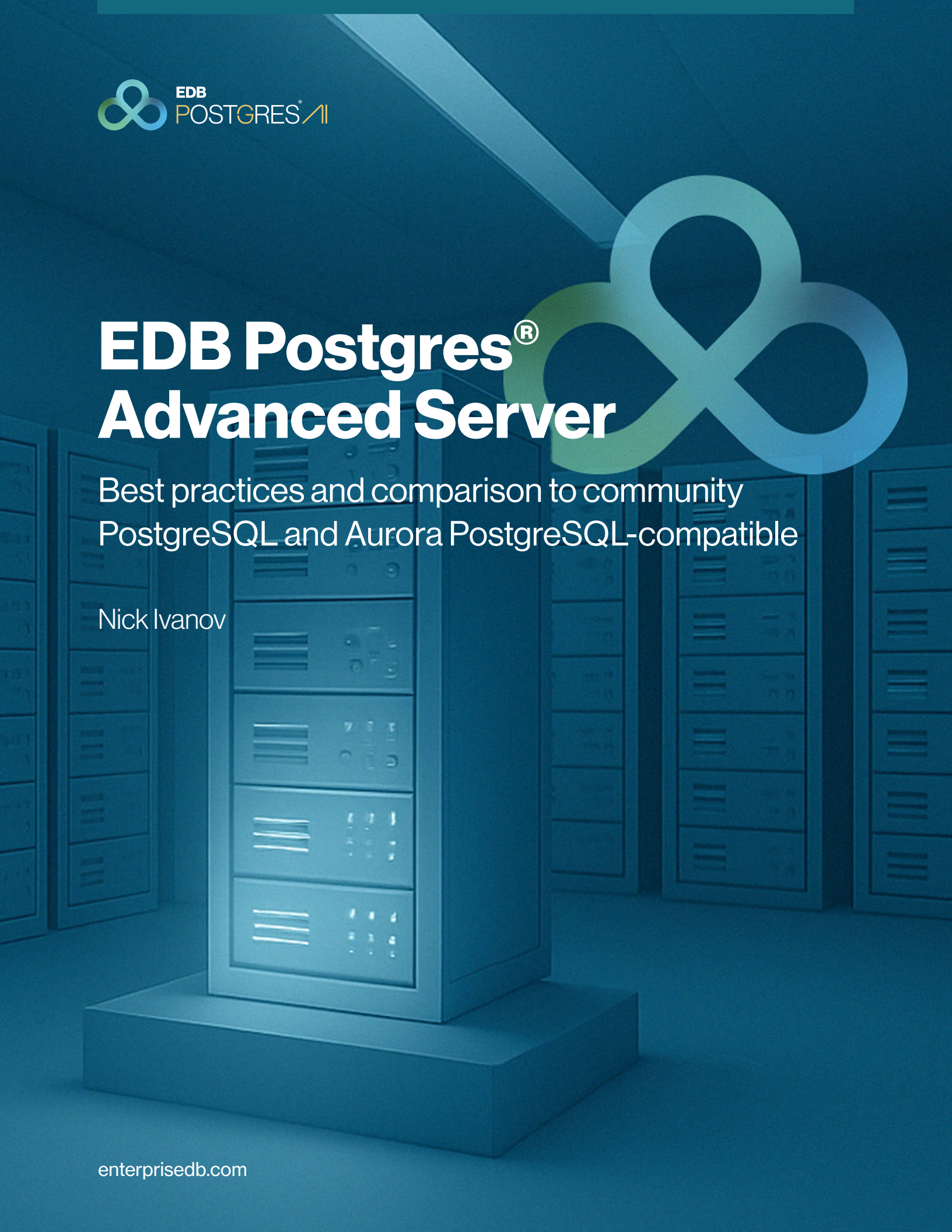
EDB POSTGRES AI

EDB Postgres® Advanced Server

Best practices and comparison to community PostgreSQL and Aurora PostgreSQL-compatible

Nick Ivanov



enterprisedb.com

Table of contents

Introduction: Why EDB Postgres Advanced Server 3

Database administration: Fine-tuning governance and visibility 4

Enhanced SQL capabilities: Oracle semantics to reduce migration risk 5

Database and application security 7

Performance monitoring and analysis: Observe, hypothesize, fix 11

Application development utilities: Familiarity lowers migration risk 12

Advanced replication: Design with portable guardrails 13

EPAS-only configuration deep dive) 15

Implementation blueprint 16

Conclusion: Where EPAS wins 19

Introduction: Why EDB Postgres Advanced Server

When enterprises evaluate PostgreSQL, they typically choose between community PostgreSQL and a cloud-managed service. While there are a few credible offerings across clouds (RDS for PostgreSQL, Azure Database for PostgreSQL, Cloud SQL), in AWS-first organizations the primary managed alternative considered is Amazon Aurora PostgreSQL-compatible. This paper therefore focuses on **community PostgreSQL** (the upstream baseline) and **Aurora PostgreSQL-compatible** (the most common "default" in AWS shops), because those two baselines resolve the engine-level contrasts that matter for **EDB Postgres Advanced Server (EPAS)**.

EPAS is "Postgres plus": It adds additional pillars on upstream PostgreSQL—database administration, enhanced SQL, database/application security, performance monitoring/analysis, application development utilities, and advanced replication—available in both Postgres mode and Oracle compatibility mode (which exposes extra migration helpers).

This paper aims to demonstrate that **EPAS reflects better engineering practice** for three recurring enterprise problems: leaving Oracle without destabilizing rewrites, enforcing security where the data actually lives, and operating Postgres predictably at scale.

1. Runtime Oracle compatibility in the engine (migration safety)

EPAS implements Oracle-style catalogs/keywords and PL/SQL-compatible behavior, and it includes `dblink_ora`, an OCI-based database link that lets EPAS `SELECT/INSERT/UPDATE/DELETE` directly against Oracle databases. This enables phased cutovers and live parity checks instead of high-risk "big-bang" migrations; community PostgreSQL and Aurora don't ship an engine-level Oracle link or compatibility layer.

Why this is best practice: It helps to align semantics at the database boundary (and can later converge to native PG) instead of scattering fragile translation logic across apps.

2. In-engine security controls (controls live with the data)

While community PostgreSQL emphasizes client-side patterns and Aurora focuses encryption in the storage/KMS plane, EPAS moves the following key controls into the database engine: Transparent Data Encryption (TDE) for data files and write-ahead logs (WALs) (supported in EPAS 15+), data redaction (policy-driven masking at query time), SQL/Protect (role-aware SQL-injection guardrails with **learn** → **passive** → **active** modes), password profiles, and audit-log password redaction.

Why this is best practice: This allows controls to travel with the database across environments and remain auditable in one place—without duplicating masking in apps or depending solely on a cloud storage plane. (Community PostgreSQL documents encryption options but no native TDE; Aurora documents AES-256 at-rest encryption managed by AWS KMS in the cloud.)

3. Operator-grade governance AND observability (predictable day-two ops)

EPAS adds features such as Resource Manager (bind sessions to resource groups and cap usage), EDB Wait States (sampled wait-event time series stored in tables), and Dynatune (baseline by workload profile using `edb_dynatune / edb_dynatune_profile`), as well as tuning tools such as Query Advisor (hypothetical indexes + recommendations) and SQL Profiler (targeted traces).

Why this is best practice: This creates a repeatable operating model (observe → hypothesize → fix) implemented as database-native features that can be used to version, promote, and be reused in any environment.

