System Design

Building Robust Architectures from the Ground Up

Saravuth R. (Orm) - Platform Lead, Mini App Open Platform

Outline

Introduction to System Design

Understanding Client-Server Architecture

Introduction to Client-Server-Database Interaction

Load Balancing Techniques

Disaster Recovery and High Availability

Monolith vs. Microservice Architecture

Introduction to CAP Theorem

Database Type and its use case

When a Database Reaches Its Limit

Caching Strategies in System Design

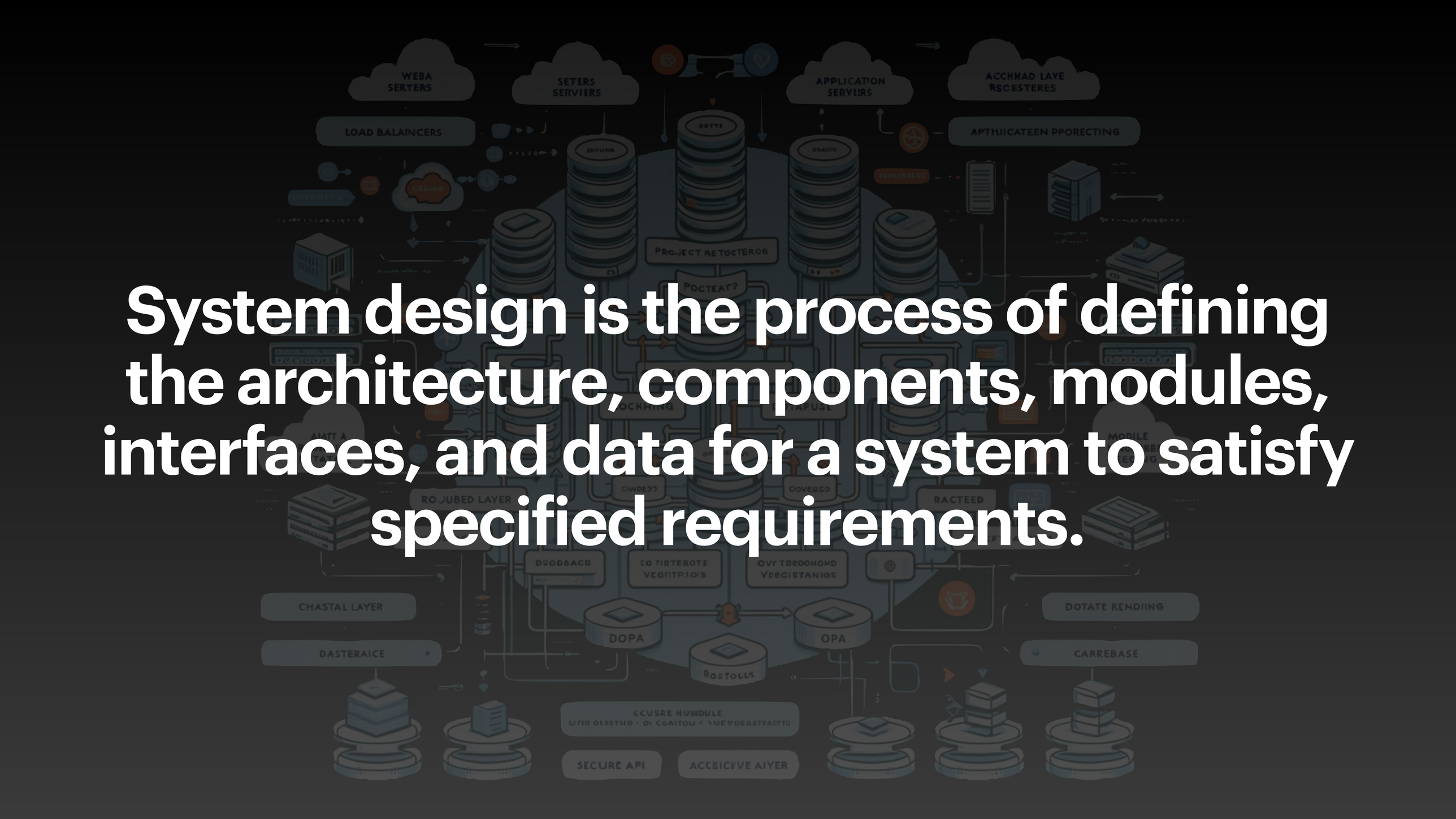
Event-Driven Architecture and CQRS

Database Transactions and Distributed Transactions

Introduction to Saga Pattern

Orchestrator vs. Choreography in Microservices

API Gateway and Its Role



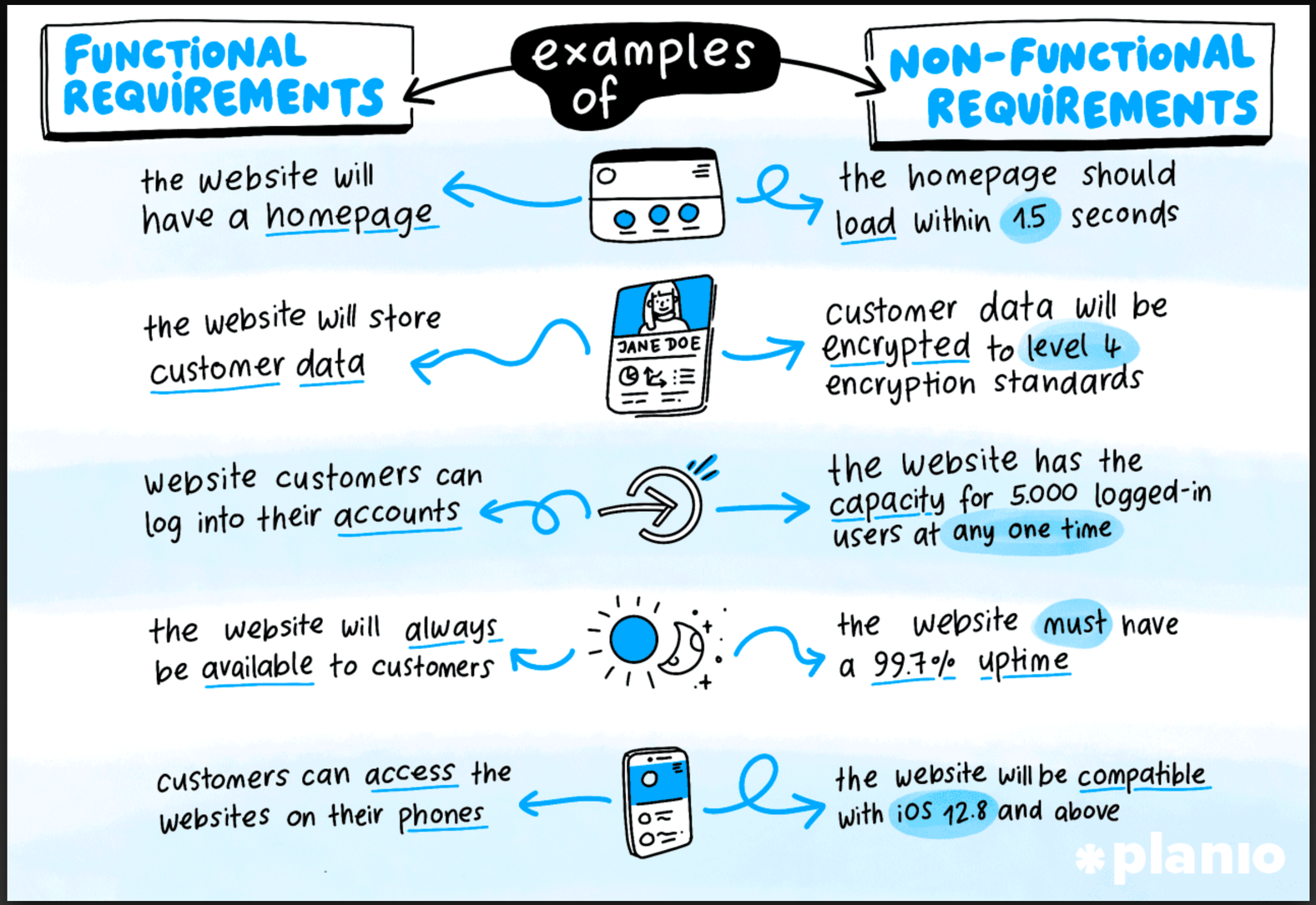
System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.

Satisfy specified requirements

What are those requirements ?

- **Performance**: Ensuring the system meets performance benchmarks and response time expectations.
- **Scalability**: Designing the system to handle growth in users or data without compromising performance.
- **Security**: Incorporating measures to protect against vulnerabilities and unauthorized access.
- **User Experience**: Focusing on intuitive interfaces and efficient workflows.
- **Reliability**: Building the system to operate consistently without unexpected failures.

Example of Non-Functional Requirements

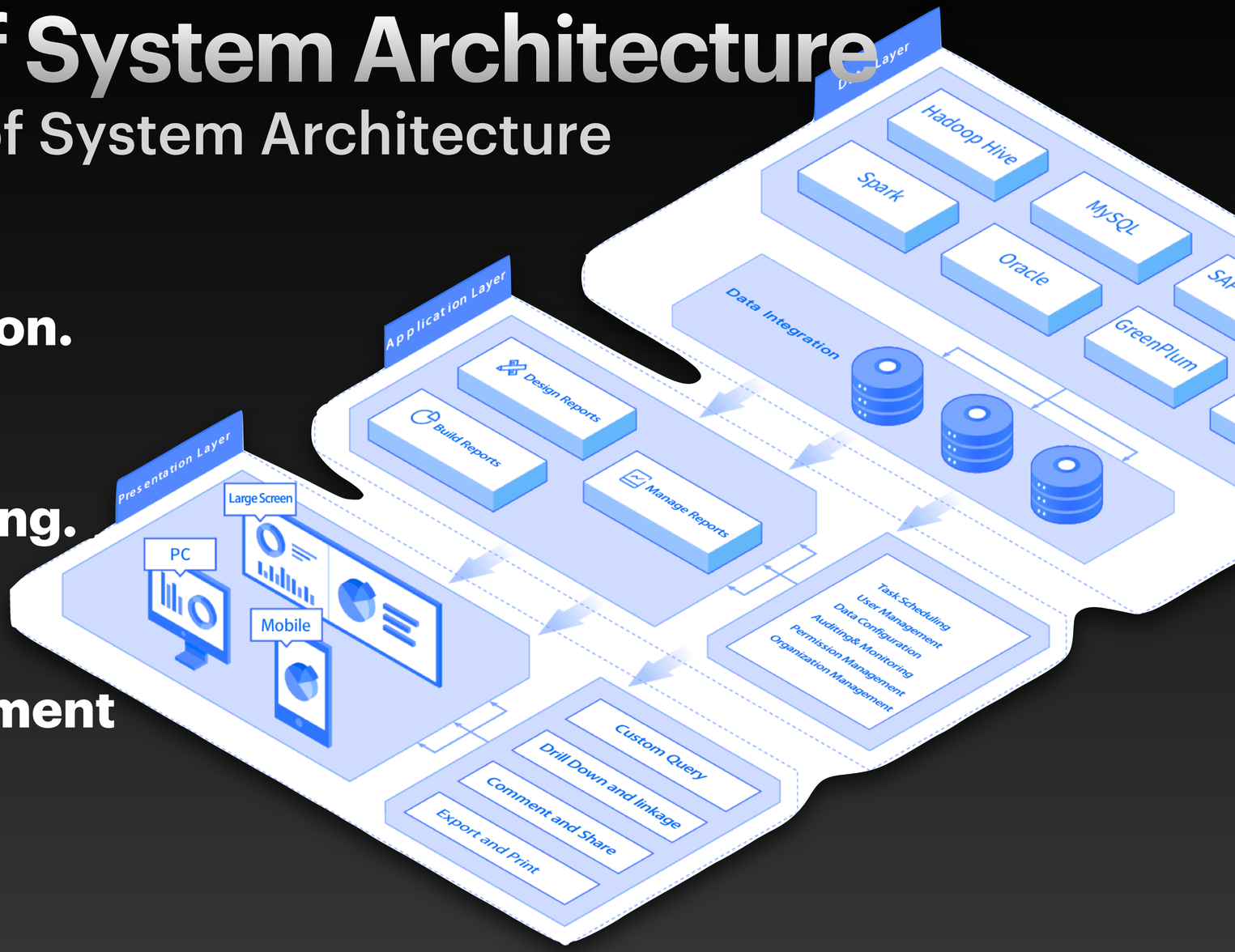


Now, Let's begin

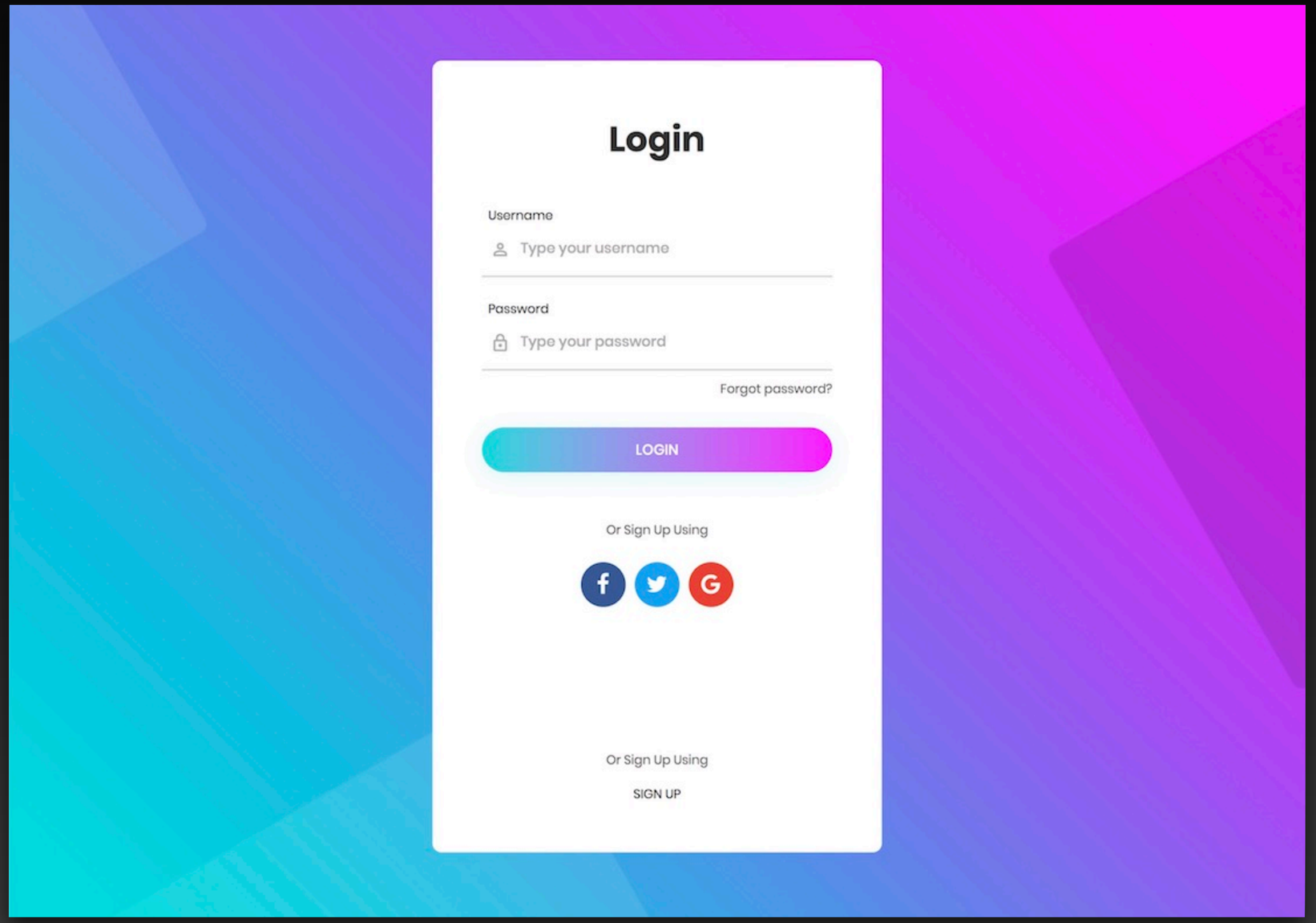
Overview of System Architecture

Layers of System Architecture

- **Presentation Layer:**
  - **User interface and interaction.**
- **Application Layer:**
  - **Business logic and processing.**
- **Data Storage Layer:**
  - **Database and data management**



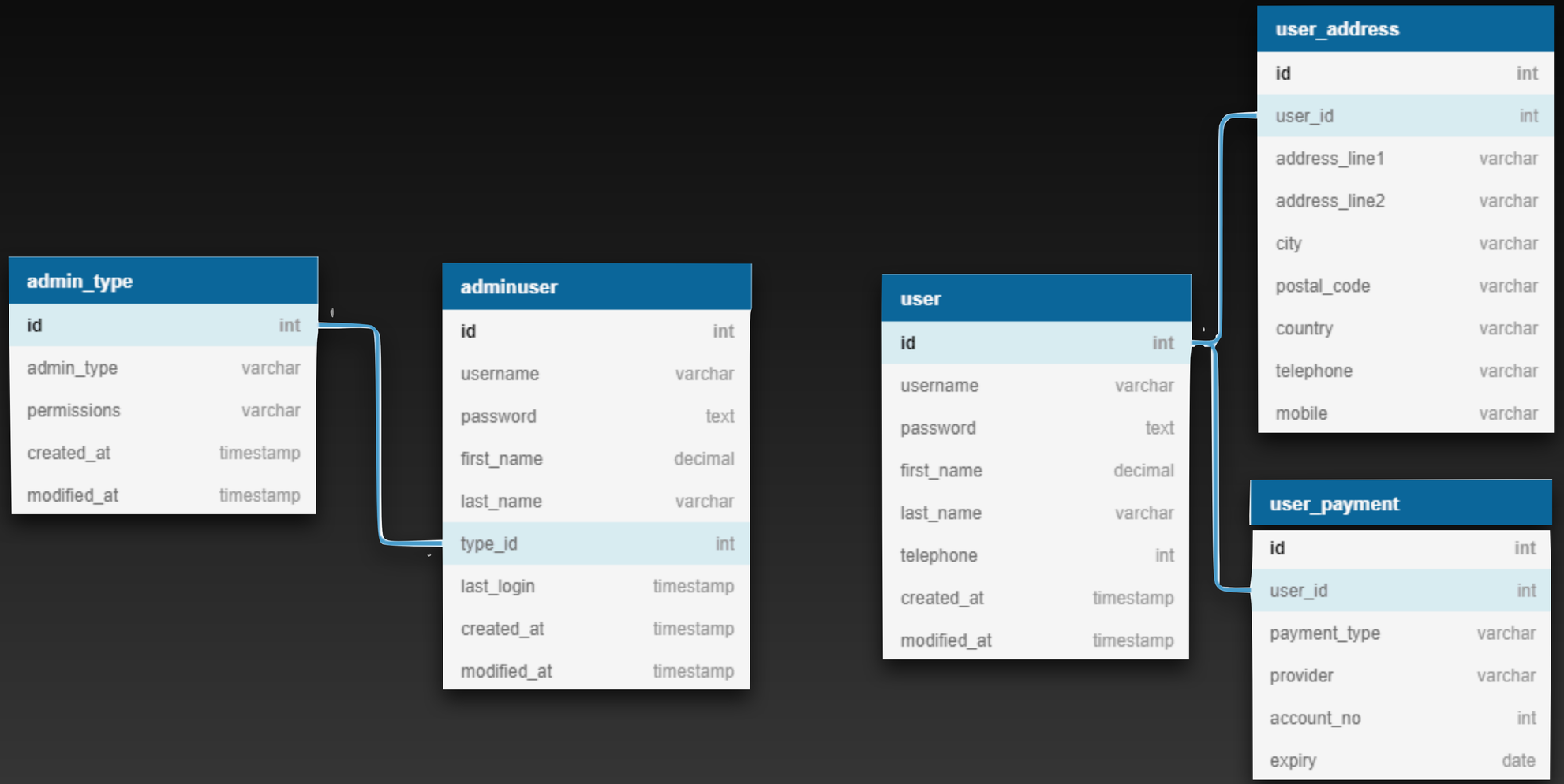
Presentation Layer: User interface and interaction.



