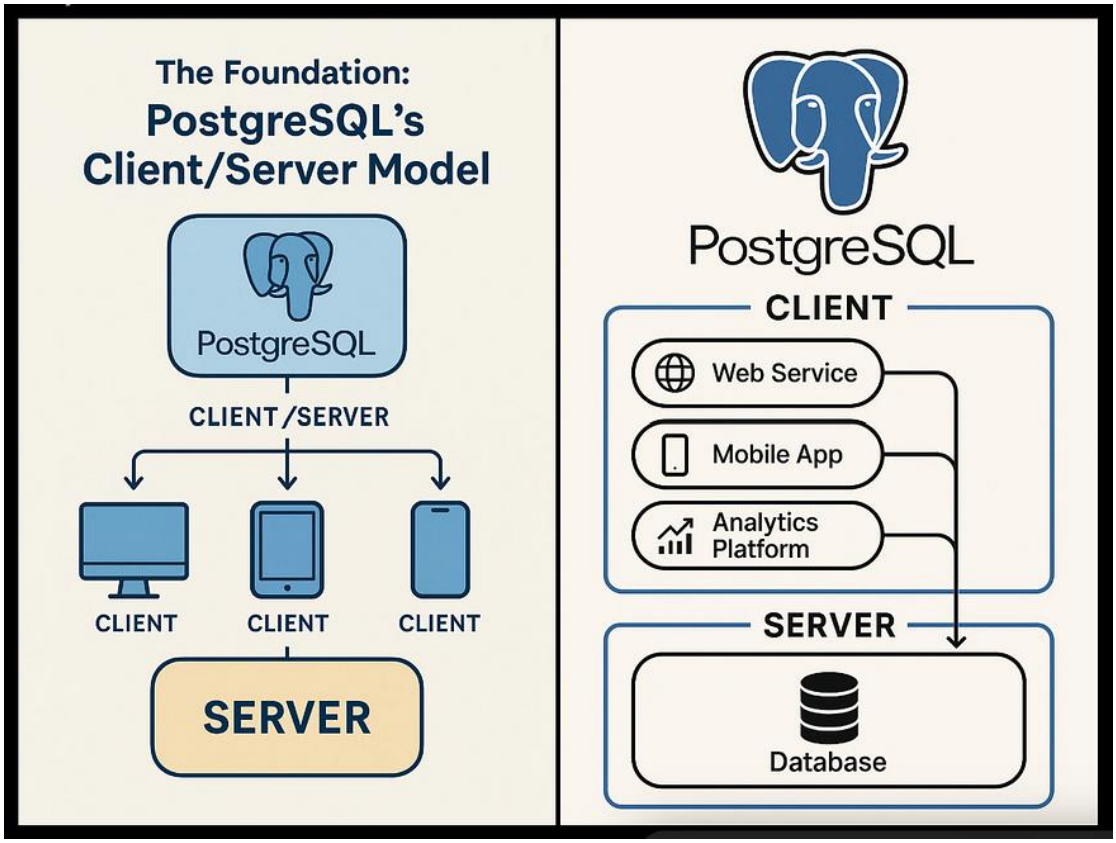
Understanding PostgreSQL Architecture: The Client-Server Model Demystified

PostgreSQL, one of the most powerful and widely used open-source relational database systems, is known for its robustness, scalability, and flexibility. Before diving into its advanced features and internal processes, it's essential to understand the core architecture that powers PostgreSQL. Grasping how its components interact will not only clarify its design but also help you troubleshoot, optimize, and architect PostgreSQL-based solutions more effectively.

The Foundation: PostgreSQL's Client/Server Model

At its heart, PostgreSQL operates on a **client/server architecture** — a time-tested model that separates the user interface (client) from the core data processing (server). This division enables PostgreSQL to serve diverse applications, ranging from web services and mobile apps to complex analytics platforms.

