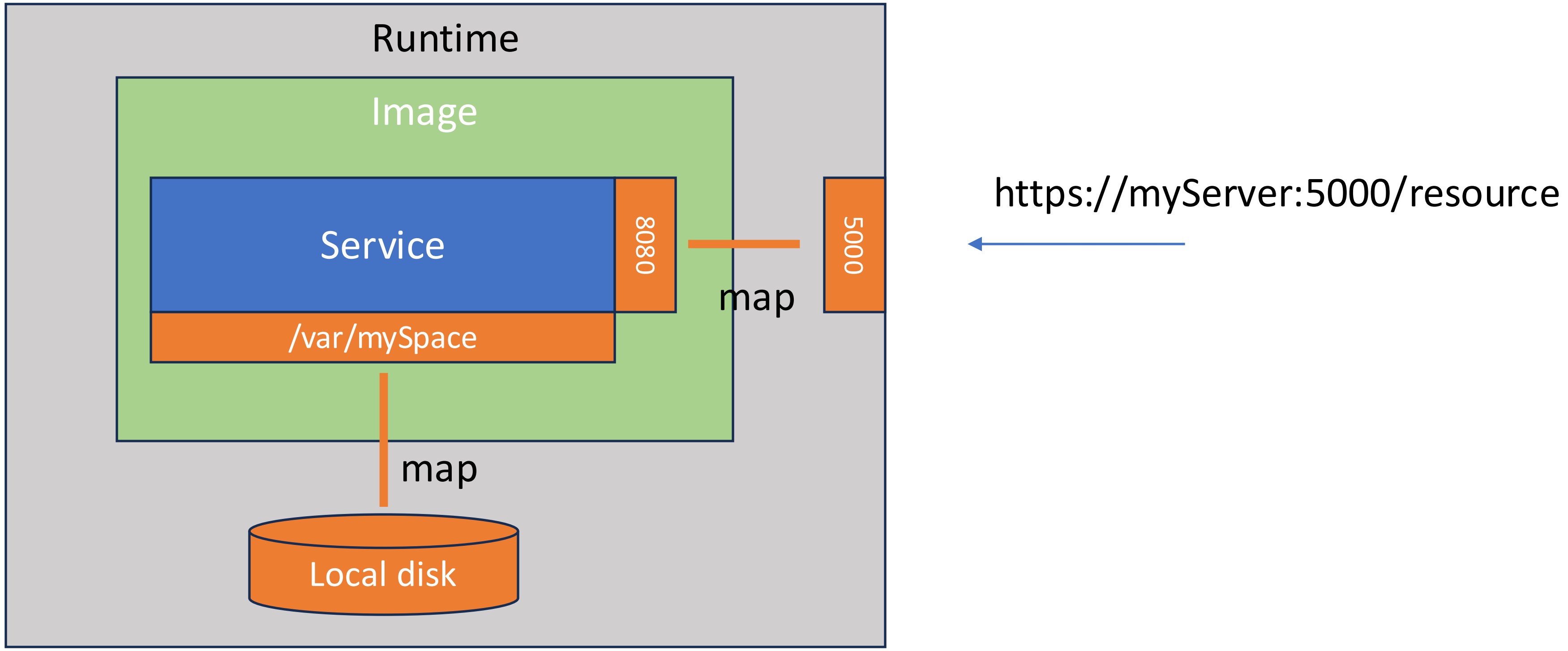
ACP / 3 + 4

Michael Glienecke, PhD

Welcome again

- A bit of recap
- Storing data

A microservice in docker

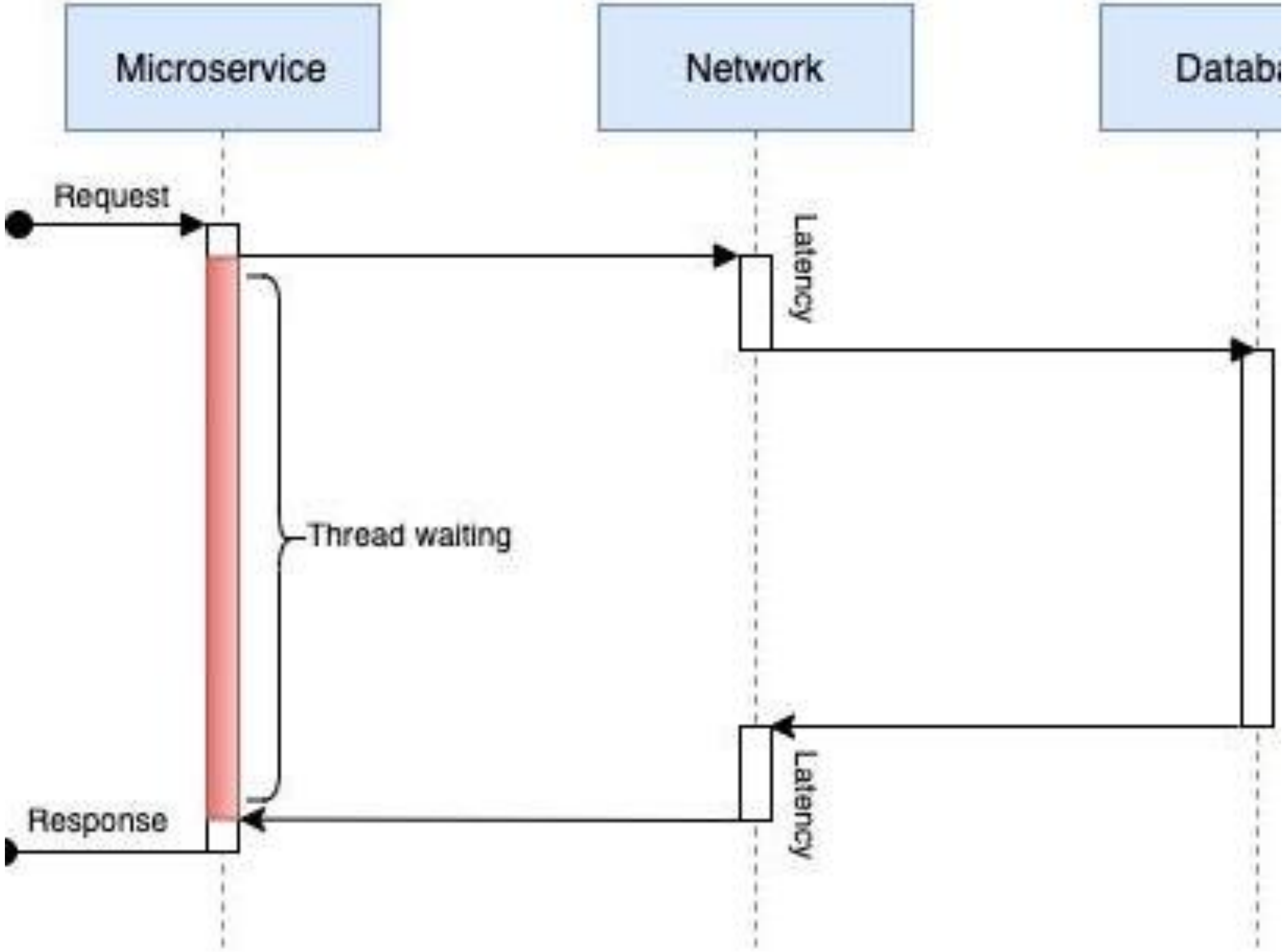


Microservices structure

Spring Boot uses Tomcat

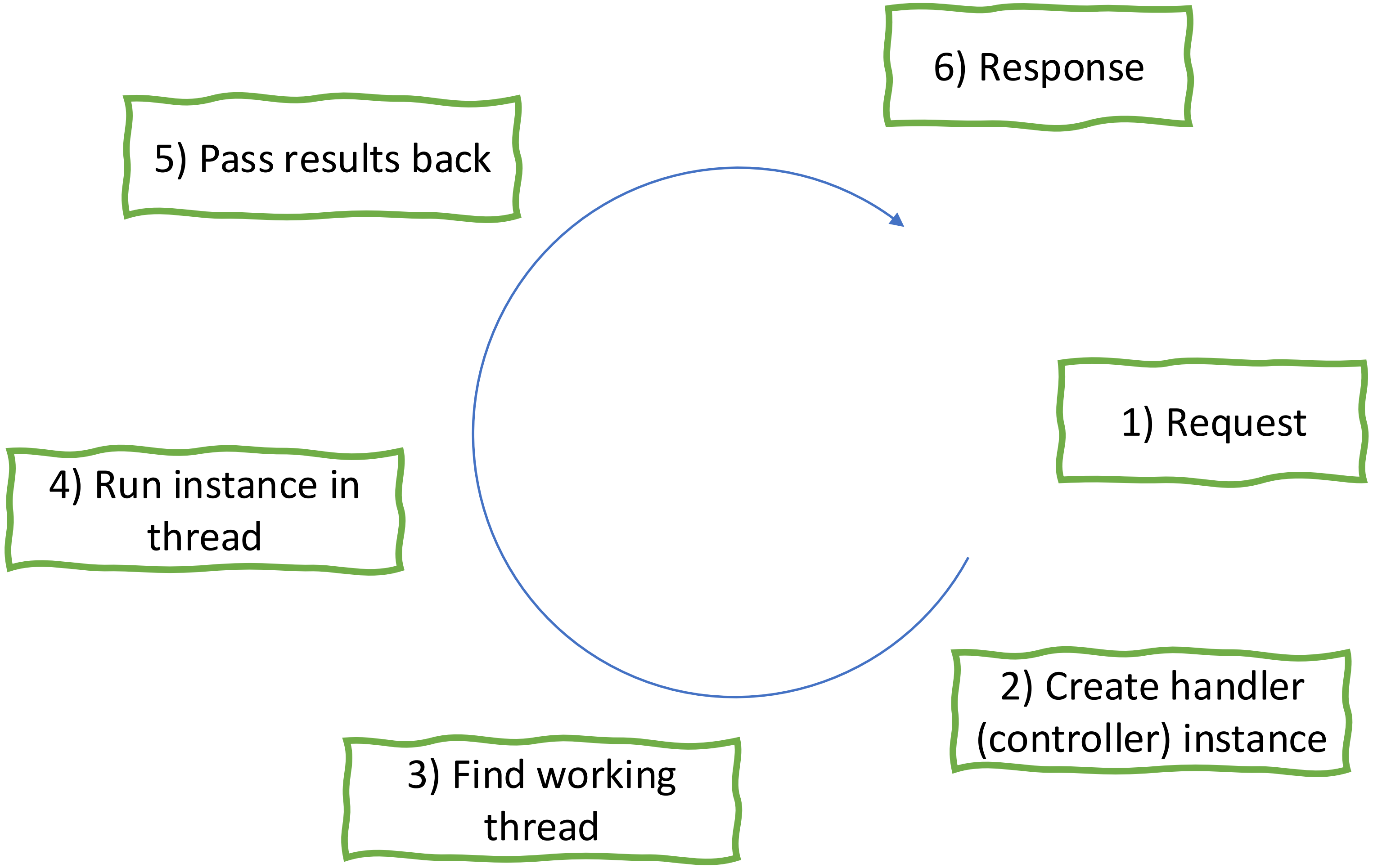
- Some limitations (https://oskar-uit-de-bos.medium.com/the-performance-challenge-in-java-microservices-e51cce3977e9)