Portability

Finally, this solution allows portability to become an architectural through line. This can help emphasize the *delivery mechanism* for security, compatibility, and operations—not as an end in itself. EPAS runs on-prem, in public cloud, and in Kubernetes; notably, the EDB Postgres for Kubernetes operator exposes TDE as a first-class setting in the

epas stanza, so that the encryption posture is kept as code and follows a given manifest. By comparison, Aurora's at-rest encryption is excellent inside AWS but is, by design, tied to AWS KMS and the RDS/Aurora storage plane. If one values optionality—hybrid today, multi-cloud tomorrow—EPAS's security and operations model is the solution.

How to read this paper

This paper addresses the *outcomes* that many enterprises need: safer migration, in-database security, and day-two predictability. It also shows how EPAS can deliver to them as engine features. Portability is the primary delivery mechanism; the same settings and runbooks work on-prem, in any cloud, and in Kubernetes.

Database administration: Fine-tuning governance and visibility

Enterprises don't just need a fast database on good days; they need one that can be governed under mixed workloads and whose behavior changes can be explained. EPAS's administration pillar is built for this reality: It puts resource controls, longitudinal visibility, and operator tooling inside the engine, so that posture is versionable and portable across on-prem, cloud, and Kubernetes environments.

Resource Manager: Shaping mixed workloads without replatforming

When transactional traffic (OLTP), data pipelines (ETL), and ad hoc analytics all share the same cluster, a **workload control** solution is needed—a way to keep business-critical transactions responsive while letting heavy jobs run under limits. EPAS's Resource Manager helps accomplish this inside the database by assigning sessions to named *resource groups* and enforcing per-group limits. Defaults and overrides become plain database settings (for example, `SET edb_resource_group …`), allowing the ability to limit how many groups can be active via `edb_max_resource_groups`. Because this lives in the engine, the same policy is easy to review, version, and promote across environments.

Why this is a best practice: Governing contention at the database boundary is predictable and auditable. This helps to solve the question of "who was running under which limits" during an incident, and one doesn't have to replatform work in order to achieve fairness. Keeping the controls in the engine also preserves portability as one moves between on-prem and cloud.

How it compares: Community PostgreSQL doesn't ship a first-class, SQL Server–style "resource governor." AWS's own migration playbook states plainly that community PostgreSQL lacks built-in resource management of that kind. Aurora PostgreSQL likewise doesn't expose a per-session resource governor; capacity is managed in the AWS control plane (cluster sizing/Serverless) rather than as an engine feature. EPAS, by contrast, gives you database-native workload controls that travel with the database.

EDB Wait States: A durable, query-aware record answering, "What changed?"

When latency jumps at 10:37, guesses are expensive. EPAS's Wait States component registers as a lightweight sampler that probes each session on a schedule and records the database, user, currently running SQL, and the wait events involved (locks, I/O, CPU, inter-process communication). The history is stored in the database with practical defaults—one-second sampling and seven-day retention—and you can tune both via `edb_wait_states.sampling_interval` and `edb_wait_states.retention_period`. That rolling time series lets teams reconstruct incidents, prove regressions, and justify capacity changes with evidence rather than anecdotes.

Why this is a best practice: Postmortems and SLO reviews need longitudinal, query-aware evidence that survives restarts and lines up with code/config changes. Capturing it *inside the engine* gives you a single, portable source of truth you can keep with your cluster.

How it compares: In community PostgreSQL and Aurora, teams often stitch together system catalogs and external telemetry. Aurora's Performance Insights is excellent for visualizing waits in the AWS console, but it lives outside the database. EPAS's sampler and retention are database settings, so the evidence model follows your cluster wherever it runs.

Dynamic resource tuning: A transparent baseline you can defend—and refine

Every system needs a sane starting configuration that reflects hardware and intended workload. EPAS supports dynamic resource tuning via two explicit parameters: `edb_dynatune` (how aggressively to use host resources) and `edb_dynatune_profile` (declares workload intent—for example, OLTP vs. data warehouse). You set them in `postgresql.conf`, start the server, and then iterate using your own operational evidence. The point is not a black box; it's a defensible baseline you can explain to reviewers and adjust over time.

Why this is a best practice: Baselines should be explainable and repeatable. Encoding workload intent as a first-class knob avoids copy-pasted configs and micro-benchmark "voodoo," and it keeps your starting posture portable as code.

How it compares: You can reach good baselines on community PostgreSQL or Aurora, but upstream doesn't treat "workload profile" as a native setting, and cloud consoles don't encode that intent inside the engine's own configuration. EPAS's approach keeps the baseline reviewable and portable as database configuration.

SQL Profiler + Query Advisor: Turn observations into targeted, low-risk changes

Observability only matters if it leads somewhere. EPAS provides a **Profiler → Advisor** workflow entirely in the database toolkit. SQL Profiler captures focused traces—short windows filtered to specific users or modules—so you can grab representative workloads without turning on heavyweight global tracing. Index Advisor then analyzes those captured statements by creating hypothetical indexes and having the planner cost the workload as if those indexes existed, logging concrete recommendations and exposing them via views. This shortens the loop from symptom to fix: Observe a wait pattern, capture a tight trace, apply one or two indexes that actually shift p95/p99.

Why this is a best practice: A reliable tuning loop is **observe → hypothesize → implement**. Because Profiler and Advisor are part of EPAS, the loop is consistent and repeatable across teams and environments.

How it compares: In community/Aurora, you can assemble similar pieces—e.g., the **HypoPG** extension for hypothetical indexes is supported on RDS/Aurora—but you still need to wire up workload capture and normalize outputs. EPAS ships an integrated, supported **trace → advise** path so you spend time changing the system, not integrating tools.

Why this matters to enterprises

What distinguishes a mature program from a heroic one is a portable operating model: the same QoS (quality of service) controls (resource groups), the same evidence trail (wait-state time series), and the same tuning loop (Profiler → Advisor) in dev, staging, and prod—and after a platform move. EPAS encodes these as engine-level features and parameters (`edb_resource_group, edb_max_resource_groups, edb_wait_states.*, edb_dynatune*`), so governance and evidence move with the database rather than living in a vendor console.

Outcome now: Predictable mixed-workload behavior and a durable record of changes make incidents explainable and tunings defensible.

Portability next: Because the limits and the baseline live in the engine, the same posture moves from dev to prod and from on-prem to cloud without retooling.

Enhanced SQL capabilities: Oracle semantics to reduce migration risk

For teams exiting Oracle, the hard part isn't moving data—it's matching behavior. Seemingly small semantic gaps (how DATE is stored, how NULL behaves in string ops, whether an error rolls back prior statements) can turn into late-stage defects. EPAS treats this as an engine problem, not a tooling problem: You can enable Oracle-compatible behavior where it matters, connect to the source over an OCI database link to phase your cutover, and keep developers on an Oracle-familiar surface while you stabilize. Because these controls are just database settings, you keep the option to dial them back as you converge on pure PostgreSQL semantics later—so you buy down risk now without sacrificing portability.

Match semantics at the database boundary—then taper as you harden

EPAS exposes compatibility switches that let you align behavior with Oracle for the duration of the migration. The intent isn't to "pretend to be Oracle" forever; it's to move the semantic reconciliation into one auditable, reversible place: the database.

A common example is date handling. In PostgreSQL, a `DATE` column stores only a date. With EPAS, when `edb_redwood_date` is set at the time you create or alter a table, a `DATE` column is defined with Oracle-like behavior (the time component is preserved). This setting is evaluated during table creation or redefinition; changing it later doesn't retroactively affect existing columns. In other contexts within Redwood-compatible EPAS (for example, SPL variables and parameters), Oracle-style `DATE` behavior applies regardless.

