EDB POSTGRES® AI

# Best Practices for DevOps with Postgres®

Nick Ivanov

enterprisedb.com

## Table of Contents

Postgres in a DevOps context ..... 3

Automated provisioning and configuration ..... 3

- Trusted Postgres Architect ..... 3
- Cloud-native Postgres ..... 4

Schema and code management ..... 5

- Automated testing ..... 5

Continuous deployment for Postgres ..... 7

- Blue/green deployment ..... 7
- Canary deployment ..... 7

Monitoring and observability ..... 8

- Performance optimization ..... 8
- Proactive issue detection ..... 8
- Faster feedback loop ..... 9
- Resource management and cost control ..... 9
- EDB monitoring tools ..... 9
  - Postgres Enterprise Manager ..... 9
  - Prometheus and Grafana integration ..... 10
  - EDB Postgres AI Hybrid Manager ..... 10
  - OpenTelemetry ..... 11

Appendix I. Example TPA configuration file ..... 11

Appendix II. Important monitoring metrics ..... 14

- Server events ..... 14
- Operating system ..... 15
- Application behavior ..... 15
- Replication performance ..... 16
- Vacuum performance ..... 16

## Postgres in a DevOps context

The term *DevOps* is a portmanteau of development and operations. It describes a methodology that bridges the gap between software development and IT operations teams, aiming to make organizations more agile, shorten the system development life cycle (SDLC), and provide continuous delivery of high-quality software with maximum consistency and reliability.

The DevOps framework relies heavily on specific practices and tools:

- **Continuous integration and continuous delivery** (CI/CD) are two related concepts, often mentioned together, that define the rapid deployment of software and frequent system update cycles to accommodate changing business demands and enable constant improvement of user experience.
- **Automation** is a central focus for DevOps engineers and site reliability engineers. Reducing the number of manual steps in software testing and deployment relieves engineers from mundane tasks, improves software quality, and ensures repeatable execution of deployment and infrastructure management tasks.
- **Infrastructure as code** (IaC) is a principle that involves managing infrastructure using code (e.g., using tools such as Terraform or Ansible). IaC enables automated and repeatable deployments, ensuring that development, testing, and production environments look alike.
- **Observability and monitoring** allow DevOps teams to gain visibility into performance, health, and cost of systems. Observability provides valuable and contextual insights at every phase of the software lifecycle, feeding these data back to the developers. Reliable and configurable monitoring tools help engineers react to the changes in system status while avoiding being overwhelmed by unnecessary alerts.

EnterpriseDB (EDB) as a company embraces the DevOps principles, both as the developer of software that is used in critically important systems by millions of users around the world and as the provider of tools that enable users to practice DevOps in their environments when it comes to deploying and managing Postgres and related tools.

CloudNativePG™ and two EDB products that are built on the same foundation, EDB Postgres AI for CloudNativePG **Cluster** and EDB Postgres AI for CloudNativePG **Global Cluster**, are mature Kubernetes operators. They allow you to use a declarative approach to designing highly available Postgres database clusters that seamlessly integrate with the same CI/CD pipelines you use to deploy and manage other applications, making the database a first-class citizen in your DevOps processes.

EDB maintains Trusted Postgres Architect, an Ansible-based orchestration tool that deploys Postgres clusters according to EDB's best practices and recommendations. It extends the same declarative, infrastructure-as-code approach to bare metal servers, virtual machines, cloud-based compute resources, and containers.

It is not surprising, then, that in 2024, EDB Postgres AI won that year's DevOps Dozen Award in the *Best DevOps for DataOps/Database Solution* category.

## Automated provisioning and configuration

### Trusted Postgres Architect

Trusted Postgres Architect, or TPA, is an Ansible-based automation tool for deploying highly available Postgres clusters that follow proven architecture patterns. It embodies EDB's best practices and recommendations, derived from years of experience implementing and supporting Postgres database systems. TPA is used internally to facilitate automated testing. EDB professional services consultants and customers alike also use it externally to ensure consistent deployment of Postgres clusters across development, test, and production environments, making the tool a core component of DevOps practices for Postgres.

TPA core functionality and design principles include:

- **Orchestration and automation:** TPA uses Ansible, a widely adopted, powerful, and flexible automation platform that orchestrates deployment and configuration of multiple servers consistently.