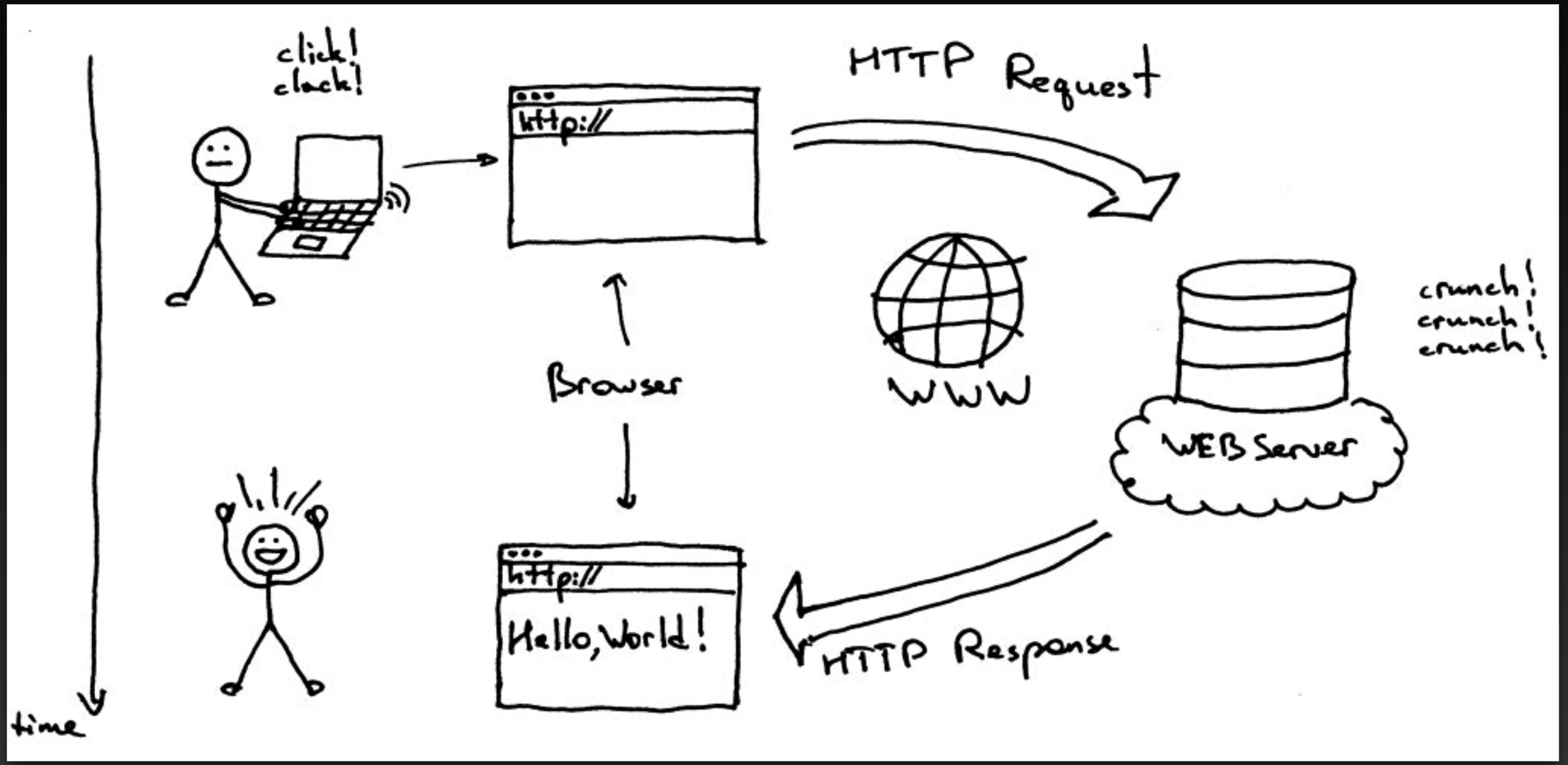
Application Layer: Business logic and processing.

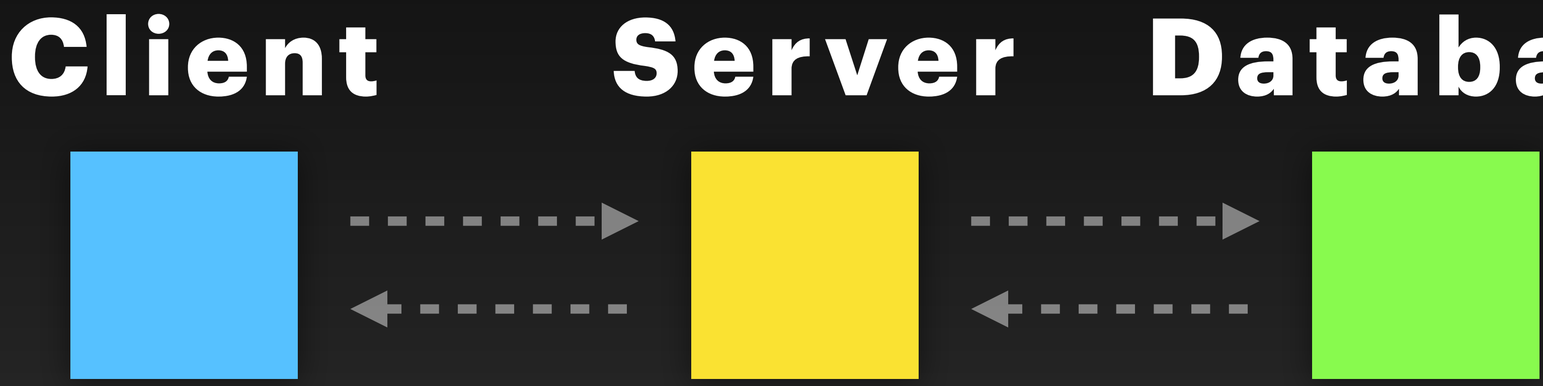
```
bhatara.c, 3 months ago | 1 author (bhatara.c)
type loginPartnerHandler struct {
    loginpartner corepartner.LoginPartnerUserFunc
}

func NewLoginPartnerHandler(loginpartner corepartner.LoginPartnerUserFunc) *loginPartnerHandler {
    return &loginPartnerHandler{
        loginpartner,
    }
}

func (h *loginPartnerHandler) LoginPartnerUser(c echo.Context) error {      bhatara.c, 3 months ago • add logout and login handler
    authHeader := c.Request().Header.Get("Authorization")

    accessToken, err := utils.ExtractJWTTokenFromHeader(authHeader)
    if err != nil {
        logger.Error(err.Error(), zap.String("tag", "fail to extract JWT token from header"))
        return apiresponse.WriteErrorPortal(c, http.StatusBadRequest, err)
    }

    response, err := h.loginpartner(c.Request().Context(), accessToken)
    if err != nil {
        logger.Error(err.Error(), zap.String("tag", "login partner user user fail"))
        return apiresponse.WriteErrorPortal(c, http.StatusInternalServerError, err)
    }

    return apiresponse.WriteSuccessPortal(c, response)
}
```

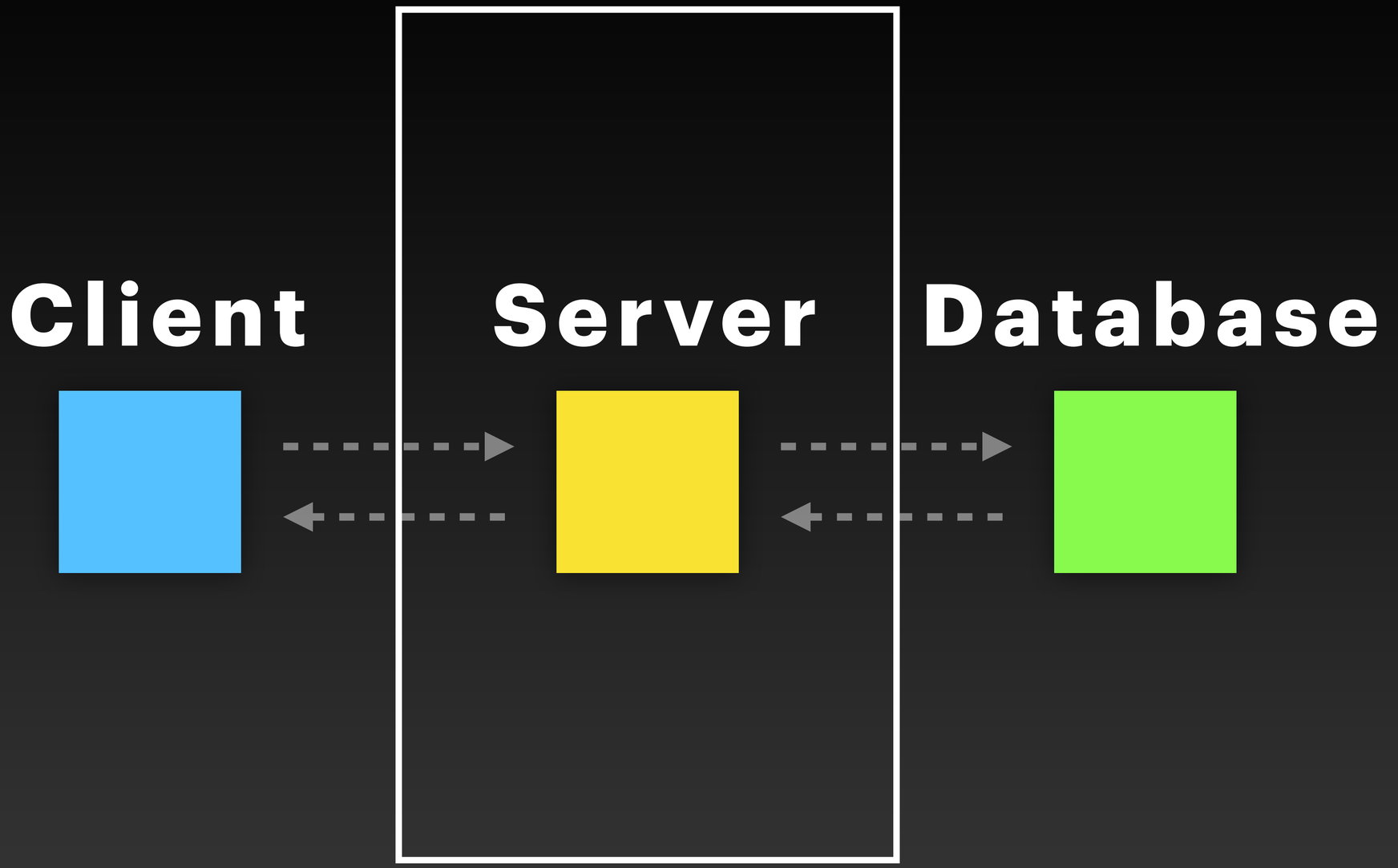
Data Storage Layer: Database and data management.