Blocking threads and consuming resources



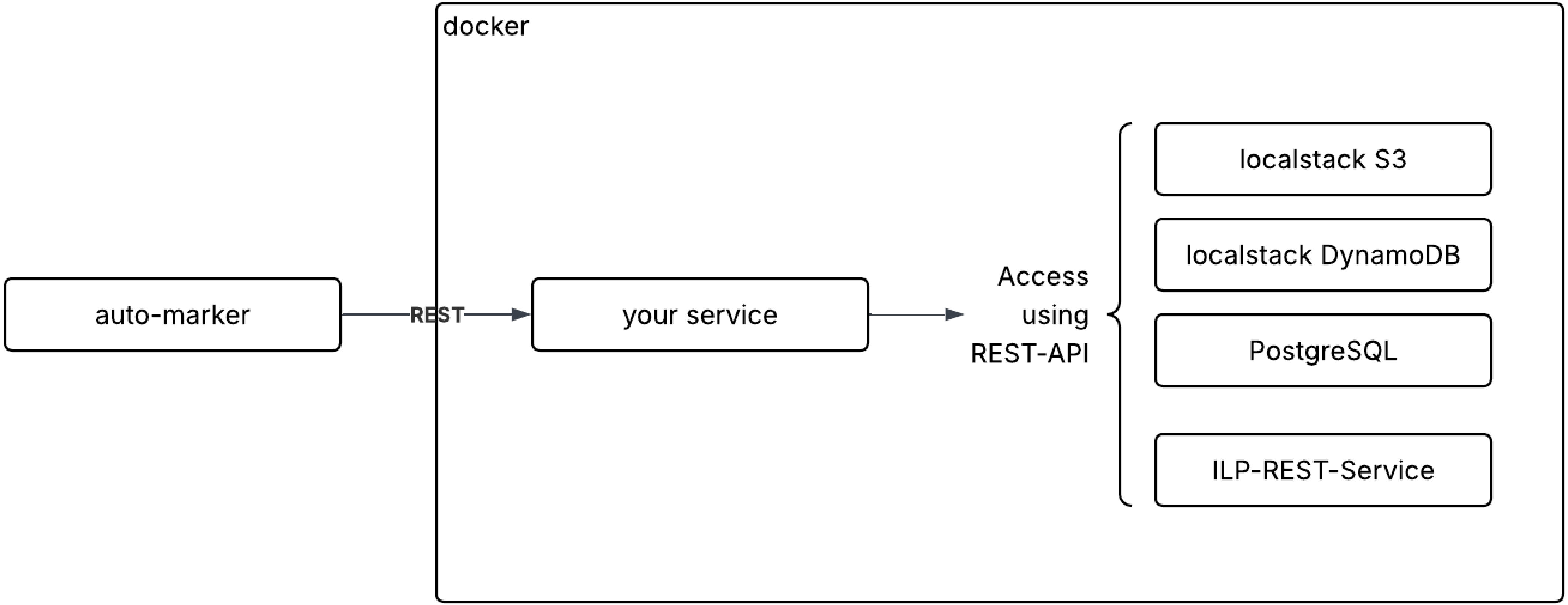
Inspection of a Java Spring Boot based service

The structure of a microservice



1) Request
2) Create handler (controller) instance
3) Find working thread
4) Run instance in thread
5) Pass results back
6) Response

Service landscape



Looking at other languages for services

- Pure Java (no Spring Boot)
- Rust
- Go

Why store data?

- Temporary data
- State

- General Persistence / long term
- Microservice structure / data

- Alternatives? In memory / queues, …

Storage options

- Files
- SQL / No-SQL databases
  - Postgres, SQL Server, DB/2, ORACLE, … (SQL)
  - DynamoDB (Key-Value database)
  - S3, Azure Blobs (Blob-Storage)
  - MongoDB / CosmosDB / MariaDB (No-SQL)

When do you use what?

- Files
  - Temporary, volatile things
- SQL
  - Transactional business Domain Data
  - Data Warehousing
- Key-Value database
  - Quick paced data with single key
  - Often huge volumes
  - Serverless Apps

When do you use what /2?

- Blob-Storage
  - Mass storage of data items (images, …)
  - Big Data analysis
- No-SQL
  - Documents
  - JSON / XML data

What we will be using

- DynamoDB + S3 using localstack
    - UI in localstack UI
- PostgreSQL
    - dbeaver and pgAdmin as UI tools

localstack

- https://www.localstack.cloud/
- Runs in your local system
- Emulates many (depending on level) aws-Services
- Access using the aws SDK (preferably v2) https://aws.amazon.com/sdk-for-java/

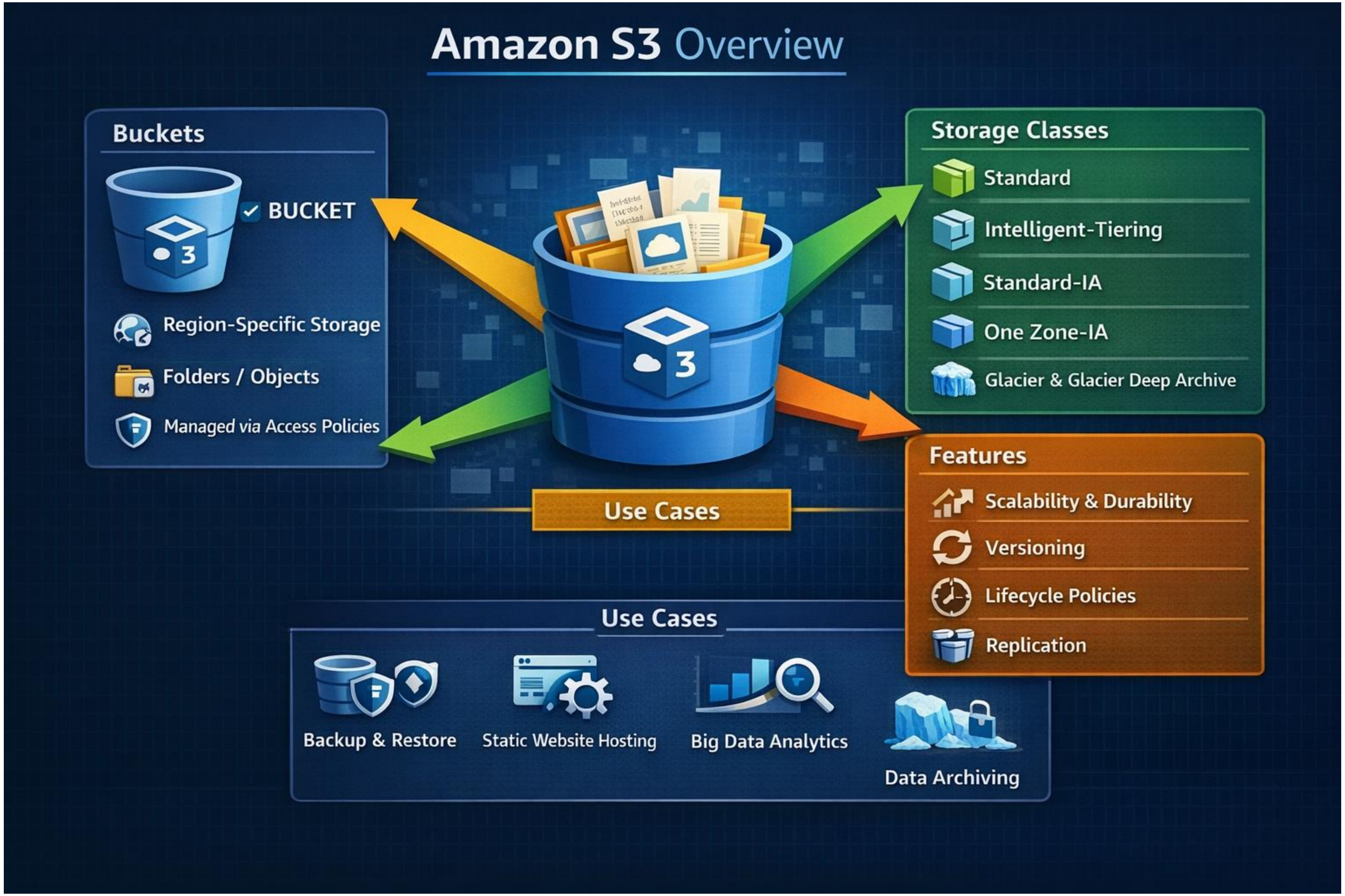
- “test” / “test” as credentials for local usage

Localstack /2

- DynamoDB
  - https://docs.aws.amazon.com/code-library/latest/ug/dynamodb_example_dynamodb_Scenario_GettingStartedMovies_section.html
  - https://docs.aws.amazon.com/code-library/latest/ug/dynamodb_code_examples_actions.html
- S3:
  - https://docs.aws.amazon.com/code-library/latest/ug/s3_example_s3_Scenario_GettingStarted_section.html
- Localstack:
  - https://docs.localstack.cloud/aws/integrations/aws-sdks/java/

S3

- Overview
- Integration
- Structure



Amazon S3 Overview

Buckets

- BUCKET
- Region-Specific Storage
- Folders / Objects
- Managed via Access Policies

Storage Classes

- Standard
- Intelligent-Tiering
- Standard-IA
- One Zone-IA
- Glacier & Glacier Deep Archive