- **Declarative configuration:** TPA operates using a declarative configuration mechanism. It is driven by a single YAML configuration file that describes the desired cluster architecture and configuration parameters. TPA can help the user create a basic configuration defined with a few command-line arguments; the configuration can then be customized as necessary.
- **Deployment stages:** TPA operates in four distinct stages to bring up a cluster: configuration generation, provisioning, deployment, and testing.
    - **Provisioning:** TPA can provision AWS EC2 instances and Docker containers, or deploy to existing servers and virtual machines.
    - **Deployment:** The deployment stage installs and configures the necessary software, including Postgres, and associated components (such as EDB Postgres Distributed, Barman, PgBouncer, or EDB Failover Manager). It also configures the operating system (e.g., kernel settings, users, log rotation).
- **Idempotency:** The provisioning, deployment, and testing stages are designed to be idempotent. If the process is rerun and nothing has changed in the configuration or on the instances, no action will be performed. If changes were made to the configuration file, TPA safely deploys those changes.
- **Cluster management:** Once deployed, TPA provides convenient cluster management functions, including configuration changes, switchover, and zero-downtime minor version upgrades of all software components.
- **Extensibility:** TPA is extensible through Ansible, allowing users to write custom commands, custom tests, and hook scripts that are invoked during various stages of deployment.

Postgres instances deployed with the help of TPA are automatically configured based on the EDB experience and industry best practices, taking into account the available server hardware resources and cluster architecture. This primarily applies to the Postgres configuration parameters in the `postgresql.conf` file and the host-based authentication properties in `pg_hba.conf`. Users can customize the database cluster configuration by supplying the desired values in the TPA configuration manifest.

## Cloud-native Postgres

Cloud-native Postgres is the embodiment of DevOps culture—it follows the microservices architecture pattern and relies on Kubernetes container management and orchestration mechanisms to achieve database cluster scalability and availability goals. It allows application developers to accelerate delivery and adapt to continuous change. All cloud-native Postgres variants implement Kubernetes operators that allow users to deploy and maintain Postgres clusters in various architecture patterns using a declarative, infrastructure-as-code approach. All aspects of a database cluster can be defined in the deployment manifest: topology, replication configuration and consistency settings, database configuration parameters, backup schedule, storage and network characteristics, and so on. The manifest can be maintained in a source code repository integrated with a CI/CD pipeline; any change in the manifest can trigger automatic deployment of the new configuration.

Cloud-native Postgres offers users these benefits:

- **Automatic scaling and elasticity:** Resources can be configured to automatically scale up or down based on demand, optimizing performance and cost efficiency.
- **Reduced operational overhead:** Routine maintenance tasks such as patching, backups, and updates can be defined declaratively and performed automatically by the Kubernetes operator.
- **High availability:** Built-in high availability configurations, such as multi-zone replication and automatic failover, ensure minimal downtime.
- **Observability:** Monitoring and logging tools are built in, allowing easy integration with cloud-native monitoring solutions (including CloudWatch, Azure Monitor, Prometheus, and Grafana).
- **CI/CD integration:** Easy integration with continuous integration and continuous deployment pipelines supports agile development and business practices.
- **Advanced deployment methodologies:** These support sophisticated DevOps practices for application updates, such as blue-green deployments, canary releases, and rolling upgrades.

## Schema and code management

Database migration tools enforce a controlled, versioned, and repeatable approach to database evolution, treating the schema as part of application code. They are essential for managing and automating PostgreSQL schema changes across different environments (development, staging, production) in a CI/CD process that is important to the users following DevOps practices.

Using schema migration tools offers the following benefits:

- **Consistency** of changes is guaranteed in all environments (development, test, production); the databases have the exact same schema structure, reducing "works on my machine" issues.
- **Version control** supports the "infrastructure as code" principle. The migration scripts are committed alongside application code in the source control systems, providing a review and approval workflow and a complete history of the database schema's evolution.
- **Automated CI/CD** pipelines natively incorporate schema changes, running the corresponding migration or update commands after the application is built.
- **Safe evolution** of schema artifacts is ensured by the immutability of each migration file. Once the change is applied, a new script must be created for every subsequent change, forcing a clear, auditable timeline of schema modifications.

There are many schema migration tools that natively support Postgres. The alternatives often fall into categories based on their primary approach: versioned vs. declarative.

Version-based solutions include Flyway, Liquibase, pgroll, and others; they manage changes via a sequence of numbered or named scripts that implement the necessary modifications or roll them back.