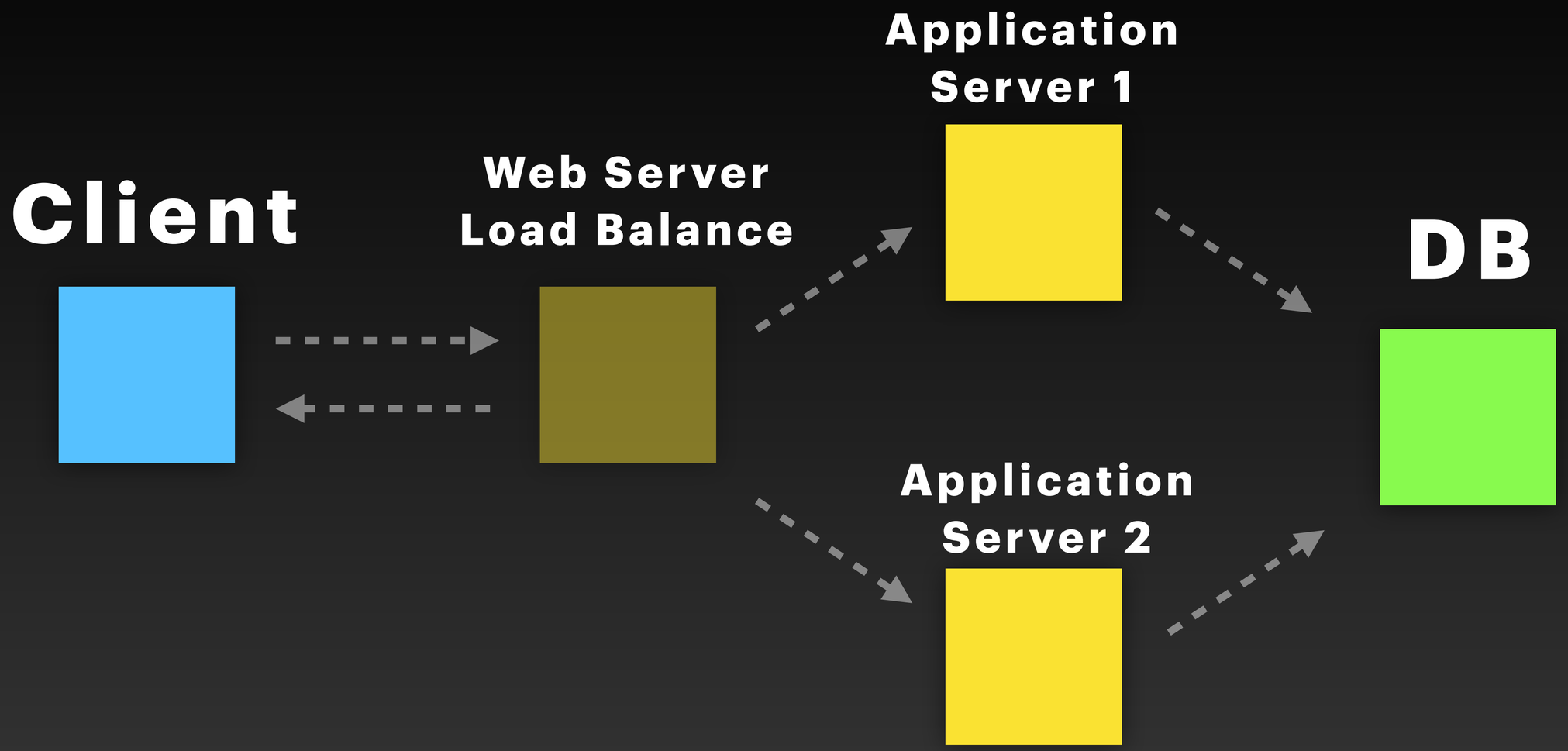


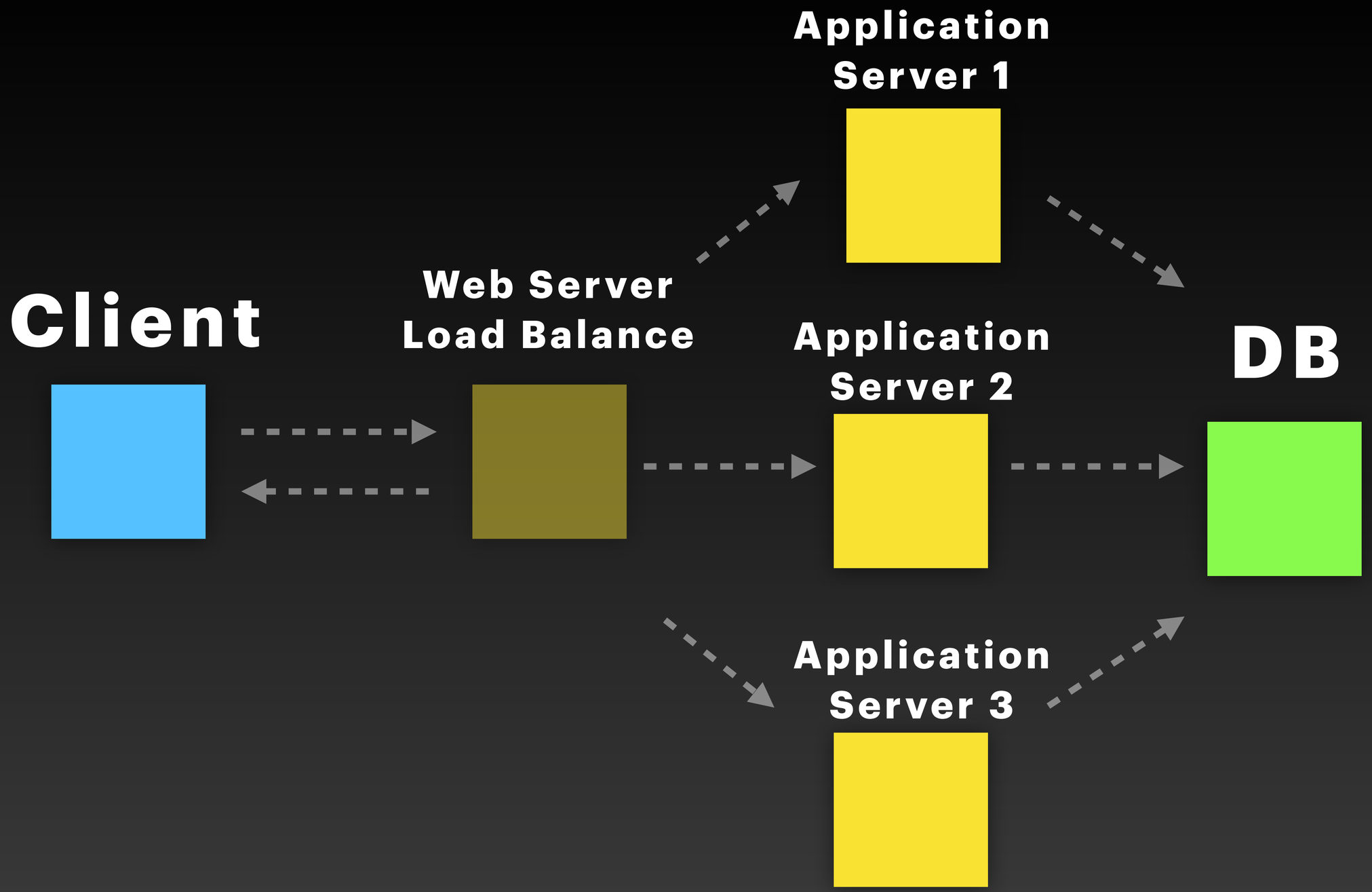
Client - Server



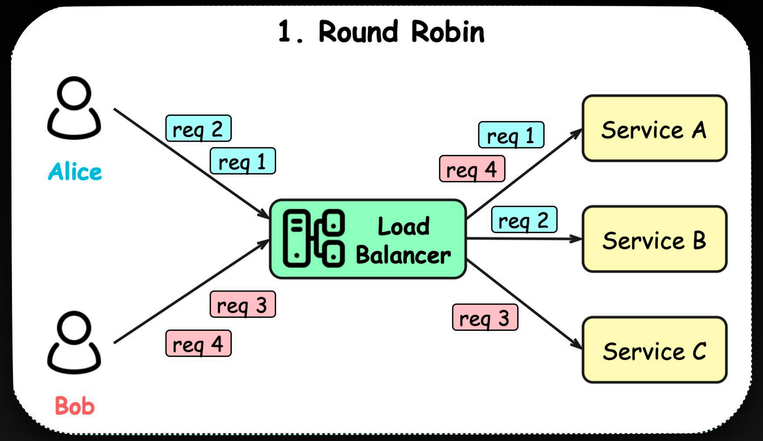

Client
Server
Database

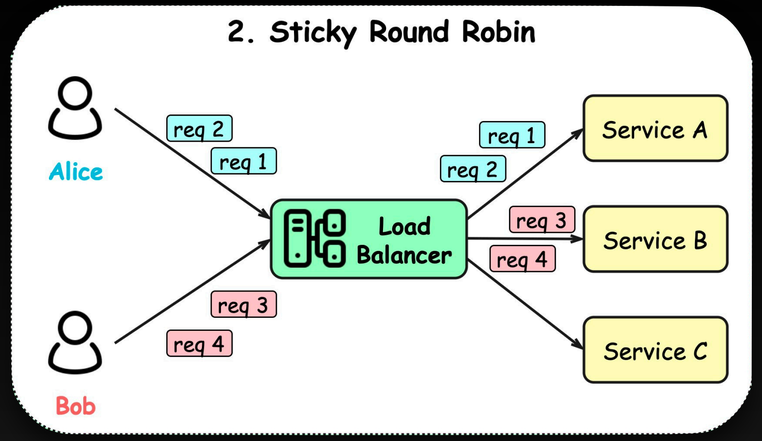

Client
Server
Database

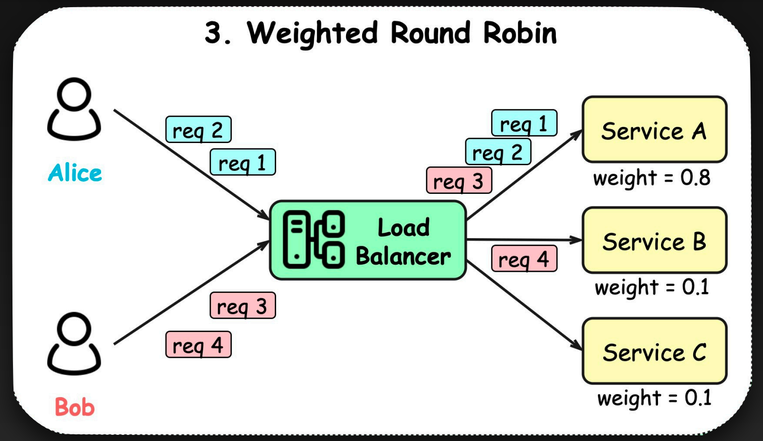

Client
Web Server
Load Balance
Application
Server 1
Application
Server 2
DB

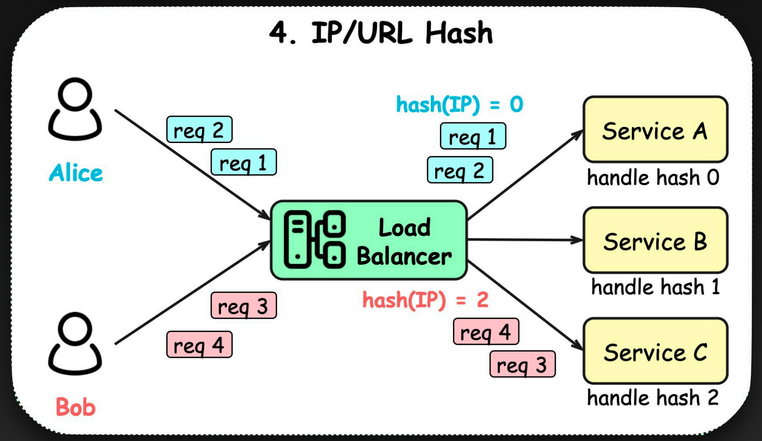

Application
Server 1
Client
Web Server
Load Balance
Application
Server 2
DB
Application
Server 3