Features

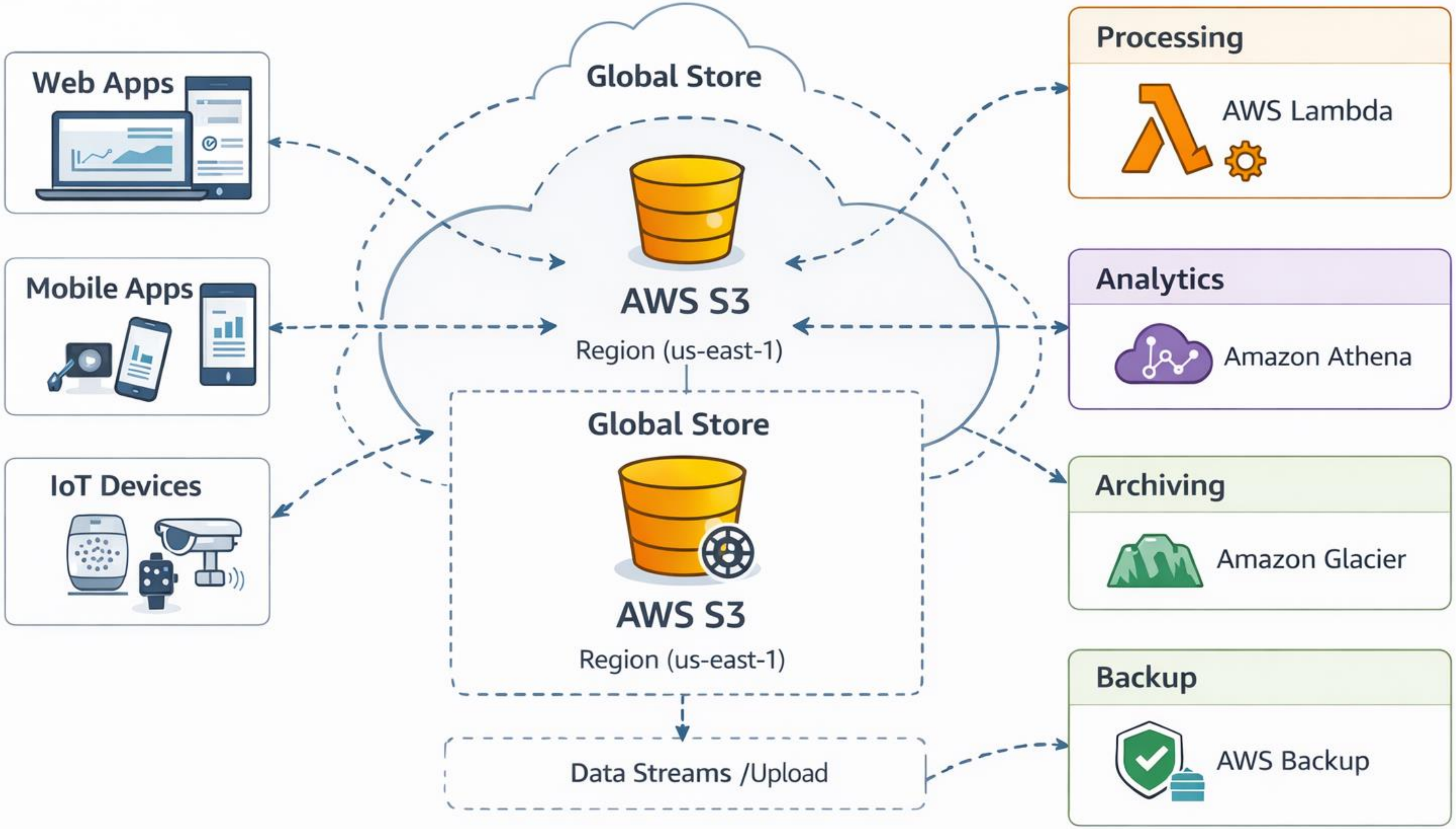
- Scalability & Durability
- Versioning
- Lifecycle Policies
- Replication

Use Cases

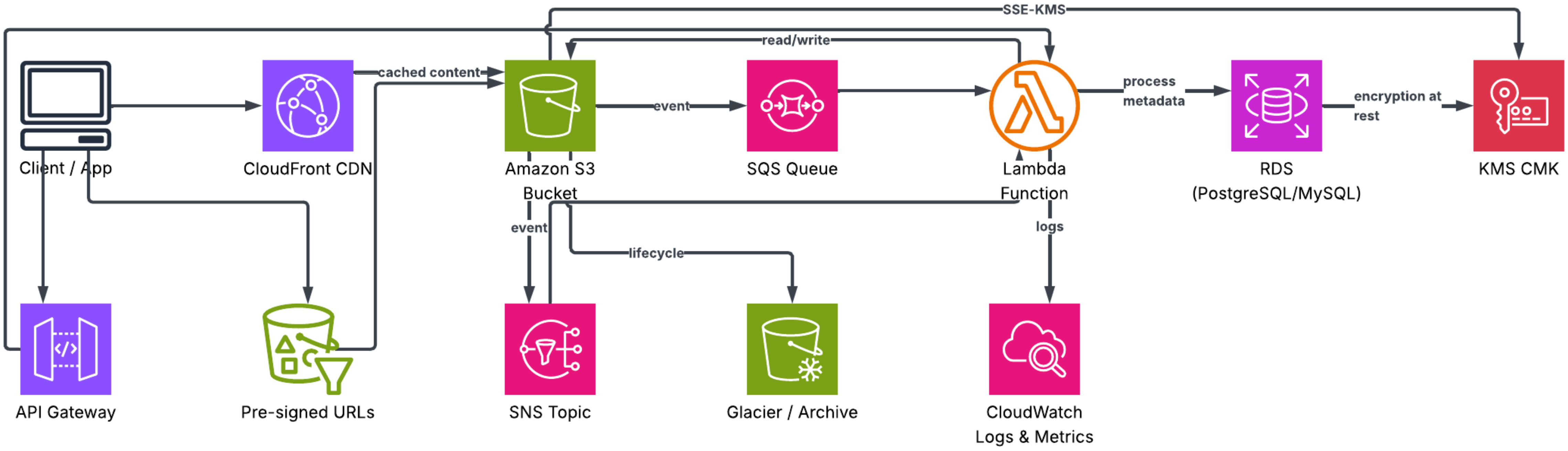
Use Cases

- Backup & Restore
- Static Website Hosting
- Big Data Analytics
- Data Archiving

Typical AWS S3 Integration Into Global Process



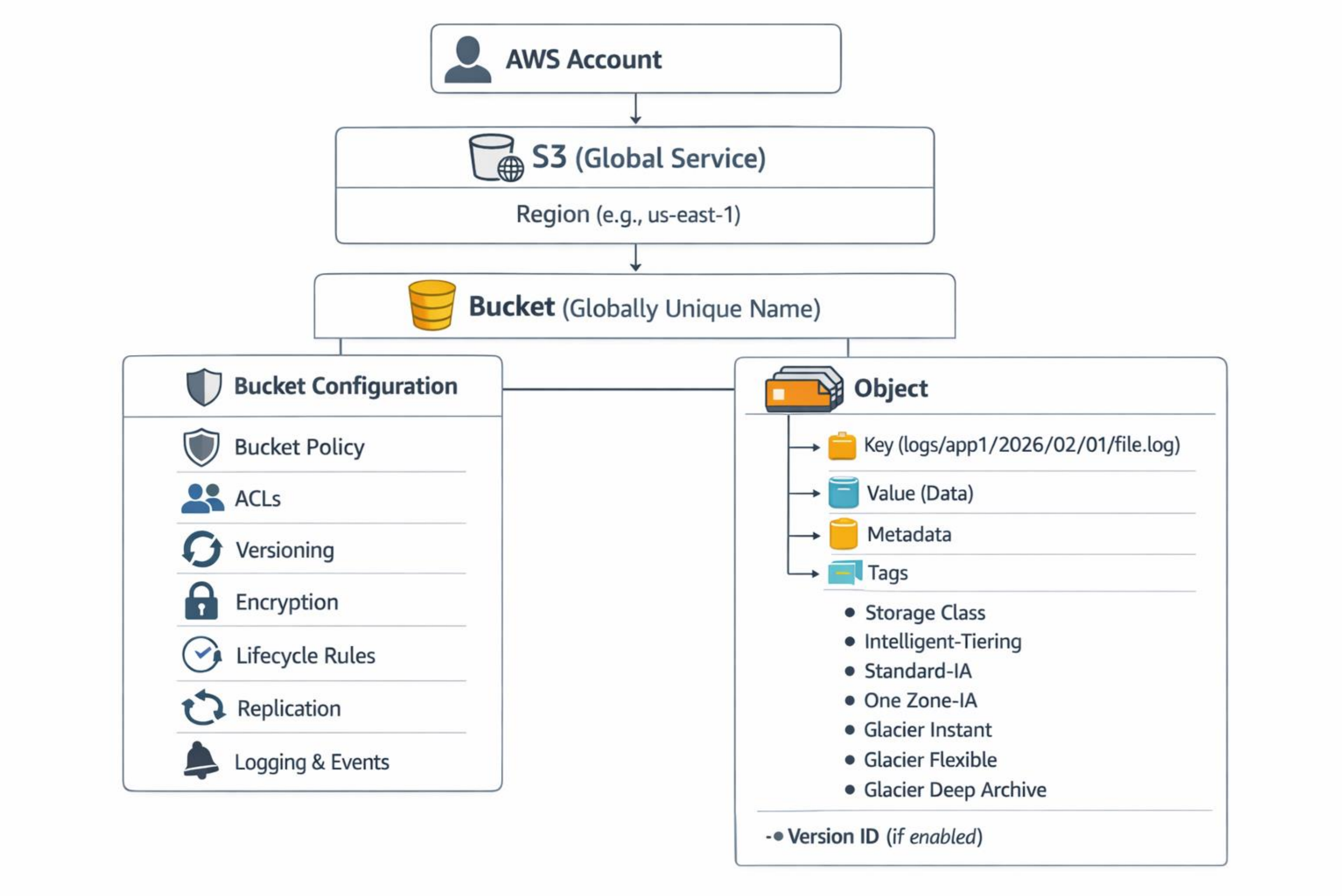
Integration scenario



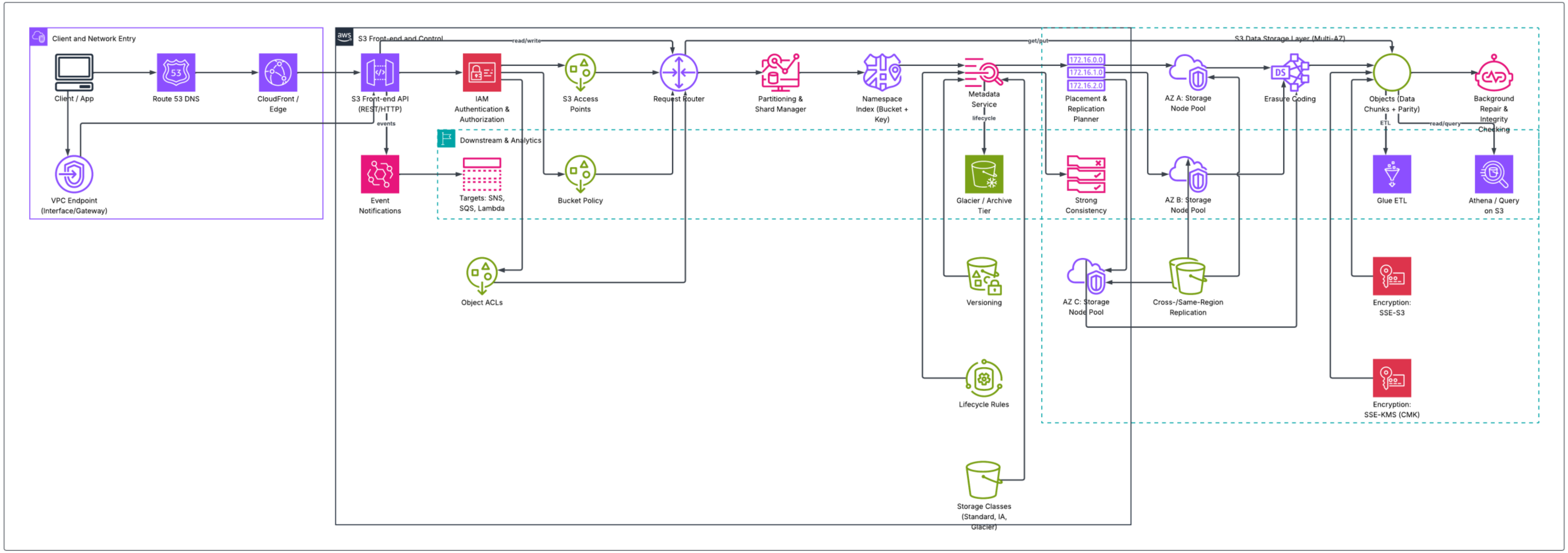
Structure

- No folders (folders are just a prefix in the object tree)
- Everything is an object
- Keys are flat
  `a/b/c/file.txt` is a single string, not a directory tree
- Buckets are regional (but names are global)