Other tools, such as Atlas or pgschema, take the declarative approach. They let users define the final, desired state of the database schema, often in a simple language such as HCL or a set of SQL DDL files. The tool handles the tasks of performing a "diff" between the desired state and the actual database state, then automatically generates the safe, incremental SQL migration steps needed to bridge the gap. This approach significantly reduces manual effort and can help detect destructive changes before they are applied.

### Automated testing

Integrating database schema evolution into the CI pipeline is incomplete without provisions for automated testing. Database test automation solves two critical problems:

- Ensuring data integrity and schema consistency by continuously validating all database constraints and applying changes consistently across all environments
- Validating application logic embedded in the database objects, such as functions, stored procedures, triggers, and views

Users should set up temporary, disposable database instances for the purposes of automated testing. Running tests in shared environments or production instances is never acceptable. Container-based instances are ideal for this purpose.

Automated tests use the schema migration tools discussed previously to set up versioned database schemas and populate them with predefined test data to ensure deterministic results.

Two types of automated tests can be integrated into the CI pipelines:

- **Unit tests (logic verification):** These focus on individual database objects, isolating their logic.
  - **What they test:** Stored procedures, functions, and complex triggers.

    - **How they work:**
        - **Arrange (setup):** Insert specific, minimal seed data directly into the necessary tables.
        - **Act (execute):** Call the stored procedure or function being tested.
        - **Assert (verify):** Check the resulting state of the database or the return value of the function against the expected result.
    - **Tools:** PostgreSQL-native testing framework pgTap allows users to write unit tests directly in SQL, which are then run by the CI pipeline.
- **Integration tests (application stack):** These tests verify how the application code interacts with the database.
    - **What they test:** ORM (object-relational mapping) queries, application service layers, transaction boundaries, and complex SQL generated by the application.
    - **How they work:**
        - **Start:** The CI pipeline starts both the database container and the application server (e.g., a Spring Boot or Node.js service).
        - **Execute**: The application's test suite calls the service endpoints or business logic, which in turn executes database commands.
        - **Assert:** The test verifies that the final outcome (the data stored in the database, the API response, etc.) is correct.

Some of the challenges inherent in automated database testing, as well as the approaches that help mitigate them, are shown in the following table:

| Challenge | Mitigation |
|---|---|
| Test data management | **Use fixtures:** Develop versioned data migration scripts (separate from schema migrations) to populate reference data, lookups, and minimal test data required for logic execution. |
| Speed and performance | **Limit scope:** Use in-memory databases (if applicable) for application unit tests; for full database tests, use small, highly optimized data fixtures to minimize setup and teardown time. |
| State consistency | **Transactional tests:** Ensure that each test runs within its own transaction. The transaction should be rolled back at the end of the test, guaranteeing that the next test starts with the same clean, migrated state. |
| Tooling fragmentation | **Standardize:** Choose a migration tool and a testing framework that can be easily invoked via command-line interface (CLI) commands within your CI script (e.g., Jenkins, GitHub Actions, GitLab CI). |

## Continuous deployment for Postgres

As discussed earlier, deployment automation and integration features of EDB Postgres AI enables users to implement continuous deployment. However, this practice comes with its own set of challenges:

- **Software release velocity:** DevOps teams are pressured by the CI/CD model to deploy software changes quickly. Companies aim to release significant changes every two weeks or more frequently, instead of releasing yearly.
- **Risk of instability:** While developers are under constant pressure to create new features, every change can potentially break the application in production. The DevOps team is responsible for successfully getting these features to customers while having the capability to roll back changes if the application falters.
- **Need for zero-downtime:** Implementing continuous deployment necessitates advanced techniques to ensure the application availability, such as zero-downtime deployment techniques for schema changes and version upgrades, or sophisticated strategies such as blue-green deployments and canary releases.

EDB Postgres AI is well positioned to help developers address these challenges. Automation tools and declarative database infrastructure definitions enable rapid, repeatable deployment.

Rich replication features inherent in Postgres allow users to set up multi-node database clusters that allow minor version upgrades of the operating system, database software, and custom applications in a rolling fashion, without incurring downtime. Replicas are upgraded first, then one of the replicas is promoted to the primary role, and the remaining cluster node is finally upgraded.

EDB Postgres Distributed (PGD) supports even more sophisticated upgrade procedures, allowing major database software version upgrades, as well as blue/green and canary deployments while the database cluster remains operational to serve client workloads.