Another example is error behavior. In native PostgreSQL, raising an exception inside a transaction rolls back prior uncommitted statements. If your PL/SQL code expects statement-level semantics, EPAS's `edb_stmt_level_tx` can emulate the Oracle behavior: An exception *doesn't* automatically unwind earlier updates.

Why this is best practice: You centralize the "translation layer" in the engine—auditable, source controlled, and portable—rather than scattering work-arounds through applications. As tests prove a switch is no longer needed, you turn it off and keep going. That's how you reduce migration risk without creating long-term technical debt.

How it compares: Community PostgreSQL intentionally preserves PostgreSQL semantics; there's no native "Oracle mode." Aurora PostgreSQL likewise doesn't ship engine-level Oracle semantics; AWS's Oracle → Aurora guidance centers on adapting application code/types, not flipping a runtime Oracle compatibility switch. EPAS lets you choose Oracle-compatible behavior now and converge later without spraying work-arounds through apps.

Phase the cutover and prove parity with an OCI database link

Big-bang weekends are avoidable. EPAS provides `dblink_ora`, an OCI-based database link that lets you `SELECT`, `INSERT`, `UPDATE`, and `DELETE` against Oracle directly from EPAS. Teams use it to shadow traffic, reconcile deltas, and run parity tests during staged cutovers. The docs walk through the call patterns (`dblink_ora_connect` plus helpers), so scripting dual-write windows and back-to-back comparisons is straightforward.

Why this is best practice: You measure real behavior before you flip the switch. By running production-like paths through both systems, you catch semantic mismatches under load—when you still have an exit—rather than discovering them in week three of production.

How it compares: There's no native equivalent to EPAS's `dblink_ora` in community PostgreSQL or Aurora PostgreSQL. AWS migration playbooks focus on schema/code conversion and data movement to Aurora rather than issuing Oracle CRUD from within Aurora/PostgreSQL. EPAS's engine-level link is the enabler for phased migrations.

Keep the surface familiar so people can ship while you stabilize

Semantics aren't the only friction point. Human workflows matter, too. EPAS provides Oracle-compatible catalog views, keywords, functions, and PL/SQL-compatible behavior, so scripts and habits largely carry over. Utilities such as **EDB*Plus** (an SQL** *Plus-style CLI*) and ** *EDB* Loader** (an Oracle SQL*Loader-style bulk loader) preserve muscle memory for developers and DBAs, which speeds validation and reduces the volume of rewrites during the critical early phase.

Why this is best practice: Migration risk is as much people and process as code. A familiar surface lowers the cognitive load so teams can focus on testing and parity, not on relearning basic tooling.

How it compares: Community PostgreSQL is intentionally PostgreSQL-first; Oracle-style views and tools aren't part of core. Aurora provides managed Postgres on AWS but does not add an Oracle runtime layer or SQL*Plus*/*SQL*Loader-compatible utilities in the engine. EPAS's approach keeps these in-engine and portable, independent of a specific cloud console.

How this compares to community Postgres and Aurora PostgreSQL

Neither community PostgreSQL nor Amazon Aurora PostgreSQL-compatible ships an engine-level Oracle runtime like EPAS. Community PostgreSQL, by design, preserves PostgreSQL semantics; Oracle compatibility isn't part of core. Aurora provides strong managed-service capabilities on AWS, but its Oracle migration guidance focuses on adapting applications and types, not enabling a runtime Oracle mode in the engine. In practice, that means more rewrite work up front or more logic at the application edge. EPAS's approach lets you choose Oracle-compatible behavior at the database boundary now, phase the move with `dblink_ora`, and converge later—preserving optionality.

Why this matters for enterprises

Because these are engine-level configuration choices, not cloud add-ons, you control migration risk and future optionality at the database boundary. You can enable Oracle-like behavior per session or per database while you migrate, track those settings in source control, and turn them off once tests prove you're safe—without touching application code. The same pattern works on-prem, in any cloud, and in Kubernetes, so your approach to compatibility, validation, and cutover is portable, auditable, and repeatable. In practice, that means fewer rewrites, faster parity checks, cleaner change reviews, and no architectural lock-in—exactly the properties large organizations need when they're moving critical systems under tight timelines and strict compliance.

Outcome now: You cut migration risk by matching semantics at the database boundary and proving parity before cutover.

Portability next: And because those choices are GUCs, you can taper them off later—identically—in every environment.

Database and application security

Security controls are most dependable when they live where the data lives. EPAS's security pillar is unapologetically engine-centric: encryption of files/WAL at the database layer, dynamic masking at query time, a database-side safety net for SQL-injection classes, and centralized password hygiene—with audit-log redaction to close easy gaps. Because these are database features and configuration parameters, the policies travel with the cluster across on-prem, cloud, and Kubernetes, rather than being tied to a single provider console. That's the through line from the Introduction: Keep security with the data so your posture remains portable and auditable.

Transparent Data Encryption (TDE): Encrypt at the engine, keep apps unchanged

In EPAS, TDE encrypts table data and catalog content on disk and in WAL files, and it does so transparently to applications—no code changes, no driver twiddling. That's an engine-level guarantee about what is written to storage. Moreover, it provides separation of duties with encryption control handled by the database administrators rather than system administrators. Database level encryption ensures that the data is encrypted even when there is a change in custody, such as in the case of backups.

Why this is a best practice: Placing encryption inside the database keeps the control close to the asset being protected, makes the policy part of database configuration (reviewable, testable, repeatable), and ensures that the protection moves with the database wherever it runs. It also aligns with compliance reviews that expect a clear system-of-record statement about how data files and redo/WAL are handled.

How it compares:

- **Community PostgreSQL:** The core project documents encryption mechanisms but has no native TDE in upstream PostgreSQL; the status is tracked on the community wiki, and it is unlikely to be implemented anytime soon.
- **Aurora PostgreSQL-compatible:** Encryption is excellent at the cluster storage plane using AWS KMS (AES-256). That secures storage within AWS, but it's not PostgreSQL-engine TDE and is, by design, tied to AWS KMS.

Portability note: If you run on Kubernetes, EPAS exposes TDE as a first-class manifest setting (`epas.tde.enabled: true` with a referenced secret), so your encryption posture is literally kept as code and follows your cluster definitions.

Data redaction: Policy-driven masking at query time

EPAS can dynamically redact sensitive fields at query time, so non-privileged users see only sanctioned representations (for example, last-4 of an SSN), and privileged users see full values. The policy is enforced in the database, not in each application.

Why this is a best practice: Masking implemented once, centrally, is easier to audit and less brittle than per-app views or middleware rewrites. It guarantees one source of truth for who sees what, and it stays with the data wherever the database runs. Learn More

How it compares: Upstream Postgres and Aurora customers often build masking at the app/reporting layer or via ad hoc views. EPAS bakes the rule into the engine, reducing drift across services and eliminating the "forgot to mask in this microservice" class of findings. (Neither community PostgresQL nor Aurora ship EPAS's data redaction feature in the database engine.) Learn More

SQL/Protect: A database-side safety net for SQL-injection patterns

EPAS includes SQL/Protect, which can run in **learn** → **passive** → **active** modes and records statistics about potential SQL-injection attempts in the `edb_sql_protect_stats` view, giving you a staged path from observation to enforcement. Policies are role aware and auditable in SQL.