Press enter or click to view image in full size



The Server Process — The Brain of PostgreSQL

The server side of PostgreSQL is managed by the `postgres` process. This server process is responsible for:

- Managing database files and ensuring data integrity.
- Accepting incoming connections from client applications.
- Executing SQL commands on behalf of clients.
- Performing background tasks such as checkpointing, vacuuming, and replication.

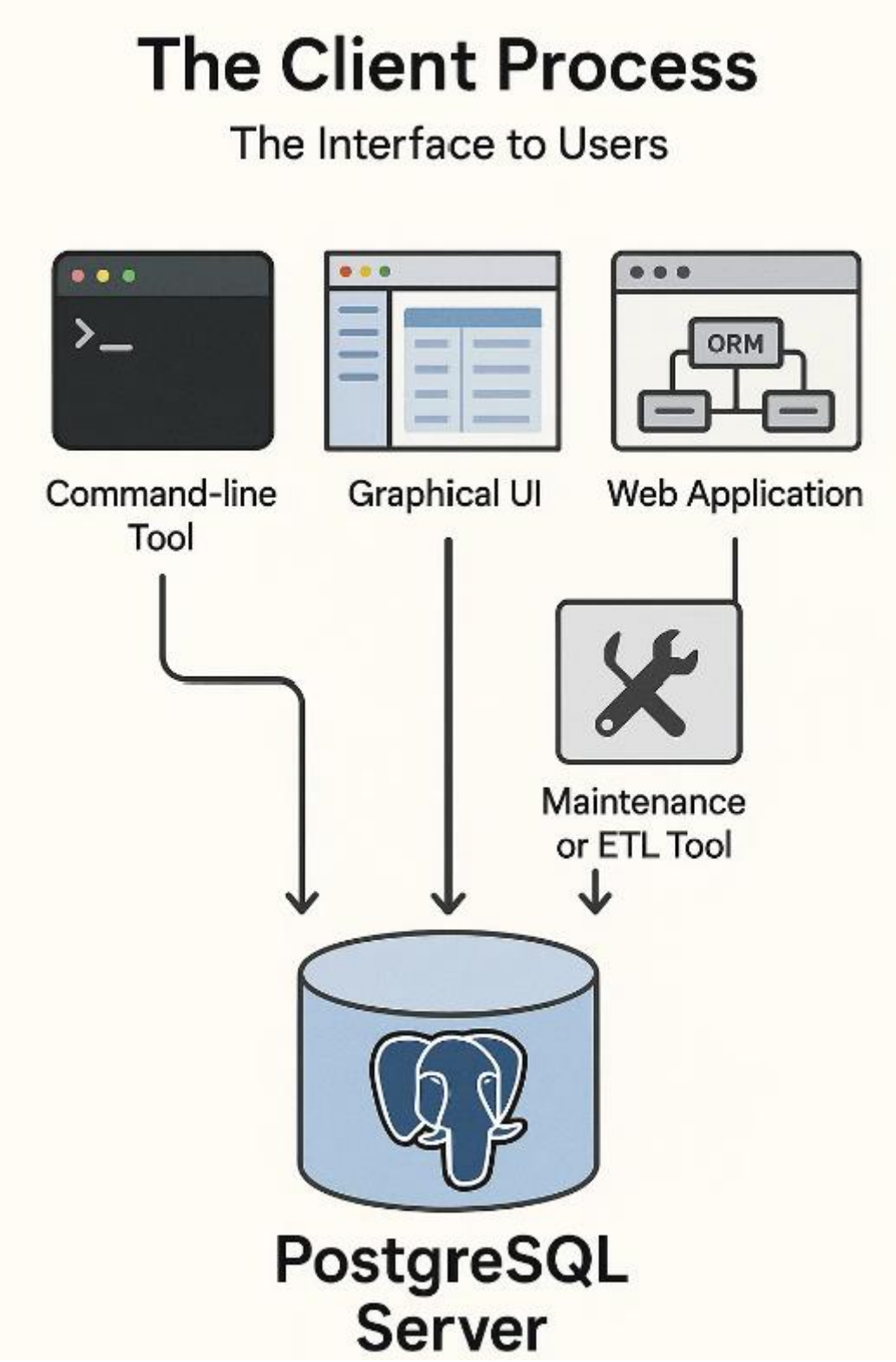
Whenever you install PostgreSQL and start the service, it is the `postgres` server process that comes alive, ready to handle database operations.