### Blue/green deployment

The term *blue/green deployment* refers to a software implementation technique that relies on the presence of two identical—in terms of the infrastructure capacity—production environments, only one of which is “live” and serves the application workload at any given time. The “blue” environment is the current production environment. The “green” environment is the future production environment that contains a new version of the application or the database; it is where the deployment, configuration, and acceptance testing are performed without affecting live users. Once the tests are successfully completed, application traffic is redirected to the blue environment and begins processing the workload.

To use this technique, it is necessary to set up replication of data from green to blue while the configuration and testing in the blue environment is ongoing, to ensure that both databases are in sync up to the moment of the blue system promotion. At this point, replication must be enabled in the opposite direction, from blue to green, to allow rollback without data loss if any major issues or bugs are discovered shortly after promotion.

### Canary deployment

The so-called “canary deployment” techniques can be more complicated, as they entail simultaneously maintaining the old and new versions of the database and applications, allowing a subset of users to access the new version for testing while maintaining the original version for the rest. This limits the potential impact of the new version to a small group of early adopters or beta testers. However, this means that the data are being simultaneously modified on both systems, requiring bidirectional replication between them and robust conflict avoidance and resolution capabilities.

Under this methodology the volume of application transactions is gradually shifted from the current database system to its new version, while functional and performance tests are conducted. Finally, when 100% of the application traffic is handled by the new system, the old servers can be decommissioned.

EDB Postgres Distributed offers features allowing users to perform canary deployments across PGD clusters without incurring any downtime:

- **Multidirectional replication:** PGD uses logical replication, asynchronous by default, to replicate changes to data, schemas, and database configuration parameters between all nodes in the cluster. All nodes are active simultaneously and can process write as well as read operations.
- **Node groups:** Cluster nodes are organized into groups that manage themselves independently of each other. Each group can have a leader that handles all read/write connections within the group, minimizing the risk of data conflicts.
- **Replication sets:** These allow you to manage which tables are replicated to which nodes, helping with geographic segregation of data and isolating incompatible database objects between the schema versions.
- **Conflict resolution:** PGD has an extensive set of conflict resolution rules, allowing you to fine-tune how incompatible changes occurring on multiple nodes simultaneously should be handled. Users can also define stream triggers to implement more complex conflict resolution rules that help deal with small schema differences in the old and new database versions.

Of course, setting up a PGD cluster for canary deployments can be automated using TPA to ensure consistent configuration and reduce the risk of errors.

## Monitoring and observability

Monitoring of the deployed Postgres databases is important in the DevOps context as it enables users to reach the key framework goals: reliability, performance, and rapid feedback. Monitoring provides the necessary visibility to ensure that the database, a critical application component, is healthy and operating optimally throughout the development, testing, and production lifecycle.

The list of suggested metrics to monitor is presented in Appendix II.

Monitoring in the following key areas should be established:

### Performance optimization

**Slow queries** are exposed by the **pg_stat_statements** module, which is recommended to be set up by default in all Postgres instances. Identifying slow queries early in the development cycle allows engineers to promptly address potential issues, rewrite queries, or modify the database model as necessary.

EDB Postgres AI platform provides additional features for slow query monitoring, maintaining query performance history, identifying execution plan drift, and providing optimization suggestions.

Resource bottlenecks, such as excessive CPU utilization, growing use of memory, or high I/O activity, are reported by the server operating system monitoring tools.

### Proactive issue detection

A core DevOps goal is to improve system reliability and minimize downtime. Monitoring enables proactive measures:

- **Health checks:** Tracking metrics including disk space, replication lag (for high availability), and the status of background processes (such as the autovacuum daemon).
- **Alerting:** Setting up alerts for critical thresholds (e.g., disk usage above 80%, inactive replication slots, growing replication lag, transaction ID wraparound risk) allows operations teams to address issues before they impact users.

## Faster feedback loop

Early feedback about database performance and reliability allows teams to shift-left the efforts required to ensure optimal database operation. Monitoring helps integrate database performance into the entire software lifecycle.

- **Development/staging insight:** Performance issues identified in lower environments can be caught earlier in the cycle, preventing costly fixes in production.
- **Deployment validation:** Monitoring metrics immediately after a new deployment helps validate that the changes haven't introduced any performance regressions or new errors.

