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.