Why this is a best practice: Adding a defense-in-depth layer inside the database catches certain classes of injection even if a web tier or API slips. The learn/passive/active rollout lets you baseline real traffic before you turn the screws, minimizing false positives on day one. The telemetry (`edb_sql_protect_stats`) provides durable evidence for security reviews.

How it compares: Community PostgreSQL and Aurora rely on app-side patterns, WAFs, and parameterized SQL (all still recommended), but they don't ship an engine-level SQL-injection guard with staged rollout and stats in the database. EPAS gives you that extra, portable layer at the data boundary itself.

Password hygiene and audit hygiene: Profiles and log redaction where they belong

EPAS Password Profiles let you centralize password policy—reuse windows, complexity, lockout—so a single profile can govern multiple roles. It's implemented in SQL (`CREATE PROFILE … LIMIT PASSWORD_REUSE_TIME …`), which keeps the rule visible and reviewable. Learn More

Paired with that, a single GUC, `edb_filter_log.redact_password_commands`, ensures that credentials aren't written to logs during `CREATE/ALTER ROLE` flows (the docs spell out the recognized syntaxes), closing a common audit gap without fragile log scrubbing.

Why this is a best practice: Password rules and redaction live with the database rather than scattered across applications or SIEM filters. That reduces drift, simplifies audits, and keeps sensitive content out of log archives from the start.

How it compares: Community/Aurora shops can approximate pieces with conventions and external tooling, but engine-level **profiles + log redaction** aren't provided as a unified, database-native feature set. EPAS's approach makes the policy part of the schema/config you ship and review.

Privilege Analysis

EDB provides a unique capability for Privilege Analysis, enabling organizations to strengthen database security through the principle of least privilege. This is achieved using privilege captures.

Privilege Analysis dynamically records the privileges that database users actually use over a defined period of time. This feature helps identify security vulnerabilities by distinguishing between used and unused privileges, making it easier to enforce the least privilege model. By so doing, you can safely remove excessive or unnecessary privileges from users, reducing the risk of accidental or intentional data access and modification.

This feature also supports preproduction testing, allowing you to simulate full application workloads, analyze privilege usage, and implement privilege lockdowns before deploying to production. The resulting reports can be shared with auditors to demonstrate database privilege usage and compliance across roles.

A privilege capture policy defines how privileges are captured. Once enabled, you can query system views to review the captured privileges.

For detailed syntax and configuration options, refer to the CAPTURE PRIVILEGES POLICY section of the EPAS documentation. EPAS supports Privilege Capture through both:

- SQL commands – for users familiar with PostgreSQL syntax
- DBMS_PRIVILEGE_CAPTURE – for users preferring Oracle-compatible syntax

```
SELECT * FROM dba_used_privs;

policy_name | run_name  | object_class | object_name    | column_name | application | role_name | privilege_type
------------ | --------  | ------------- | --------------- | ------------ | ------------ | ---------- | ---------------
q3audit     | q3_review | table        | banking.loans  | -           | edb-psql    | teller1   | SELECT

-- myuser actually used the SELECT privielge on banking.loans during the capture

SELECT * FROM dba_unused_privs where role_name = 'myuser';

policy_name | run_name  | object_class | object_name    | column_name | application | role_name | privilege_type
------------ | --------  | ------------- | --------------- | ------------ | ------------ | ---------- | ---------------
q3audit     | q3_review | table        | banking.loans  | -           | edb-psql    | teller1   | INSERT
q3audit     | q3_review | table        | banking.loans  | -           | edb-psql    | teller1   | UPDATE

-- myuser did not use the INSERT or DELETE privielges on banking.loans during the capture
```

Why this is a best practice: Under the least privilege model, users are granted only the permissions required to perform their tasks. In many cases, users are given broader access than necessary, leading to potential security risks. Without Privilege Analysis, determining the minimal required privileges for each user can be complex and error prone.

How it compares: PostgreSQL and AWS Aurora Postgres lack this capability. They only allow static analysis of granted privileges by querying the catalog tables, without the ability to determine whether these privileges are actually used.

Virtual private database (advanced row-level security)

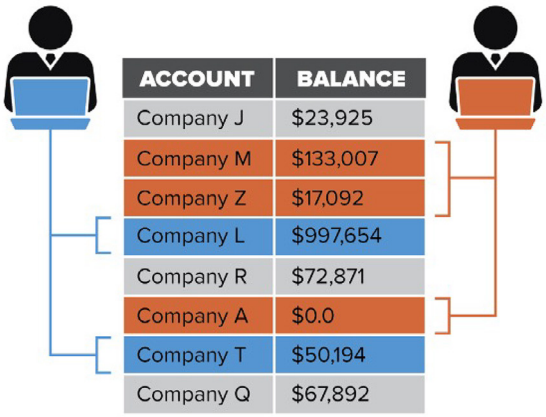
EPAS extends PostgreSQL's native row-level security (RLS) with an advanced, Oracle-compatible implementation known as the Virtual Private Database (VPD). This capability, provided through the DBMS_RLS package, enables fine-grained access control by dynamically restricting which rows and even which columns a user can view or modify. Each user sees only the data they are authorized to access, ensuring data isolation and compliance at the database level.

VPD is particularly valuable in multi-tenant and security-sensitive environments, where multiple users or organizations share a database schema. Administrators can define row- and column-level policies that enforce access rules transparently, without requiring application code changes.

Why this is a best practice: With these enhancements, EDB Virtual Private Database provide a more flexible, manageable, and performant approach to data-level security than standard PostgreSQL RLS, making it ideal for enterprise and regulated environments.

How it compares: While PostgreSQL and Aurora provide standard RLS functionality, EPAS enhances it with several enterprise-grade improvements:

- **Column-sensitive policies:** EPAS allows policies that filter access not just by rows but also by specific columns within a table—something not supported in PostgreSQL.
- **Simplified policy management:** A single policy function can be defined once and applied to multiple tables, reducing redundancy. Policies can also be temporarily disabled without being deleted, making testing and policy lifecycle management easier.
- **Flexible policy scope:** Policies can target individual operations (`INSERT`, `UPDATE`, `DELETE`, `SELECT`) or all operations together. Multiple policies on a table are logically OR'ed, allowing access if any policy condition is satisfied.
- **Optimizer-aware performance:** EPAS includes dictionary views that describe policies for better transparency and diagnostics. The query optimizer is aware of VPD constraints, allowing it to use indexes and improve execution efficiency.
- **Performance and visibility trade-offs:** Because EPAS makes policy conditions visible to the optimizer, constraints may be evaluated more than once per qualifying row—but overall query performance is typically faster for indexed or straightforward policies. PostgreSQL, by contrast, hides RLS constraints from the optimizer, which can simplify complex policy evaluations but limit optimization opportunities.
- **Oracle-compatible implementation:** The DBMS_RLS package follows the familiar Oracle VPD model, simplifying migration for organizations transitioning from Oracle to EPAS.



Why this matters for enterprises

These controls—engine-level TDE, data redaction, SQL/Protect, Privilege Analysis, and password profiles with audit-log redaction—put security at the database boundary, where it's easiest to audit and safest to keep consistent. Because they are database features and parameters, you can version them, promote them across environments, and carry them from on-prem to any cloud to Kubernetes without re-architecting. In practice, that means fewer moving parts to satisfy compliance, less duplication of masking and policy logic in apps, and no lock-in to a single cloud's storage plane—exactly the outcomes large organizations want when they secure critical systems at scale.

Outcome now: Encryption, masking, injection guardrails, and password hygiene are enforced where the data lives—easier audits, fewer gaps.