## Resource management and cost control

By understanding the database workload and resource usage, teams can make informed decisions:

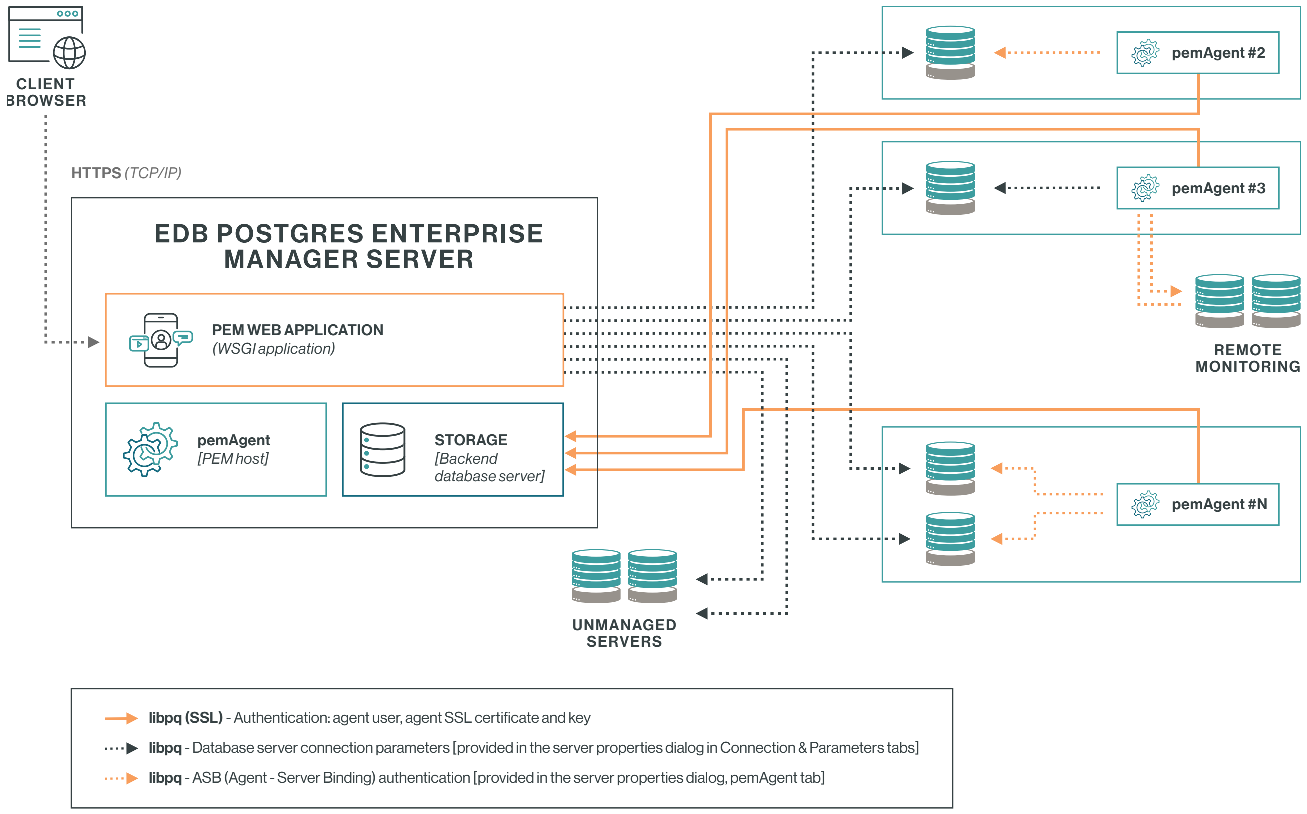
- **Accurate scaling:** Ensuring that resources are provisioned appropriately—neither over-provisioning, thus wasting money, nor under-provisioning, causing performance issues and availability problems.
- **Workload analysis:** Identifying peak usage times helps schedule maintenance, backups, or batch jobs for periods of low impact.

## EDB monitoring tools

### Postgres Enterprise Manager

Postgres Enterprise Manager, or PEM, is a comprehensive database design and management tool, allowing its users to monitor a wide range of predefined and custom metrics collected not only on the database servers but also on other components of the database infrastructure, such as failover managers, connection poolers, backup servers, and so on. Users can also define metric thresholds and configure PEM to alert them when these thresholds are crossed.

PEM is deployed using client-server architecture, with agents collocated with database servers or monitoring servers remotely, as shown in the following diagram:


CLIENT BROWSER
HTTPS (TCP/IP)
EDB POSTGRES ENTERPRISE MANAGER SERVER
PEM WEB APPLICATION (WSGI application)
pemAgent [PEM host]
STORAGE [Backend database server]
pemAgent #2
pemAgent #3
REMOTE MONITORING
pemAgent #N
UNMANAGED SERVERS
libpq (SSL) - Authentication: agent user, agent SSL certificate and key
libpq - Database server connection parameters [provided in the server properties dialog in Connection & Parameters tabs]
libpq - ASB (Agent - Server Binding) authentication [provided in the server properties dialog, pemAgent tab]

PEM comes with a command-line interface, **pemworker**, which allows users to register agents and Postgres servers with the PEM server programmatically, building PEM integration into the DevOps deployment pipelines.