S3 internal


AWS Account
S3 (Global Service)
Region (e.g., us-east-1)
Bucket (Globally Unique Name)
Bucket Configuration
Bucket Policy
ACLs
Versioning
Encryption
Lifecycle Rules
Replication
Logging & Events
Object
Key (logs/app1/2026/02/01/file.log)
Value (Data)
Metadata
Tags
Storage Class
Intelligent-Tiering
Standard-IA
One Zone-IA
Glacier Instant
Glacier Flexible
Glacier Deep Archive
Version ID (if enabled)

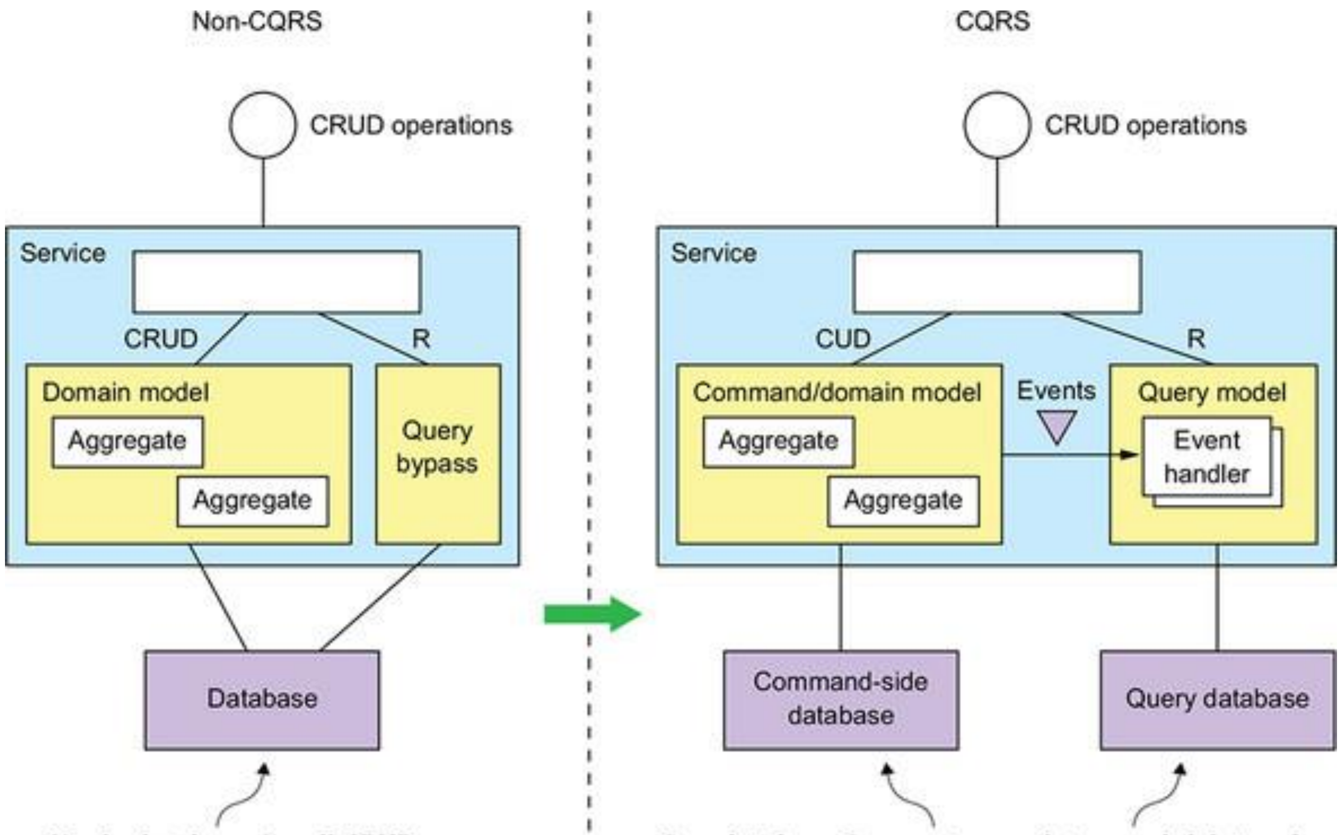
AWS S3 Internal Structure Diagram



Applying storage in a real-world problem

- CQRS (Command Query Responsibility Segregation)

CQRS
Command Query Responsibility Segregation



Single database for all CRUD

One database for creates, updates, and deletes. A separate database for queries. It is kept up-to-date by using events that are published whenever the command-side database changes.

https://livebook.manning.com/book/microservices-patterns/chapter-7/138

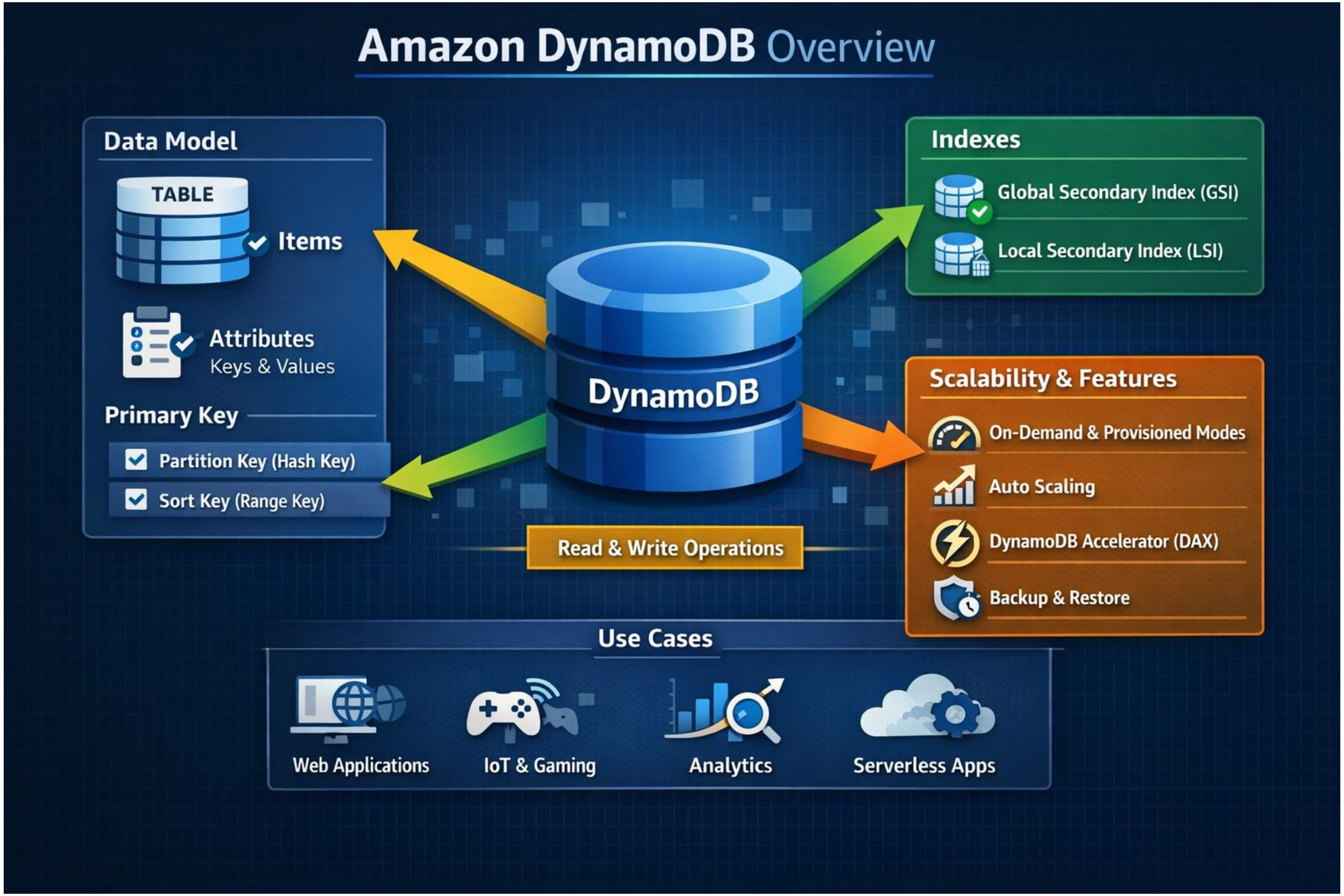
CQRS Pros / Cons

- Enables the efficient implementation of queries in a microservice architecture
- Enables the efficient implementation of diverse queries
- Makes querying possible in an event sourcing-based application
- Improves separation of concerns

- More complex architecture
- Dealing with the replication lag

DynamoDB

- Overview
- Integration
- Structure



Amazon DynamoDB Overview

Data Model

TABLE

Items

Attributes
Keys & Values

Primary Key

☑ Partition Key (Hash Key)

☑ Sort Key (Range Key)

Indexes

Global Secondary Index (GSI)

Local Secondary Index (LSI)

Scalability & Features

On-Demand & Provisioned Modes

Auto Scaling

DynamoDB Accelerator (DAX)