The Client Process — The Interface to Users

On the other side of the architecture sits the client application. This could be:

- A command-line tool like `psql`.
- A graphical user interface (GUI) such as pgAdmin.
- A web application backend accessing the database through an ORM (Object Relational Mapper).
- Specialized database maintenance tools or ETL (Extract, Transform, Load) jobs.

Clients initiate connections to the PostgreSQL server to issue queries, retrieve results, and perform various operations on the database.



Communication: Local or Remote

A key advantage of the client/server model is its flexibility in deployment. The client and server can reside on the same machine or on entirely

different hosts. When they are on separate systems, communication happens securely over TCP/IP.

It's important to recognize that:

- Files available on the client machine may not be directly accessible to the server and vice versa.
- File paths and system resources referenced by the client might not exist on the server.

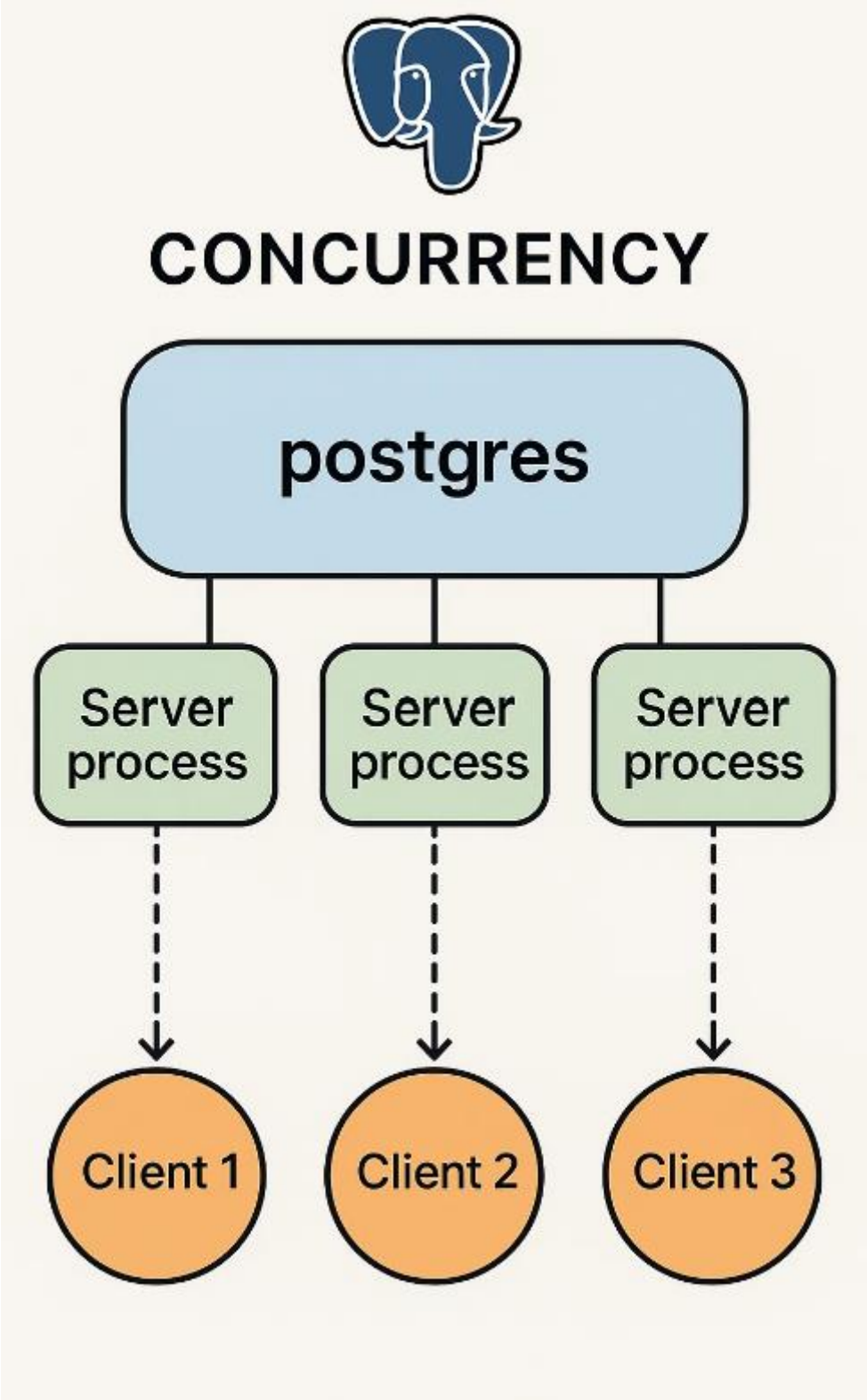
This separation reinforces security and allows PostgreSQL to support diverse deployment scenarios — from local development to large-scale, cloud-hosted architectures.