## Prometheus and Grafana integration

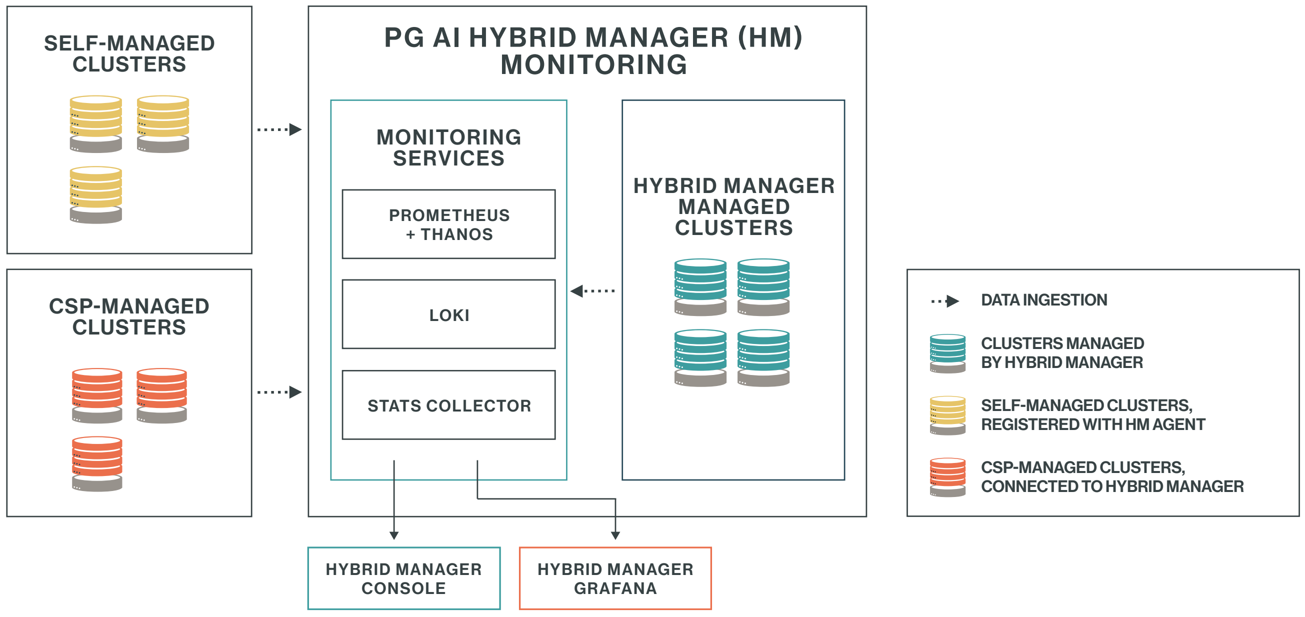
Prometheus is a popular solution for metrics collection, storage, and retrieval. In combination with Grafana, a flexible metrics dashboard and alerting service, it is becoming a de facto standard for open source metrics management, particularly in the context of microservices-based applications running on various Kubernetes platforms. CloudNativePG offers built-in integration with both, supplying a Prometheus exporter endpoint, a set of predefined metrics, and a collection of sample Grafana dashboards for Postgres, providing users with insights into the operation of their Postgres clusters.

EDB developers do not only use Prometheus as the principal monitoring system in the EDB Postgres AI platform and support its use by our customers; they also contribute to the continuing development of the software.

## EDB Postgres AI Hybrid Manager

EDB Postgres AI Hybrid Manager is the next-generation observability platform for databases. It can collect metrics, logs, and traces not only from Hybrid Manager–managed Postgres instances but also from customer-maintained Postgres and Oracle databases, as well as database services deployed on popular cloud computing platforms such as Amazon AWS, Google Cloud, and Microsoft Azure.