Backup & Restore

DynamoDB

Read & Write Operations

Use Cases

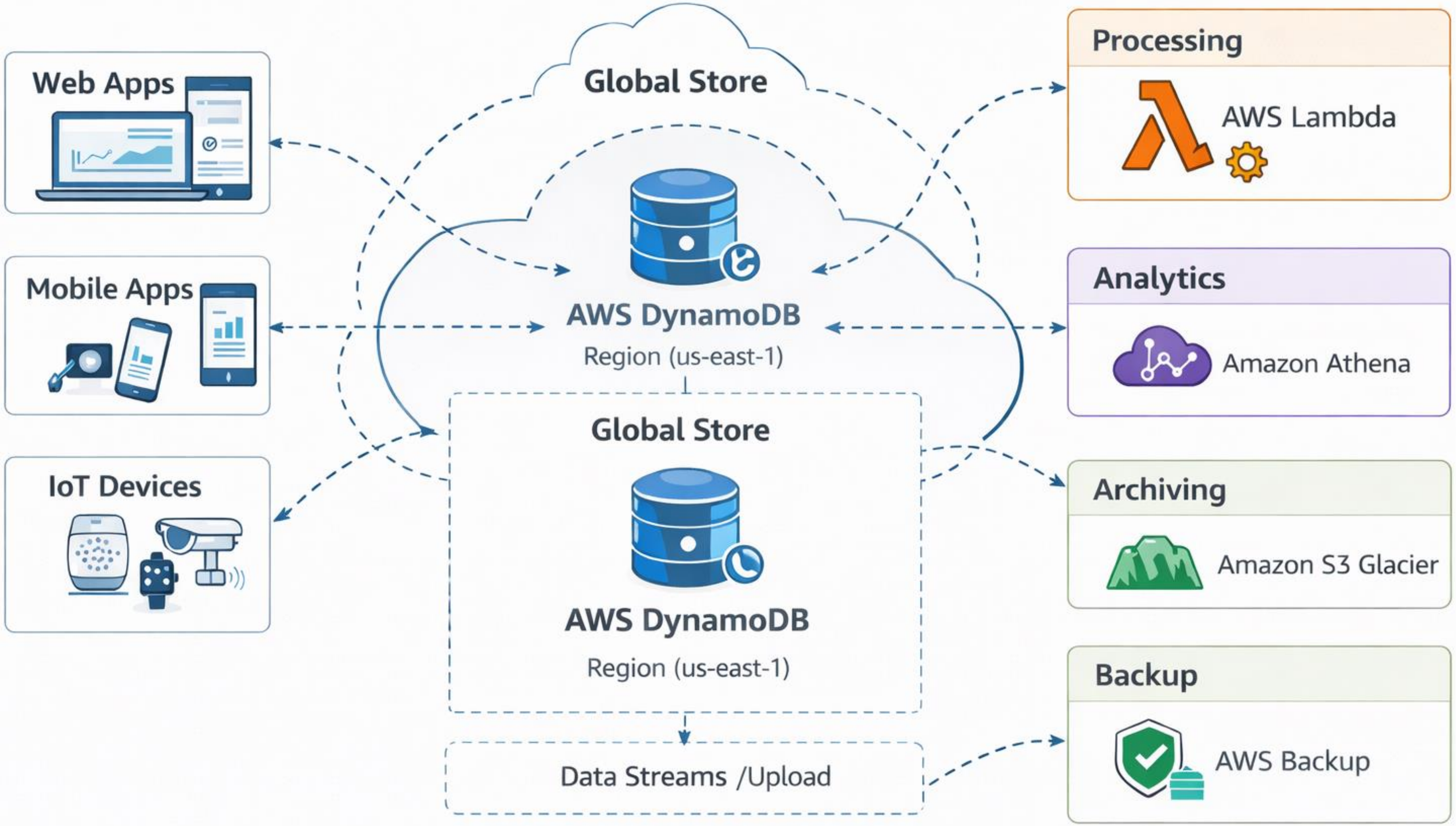
Web Applications

IoT & Gaming

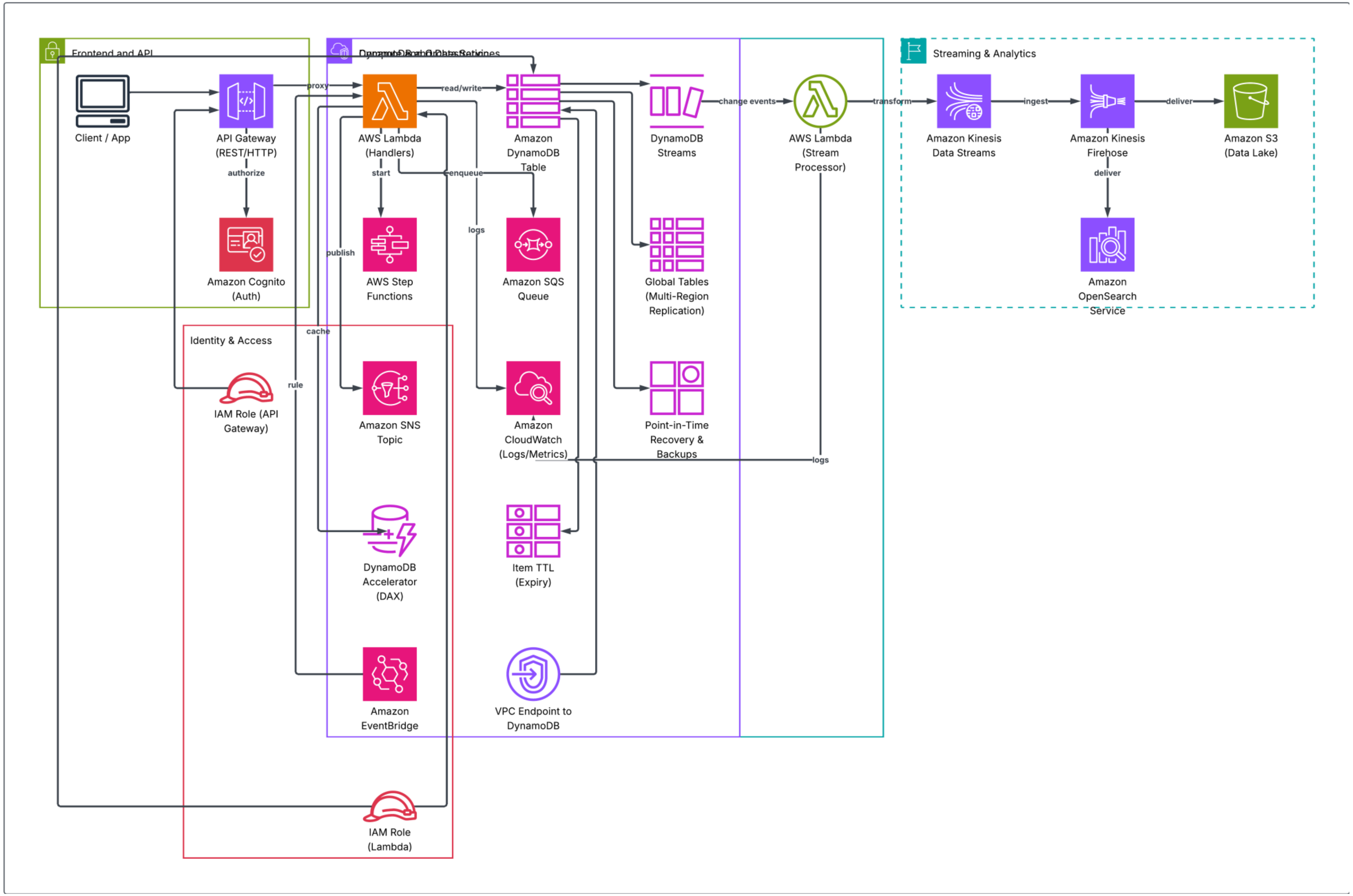
Analytics

Serverless Apps

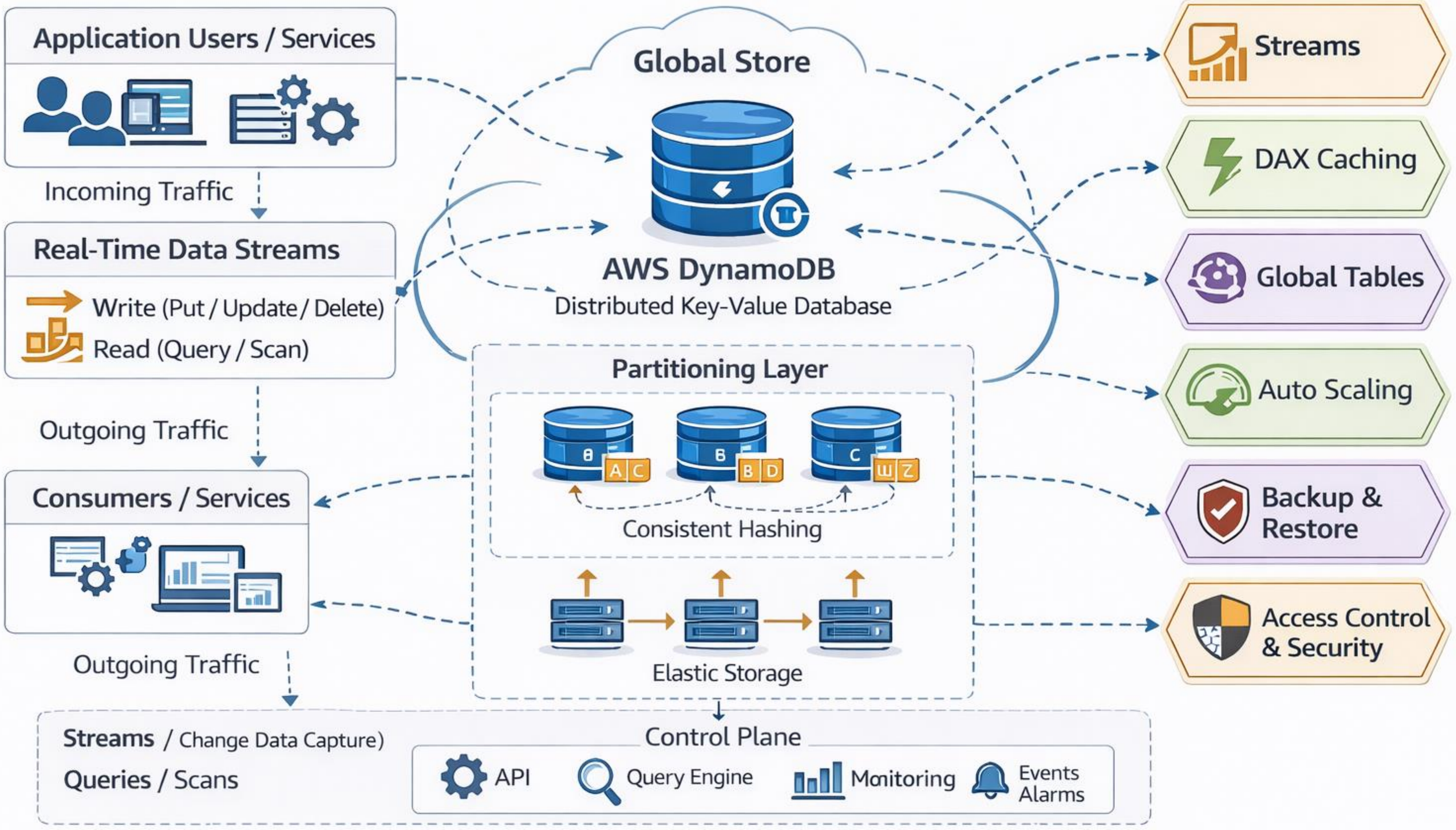
Typical AWS DynamoDB Integration Into Global Process