Concurrency: How PostgreSQL Handles Multiple Clients

PostgreSQL is designed to efficiently handle multiple client connections simultaneously. To achieve this, it employs a **process-per-connection model**:

1. The main `postgres` server process listens continuously for new client connections.
2. When a client connects, PostgreSQL **forks** (spawns) a new server process dedicated to handling that connection.
3. Each client-server pair communicates directly, independent of other connections.

This design allows PostgreSQL to isolate sessions, improving stability and ensuring that one misbehaving session doesn't easily impact others. While this detail is mostly hidden from end users, it's valuable for DBAs and developers to understand, especially when tuning performance or diagnosing system resource usage.



PostgreSQL Background Processes: The Invisible Heroes Powering Your Database

When most people think of PostgreSQL, they immediately recall its powerful SQL capabilities, robust ACID compliance, and flexible data

types. However, what often goes unnoticed are the internal background processes that work silently, 24/7, to ensure PostgreSQL remains fast, stable, and resilient even under demanding workloads.

We'll explore the internal background processes that make PostgreSQL one of the most reliable and popular open-source relational database systems in the world.

Behind the Scenes: Why Background Processes Matter in PostgreSQL

PostgreSQL follows a **multi-process architecture** — unlike some database systems that rely on a single multithreaded process. This architecture allows PostgreSQL to break its workload into several specialized background processes, each focusing on critical database tasks such as:

- Writing data to disk
- Managing transactions safely
- Handling replication and backups
- Reclaiming unused space
- Collecting system and query performance statistics

By delegating responsibilities to these dedicated processes, PostgreSQL is able to maintain high concurrency, reduce bottlenecks, and ensure excellent system stability.

To observe these background processes in action, you can run the following command on your PostgreSQL host:

```
ps -ef | grep postgres
```

This will display a list of PostgreSQL system processes, each performing specific database operations in the background. Let's now take a detailed look at these essential PostgreSQL components.

```
postgres   23442       1  0 16:49 ?        00:00:00 /usr/pgsql-17/bin/postgres -D
/var/lib/pgsql/17/data/
postgres   23443   23442  0 16:49 ?        00:00:00 postgres: logger
postgres   23444   23442  0 16:49 ?        00:00:00 postgres: checkpointer
postgres   23445   23442  0 16:49 ?        00:00:00 postgres: background writer
postgres   23447   23442  0 16:49 ?        00:00:00 postgres: walwriter
postgres   23448   23442  0 16:49 ?        00:00:00 postgres: autovacuum launcher
postgres   23449   23442  0 16:49 ?        00:00:00 postgres: logical replication
launcher
root       23451   22870  0 16:49 pts/0    00:00:00 grep --color=auto postgres
```

□ Core Background Processes in PostgreSQL

1□ Checkpointer: Safeguarding Data Durability

The Checkpointer ensures that modified data in memory (known as *dirty buffers*) is safely written to disk periodically. This protects against data loss in the event of a system crash and prevents overwhelming I/O spikes during large checkpoint operations.