The Hybrid Manager platform comprises Prometheus for the storage and retrieval of metrics, with Thanos enabling scalability and availability; Loki for log consolidation; and Grafana for customizable dashboards. It also incorporates a rich set of dashboards in its own user interface, giving users immediate access to performance and resource utilization insights across their entire database estate, as a single-pane-of-glass view. The diagram below provides a high-level overview of the Hybrid Manager deployment architecture:


SELF-MANAGED CLUSTERS
CSP-MANAGED CLUSTERS
PG AI HYBRID MANAGER (HM) MONITORING
MONITORING SERVICES
PROMETHEUS + THANOS
LOKI
STATS COLLECTOR
HYBRID MANAGER MANAGED CLUSTERS
HYBRID MANAGER CONSOLE
HYBRID MANAGER GRAFANA
DATA INGESTION
CLUSTERS MANAGED BY HYBRID MANAGER
SELF-MANAGED CLUSTERS, REGISTERED WITH HM AGENT
CSP-MANAGED CLUSTERS, CONNECTED TO HYBRID MANAGER

### OpenTelemetry

OpenTelemetry, or OTel, is an observability framework for generating, publishing, and collecting telemetry data, such as metrics, logs, or application traces. The **`edb_otel`** extension allows users to instrument their database procedural code, such as stored procedures and functions, to emit metrics and traces using the OTel protocol, which can then be easily integrated into the existing monitoring infrastructure, providing valuable insights into the database application performance and aid in troubleshooting of newly deployed code. This helps close the deployment feedback loop, allowing teams to follow DevOps practices.

## Appendix I. Example TPA configuration file

```
---
architecture: M1
cluster_name: my_first_cluster
cluster_tags:
Owner: nick.ivanov@enterprisedb.com

cluster_rules:
# PEM
- cidr_ip: 0.0.0.0/0
from_port: 443
proto: tcp
to_port: 443
- cidr_ip: 172.31.80.0/20
from_port: 0
to_port: 65535
proto: tcp
- cidr_ip: 172.31.80.0/20
from_port: -1
to_port: -1
proto: icmp

ec2_ami:
Name: RHEL-9.4.0_HVM-20240423-x86_64-62-Hourly2-GP3
Owner: '309956199498'
ec2_instance_reachability: public
ec2_instance_key: key_name
cluster_bucket: auto

ec2_vpc:
us-east-1:
Name: tpa
cidr: 172.31.80.0/20
cluster_vars:
default_barman_minimum_redundancy: 3
enable_pg_backup_api: true
edb_repositories:
- enterprise
epas_redwood_compat: false
failover_manager: efm
efm_version: '5.0'
postgres_coredump_filter: '0xff'
postgres_version: '17'
postgresql_flavour: epas
pem_agent_package_version: '10.1.0'
pem_server_package_version: '10.1.0'
preferred_python_version: python3
packages:
```

```
RedHat:
- kernel-modules-extra
- barman-cli-cloud
common:
- edb-lasso

yum_repository_list:
- EPEL

locations:
- Name: a
az: us-east-1b
region: us-east-1
subnet: 172.31.80.0/20

instance_defaults:
default_volumes:
- device_name: root
encrypted: false
volume_size: 50
volume_type: gp3
platform: aws
# 4/32 GB, Xeon 4th gen.
type: r7i.large
vars:
ansible_user: ec2-user
packages:
RedHat:
- barman-cli-cloud
- python3-boto3

instances:
- Name: pg1
location: a
node: 1
backup: barman
role:
- primary
- pem-agent
volumes:
- device_name: /dev/sdf
encrypted: false
vars:
volume_for: postgres_data
volume_size: 100
volume_type: gp3
- device_name: /dev/sdg
encrypted: false
vars:
volume_for: postgres_wal
volume_size: 50
volume_type: gp3
- Name: pg2
location: a
node: 2
role:
- replica
- pem-agent
upstream: pg1
volumes:
- device_name: /dev/sdf
```

```
encrypted: false
vars:
volume_for: postgres_data
volume_size: 100
volume_type: gp3
- device_name: /dev/sdg
encrypted: false
vars:
volume_for: postgres_wal
volume_size: 50
volume_type: gp3
- Name: pg3
location: a
node: 3
role:
- replica
- pem-agent
upstream: pg1
volumes:
- device_name: /dev/sdf
encrypted: false
vars:
volume_for: postgres_data
volume_size: 100
volume_type: gp3
- device_name: /dev/sdg
encrypted: false
vars:
volume_for: postgres_wal
volume_size: 50
volume_type: gp3
- Name: barman
location: a
node: 3
role:
- barman
- pem-agent
volumes:
- device_name: /dev/sdf
encrypted: false
vars:
volume_for: barman_data
volume_size: 200
volume_type: gp3

- Name: pemserver
location: a
node: 4
role:
- pem-server
```

## Appendix II. Important monitoring metrics

This document discusses the various metrics available from the EDB PostgreSQL–based monitoring interface as well as from the Postgres Enterprise Manager (PEM) database. The list of metrics included herein is not exhaustive; it is a baseline recommendation. Customers should consider adapting these recommendations to their specific needs and environments.