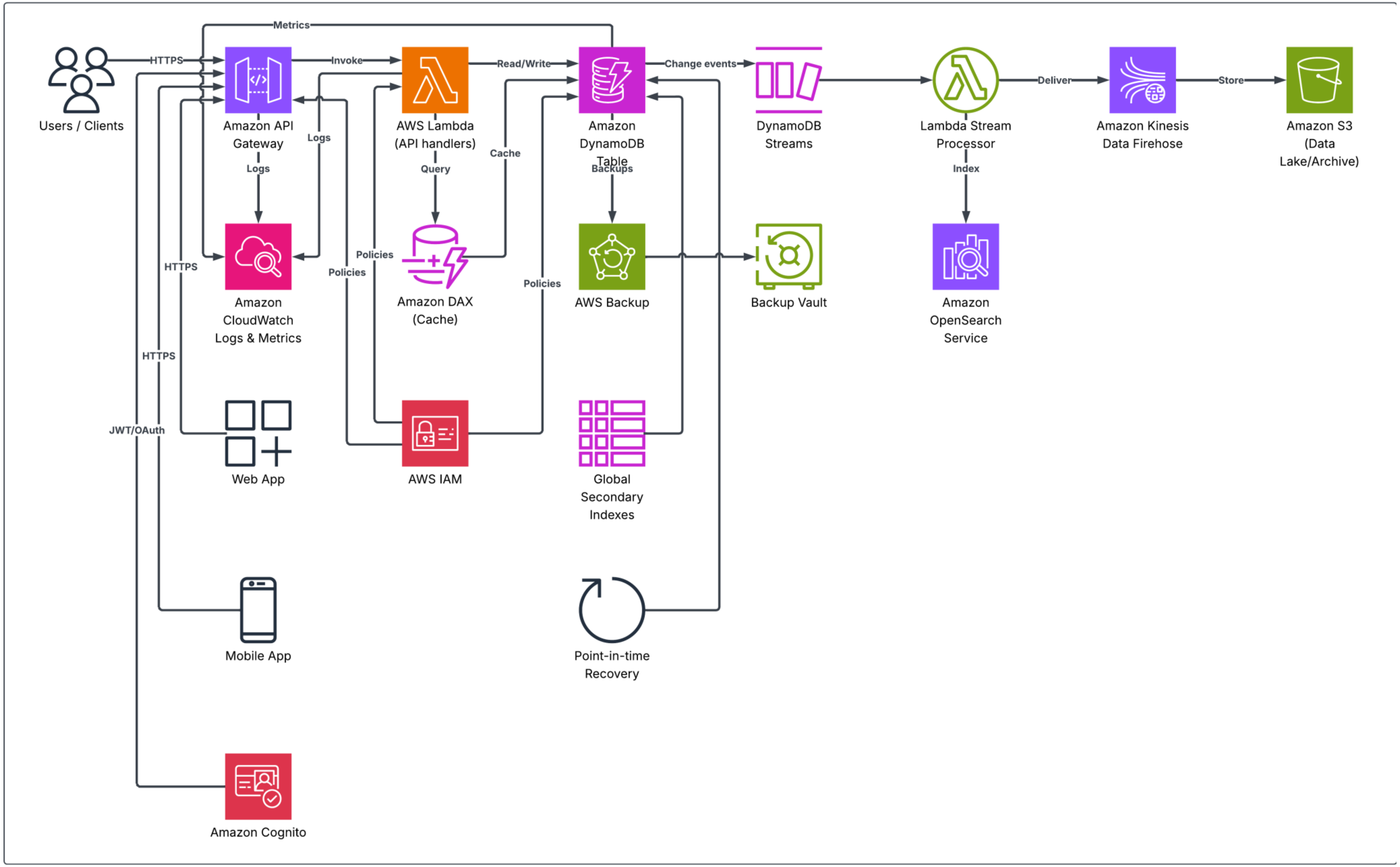
AWS DynamoDB Integration Flow Diagram



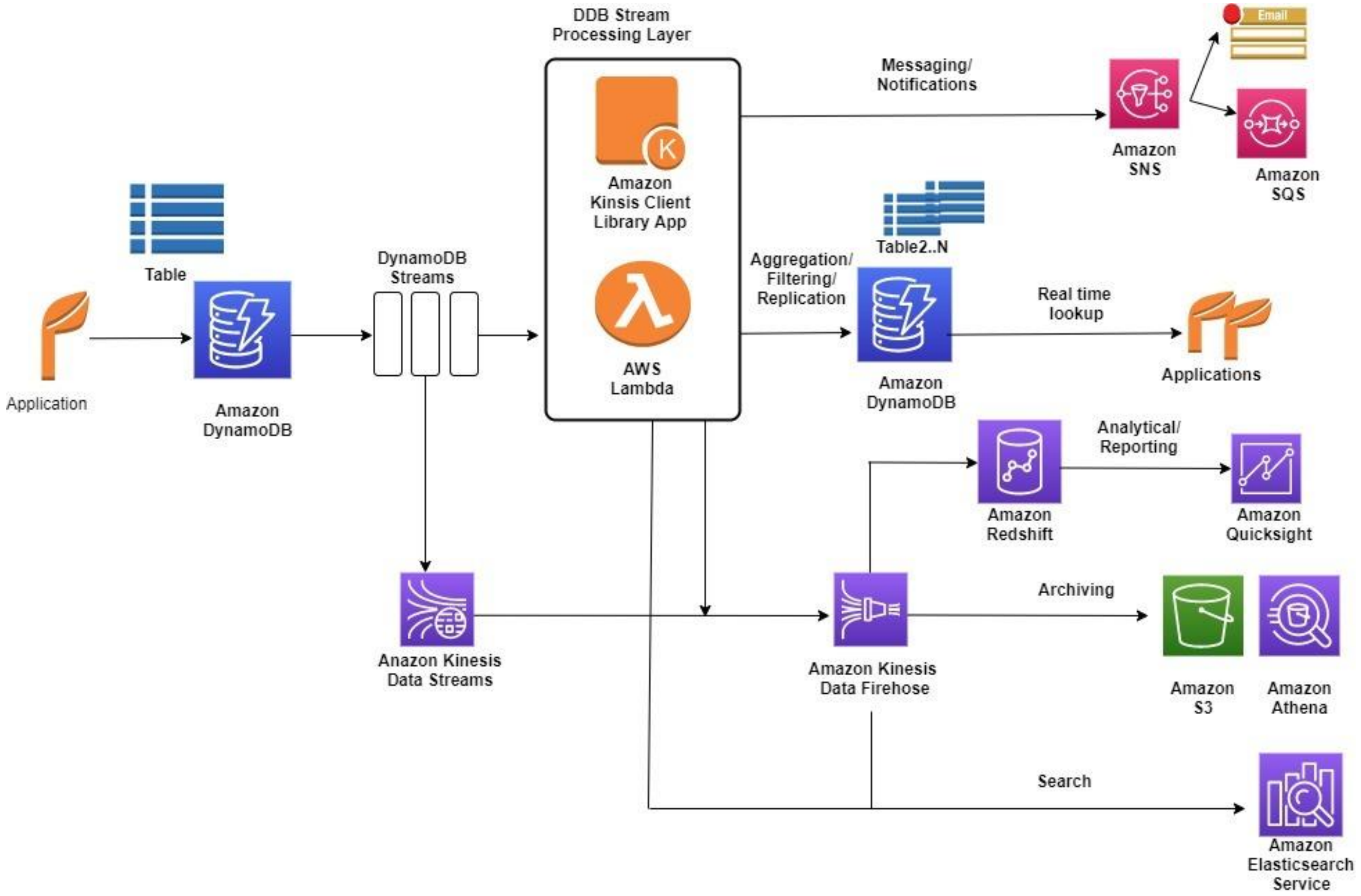
AWS DynamoDB Architecture

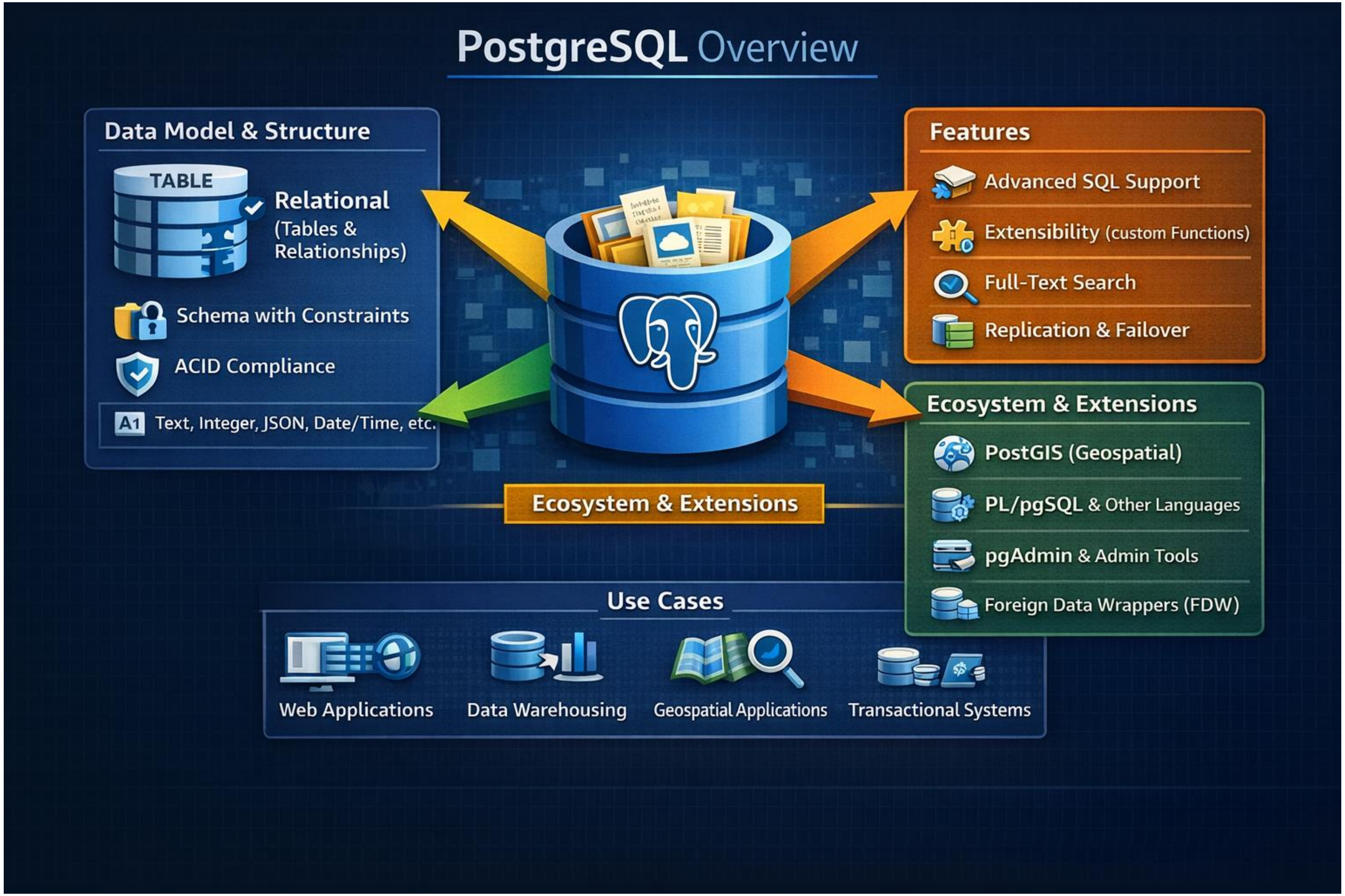


AWS DynamoDB Architecture



DynamoDB Streams Usage pattern





PostgreSQL Overview

Data Model & Structure

- TABLE
- **Relational** (Tables & Relationships)
- Schema with Constraints
- ACID Compliance
- A1 Text, Integer, JSON, Date/Time, etc.