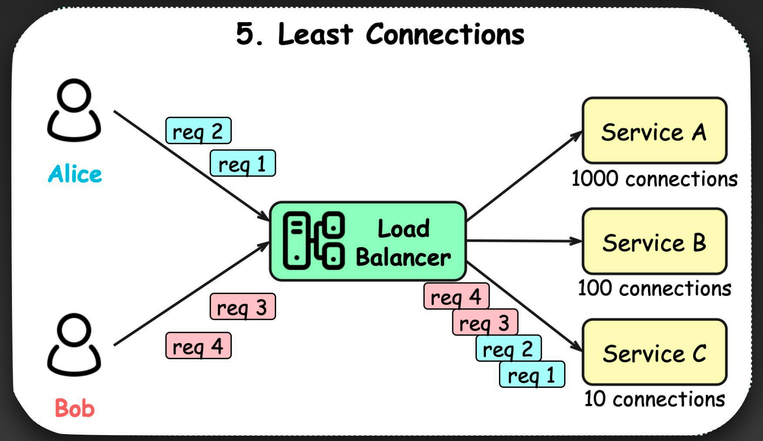
Top 6 Load Balancing Algorithms

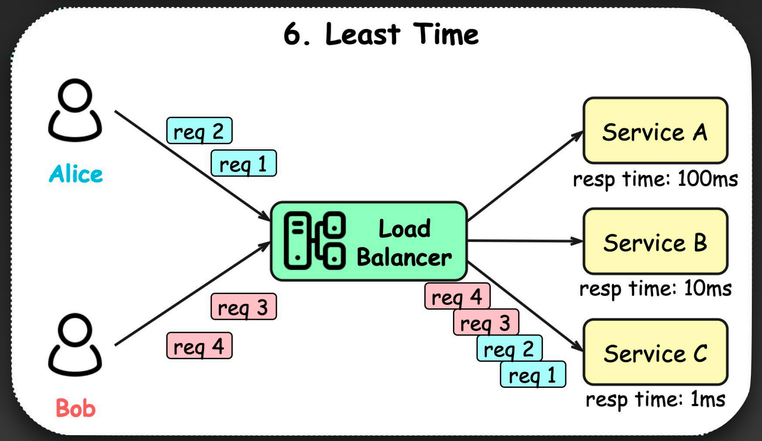




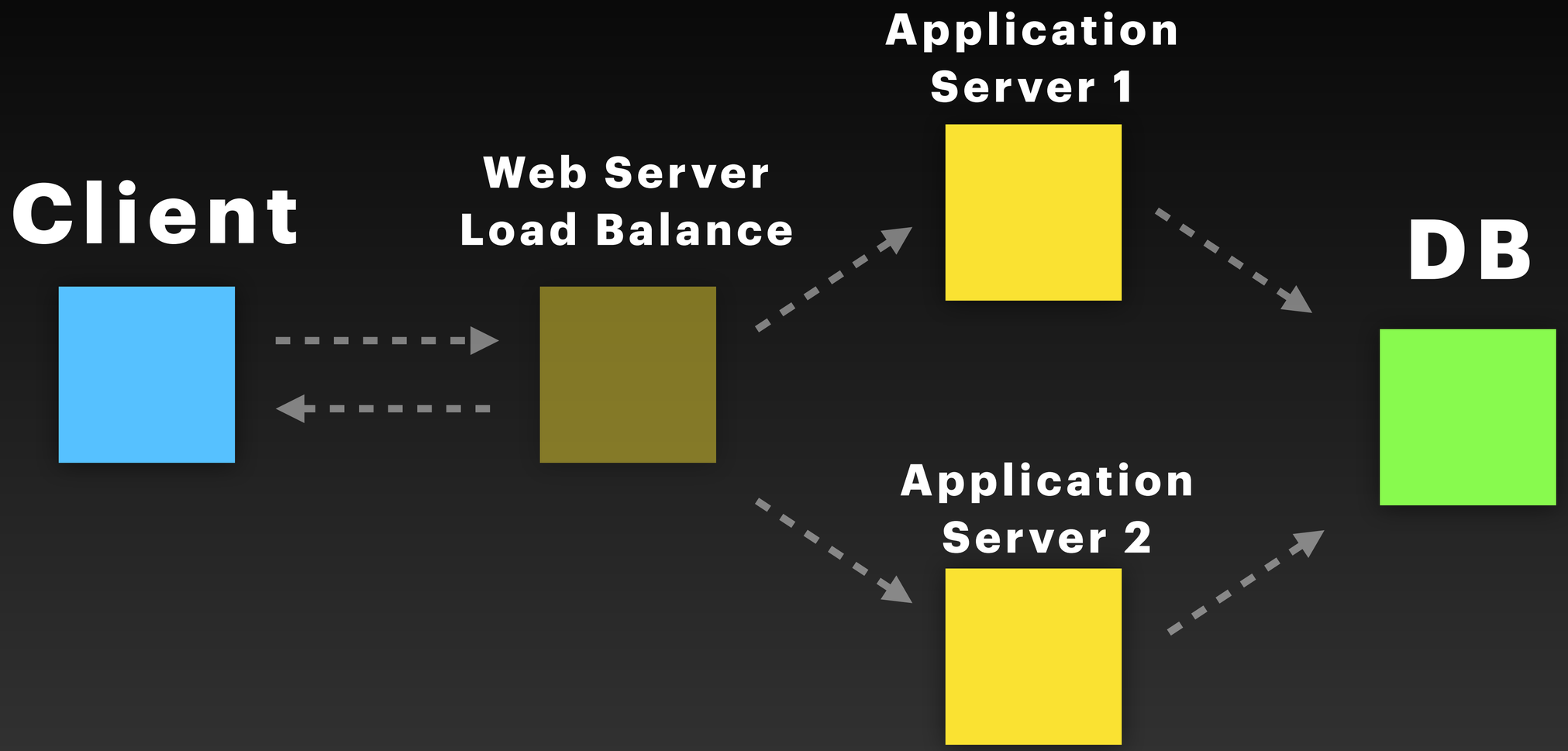


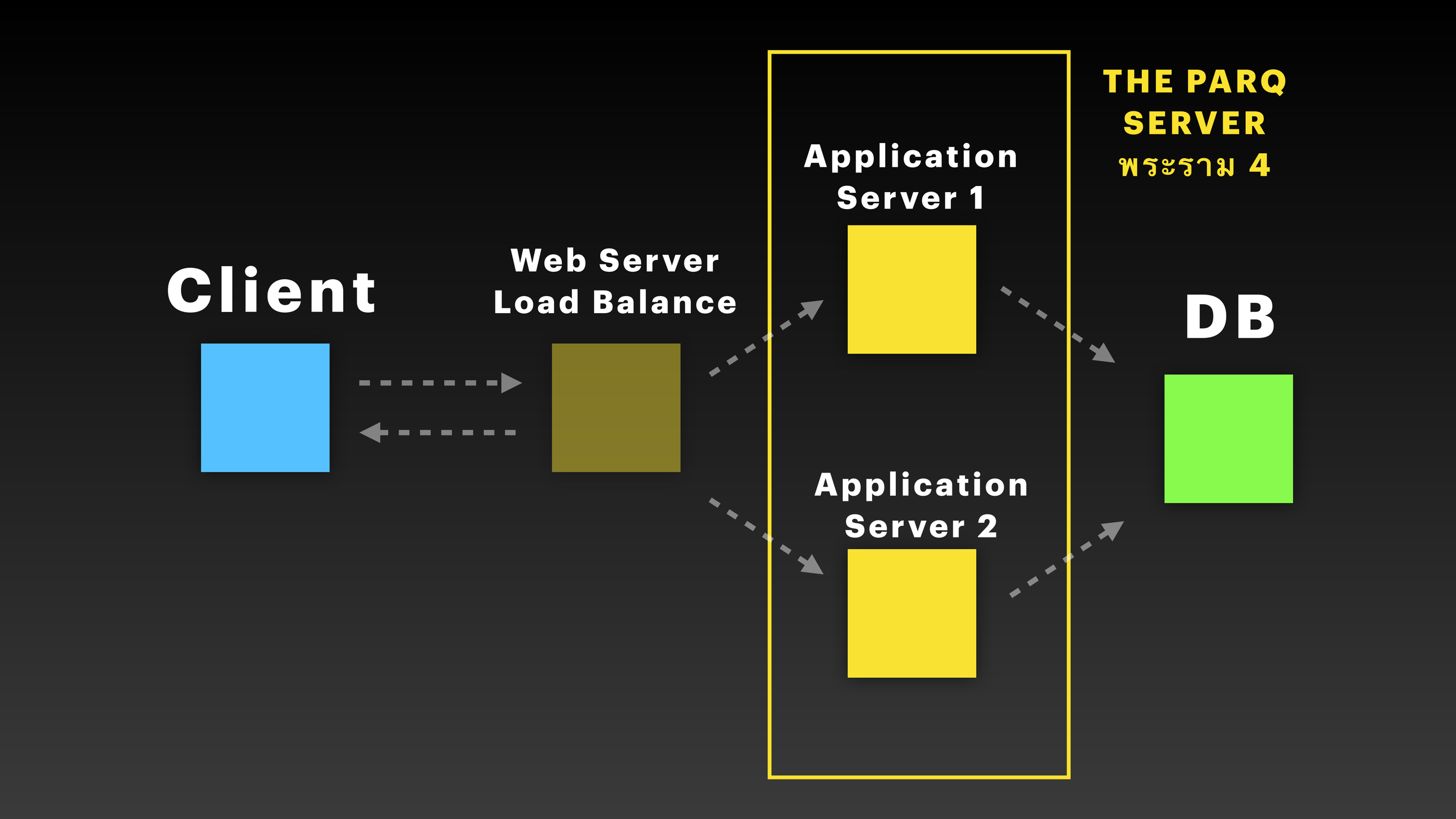


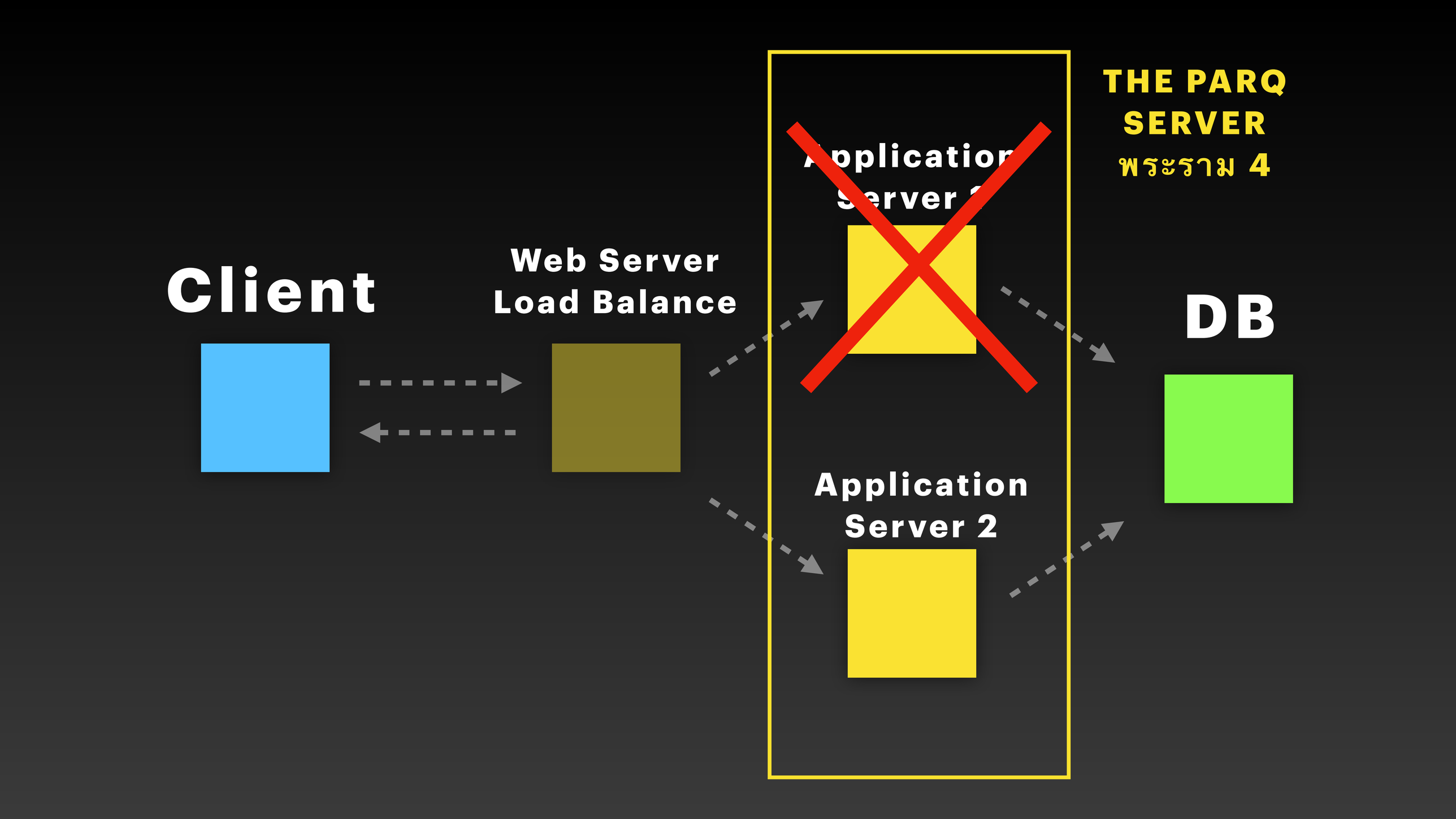


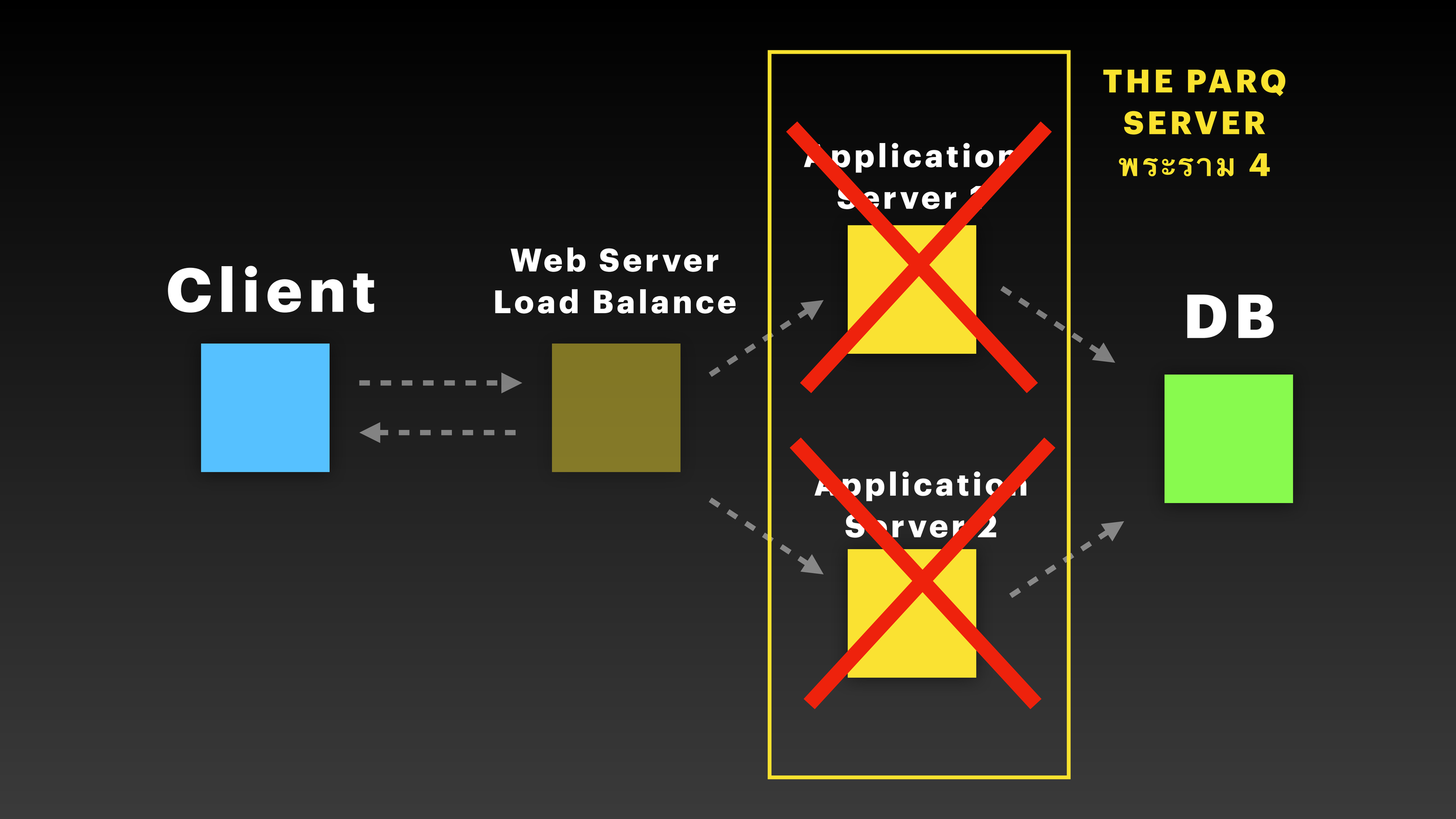


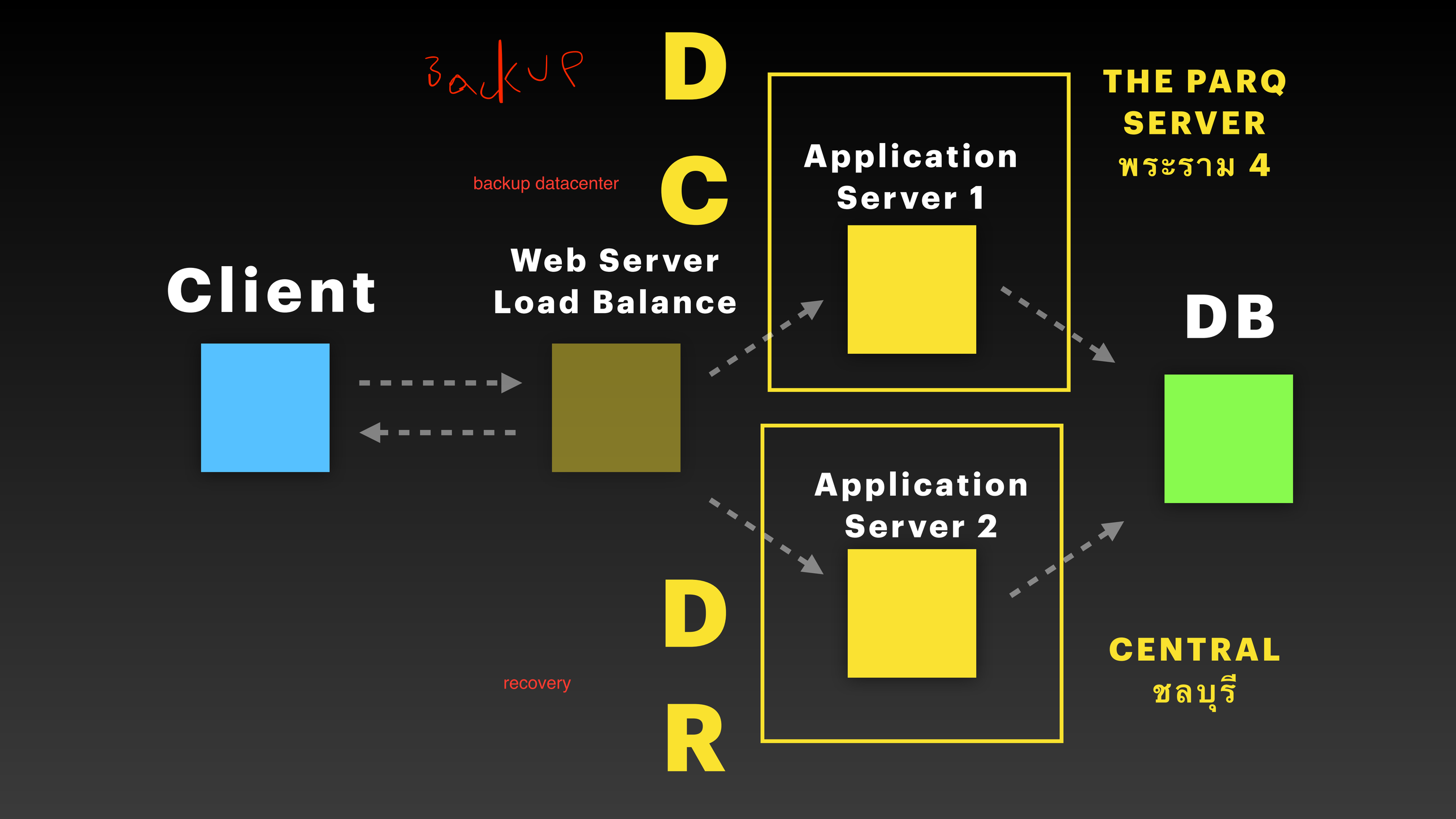
High Availability & Disaster Recovery

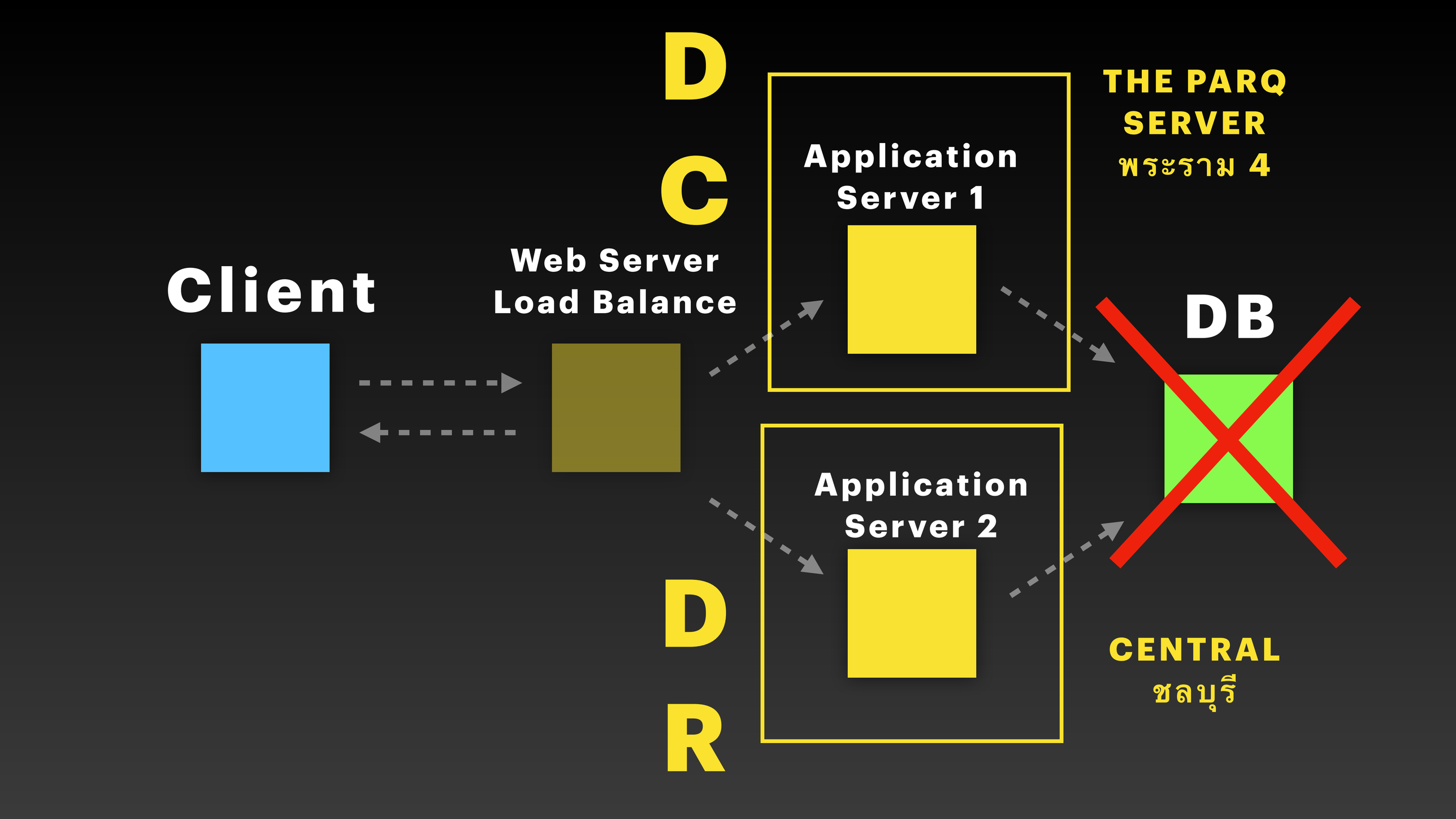

Client
Web Server
Load Balance
Application
Server 1
Application
Server 2
DB


THE PARQ
SERVER
พระราม 4
Client
Web Server
Load Balance
Application
Server 1
Application
Server 2
DB

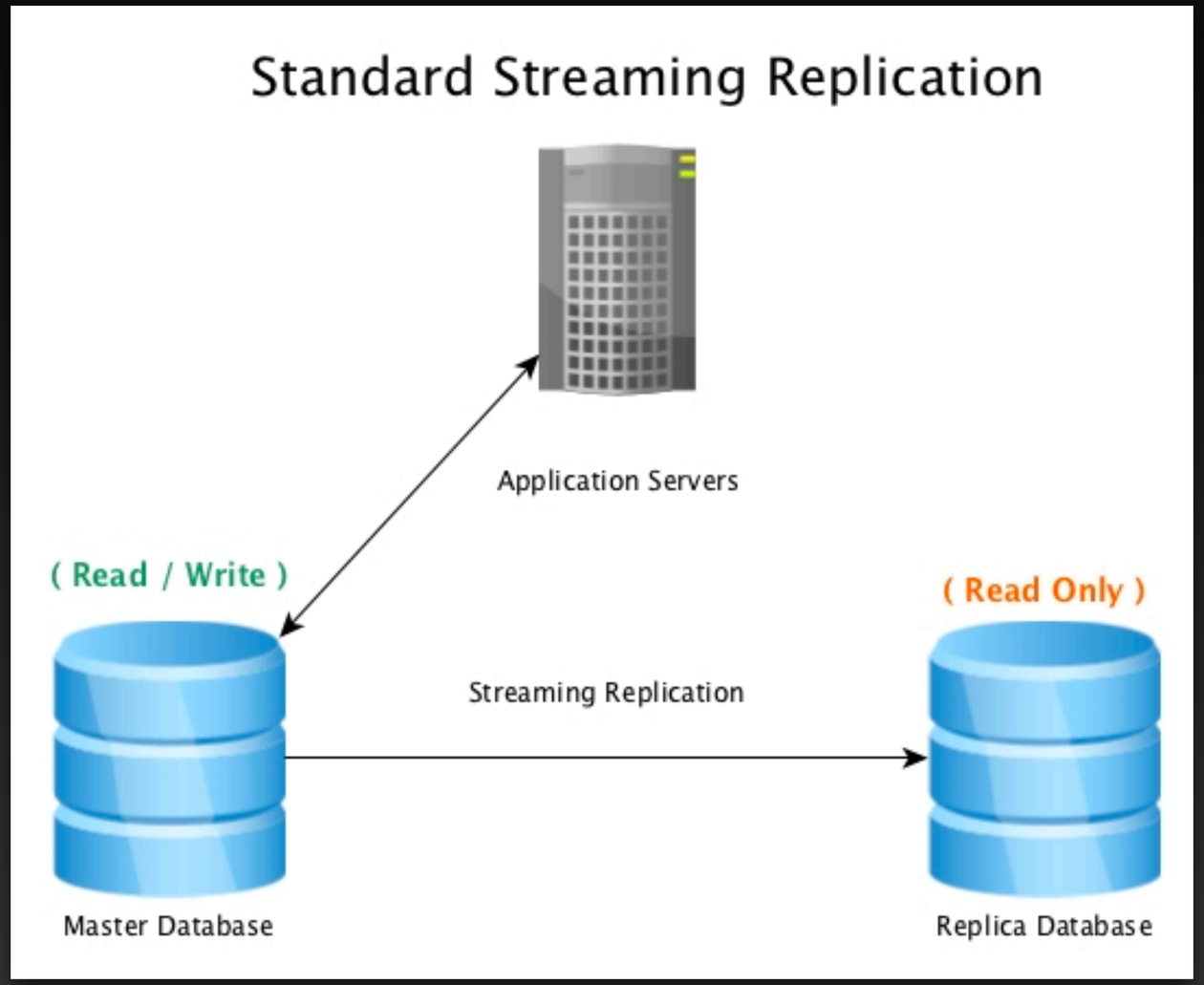

THE PARQ
SERVER
พระราม 4
Client
Web Server
Load Balance
Application
Server 1
Application
Server 2
DB