The Alert column in the tables below shows which metrics may warrant setting up alerts; specific alert threshold values depend on the specific application workload pattern and service-level requirements.

### Server events

| Metric | Sources | Alert | Description |
|---|---|---|---|
| PostgreSQL instance start-up or shutdown | PostgreSQL log file | Y | Scan log for lines including **received SIGHUP**, **reloading configuration files**, or **database system was shut down**. |
| Backup success or failure | Backup job log | Y | Monitor the backup job log for error messages to ensure backups complete successfully on schedule. |
| WAL archiving failure | PostgreSQL log file `pg_stat_archiver` | Y | While occasional WAL archiving failure is not critical, consistent failures will lead to the WAL files accumulating on disk, potentially leading to disk space shortage, and affect the database RPO compliance. |
| Database errors | PostgreSQL log file | Y | Scan log for lines including the error levels **`ERROR`**, **`FATAL`**, **`PANIC`**. |
| Checkpoint frequency | `pg_stat_checkpointer` | Y | Too-frequent checkpoints and a high number of requested checkpoints can indicate suboptimal WAL and checkpointer configuration. |

### Operating system

| Metric | Sources | Alert | Description |
|---|---|---|---|
| Disk utilization | Data directory<br>WAL directory<br>Backup directory | Y | Monitor for sudden growth of used disk space. The alert threshold should depend on the volume size rather than percentage of used space and must allow sufficient time to react. |
| CPU utilization | OS | N | Consistent high CPU utilization can indicate suboptimal tuning of the database (missing indexes, wrong parallelization settings, misbehaving applications, etc.) and requires detailed analysis. |
| Memory utilization | OS | N | It is normal for a database server to acquire a large amount of memory over time and never release it to the operating system. However, memory utilization should be monitored to identify possible memory leaks and support resource planning. |

### Application behavior

| Metric | Sources | Alert | Description |
|---|---|---|---|
| Number of sessions | `pg_stat_activity` | N | The number of sessions should stay below the server **`max_connections`** setting. |
| Number of idle sessions | `pg_stat_activity` | N | A large number of sessions in the idle state might indicate an incorrect configuration of the connection pool or poorly designed applications. Although the overhead of one idle connection is not significant, a large number of such connections can impact the server performance. |
| Number of idle-in-transaction sessions | `pg_stat_activity` | Y | The presence of sessions that remain in the "idle in transaction" state for more than a few seconds likely indicates poor application design or application errors. Such sessions can block concurrent transactions and prevent efficient vacuuming of tables. |
| Deadlocks | `pg_stat_database` | N | Deadlocks are often an indicator of logical errors in the applications. |
| Locks | `pg_locks`<br>`pg_stat_activity` | N | A constantly high number of locks and locks held for a long time can severely affect application performance. They can be an indicator of poor application design and user errors. |

### Replication performance

| Metric | Sources | Alert | Description |
|---|---|---|---|
| Inactive replication slots | `pg_replication_slots` | N | Inactive replication slots will cause WAL files to accumulate, which can eventually cause WAL disk space exhaustion. |
| Replication lag | `pg_stat_replication` | N | Monitor for growing replication lag or constant lag that exceeds the RPO requirements. |

### Vacuum performance

| Metric | Sources | Alert | Description |
|---|---|---|---|
| Transaction wraparound risk (age of **datfrozenxid**) | `pg_database` | Y | See discussion in www.postgresql.org/docs/current/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND. |
| Tables not vacuumed or analyzed recently (**`last_vacuum`**, **`last_autovacuum`**, **`last_analyze`**, **`last_autoanalyze`** columns) | `pg_stat_replication` | N | Tables that are constantly modified should be regularly analyzed and vacuumed. |

## About the author

**Nick Ivanov**
Solutions Architect, EDB
Connect on LinkedIn

Nick Ivanov is a seasoned solutions architect at EDB. Since April 2022, he has brought extensive expertise in database architecture and analytics from a notable tenure at IBM from May 2015 to March 2022. He holds a Dipl.-Ing. degree in computing systems and networks from Bauman Moscow State Technical University.