Features

- Advanced SQL Support
- Extensibility (custom Functions)
- Full-Text Search
- Replication & Failover

Ecosystem & Extensions

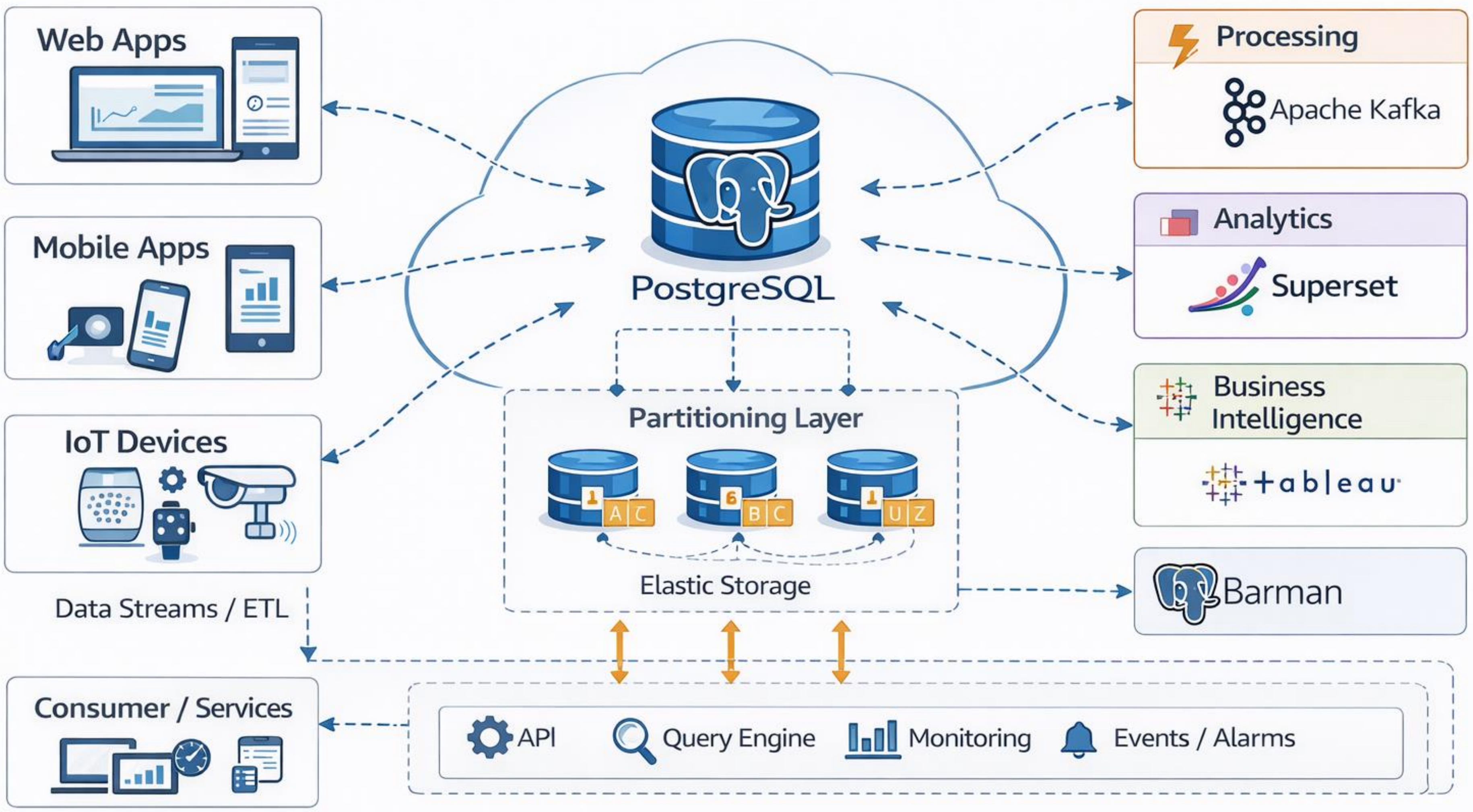
- PostGIS (Geospatial)
- PL/pgSQL & Other Languages
- pgAdmin & Admin Tools
- Foreign Data Wrappers (FDW)

Ecosystem & Extensions

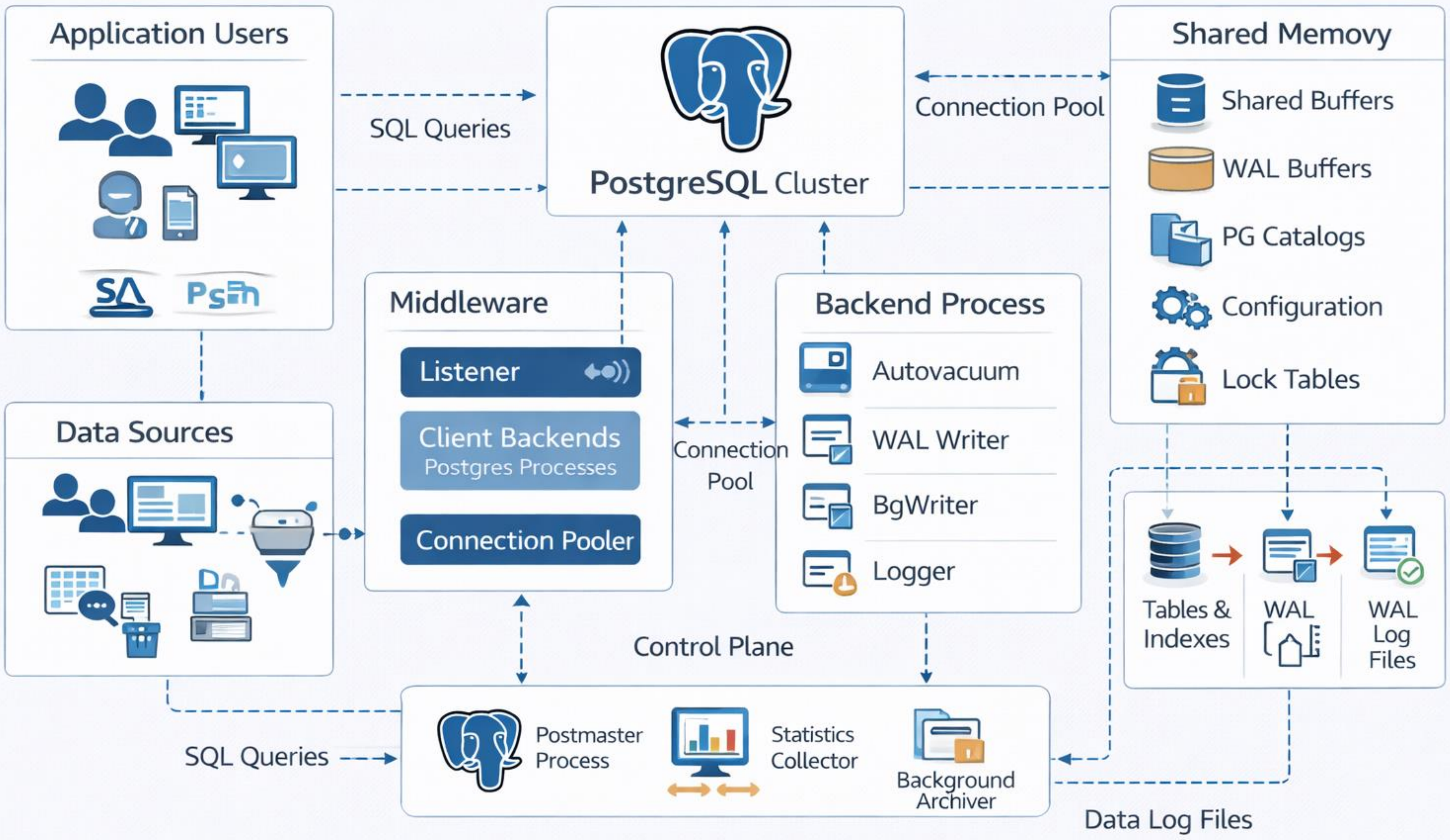
Use Cases

- Web Applications
- Data Warehousing
- Geospatial Applications
- Transactional Systems

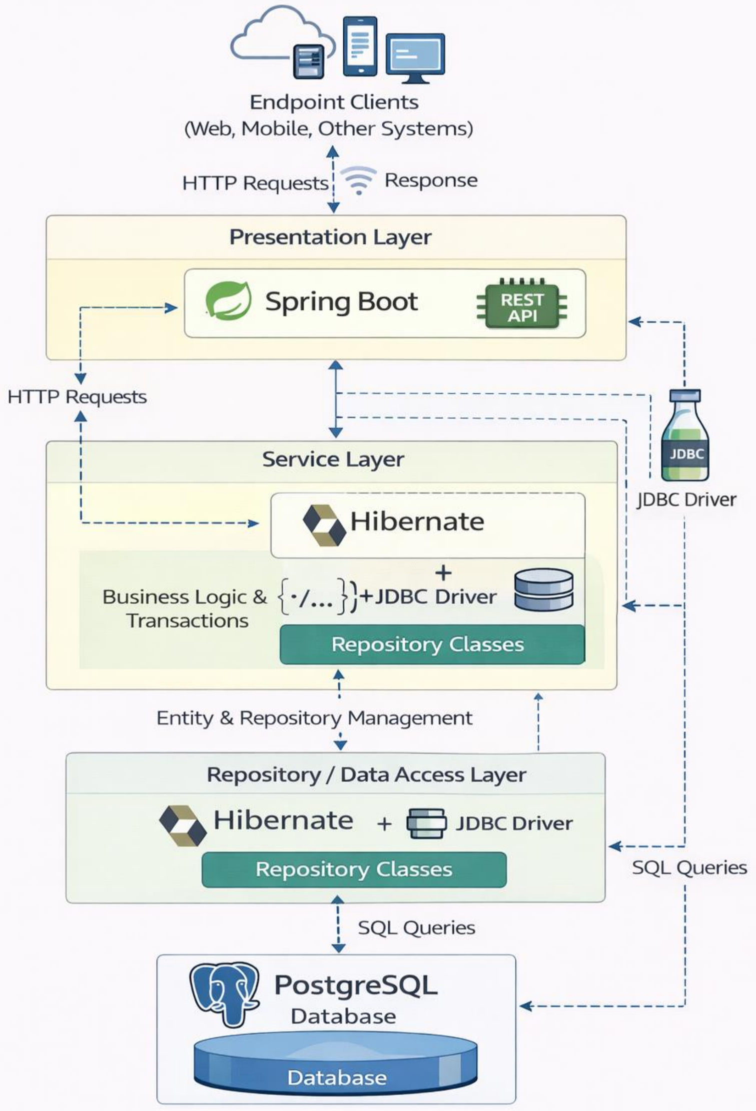
Typical PostgreSQL Integration Into Global Process