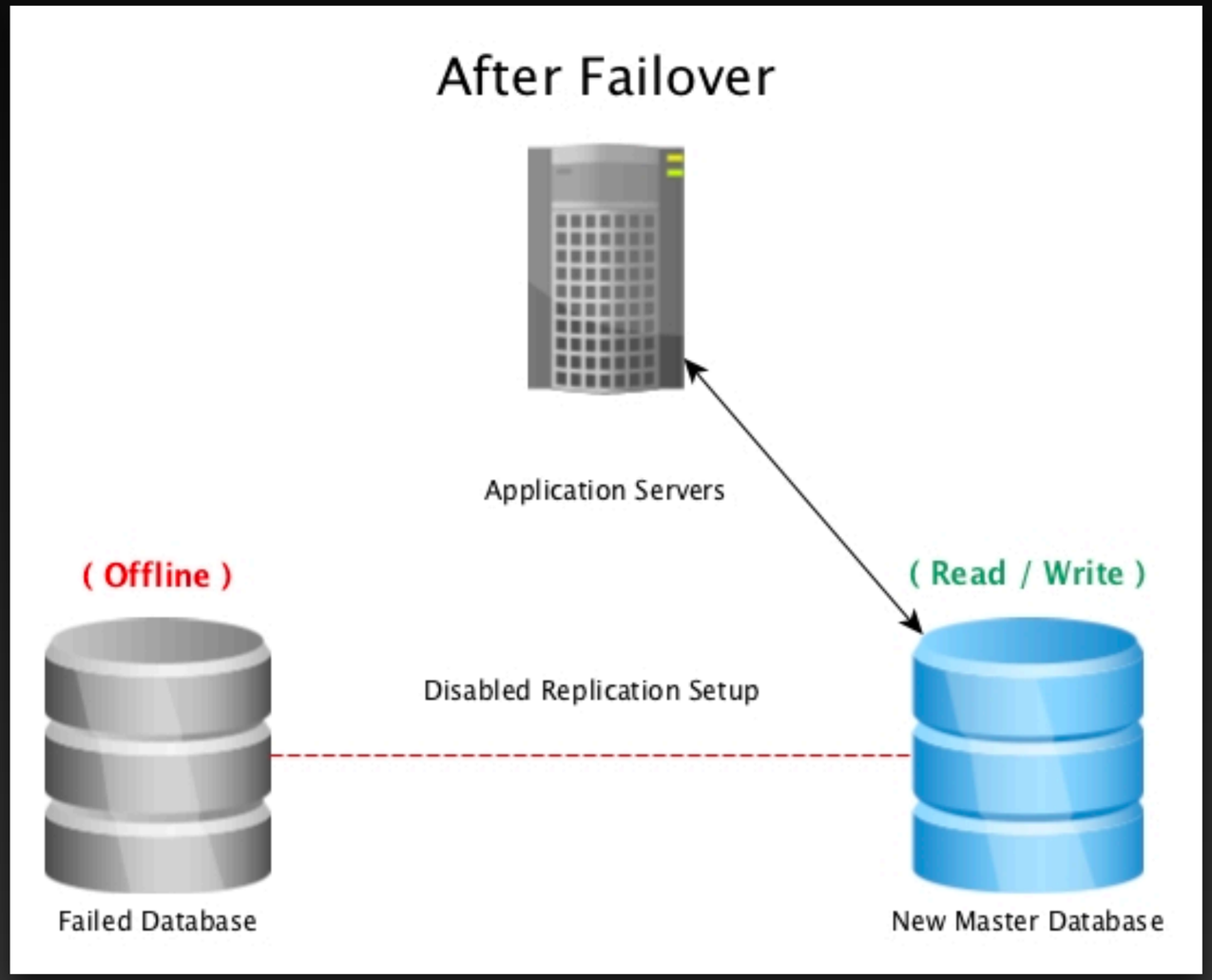

THE PARQ
SERVER
พระราม 4
Client
Web Server
Load Balance
Application
Server 1
Application
Server 2
DB


backup
D
C
backup datacenter
Client
Web Server
Load Balance
Application
Server 1
THE PARQ
SERVER
พระราม 4
DB
Application
Server 2
D
R
recovery
CENTRAL
ชลบุรี


D
C
Client
Web Server
Load Balance
Application
Server 1
THE PARQ
SERVER
พระราม 4
DB
Application
Server 2
D
R
CENTRAL
ชลบุรี

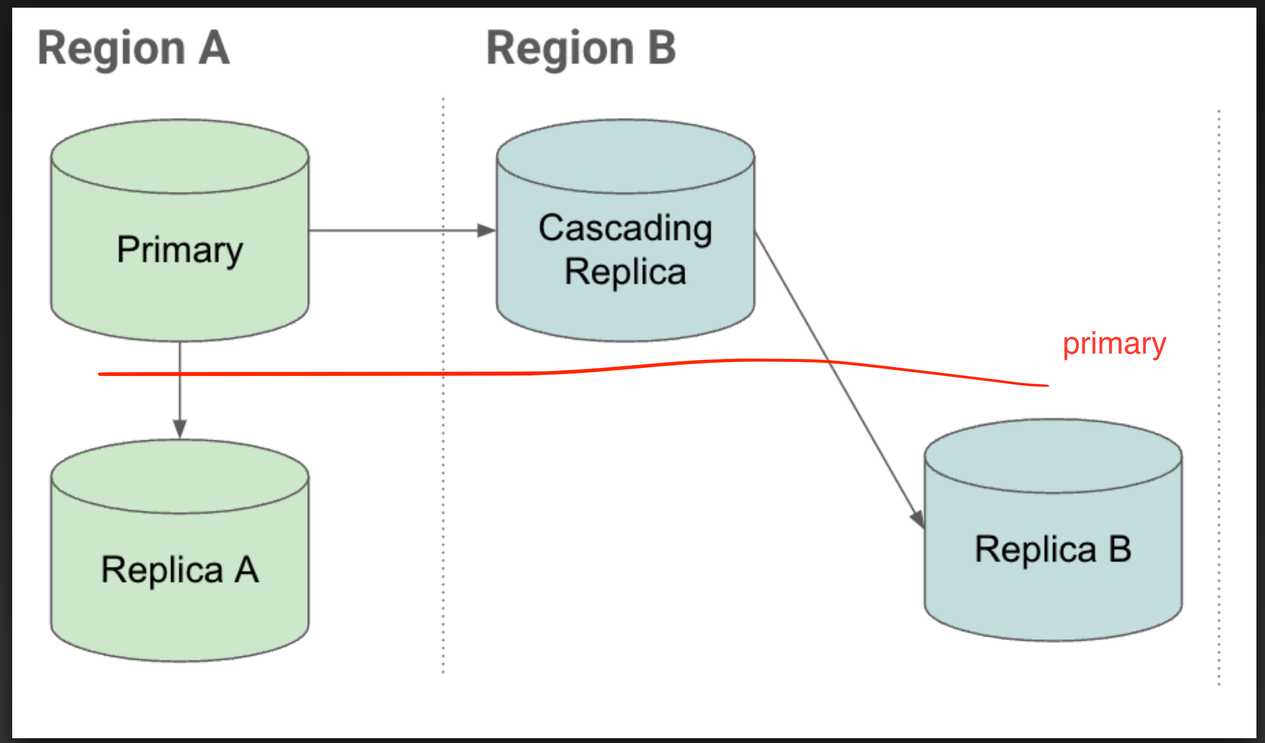
Database Replication

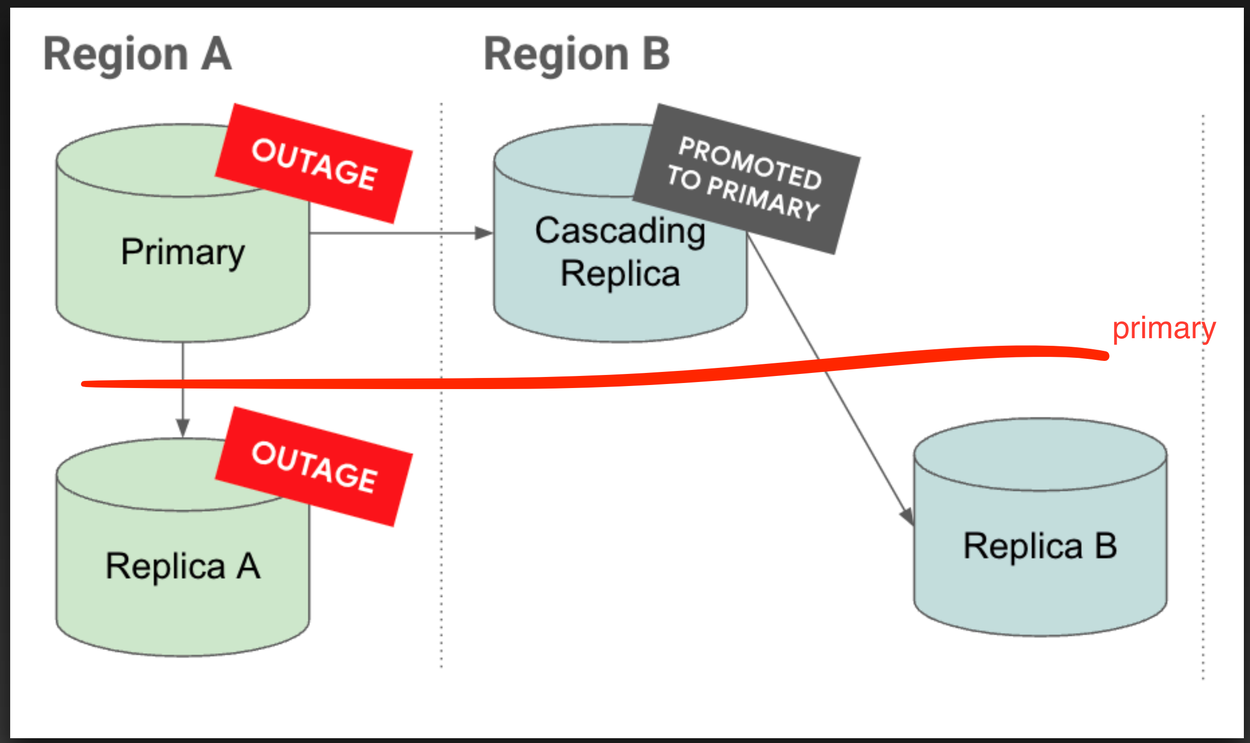


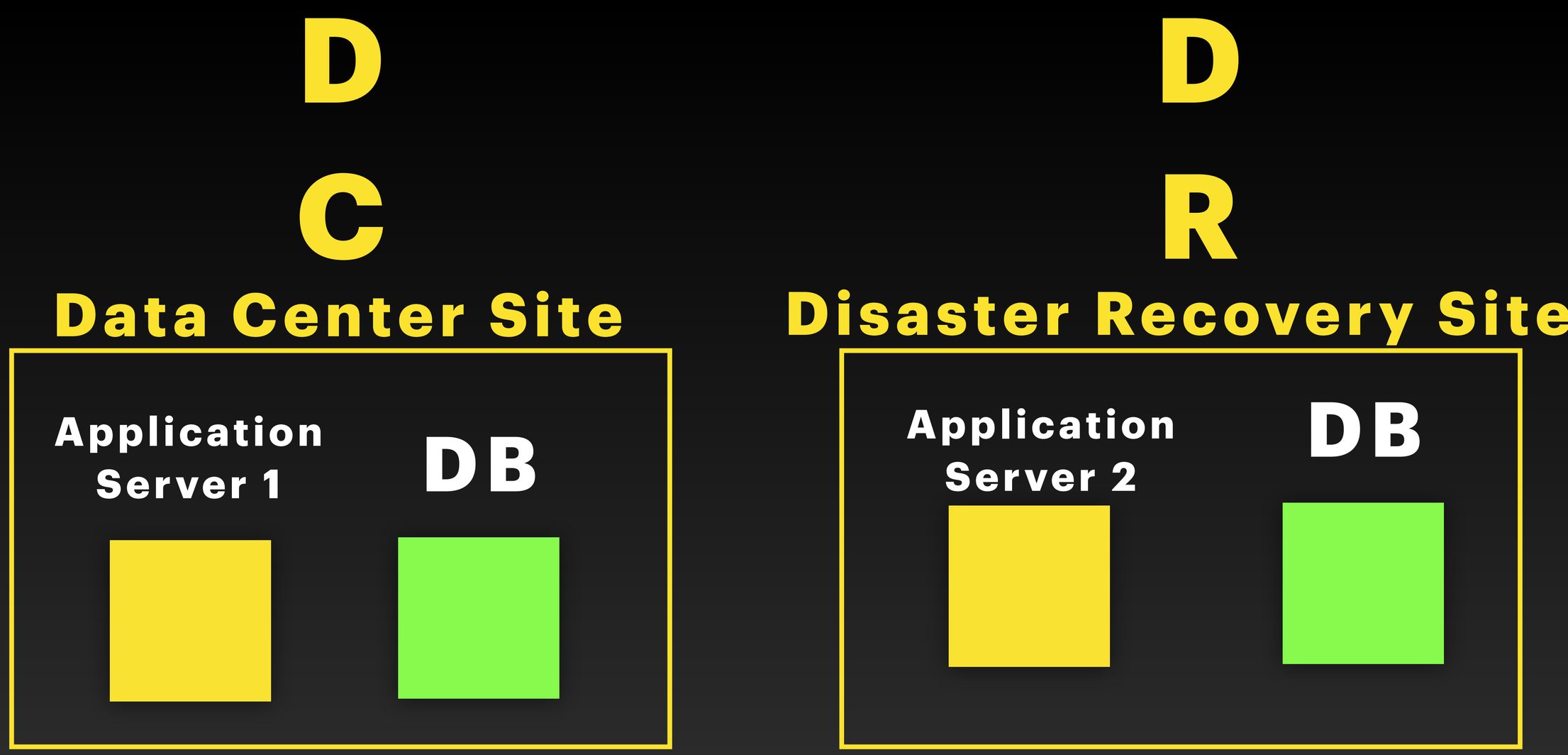


GCP Multi-region Replication Configuration

https://cloud.google.com/sql/docs/postgres/replication#configuration

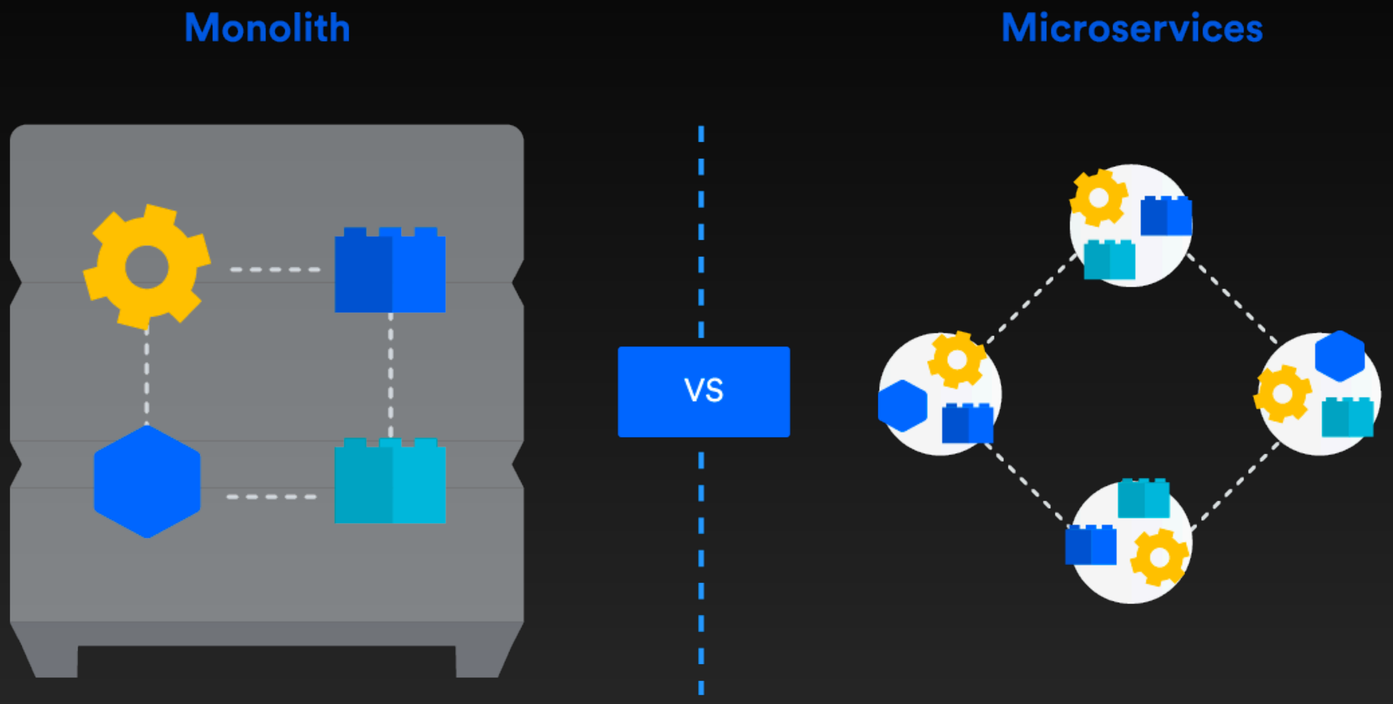






ห่างกันอย่างน้อย "160" KM

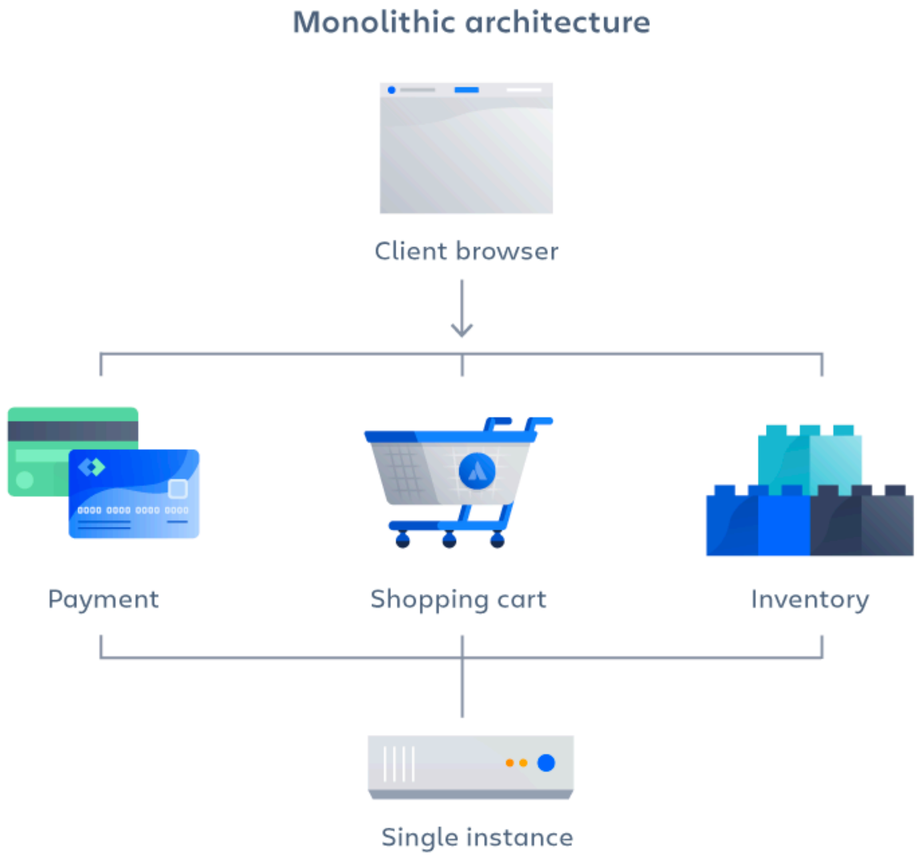
Microservices vs. Monolithic architecture



A monolithic application is built as a single unified unit while a microservices architecture is a collection of smaller, independently deployable services.