Portability next: As database features, these controls travel with the cluster, not with any single provider console.

Performance monitoring and analysis: Observe, hypothesize, fix

Performance isn't a one-off "tune and forget." It's a loop: Observe real behavior, form a hypothesis, change one thing, then verify. In EPAS, the loop is implemented in the engine, so it works the same on-prem, in any cloud, and in Kubernetes. This section assumes the Wait States sampler from the database and application security section above; here we focus on how to use that baseline with SQL Profiler and Query Advisor to make safe, targeted improvements.

This section assumes the posture is in place from the Database Administration section above: Resource Manager configured, Wait States extension installed. Here we use that posture to run the performance loop **observe** → **hypothesize** → **fix** → **verify**, with three engine-native pieces: your standing wait-event baseline (EDB Wait States); short, filtered traces when needed (SQL Profiler); and concrete recommendations to act on (Query Advisor).

Use your Wait States baseline to aim the investigation

Treat Wait States as the always-on, low-impact meter you consult first. Trend the top wait classes (locks vs I/O vs CPU), bracket changes by comparing 24–48h "before" vs. "after" a release or parameter shift, and localize spikes by correlating a wait surge with the statements and roles active at that minute. In addition, session sampling apportions total database time, so you can see what share of overall database time each query consumed—and how that time breaks down by wait type—making it trivial to ask "top 10 by total runtime," "top 10 by I/O wait," or "which queries grew CPU share after the change." Under the hood, Wait States probes each session at a configurable interval (default 1s) and retains samples for a configurable window (default 7 days) via the `edb_wait_states.sampling_interval` and `edb_wait_states.retention_period` settings, persisting results in database-managed logs/tables you can query and export.

Why this is best practice: You're making decisions from longitudinal, query-aware evidence that survives restarts and lines up with code/config changes—and because it's configured in the engine, the baseline is versionable and portable across environments.

How it compares: Community Postgres and Aurora often pair system views with external telemetry. Aurora's Performance Insights is excellent for visualizing database load and filtering by waits, SQL, users—but it lives in the AWS control plane, not as engine config you ship with the database. EPAS keeps the baseline in-engine, so the evidence model follows your cluster on-prem, in any cloud, or on Kubernetes.

When the baseline flags a suspect window, use SQL Profiler (short, filtered traces)

Once your Wait States trend isolates a time window and actors (e.g., a monthly report driving buffer-content waits), use SQL Profiler to capture a short, filtered trace—minutes, not hours—scoped to the database, users, or modules involved. Profiler supports on-demand or scheduled traces, filters, and a trace output analyzer so you zero in on the statements that actually matter.

Why this is best practice: Tracing is powerful but intrusive; being surgical avoids observer overhead while producing a precise workload slice your team can review and reproduce in lower environments.

How it compares: Upstream/Aurora workflows can combine logs, `pg_stat_*`, and console captures; EPAS provides a database-supplied profiler with consistent behavior everywhere—no dependency on a specific cloud UI.

Query Advisor: Sample workload → plan improvements (close the loop)

Query Advisor samples the live workload, evaluates statements by creating hypothetical indexes, replans queries as if those indexes existed, and emits quantified recommendations—including exact `CREATE INDEX` statements—that can be queried via the documented `index_recommendations` view/logs. Apply one or two changes, then verify in your Wait States baseline that the targeted wait class recedes and that you didn't create a new hotspot (for example, a shift from CPU to I/O wait or increased row locks).

Include Statistics Advisor in the loop: Statistics Advisor improves plan quality by fixing selectivity estimates before you add indexes. It monitors queries with multicolumn filters, detects estimation errors that suggest column

dependency, and proposes extended statistics on pairs of columns. Candidates are weighted by how many queries benefit and their execution cost, helping you apply the highest-impact changes first. Generate recommendations with `query_advisor_statistics_recommendations(...)`, create the suggested `CREATE STATISTICS` objects, run `ANALYZE`, and recheck plans before considering new indexes.

Why this is best practice: It compresses the loop to **observe** → **hypothesize** → **fix** → **verify** with a defensible, minimal change set, avoiding shotgun indexing. (Docs note limits and review guidance, keeping changes honest.)

How it compares: Community Postgres and Aurora support HypoPG (hypothetical indexes), but teams typically assemble the workflow themselves—capture workload in views/logs or Performance Insights, then analyze in the extension. EPAS ships an integrated experience: Profiler for trace-level visibility and history, and Query Advisor for live workload sampling with concrete recommendations (including Statistics Advisor for extended stats). The result is the same **observe** → **hypothesize** → **fix** → **verify** loop, without stitching tools together. Aurora explicitly lists HypoPG among supported extensions.

Why this matters for enterprises

This is the operator model mentioned earlier in the Introduction section—observe, hypothesize, fix, verify—made portable. By keeping the baseline (Wait States), the deep-dive tool (SQL Profiler), and the change engine (Query Advisor) in the database, EPAS lets you version, audit, and move the entire performance workflow across on-prem, any cloud, and Kubernetes without retooling. In practice: fewer moving parts; faster postmortems; and safer, targeted performance wins.

Outcome now: Teams run a tight loop—**observe** → **hypothesize** → **fix** → **verify**—anchored in evidence, not hunches.

Portability next: Because the loop is in-engine (baseline, profiler, advisor), it runs the same anywhere you deploy.

Application development utilities: Familiarity lowers migration risk

Exiting Oracle isn't just about data and DDL—it's about people and workflows. EPAS ships developer-facing tools that feel familiar on day one and stay useful long after cutover. Two stand out: **EDB*Plus**, an SQL*Plus-compatible CLI for running SQL and PL/SQL-style blocks, and EDB*Loader, a high-throughput bulk loader with an Oracle SQL*Loader-style interface. Because both are engine-aligned and cross-platform, they reduce early friction and keep your scripting muscle memory portable across on-prem, cloud, and Kubernetes.

EDB*Plus: Keep your CLI habits while you stabilize

Teams coming from Oracle often have a decade of scripts, habits, and runbooks built around SQL*Plus. EDB*Plus provides a command-line interface to EPAS that accepts SQL, SPL (EPAS's PL/SQL-compatible blocks), and SQL*Plus-style commands for querying objects, executing stored procedures, formatting output, running batch scripts, calling OS commands, and recording output. In other words, most of what operators and developers expect to "just work" in an SQL*Plus session does, with EPAS.

Why this is best practice: Migration risk is as much human as technical. A familiar CLI shortens the learning curve, lets you validate parity quickly, and avoids rewriting operational scripts during the most fragile phase of the move. Because EDB*Plus is part of the EPAS distribution, your CLI surface is consistent across environments and versionable alongside database changes.

How it compares: Community PostgreSQL centers on `psql` (excellent, but *not* SQL*Plus-compatible); AWS guidance typically maps SQL*Plus commands to `psql` rather than offering a drop-in SQL*Plus-style client for Aurora. EPAS gives you an SQL*Plus-compatible CLI so teams don't burn cycles relearning syntax in the middle of a migration.

EDB*Loader: Bulk load with an Oracle-style interface

Bulk load is often where schedules bend. EDB*Loader is a high-performance bulk data loader for EPAS that exposes a subset of Oracle SQL*Loader's parameters and command-line shape. You invoke `edbldr` to push files into one or more tables, with control files and options that look familiar to Oracle operators. The docs spell out interface compatibility, key options, and operational cautions (match client/server versions, etc.).