How It Works:

- Monitors shared memory for modified data pages.
- Flushes dirty buffers to disk incrementally, according to parameters like `checkpoint_timeout` and `checkpoint_completion_target`.
- Helps spread disk I/O load evenly over time.

Without the Checkpointer, PostgreSQL would be forced to write huge amounts of data all at once during checkpoints, leading to unpredictable performance and high disk utilization.

2□ Background Writer: Keeping Memory Buffers Clean

The Background Writer works continuously to flush dirty pages from shared memory to disk, helping to maintain an optimal balance between memory usage and disk writes. This allows client transactions to complete faster since they rarely have to perform their own I/O writes.

How It Works:

- Scans shared memory at regular intervals.
- Writes dirty buffers proactively before they accumulate.
- Reduces write latency and improves response times for client transactions.

By handling small amounts of I/O constantly, the Background Writer smoothens PostgreSQL's disk activity, preventing I/O spikes that would otherwise slow down query processing.

3 WAL Writer: Ensuring Transactional Integrity

PostgreSQL uses Write-Ahead Logging (WAL) to guarantee that every transaction is recoverable. The WAL Writer handles flushing changes from WAL buffers in memory to WAL files on disk, ensuring that committed transactions are safe even if the database crashes.

How It Works:

- Monitors WAL buffers in shared memory.
- Periodically writes WAL records to disk.
- Enables crash recovery and supports streaming replication to standby servers.

WAL is the backbone of PostgreSQL's ACID compliance — ensuring that every committed transaction is always recoverable.

4 Autovacuum Launcher: Managing Dead Tuples Efficiently

PostgreSQL uses Multiversion Concurrency Control (MVCC), which allows multiple transactions to occur simultaneously. When rows are updated or deleted, old versions (called *dead tuples*) remain in storage until vacuuming occurs. The Autovacuum Launcher ensures that these obsolete rows are cleaned up automatically, reclaiming space and keeping query performance high.

How It Works:

- Continuously monitors table activity using visibility maps.
- Launches autovacuum worker processes based on thresholds (like `autovacuum_vacuum_threshold` and `autovacuum_analyze_threshold`).
- Cleans up dead tuples and updates table statistics for the query planner.

Without regular autovacuum activity, tables would grow unnecessarily large, leading to slower queries and inefficient disk usage.

5 Stats Collector: Fueling the Query Planner with Accurate Data

PostgreSQL's query planner relies on accurate table and index statistics to generate efficient query execution plans. The Stats Collector gathers these statistics in real-time.

How It Works:

- Collects activity and performance metrics from backend processes.
- Stores aggregated statistics in temporary files.
- Supplies data to the query planner for better query optimization.

Accurate statistics are critical for PostgreSQL to choose the fastest execution path when running SQL queries.

6 Logical Replication Launcher: Streaming Data Across Systems

PostgreSQL supports logical replication, allowing selective replication of tables and databases across different PostgreSQL instances. The Logical Replication Launcher manages the replication worker processes that ensure data consistency across distributed systems.

How It Works:

- Starts logical replication worker processes for each active subscription.
- Continuously streams changes (INSERT, UPDATE, DELETE) to the target system.
- Ensures data consistency between primary and replica databases in near real-time.

Logical replication allows PostgreSQL users to build highly available systems, perform zero-downtime upgrades, or replicate data across regions for disaster recovery.