Monolithic

x functions 1 sever



Monolithic
Advantages

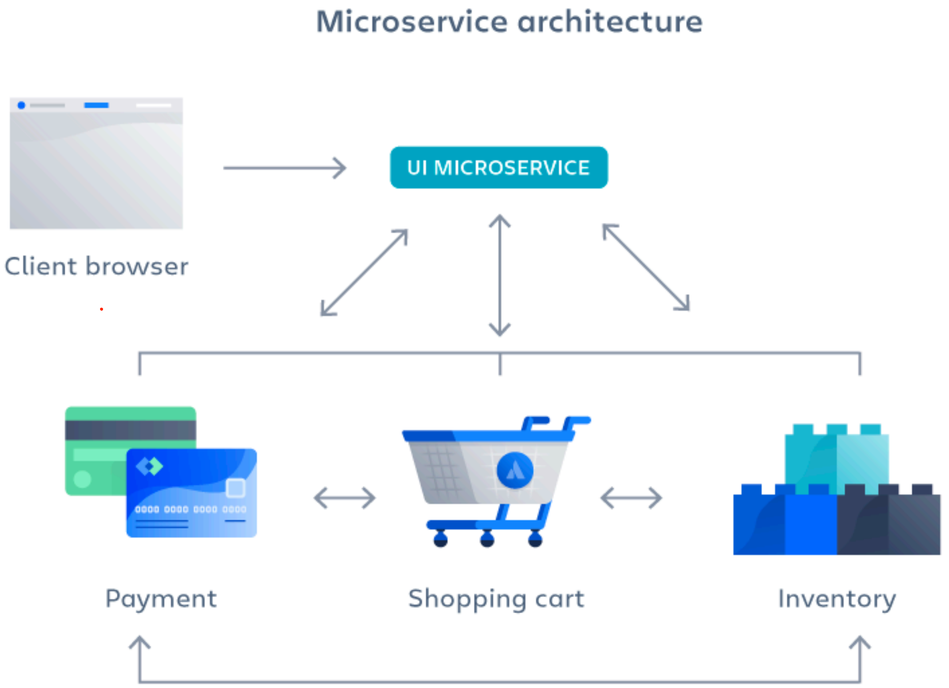
- Easy deployment – One executable file or directory makes deployment easier.
- Development – When an application is built with one code base, it is easier to develop.
- Performance – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
- Simplified testing – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.
- Easy debugging – With all code located in one place, it's easier to follow a request and find an issue.

Monolithic
Disadvantages

- Slower development speed – A large, monolithic application makes development more complex and slower.
- Scalability – You can’t scale individual components.
- Reliability – If there’s an error in any module, it could affect the entire application’s availability.
- Barrier to technology adoption – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.
- Lack of flexibility – A monolith is constrained by the technologies already used in the monolith.
- Deployment – A small change to a monolithic application requires the redeployment of the entire monolith.

Microservices

1 service 1 server



Microservices

Advantages

- Agility – Promote agile ways of working with small teams that deploy frequently.
- Flexible scaling – If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure. We are now multi-tenanant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes.
- Continuous deployment – We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
- Highly maintainable and testable – Teams can experiment with new features and roll back if something doesn’t work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.
- Independently deployable – Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- Technology flexibility – Microservice architectures allow teams the freedom to select the tools they desire.
- High reliability – You can deploy changes for a specific service, without the threat of bringing down the entire application.

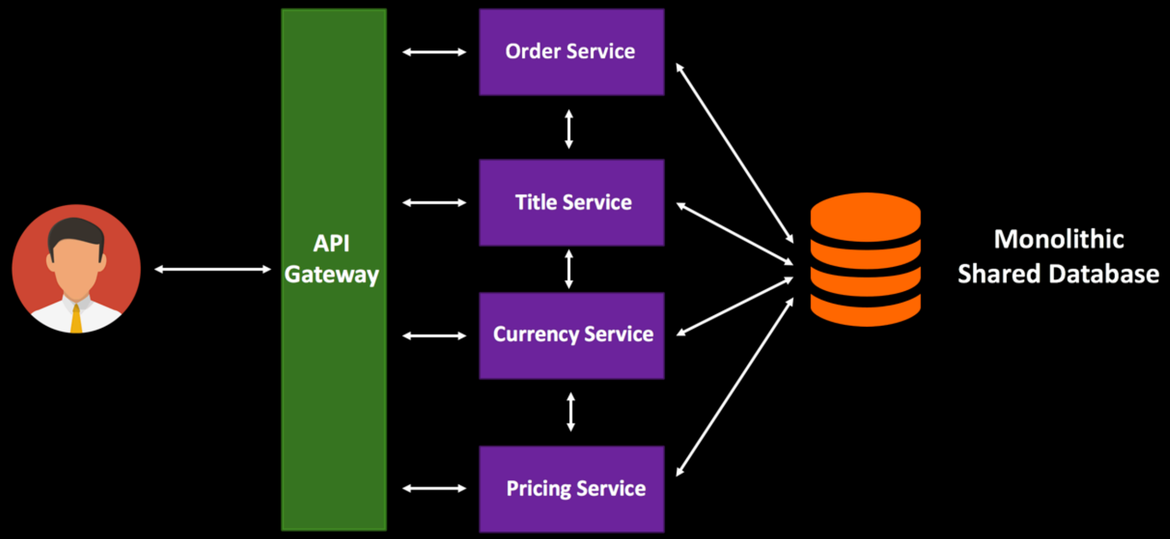
Microservices
Disadvantages

- Development sprawl – Microservices add more complexity compared to a monolith architecture, since there are more services in more places created by multiple teams. If development sprawl isn’t properly managed, it results in slower development speed and poor operational performance.
- Exponential infrastructure costs – Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.
- Added organizational overhead – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.
- Debugging challenges – Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
- Lack of standardization – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.
- Lack of clear ownership – As more services are introduced, so are the number of teams running those services. Over time it becomes difficult to know the available services a team can leverage and who to contact for support.

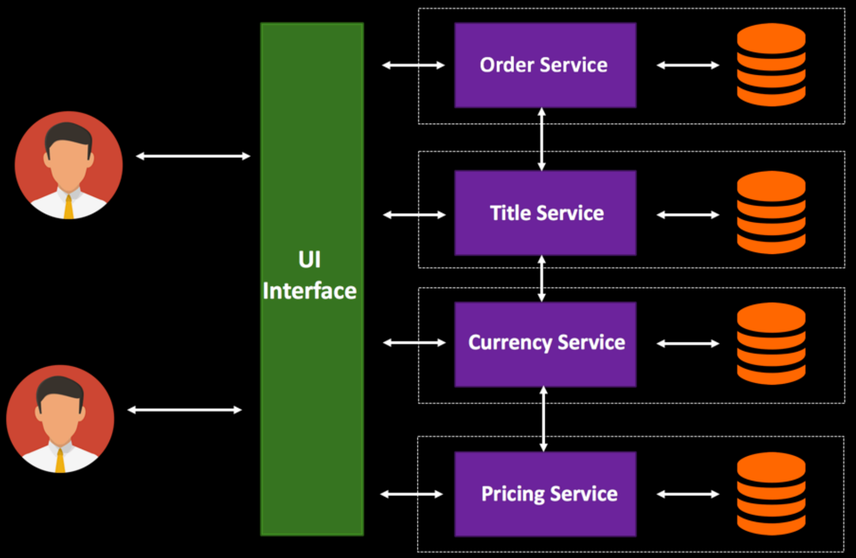
Centralized VS Decentralized DB

ต้องดู case study ก่อนว่า เป็นยังไง

Monolith Architecture – Centralized Database



Decentralized Data Management



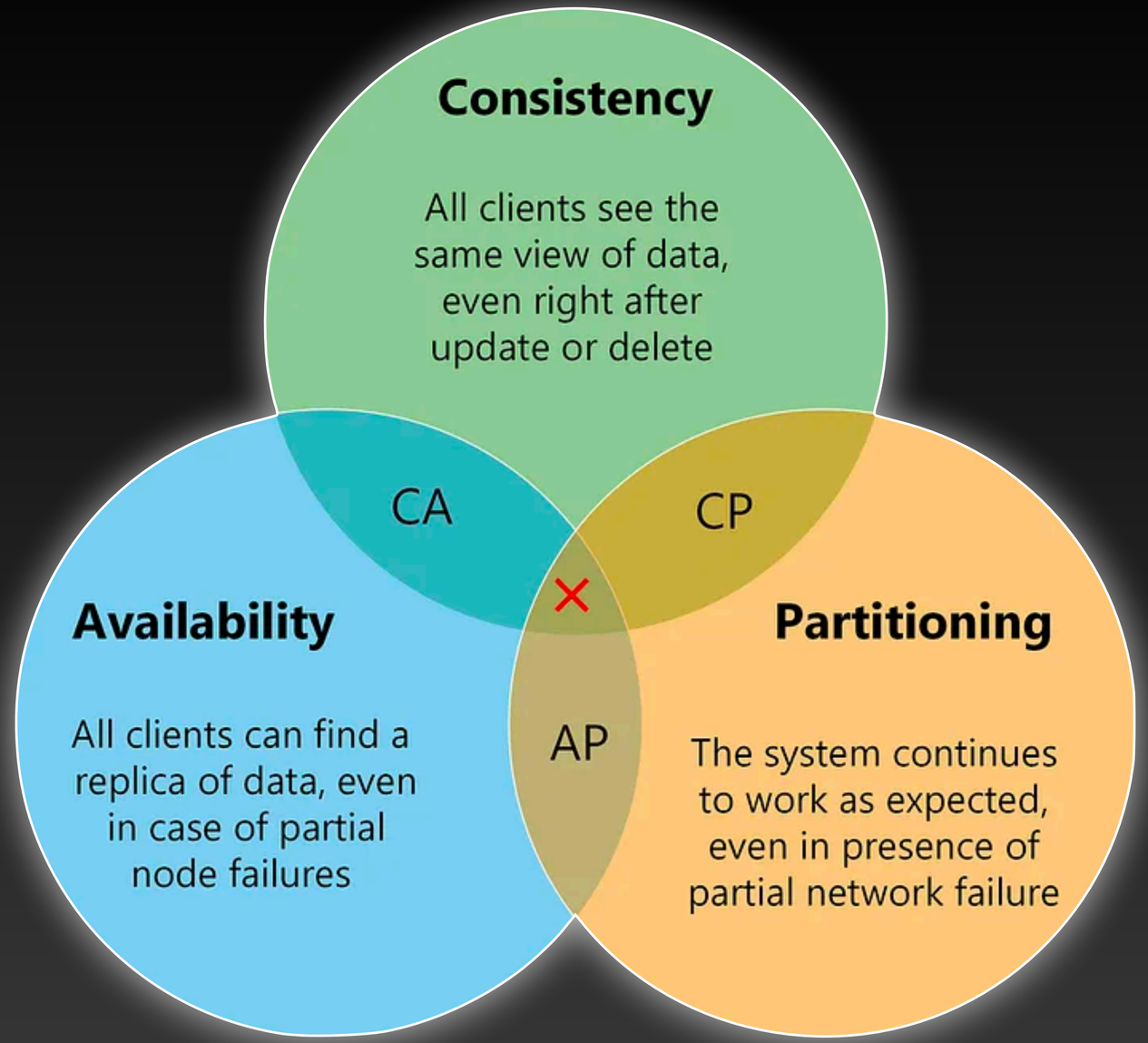
Centralized DB

- Pros:
  - Data Consistency: Easier to maintain data consistency as all services share the same database.
  - Simplicity: Simpler data management, as there's a single source of truth.
  - Simplified Transactions: Transactions involving multiple microservices are easier to manage.
- Cons:
  - Scalability Challenges: Scaling becomes more challenging as all services are dependent on the same database.
  - Tight Coupling: Services may become tightly coupled due to shared data structures and schema.
  - Complexity with Different Data Models: If microservices have significantly different data models, managing a unified schema can be challenging.

Decentralized DB

- Pros:
  - Isolation: Improved isolation between services, reducing the risk of one service affecting another.
  - Independence: Each microservice team can choose a database technology that best fits its needs.
  - Scalability: Easier to scale individual services independently.
- Cons:
  - Data Consistency Challenges: Achieving consistency across microservices may require additional effort.
  - Increased Complexity: More complex coordination is required for transactions spanning multiple databases.
  - Data Duplication: There may be duplication of data across databases, leading to potential synchronization issues.

CAP Theorem


Consistency
All clients see the same view of data, even right after update or delete
CA
CP
Availability
All clients can find a replica of data, even in case of partial node failures
AP
Partitioning
The system continues to work as expected, even in presence of partial network failure

CAP Theorem

Key Characteristics

- Consistency: All clients see the same data simultaneously.
- Availability: The system is operational at any given time.
- Partition Tolerance: System's ability to function during temporary network failures.

Misconception: Often presented as pick 2 out of 3

but network partitions are inevitable.

Better Description: Choose between Consistent or Available when Partitioned.

CAP theorem by Eric Brewer doesn't explicitly state that you can only choose two; it emphasizes the trade-offs that must be made in the presence of network partitions.

Real-Life Example

Scenario: E-commerce Stock Operation

- Consistency-Oriented Approach:
  - Description: The e-commerce platform prioritizes consistency in its stock operations.
  - Behavior During Normal Operation:
    - When a customer places an order, the system ensures that the product quantity is instantly updated across all stock databases.
    - All users see the same product quantity in real-time, maintaining a consistent view of available stock.
  - Behavior During Network Partition:
    - If a network partition occurs between stock databases, the system might temporarily halt order placements to avoid inconsistencies.
    - Customers might experience delays during partitioned periods, but the stock data remains consistent.

Real-Life Example

Scenario: E-commerce Stock Operation

- Availability-Oriented Approach:
  - Description: The e-commerce platform prioritizes availability in its stock operations.
  - Behavior During Normal Operation:
    - When a customer places an order, the system allows the order even if it can't instantly update the product quantity in all stock databases.
    - The platform continues to function, providing an available service to users even if stock data is momentarily inconsistent.
  - Behavior During Network Partition:
    - The system accepts orders during network partitions, keeping track of transactions locally.
    - Once the partition is resolved, the platform reconciles and updates stock data across databases, potentially leading to temporary inconsistencies.