Why this is best practice: When you can reuse control files and routines, you cut the riskiest kind of work—ad-hoc rewrites under deadline. Because EDB*Loader is purpose-built for EPAS, you also keep bulk ingestion close to the engine, which makes runs easier to standardize and audit across environments.

How it compares: In Community Postgres and Aurora, bulk load normally means `COPY`/`\copy` via `psql` (fast, but not SQL*Loader-style) or custom ETL. AWS's own docs point customers to `\copy` on RDS/Aurora; there's no native SQL*Loader-compatible utility in those stacks. EPAS's EDB*Loader gives Oracle-familiar semantics and flags, reducing translation effort.

Why this matters for enterprises

Big programs succeed when they trade novelty for repeatability. EDB*Plus and EDB*Loader let teams reuse what they already know—scripts, control files, and daily muscle memory—so they can focus on parity, testing, and incremental convergence to native PostgreSQL rather than tool rewrites. Because these utilities are part of EPAS and documented as Oracle-compatibility tools, they're portable (on-prem, any cloud, Kubernetes) and auditable, like the rest of your database stack. That combination—familiarity now, optionality later—is what lowers migration risk without locking you in.

Outcome now: People ship sooner by reusing familiar CLIs and loaders; less time is lost rewriting tools.

Portability next: The same utilities and scripts work on-prem, in cloud, and in Kubernetes with no extra adapters.

Advanced replication: Design with portable guardrails

Replication isn't just about moving changes; it's about designing, rehearsing, and switching over with enough control that the plan works the same way in every environment. EPAS uses PostgreSQL's replication primitives (physical and logical) as the foundation, then surrounds them with engine-level governance (Resource Manager) and longitudinal evidence (EDB Wait States) so you can cap rehearsal load, observe exactly what happened, and repeat it on-prem, in any cloud, or in Kubernetes. For topologies beyond vanilla upstream, EDB provides EDB Postgres Replication Server (EPRS)—unidirectional or bidirectional, and even non-Postgres sources. In other words: standard PostgreSQL where possible, portable guardrails where enterprises need them.

Start with PostgreSQL fundamentals, then add EPAS guardrails

PostgreSQL supports physical (byte-for-byte) and logical (row-level, publish/subscribe) replication; both often coexist in real programs (high availability on physical replicas; selective sharing with logical). That gives you compatibility and choice. EPAS doesn't replace this; it operationalizes it. Use Resource Manager to fence rehearsal load (assign replication tests to a capped group), and keep Wait States sampling so each dry run produces a query-aware timeline of waits (locks, I/O, CPU, IPC) that you can compare before/after. The result is not just "We replicated" but "We replicated under control, with evidence."

Why this is best practice: Replication plans tend to fail on operational details—noisy neighbors, unbounded apply workers, or mysteries during cutover. Shaping rehearsals at the engine boundary and capturing a time series of what happened makes failures diagnosable and successes repeatable across environments.

How it compares: Aurora PostgreSQL implements PostgreSQL logical replication and provides rich dashboards in the AWS console (Performance Insights by waits), but those controls and visuals live in the cloud plane. EPAS's shaping knobs and evidence are database configuration, so they travel with your cluster—useful when your target isn't always AWS.

Pick the topology you need, and keep it portable

Choosing the right replication topology is less about brand names and more about preserving portability, resiliency, and operational simplicity as your estate spans regions, clouds, and Kubernetes. For Postgres-to-Postgres movement, EDB Postgres Distributed (PGD) supplies write-anywhere replication with conflict handling, global sequences, DDL propagation, and commit scopes that let you tune confirmation—from local acks to multi-node quorums—for the latency and durability profile you need. Its built-in connection routing and pooling reduce tool sprawl and the number of moving parts you must configure and monitor.

Where heterogeneous systems must feed Postgres (for example, Oracle or SQL Server), EDB Postgres Replication Server captures and replicates changes into Postgres, after which PGD carries them across your Postgres footprint. The result is a consistent, cloud-smart replication playbook that keeps topology a configuration choice rather than a provider constraint.

Why this is best practice: Standardizing on a replication stack that isn't tied to a single cloud keeps cutovers, DR tests, and day-to-day sharing patterns consistent—whether the next site is another region, another cloud, or on-prem. With PGD for Postgres-to-Postgres and EPRS for heterogeneous-to-Postgres, replication becomes part of your architecture rather than an account-level feature.

How it compares: Aurora offers strong in-cloud options (Aurora Replicas, cross-Region replication, DMS patterns), but they're designed inside AWS and administered through AWS constructs. If your estate spans more than AWS, combining PGD and EDB Postgres Replication Server with engine-native Postgres controls lets you keep one playbook for active-active, failover, region expansion, and heterogeneous ingestion—without provider switch costs when topology changes.

Rehearse switchovers like releases—then execute with evidence

Treat failovers and switchovers like application releases: Throttle rehearsal load with Resource Manager so OLTP stays healthy during tests, sample the system with Wait States to capture the "at 10:37 during switchover" picture, and keep the results with your change ticket. Repeat until the timeline is boring. On the day, you have a portable checklist and a baseline that tells you whether the new primary is behaving as expected.

Why this is best practice: Enterprises don't just need replication—they need predictable replication. Guardrails + evidence turns "We think it's fine" into "We can show it's fine," which shortens audits and speeds up recovery runbooks.

How it compares: Aurora encourages similar rigor, but the visibility and throttles are primarily console-centric (Performance Insights, cluster settings). EPAS keeps the controls and proofs in the database itself, so your method doesn't depend on a particular vendor UI.

Why this matters for enterprises

With EPAS, "advanced replication" isn't a new kind of WAL—it's a portable operating pattern around standard PostgreSQL replication: Shape rehearsal load in the engine (Resource Manager), record what happens as a time series (Wait States), and choose a topology that isn't bound to a single cloud (EPRS when needed). That combination makes switchovers repeatable across on-prem, cloud, and Kubernetes and keeps your guardrails and evidence with the database, not in a vendor console—exactly the qualities large programs need when they move critical systems under change control.

Outcome now: Switchovers are rehearsed, throttled, and evidenced—so they're predictable on the day.

Portability next: Guardrails and telemetry are database configuration, so the method survives environment changes.

EPAS-only configuration deep dive

Why these specific GUCs beat DIY in community PostgreSQL and cloud-plane knobs in Aurora.

EPAS exposes a focused set of engine parameters (GUCs) that you can enable precisely where they reduce risk, keep them under source control, and later turn them off as you converge to native PostgreSQL. EPAS labels these settings explicitly as (EPAS only) in the parameter summary, which speeds design and security reviews.

Oracle-compatibility switches

The safest way to exit Oracle is to align semantics at the database boundary—once, reversibly—rather than to scatter shims through every app. EPAS does that with session/database-scoped toggles:

- **`edb_redwood_date`** — Treat a `DATE` column (on create/alter) as a timestamp (Oracle-style) instead of PostgreSQL's date-only type; prevents "missing time" defects in ported code.
- **`edb_redwood_strings`** — On string concatenation with `NULL`, return the non-NULL string (Oracle behavior) instead of `NULL` (PostgreSQL). No more "vanishing" suffixes.
- **`edb_redwood_raw_names`** — Show object names in Oracle-style catalogs exactly as stored in PostgreSQL catalogs (lowercase unless quoted), avoiding case surprises in tools.
- **`edb_redwood_greatest_least`** — Choose Oracle-compatible `NULL` handling for `GREATEST/LEAST` (toggleable per session). These functions often sit in rules and reporting logic; matching semantics avoids subtle decision errors.
- **`edb_stmt_level_tx`** — Make exceptions not roll back prior uncommitted work in the block (Oracle-like) when `TRUE`; keep default PostgreSQL rollback when `FALSE`. Use sparingly and deliberately, but it's invaluable for PL/SQL-style code paths.