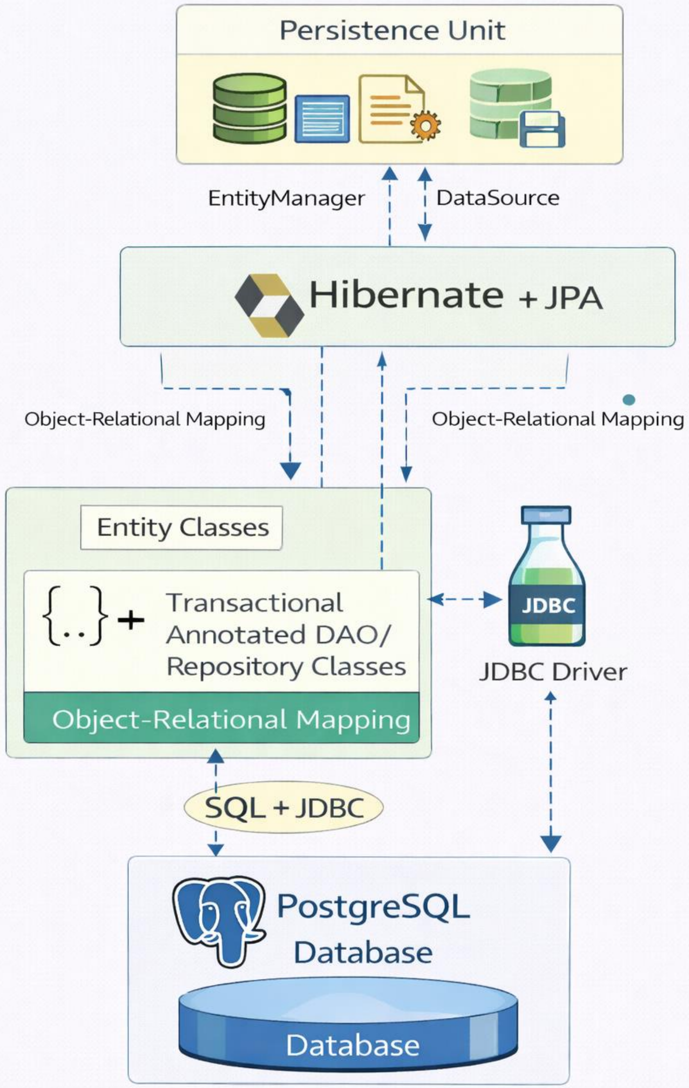
PostgreSQL Internal Architecture



Postgres Integration into Java REST Application in Spring Boot using JDBC, Hibernate, and JPA



The Interaction of JPA, JDBC, and *Hibernate*



Some words on transactions and async

- Annotating a method with **@Transactional** wraps the whole code inside a transaction
  - Isolation-Level can be configured (READ_COMMITED, etc) -> important for certain logical behaviour

- Annotating a method with **@Async** wraps the whole code in an async executed block. A CompletableFuture<> should be returned
  - If truly async, then a correlation id is passed to the caller which can be used to retrieve the request later
  - Otherwise use .join() or .get() to retrieve the result

.join() or .get() for CompletableFuture?

- **Use .join() in most cases** - it's cleaner for functional programming and doesn't require checked exception handling.

  Use .get() when you need fine-grained control over timeouts or when you explicitly want to handle InterruptedException and ExecutionException separately.

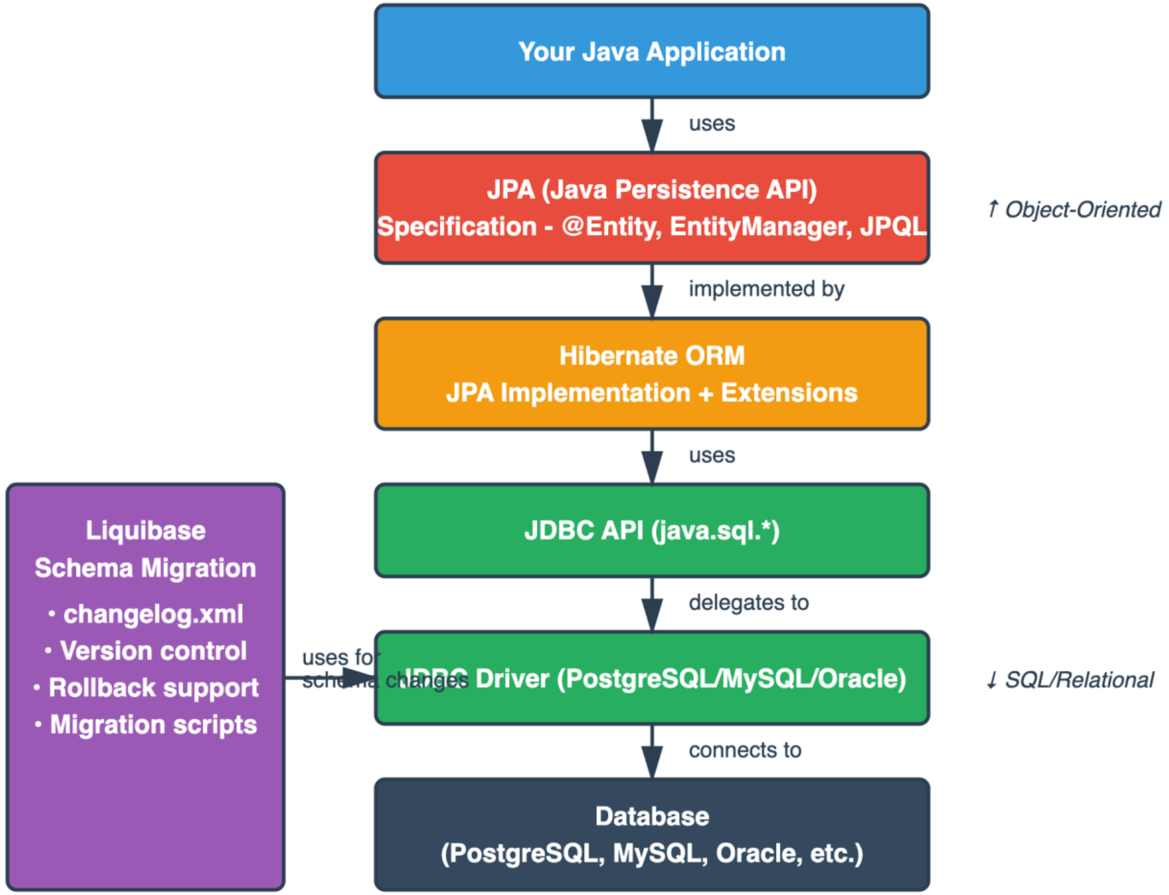
CompletableFuture() can be cool for code as well

```
public ResponseEntity<List<String>> getParallelInLoop() {
    List<String> ids = List.of("1", "2", "3", "4", "5");

    List<CompletableFuture<String>> futures =
            ids
            .stream()
            .map(id -> CompletableFuture.supplyAsync(() -> { return id + " : " +
               asyncService.asyncMethod().join();}))
            .toList(); // Wait for all

    // Wait for all and collect results - clean with .join()
    return
ResponseEntity.ok(futures.stream().map(CompletableFuture::join)
.collect(Collectors.toList()));
}
```

JPA, JDBC, Hibernate and Liquibase overview



JPA (Java Persistence API)

- **What it is:** A specification that defines how Java objects should be persisted to relational databases.
- **Key features:**
  - Annotations: @Entity, @Table, @Id, @Column, @OneToMany, etc.
  - EntityManager API for CRUD operations
  - JPQL (Java Persistence Query Language)
  - Transaction management
- **Role:** Provides a standard interface so you can switch between different ORM implementations without changing your code.

Hibernate

- **What it is:** The most popular implementation of the JPA specification (also has its own extensions).
- **Key features:**
  - o Implements all JPA interfaces and annotations
  - o Object-Relational Mapping (ORM) - converts Java objects to database tables
  - o Automatic SQL generation
  - o Caching mechanisms (first-level and second-level cache)
  - o Lazy loading and eager loading strategies
  - o HQL (Hibernate Query Language) - extends JPQL
- **Role:** Does the heavy lifting of translating between object-oriented and relational paradigms.

JDBC Driver

- **What it is:** A database-specific library that implements the JDBC API to connect to a particular database.
- **Examples:**
  - PostgreSQL JDBC Driver (org.postgresql:postgresql)
  - MySQL Connector/J (mysql:mysql-connector-java)
  - Oracle JDBC Driver (ojdbc)
  - H2 Database Driver (com.h2database:h2)
- **Role:** Provides the low-level communication protocol between Java and the specific database. Handles network communication, SQL execution, and result set processing.

Liquibase

- **What it is:** A database schema version control and migration tool.
- **Key features:**
  - Tracks database changes using changelog files (XML, YAML, JSON, or SQL)
  - Applies incremental database changes across environments
  - Supports rollback of changes
  - Maintains DATABASECHANGELOG table to track applied changes
  - Works independently of JPA/Hibernate
- **Role:** Manages the database schema structure, while JPA/Hibernate manages the data within that structure.

Behind the scenes

- **JPA** provides the @Entity annotation and EntityManager interface you use
- **Hibernate** intercepts the persist() call, generates the SQL INSERT statement
- **Hibernate** passes the SQL to the JDBC API
- **JDBC Driver** sends the SQL over the network to the database
- **Database** executes the INSERT and returns the generated ID
- **JDBC Driver** receives the result
- **Hibernate** updates the User object with the new ID

Key takeaways

- **Layered Architecture:** Each component operates at a different level of abstraction
- **JPA is the standard, Hibernate is the implementation:** You code against JPA interfaces, Hibernate does the work
- **JDBC is the foundation:** Everything ultimately goes through JDBC to reach the database
- **Liquibase handles structure, JPA handles data:** They work in parallel - Liquibase manages schema evolution, JPA manages data operations
- **Separation of Concerns:** You can swap Hibernate for another JPA provider (EclipseLink, OpenJPA) or change databases (PostgreSQL → MySQL) with minimal code changes

So, what do I need?

- Mostly JPA for standard applications
  - JPQL might come in handy
- Sometimes JDBC for special things