Real-Life Example

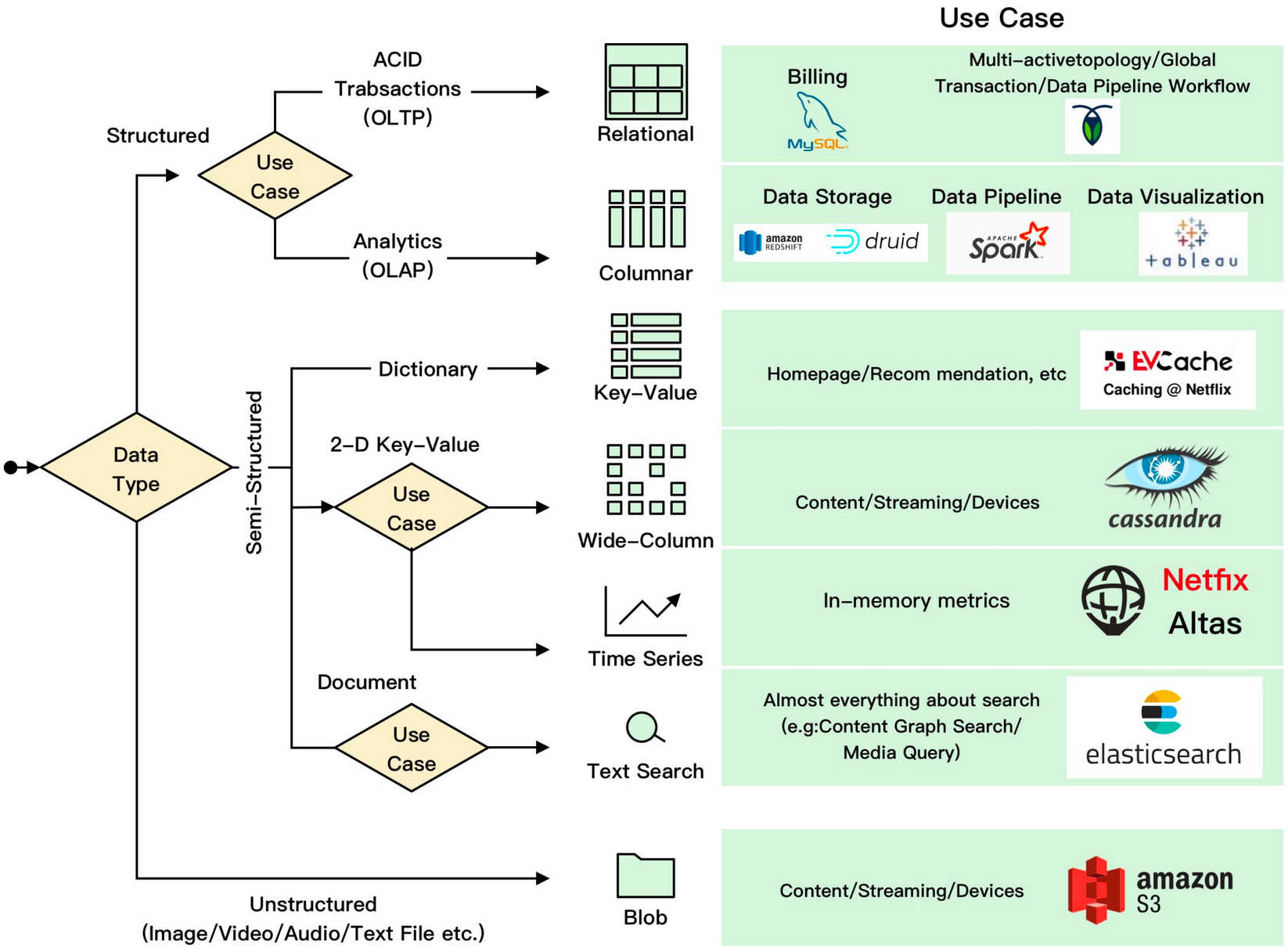
Scenario: E-commerce Stock Operation

- Balanced Consistency and Availability:
  - Description: The e-commerce platform adopts a balanced approach, considering degrees of consistency and availability.
  - Behavior During Normal Operation:
    - The system may accept partial orders or provide tentative quantity information during peak times or network fluctuations.
    - This approach introduces a level of inconsistency but ensures high availability and responsiveness to customer demands.
  - Behavior During Network Partition:
    - The platform may continue to allow limited operations, providing the best available information to users.
    - Trade-offs are made between consistency and availability based on the complexity the business is willing to introduce to maintain a balance.

Database Types and its use case

Databases used in NETFLIX

ByteByteGo.com



Relational database

- Relational databases are well-suited for scenarios where data has a clear structure, relationships between different entities are defined, and there's a need for ACID (Atomicity, Consistency, Isolation, Durability) compliance

Relational database

Use case

- Transactional Systems:
  - Example: A banking system that tracks account transactions, ensuring that each transaction is recorded accurately and reliably.
- Inventory Management:
  - Example: An e-commerce platform using a relational database to store information about products, suppliers, and inventory levels to facilitate efficient order fulfillment.
- E-commerce Applications:
  - Example: An online retail store using a relational database to manage product details, customer orders, and track inventory in real-time.

Rental ID	Movie ID	Customer Name	Rental Date	Due Date
1	1	John Smith	2024-01-22	2024-01-27
2	3	Jane Doe	2024-01-20	2024-01-23
3	2	Mike Jones	2024-01-18	2024-01-21
4	5	Sarah Lee	2024-01-16	2024-01-19
5	4	David Brown	2024-01-14	2024-01-17
6	1	Mary Williams	2024-01-12	2024-01-15
7	3	Robert Taylor	2024-01-10	2024-01-13
8	2	Jessica Miller	2024-01-08	2024-01-11
9	5	Thomas Garcia	2024-01-06	2024-01-09
10	4	Elizabeth Garcia	2024-01-04	2024-01-07

Columnar database

- Columnar databases are particularly well-suited for analytical workloads and scenarios where data needs to be queried for reporting and analytics purposes

Columnar database

Use case

- Example Sales Data
  - Date: The date of the transaction.
  - ProductID: A unique identifier for the product.
  - ProductName: The name of the product.
  - Category: The category of the product (e.g., Electronics, Clothing).
  - Price: The price of the product.
  - Quantity: The quantity of the product sold in the transaction.
  - StoreID: The identifier for the store where the sale occurred.
  - Region: The geographical region of the store.

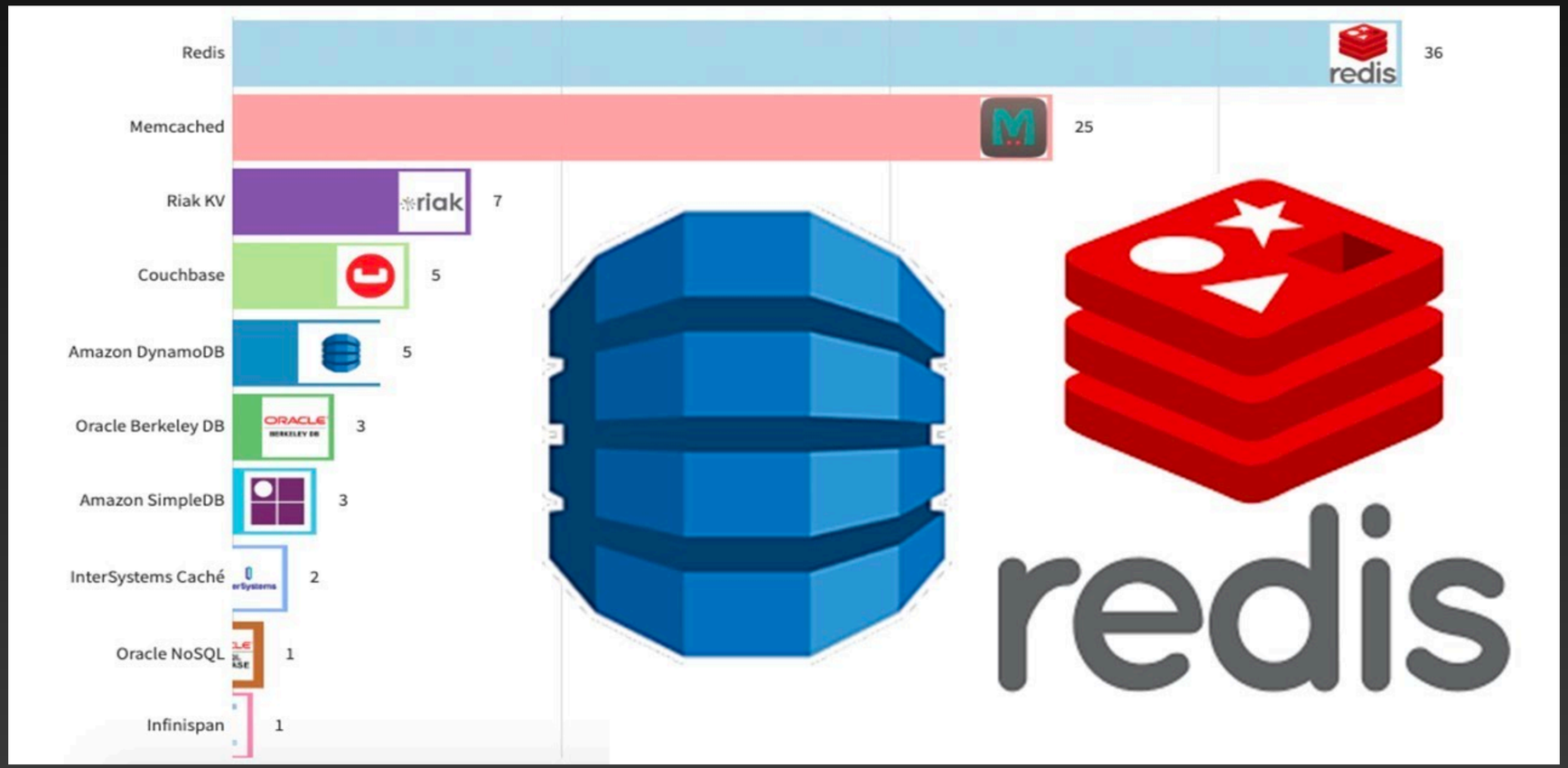
- In a columnar database, this data would be stored in a format where each of these attributes (Date, ProductID, ProductName, etc.) is stored in its own column. This is different from a row-based database where each row would contain all the data for a single transaction.
- How Data Might Be Stored in a Columnar Database:
  - Date Column: [2023-01-01, 2023-01-01, 2023-01-02, 2023-01-02, ...]
  - ProductID Column: [123, 124, 123, 125, ...]
  - ProductName Column: ["Laptop", "Shirt", "Laptop", "Smartphone", ...]
  - Category Column: ["Electronics", "Clothing", "Electronics", "Electronics", ...]
  - Price Column: [1000, 20, 1000, 500, ...]
  - Quantity Column: [1, 2, 1, 1, ...]
  - StoreID Column: [001, 002, 001, 003, ...]
  - Region Column: ["North", "South", "North", "East", ...]

- Advantages of This Approach in a Columnar Database:
  - Efficient Analytics: If the company wants to analyze total sales per product category, the database can quickly scan the 'Category' and 'Price' columns without needing to load the entire dataset.
  - Compression: Similar data in each column (like dates or product categories) can be compressed more efficiently, saving storage space.
  - Speed: Queries that need to aggregate or perform calculations over many rows (like calculating the average price of all products sold) are much faster because they involve fewer columns.

need iteration

Key-value database

- Key-value databases are best suited for scenarios where data retrieval is primarily based on simple key lookups or key-based operations.



Key-value database

Use case

- Caching:
  - Example: A web application using a key-value store to cache frequently requested user profiles or product information, reducing the load on the backend database.
- Session Storage:
  - Example: An e-commerce site using a key-value database to store session information, such as shopping cart contents and user preferences.
- User Profiles and Preferences:
  - Example: A social media platform using a key-value store to manage user profiles, storing information like profile pictures, display names, and user settings.
- Configuration Management:
  - Example: A microservices architecture using a key-value store to manage configuration parameters for individual services, making it easy to update settings without redeploying the entire application.
- Distributed Systems Coordination:
  - Example: Apache ZooKeeper, a distributed coordination service, uses a key-value data model to maintain configuration information and provide distributed synchronization.
- Distributed Caching:
  - Example: Memcached and Redis are popular key-value stores used for distributed caching in large-scale web applications.

mostly frontend?

columna upgraded

Wide column database

- Wide column databases, also known as column-family databases, are best suited for scenarios where there's a need to handle large amounts of distributed and structured data with a focus on scalability and high write throughput.

- Example of Data Storage in a Wide-Column Database
  - Consider a social media platform storing user data. Each user's data can have different attributes, and not all users will have the same set of attributes.
  - Column Family: Users
    - Row Key: UserID
    - Columns: Can vary per user, like Name, Email, DateOfBirth, FriendsList, RecentPosts, Likes, etc.
- Sample Data:
- Row Key: User123
  - Name: "Alice"
  - Email: "alice@example.com"
  - DateOfBirth: "1990-01-01"
  - FriendsList: ["User456", "User789", ...]
  - RecentPosts: ["PostID123", "PostID124", ...]
- Row Key: User456
  - Name: "Bob"
  - Email: "bob@example.com"
  - Likes: ["PostID234", "PostID345", ...]
  - ...

คล้ายjson object แต่ไม่เหมือน

Wide column database

Use case

- Content Management Systems:
  - Example: Apache Cassandra is used in some content management systems to store and retrieve diverse content types efficiently.
- Online Analytical Processing (OLAP):
  - Example: HBase can be used as a wide column store to provide the underlying storage for OLAP systems, allowing for fast and efficient analytical queries.
- Product Catalogs and E-commerce:
  - Example: A wide column store can be used to efficiently store and retrieve product information in e-commerce platforms, where products may have different attributes and specifications.
- User Profiles with Varied Attributes:
  - Example: A social networking platform might use a wide column database to manage user profiles, where users can have different sets of attributes and preferences.
- Scalable Systems with High Write Throughput:
  - Example: Apache Cassandra is known for its ability to handle high write and read throughput, making it suitable for applications with rapidly changing data.

Compare to JSON based like MongoDB

- Wide-Column Databases
  - Scalability: One of the primary strengths of wide-column stores is their ability to scale horizontally across many machines. They are designed to handle extremely large volumes of data and high write/read throughput, making them ideal for big data applications.
  - Efficient Storage of Sparse Data: Wide-column databases are highly efficient in storing sparse data, where many columns might have null or empty values. This can lead to more efficient storage and faster queries compared to JSON-based databases where each document would store all its fields, even if many are empty.
  - Flexibility in Adding Columns: They allow adding new columns to individual rows without impacting the entire schema. This means each row can have a unique set of columns, which is useful for datasets with a high degree of variability.
  - High Availability and Fault Tolerance: Many wide-column stores are designed with built-in replication and fault tolerance, making them highly available and reliable for mission-critical applications.

Compare to JSON based like MongoDB

- JSON-Based Databases (like MongoDB)
  - Flexible Document Model: JSON-based databases use a document-oriented model, which is more flexible and intuitive, especially for developers used to working with JSON data formats. This model is great for storing complex, nested data structures.
  - Ease of Use: Generally, JSON-based databases are considered more developer-friendly. They often come with features like rich query languages and secondary indexes that make them easier to use for certain types of applications.
  - Aggregation Framework: Databases like MongoDB have powerful aggregation frameworks, making it easier to perform complex queries and transformations directly in the database.
  - Real-time Applications: JSON-based databases often provide features like real-time data processing and in-built caching, making them suitable for real-time analytics and applications.

Compare to JSON based like MongoDB

- Comparative Advantages
  - Schema Flexibility: Both offer schema flexibility, but in different ways. Wide-column stores allow for dynamic columns within a column family, while JSON-based databases allow for flexible, nested document structures.
  - Data Volume and Throughput: Wide-column databases typically excel in environments with very high data volumes and write/read throughput, often outperforming JSON-based databases in these scenarios.
  - Data Model Suitability: The choice often depends on the data model that fits best with the application's needs. JSON-based models are often preferred for applications dealing with complex, nested data structures, while wide-column stores are chosen for high variability and scalability in data volume and structure.
  - Query Complexity: JSON-based databases generally offer more sophisticated querying capabilities, especially for complex data structures.

Time-series database

- Time-series column databases are specifically designed for efficiently storing and querying time-series data, where data points are associated with a timestamp.

TIMESCALE

Time-series database

Use case

- IoT Data Storage and Analytics:
  - Example: InfluxDB is a popular time-series column database used in IoT applications to store and analyze sensor data, allowing for efficient time-based queries.
- Monitoring and Alerting Systems:
  - Example: Prometheus, which is designed for monitoring and alerting, uses a time-series data model to store and query metrics from various systems.
- Financial and Market Data:
  - Example: Kdb+/q is a time-series column database commonly used in the finance industry for storing and querying large datasets of time-series data.
- Weather and Environmental Monitoring:
  - Example: TimescaleDB is a time-series database built on top of PostgreSQL and is suitable for applications that require time-series data storage, such as weather monitoring systems.

Text search database

- Text search databases are best suited for scenarios where efficient and powerful full-text search capabilities are crucial

Text search database

Use case

- Document Management Systems:
  - Example: Elasticsearch is widely used in document management systems to index and search through documents, providing features like relevance scoring and faceted search.
- Content Management Systems (CMS):
  - Example: Apache Solr is often used in CMS to power search functionality, allowing users to find relevant articles, blog posts, or other content.
- E-commerce Search:
  - Example: Amazon CloudSearch is a fully managed search service that can be employed in e-commerce applications to power product searches based on user queries.
- Log Analysis and Search:
  - Example: Elasticsearch and Kibana is a versatile platform often used for log analysis, allowing users to search and correlate log data to gain insights into system behavior.

When a Database Reaches Its Limit

Types of Database Limits

- Storage Capacity: The maximum amount of data that can be stored in the database. This limit is determined by the physical storage available and the database's architecture.
- Memory: The amount of RAM available for caching, sorting, and other in-memory operations. Insufficient memory can lead to increased disk I/O and slower performance.
- CPU Usage: The processing power available for executing queries and transactions. High CPU usage can result in slower response times and increased latency.
- I/O Throughput: The speed at which data can be read from or written to disk. I/O bottlenecks can occur when the disk subsystem is unable to keep up with the demands of the database.