Why this is better practice: You adopt only the semantics an application needs, keep the deltas reversible, and document the choice in one place (the database config). Community/Aurora don't ship runtime Oracle-mode toggles; EPAS flags these as EPAS-only in the summary so reviewers can see what's unique and how it scopes (session/database/cluster).

Rollout pattern: Define a per-app compatibility profile (commonly: `edb_redwood_date, edb_redwood_strings`, sometimes `edb_stmt_level_tx`, plus `edb_redwood_greatest_least` when needed). Enable only those tests that prove necessary; validate parity against Oracle (for example, with your existing checks) before cutover. Because they're GUCs, these choices travel with the app across environments.

Engine-side injection guardrails (SQL/Protect)

What it is: SQL/Protect learns "normal" query shapes for protected roles, then enforces policies in stages—learn → passive → active—controlled by the GUC `edb_sql_protect.level`. It records rich attempt statistics that you can watch in `edb_sql_protect_stats`.

Why this is better practice: A defense-in-depth control at the data boundary reduces dependence on uniform app behavior or a single proxy/WAF. The staged rollout limits false positives, and because policy + telemetry are in-engine, they are auditable and portable across on-prem, cloud, and Kubernetes. Community/Aurora don't provide an engine-level SQL-injection guard; you'd assemble proxies and conventions (useful but fragmented).

Operational tip: Start with a small set of protected roles, run in passive while monitoring `edb_sql_protect_stats`, then switch to active once noise is low.

Dynamic masking and audit hygiene

- **Data redaction:** Create column-level masking policies with `CREATE REDACTION POLICY`; control enforcement with the `edb_data_redaction` GUC. Masking happens at query time in the engine, so non-privileged users see only sanctioned views (e.g., last-4), and privileged users see full values.

- **Password redaction in logs:** Enable via `shared_preload_libraries += '$libdir/edb_filter_log'`, then set `edb_filter_log.redact_password_commands = true`. The module recognizes specific `CREATE/ALTER ROLE|USER` syntaxes and removes credentials from the log—no brittle parsers.

Why this is better practice: Masking and log hygiene as database features are easier to test, audit, and move with the database. In Community/Aurora, you can approximate masking with views/RLS and rely on process for log scrubs; EPAS turns both into explicit, testable settings.

Operational tip: Define policies for PII/PCI tables in non-prod, test privileged bypass, then enable broadly; keep password-log redaction on by default everywhere except disposable dev.

Workload governance and observability

- **Resource Manager:** Bind sessions to named resource groups (`SET edb_resource_group TO <group>`), and limit simultaneous groups cluster-wide with `edb_max_resource_groups` (EPAS-only). Group definitions live in a shared system catalog, so every database in the instance can use them.
- **EDB Wait States:** Keep a low-impact wait-event time series (defaults: 1s sampling; 7-day retention) tunable via `edb_wait_states.sampling_interval` and `edb_wait_states.retention_period`, persisted in engine tables/logs.
- **Dynatune:** Establish a defensible baseline with `edb_dynatune` (0–100, how aggressively to use host resources) and `edb_dynatune_profile (oltp|reporting|mixed)`, then refine with your Wait States + Profiler/Advisor evidence.

Why this is better practice: These controls are database-native and portable. Aurora offers strong in-console knobs/metrics, but posture there lives in the AWS plane; Community Postgres requires assembling extensions and conventions. EPAS standardizes governance/observability in the engine, so the same runbook works everywhere.

Operational tip: Keep Wait States on at a modest interval with 7–14 days retention; treat Dynatune as a starting line, not a finish line; encode resource-group bindings into role defaults for predictability.

Why EPAS-only GUCs matter

For migrations and day-two operations, configuration as architecture beats ad-hoc rewrites or cloud-specific toggles. With EPAS's EPAS-only GUCs you can (1) choose semantics where they reduce risk, (2) enforce security where the data lives, and (3) govern/observe workloads with settings that move with the database—on-prem, any cloud, Kubernetes. That's the practical meaning of portability in enterprise databases.

Implementation blueprint

A practical blueprint should reduce risk now, keep choices open later, and work everywhere you run. The steps below are written as actions (what to do), followed by outcomes (why it helps today) and a portability note (why the win persists across environments).

Migration track (Oracle → EPAS): Adopt only what you need, prove parity, taper later

Create a per-app compatibility profile

For each application, enable only the Oracle behaviors your tests prove it relies on—typically `edb_redwood_date` (Oracle-style DATE with time component when used in table DDL) and `edb_redwood_strings` (Oracle concatenation with NULL), plus `edb_stmt_level_tx` for specific PL/SQL-style blocks. Add `edb_redwood_greatest_least` where business logic uses those functions. Scope these as session or database GUCs so you can A/B safely.

Outcome now: You eliminate a large class of late-stage semantic bugs without forking application code.

Portable because: The profile is database configuration you can version and promote identically on-prem, in any cloud, and under Kubernetes.

Phase the cutover with dblink_ora

Stand up an OCI database link from EPAS and run shadow traffic (`SELECT/INSERT/UPDATE/DELETE`) against Oracle from EPAS during a rehearsal window. Compare results and side effects under real load before flipping write paths. Keep the link available for early-life parity checks after cutover.

Outcome now: You measure real behavior and catch mismatches while you still have an exit.

Portable because: The link is an engine feature—no reliance on a particular cloud console or middleware.

Converge to native PostgreSQL over time

As tests show a switch is unnecessary, remove it from the profile; treat each removal as a mini-release with before/after checks.

Outcome now: You retire temporary compatibility with confidence and avoid long-term semantic debt.

Portable because: The same GUC changes and validation scripts apply in every environment.

Security track: Keep protections with the data, not just the cloud plane

Turn on TDE at creation time

Encrypt data files and WAL in the engine so applications remain unchanged. In Kubernetes, declare it in the manifest (operator `epas.tde` stanza) so posture is kept as code.

Outcome now: At-rest requirements are satisfied without driver refits or app changes; audits have a single, authoritative control point.

Portable because: TDE is database-side and travels with backups, replicas, and manifests.

Centralize masking with data redaction

Define column-level policies (`CREATE REDACTION POLICY`) for PII/PCI tables; enable via `edb_data_redaction`. Test privileged bypass explicitly.

Outcome now: One enforcement point eliminates per-app view drift and “forgot to mask in this service” incidents.

Portable because: Policies live in SQL and migrate with the schema.

Add engine-side injection guardrails (SQL/Protect)

Roll out in **learn** → **passive** → **active** using `edb_sql_protect`.level, and watch `edb_sql_protect_stats` until noise is low. Protect the fewest roles that meaningfully reduce risk (e.g., web/API roles).

Outcome now: Defense-in-depth at the data boundary catches classes of injection even if a tier slips.

Portable because: Policy and telemetry are engine-native, not tied to a proxy or WAF SKU.

Enforce password hygiene and redact credentials in logs

Use Password Profiles for reuse/complexity/lockout and enable `edb_filter_log.redact_password_commands` (with the preload library) so `CREATE/ALTER ROLE|USER` never leaks secrets.

Outcome now: You close easy audit gaps and reduce incident cleanup.

Portable because: It's SQL + GUCs—no SIEM-specific filter rules to recreate elsewhere.