Bonus: Other Supporting Background Processes

Additional PostgreSQL Background Processes: Archiving and Replication Workers Explained

In our previous discussion, we explored the core PostgreSQL background processes like Checkpointer, WAL Writer, Background Writer, Autovacuum, and Logical Replication Launcher. However, PostgreSQL's internal architecture includes several other important auxiliary background processes — especially when you're working with replication, high availability, and backup configurations.

In this section, we'll focus on four additional PostgreSQL background processes that play a crucial role in data durability, replication, and business continuity.

□ 1□ Archiver Process

The Archiver process is responsible for safely archiving completed Write-Ahead Log (WAL) segments for long-term retention and disaster recovery purposes. This is essential for PITR (Point-In-Time Recovery) and backup strategies.

How It Works:

- Monitors for completed WAL segments that are ready for archiving.
- Copies these segments to a configured archive location (often NFS, cloud storage, or local backups).
- Controlled by the `archive_mode` and `archive_command` settings.

Example Configuration:

```
archive_mode = on
archive_command = 'cp %p /var/lib/postgresql/archive/%f'
```

Without the Archiver process, you would not be able to retain full WAL history necessary for restoring the database to any point in time or for building new replicas from archived WAL files.

□ 2□ Replication Sender (walsender)

The WAL Sender process (walsender) handles streaming WAL data from the primary server to standby replicas for synchronous or asynchronous replication.

How It Works:

- Continuously reads newly generated WAL entries.
- Streams WAL data over the network to connected standbys.
- Supports both physical replication and logical replication subscribers.
- One walsender process is spawned per connected replica.

The WAL Sender is the core of PostgreSQL's high availability architecture, enabling near real-time failover setups.

□ 3□ Replication Receiver (walreceiver)

Purpose:
The WAL Receiver (walreceiver) process runs on the standby server and is responsible for receiving the WAL stream sent by the primary's walsender.

How It Works:

- Continuously receives WAL data over network.
- Writes received WAL segments to the standby's local disk.
- Applies the WAL changes to the data files in real-time (or delayed, based on configuration).

This process ensures the standby instance stays in sync with the primary, allowing rapid failover with minimal data loss.

□ 4□ Logical Replication Workers

Logical Replication Workers handle more granular, table-level replication, which is especially useful for multi-master replication setups, partial data replication, and live migrations.

How It Works:

- Subscribe to logical replication publications on the source database.
- Process INSERT, UPDATE, and DELETE changes for subscribed tables.
- Apply changes to the target tables in near real-time.
- Can support multiple subscriptions and parallel workers.

Logical Replication Workers make PostgreSQL extremely flexible for use cases like zero-downtime upgrades, cross-datacenter replication, and heterogeneous system integration.

□ Why These Processes Matter

These additional background processes — Archiver, WAL Sender, WAL Receiver, and Logical Replication Workers — form the backbone of PostgreSQL's advanced replication, backup, and high availability capabilities. Understanding how they work allows DBAs and architects to design resilient systems that can withstand:

- Hardware failures
- Network outages
- Disaster recovery scenarios
- Migration or scaling events

□ Conclusion: Invisible, but Indispensable

The true strength of PostgreSQL lies not just in its SQL engine, but in its carefully engineered background processes that ensure consistent, fast, and reliable database performance at any scale. Understanding how these background processes work can help developers, DBAs, and architects:

- Optimize database configurations

- Diagnose performance issues
- Tune workloads for better concurrency and throughput
- Build resilient replication and HA setups

The next time you run PostgreSQL, remember — behind every successful query is a symphony of background processes working in harmony to keep your data safe and your application running smoothly.

In Summary

PostgreSQL's architecture is elegantly simple yet immensely powerful. By separating client applications from the core server process, and by creating isolated processes for each connection, PostgreSQL ensures reliable, concurrent access to data with excellent scalability.

Whether you're building your first application or managing large production clusters, understanding this foundational architecture is a crucial first step in mastering PostgreSQL.