Operations track: A performance loop you can version and repeat

Keep a longitudinal baseline with EDB Wait States

Run the sampler at a modest interval (e.g., 1s) with 7–14 days of retention. Use it to trend top waits, bracket releases, and localize spikes by role/query.

Outcome now: Performance work starts from evidence (not hunches) and postmortems become explainable.

Portable because: Sampling/retention are database settings; the baseline survives platform changes.

Set an initial posture with Dynatune, then tune from evidence

Declare workload intent (`edb_dynatune_profile`) and aggressiveness (`edb_dynatune`), then iterate using what Wait States shows.

Outcome now: You avoid cargo-cult configs and arrive at a defensible starting point faster.

Portable because: The same knobs exist wherever EPAS runs.

Fence heavy jobs with Resource Manager

Bind ETL/analytics/users to named groups (`SET edb_resource_group …`) and cap sprawl with `edb_max_resource_groups`. Consider role defaults for predictable bindings.

Outcome now: OLTP keeps headroom while batch and ad-hoc jobs run under limits.

Portable because: Workload control is database-native—no reliance on cloud-specific throttles.

Use SQL Profiler → Index Advisor to close the loop

When the baseline points to a window, capture a short, filtered trace with SQL Profiler; feed it to Index Advisor to evaluate hypothetical indexes and promote the minimal changes that move p95/p99.

Outcome now: You ship small, provably effective fixes with artifacts (baseline before/after, trace ID/file, Advisor recommendations) attached to the change.

Portable because: The loop is in-engine and repeatable across environments.

Kubernetes note: Keep encryption and ops posture as code

If you operate with EDB Postgres for Kubernetes, declare TDE (`epas.tde.enabled` with a secret reference) and any required EPAS settings in manifests.

Outcome now: Security and ops posture are versioned alongside deployments and reviewed like any other change.

Portable because: The same manifests work across clusters and clouds.

What this blueprint buys you

- **Risk down now:** Choose Oracle semantics per app and prove parity with a real OCI link before cutover. Learn More
- **Security that travels:** Engine-level TDE, data redaction, SQL/Protect, password policies, and log redaction move wherever the database runs. Learn More

- **Day-two predictability:** Wait-event baselines, Dynatune posture, Resource Manager shaping, Profiler → Index Advisor fixes—the same, reviewable loop in every environment. Learn More

That combination—engine semantics, engine security, engine observability—is what makes the plan truly portable across on-prem, cloud, and Kubernetes.

Conclusion: Where EPAS wins

EPAS is an engine-level answer to three enterprise problems: Safer migration, in-database security, and day-two predictability. You get runtime compatibility (to cut rewrite risk and phase cutovers), security controls where the data lives (to pass audits without app rewrites), and a performance loop you can run every day (to improve p95/p99 safely). Those wins persist because they're engine features—encryption, masking, injection guardrails, workload shaping, evidence capture, and tuning are database configuration and SQL, not one-off scripts or a single vendor console—so the same posture works on-prem, in any cloud, and under Kubernetes.

EPAS isn't only an "Oracle-exit" story—it's an engine-level story: The same capabilities that de-risk Oracle migrations (runtime compatibility and `dblink_ora`) also strengthen any Postgres program by putting security, governance, and the performance loop inside the database, so the approach remains portable across on-prem, cloud, and Kubernetes.

Security lives with the data: EPAS provides Transparent Data Encryption (TDE) at the engine layer (files and WAL) and applies dynamic controls such as data redaction, SQL/Protect, password profiles, and password-log redaction. Because these are database features, your encryption and masking posture becomes versionable, auditable, and environment-agnostic (including first-class operator settings in Kubernetes). Upstream PostgreSQL still lacks native TDE; Aurora's strong at-rest encryption is implemented in the storage/KMS plane, not as PostgreSQL-engine TDE.

Governance and observability for day-two reality: EPAS bakes operator-grade controls into the engine: Resource Manager to shape mixed workloads; EDB Wait States to keep a query-aware, sampled wait-event time series; and SQL Profiler + Index Advisor to run a portable **observe** → **hypothesize** → **fix** → **verify** loop with hypothetical indexes and quantified recommendations. Community PostgreSQL users typically assemble extensions and logs; Aurora's best visuals are console-centric. EPAS keeps the method in the database, so the same runbook travels with the cluster.

Compatibility when you need it—then optional convergence: If you are exiting Oracle, EPAS lets you choose semantics at the database boundary (for example, Redwood date/string behavior or statement-level transaction handling) and phase cutovers with `dblink_ora`—issuing `SELECT/INSERT/UPDATE/DELETE` against Oracle from EPAS for parity checks before you flip traffic. Neither community PostgreSQL nor Aurora ships an engine-level Oracle runtime or an OCI database link. Crucially, these switches are reversible: Keep them per app during transition, then taper to native PostgreSQL as tests allow.

How EPAS compares: Recap for decision-makers

Versus community PostgreSQL: Upstream is the right baseline when you want pure PostgreSQL semantics and are happy to assemble pieces yourself. What it doesn't ship is the engine-level bundle that enterprises usually need on day one: native TDE in the database engine, data redaction and SQL/Protect as in-database controls, runtime Oracle compatibility (Redwood switches and statement-level behavior) for risk-managed exits, the OCI link (`dblink_ora`) for phased cutovers, and an integrated operations loop (Resource Manager, Wait States, SQL Profiler → Index Advisor). You can approximate parts of this with extensions and external tooling, but you're stitching—and what you stitch won't automatically travel the same way across environments.

Versus Amazon Aurora PostgreSQL-compatible: Aurora excels as a managed platform inside AWS: resilient storage, fast failover, and rich console-centric observability. Its encryption story is strong in the KMS-backed storage plane, but it isn't PostgreSQL-engine TDE you can carry outside AWS. Aurora also doesn't provide an engine-level Oracle runtime or an OCI database link to phase migrations. For day-two tuning, you get good

dashboards, but the governance and performance loop (workload shaping in-engine, query-aware time series, profiler-to-advisor changes) lives largely outside the database. EPAS, by design, puts those controls in the engine so the same posture works on-prem, in any cloud, and under Kubernetes.

Net of nets: If you need engine-resident security, runtime compatibility to shrink migration risk, and a portable operating model you can version and promote everywhere, EPAS is strictly stronger in these dimensions than either Community or Aurora, because it combines those capabilities inside one distribution rather than in consoles and add-ons.

Two doors, one distribution:

- **Door A – Oracle exit:** Use EPAS's runtime compatibility and `dblink_ora` to reduce rewrite risk now; retire switches later as you converge.
- **Door B – Standardize on Postgres (no Oracle in sight):** Adopt EPAS for engine-level TDE, data redaction, SQL/Protect, and the **Profiler** → **Query Advisor** loop. These are controls and practices that stay identical on-prem, in any cloud, and under Kubernetes.

Why this is the portable choice: With EPAS, the critical levers—encryption posture, masking, injection guardrails, workload shaping, evidence collection, and tuning—are engine-resident and configurable, not tied to a vendor UI. That's why EPAS is strictly stronger than community PostgreSQL or Aurora when you need these capabilities together—and why the approach remains portable as your estate spans on-prem, multi-cloud, and Kubernetes.

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.

About EDB Postgres AI

EDB Postgres AI is the first open, enterprise-grade sovereign data and AI platform, with a secure, compliant, and fully scalable environment, on premises and across clouds. Supported by a global partner network, EDB Postgres AI unifies transactional, analytical, and AI workloads, enabling organizations to operationalize their data and LLMs where, when, and how they need them.

EDB POSTGRES AI