When a Database Reaches Its Limit

When a database reaches its limits in terms of memory, CPU, I/O, or storage, it can be scaled in two primary ways:

- Vertical Scaling:
  - Increase CPU: Add more cores or upgrade to a more powerful processor to handle more transactions and complex queries.
  - Increase Memory: Add more RAM to support larger datasets and improve cache performance.
  - Upgrade Disk: Move to faster storage solutions like SSDs to reduce I/O latency.
  - RAID Configuration: Use RAID (Redundant Array of Independent Disks) to increase storage capacity and improve reliability and performance.

upgrade cache storage

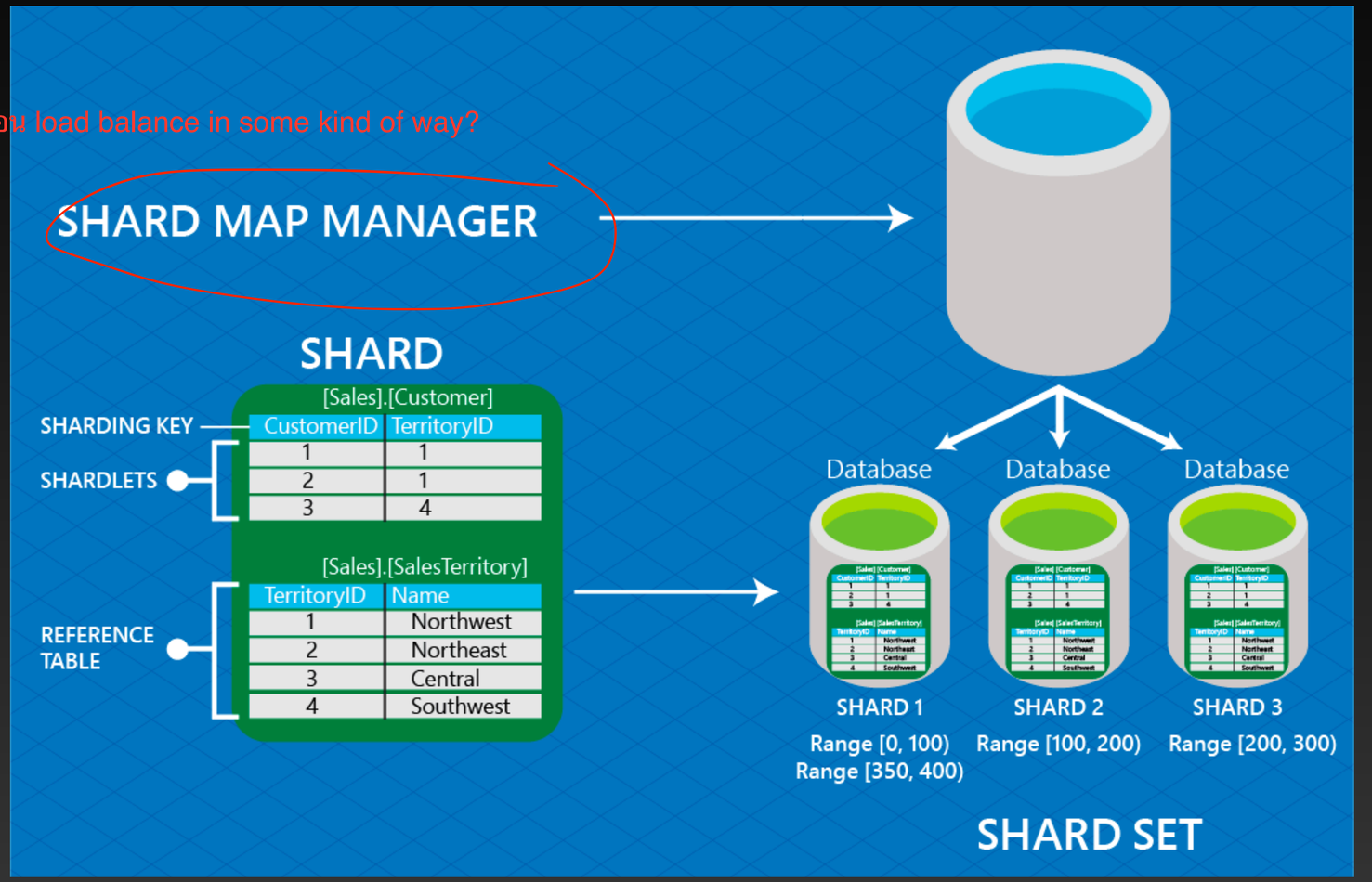
When a Database Reaches Its Limit

When a database reaches its limits in terms of memory, CPU, I/O, or storage, it can be scaled in two primary ways:

- Horizontal Scaling:
  - For Read-Heavy Workloads:
    - Replication: Create copies of the database on different servers to distribute read queries and improve availability.
    - Caching: Use caching mechanisms to store frequently accessed data in memory, reducing the need for database queries.
    - Load Balancing: Distribute queries across multiple servers to balance the load and improve performance.
  - For Write-Heavy Workloads:
    - Sharding: Distribute data across multiple database instances or servers to reduce the load on any single server. (more info https://www.citusdata.com/)
    - Partitioning: Divide a large database into smaller, manageable parts to improve write performance and scalability. (more info https://www.postgresql.org/docs/current/ddl-partitioning.html)
    - Database Federation: Combine multiple databases into a single virtual database to distribute the write load. (more info (more info https://www.postgresql.org/docs/9.3/postgres-fdw.html)

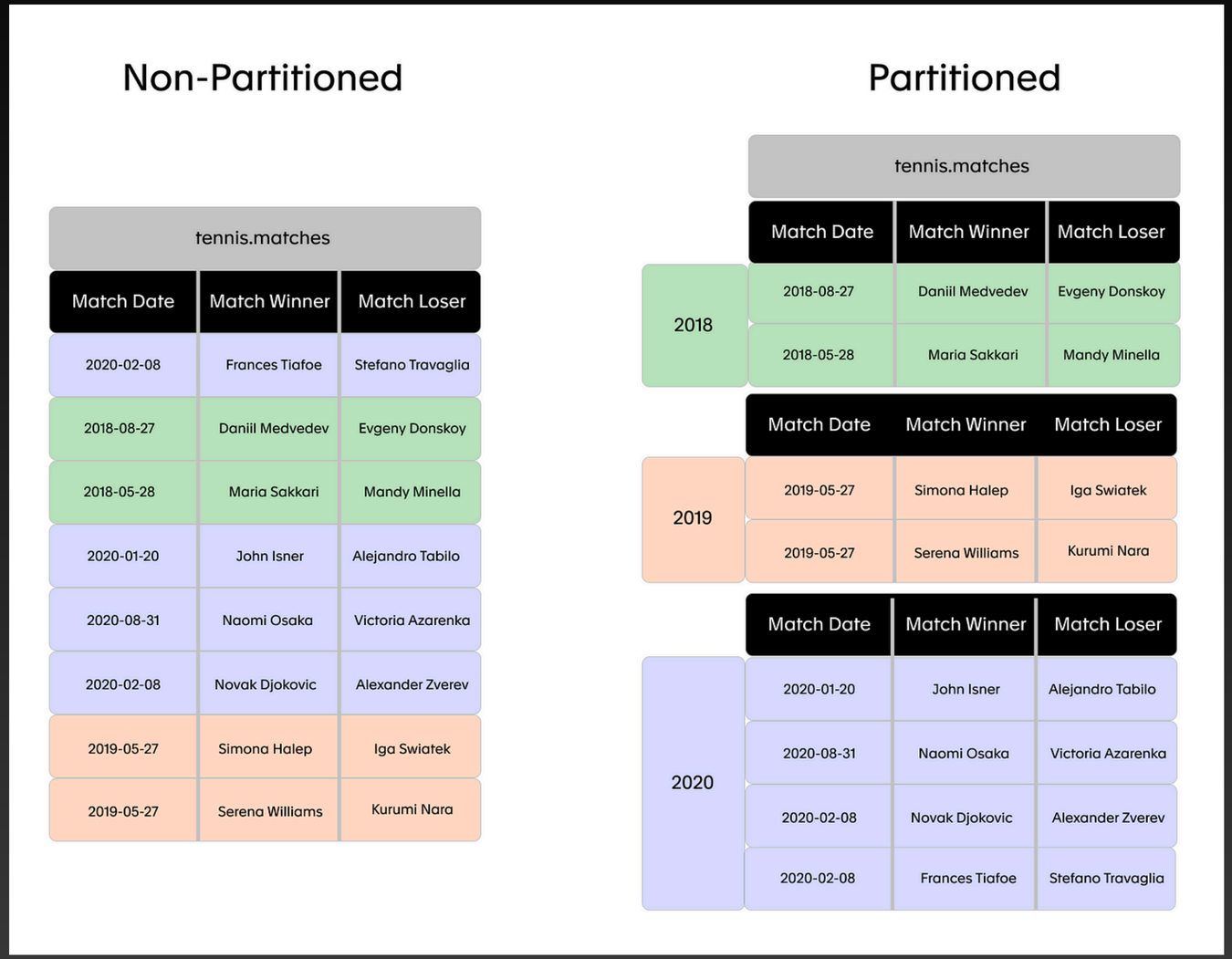
Sharding and Partitioning as Key Solutions

- What is Database Sharding?
  - Sharding is the process of splitting a large database into smaller, more manageable pieces called shards. Each shard contains a subset of the data and operates on a separate database server. This distribution allows for parallel processing and reduces the load on any single server.



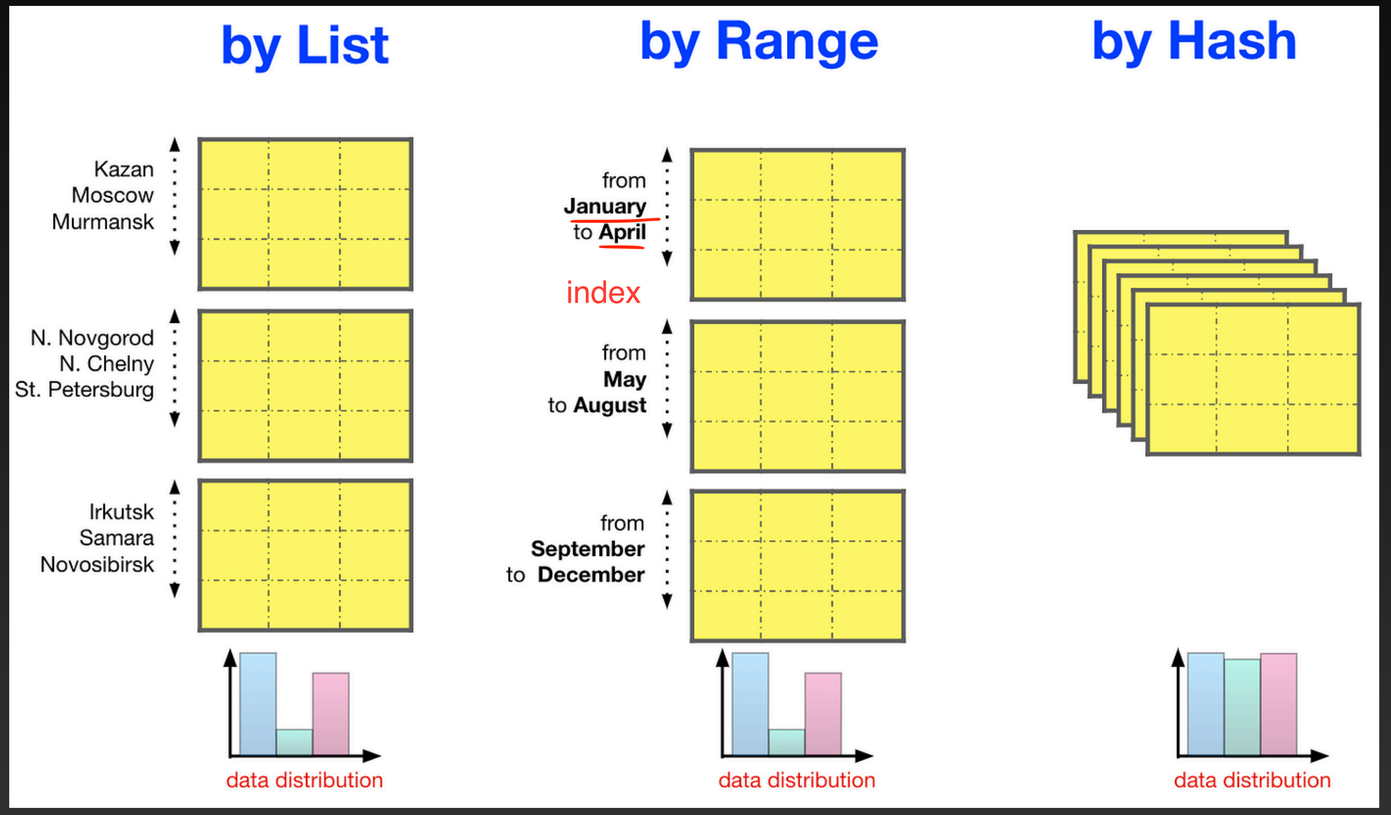
Sharding and Partitioning as Key Solutions

- What is Database Partitioning?
  แยก ตามช่วงเวลา
  - Partitioning is the process of dividing a large database table into smaller, more manageable pieces called partitions. Partitions can be based on various criteria such as ranges of values or specific keys. Unlike sharding, partitioning usually occurs within a single database server



Sharding and Partitioning as Key Solutions

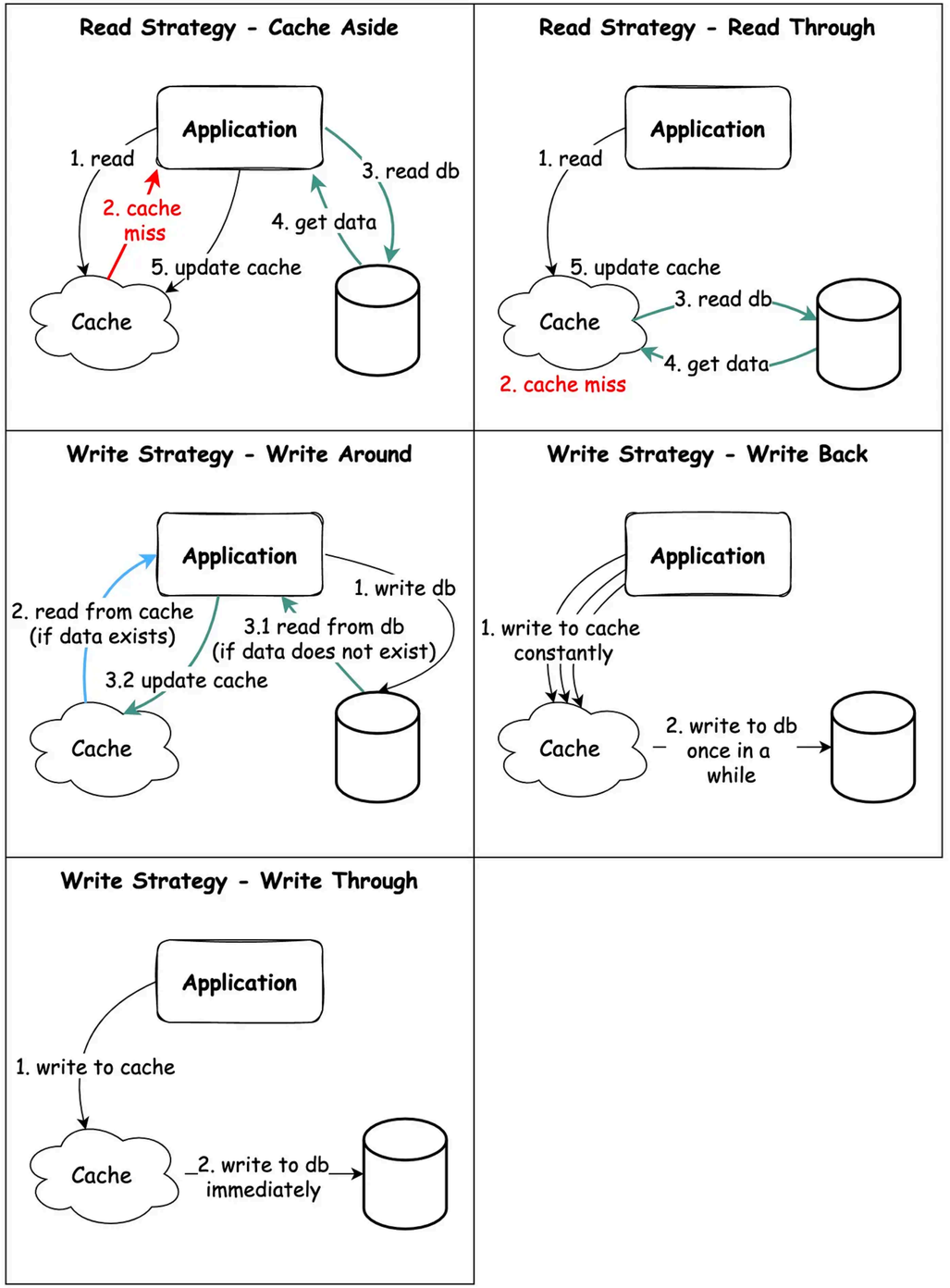
- Database partitioning, while operating within the same database service on a single server, can still make queries faster through several mechanisms:
  - **Reduced I/O**: By dividing a large table into smaller partitions, the database can limit the amount of data it needs to read from disk for a given query. If a query only needs to access data from a specific partition, the database can ignore the other partitions, reducing disk I/O.
  - **Improved Index Efficiency**: Each partition can have its own indexes. When a query is executed, the database can use the index of the relevant partition(s) instead of scanning the entire table or using a larger, less efficient index. This can significantly speed up query execution.
  - **Parallel Processing**: Some database systems can utilize parallel processing to execute queries on different partitions simultaneously. This can lead to faster query execution, especially for read-heavy operations and analytical queries that need to aggregate large amounts of data.



Benefits of Sharding and Partitioning

- Scalability: Enables databases to handle more data and more users without significant degradation in performance.
- Performance: Improves query response times and transaction throughput by distributing load.
- Flexibility: Allows for incremental scaling, adding more shards or partitions as needed.

Caching Strategies in System Design



Cache Aside

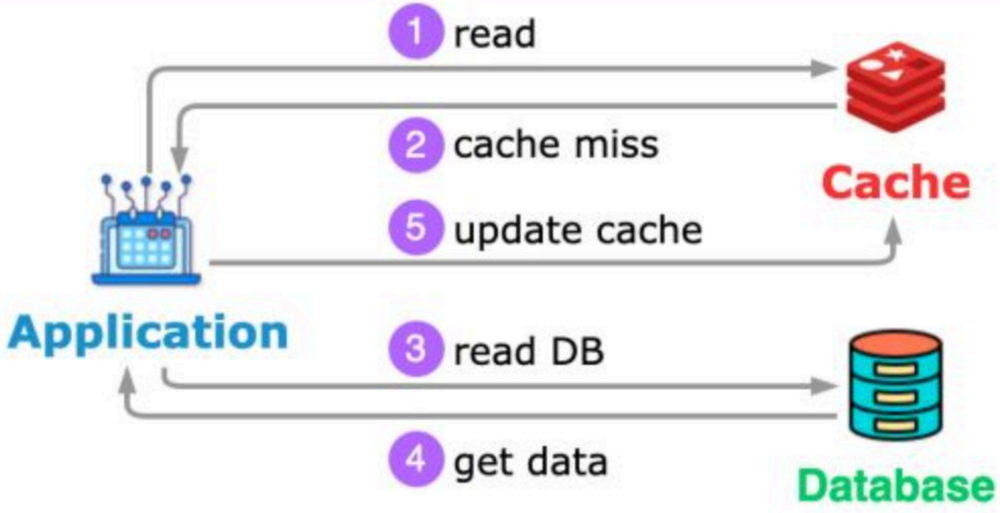
แยกกัน ไม่ยุ่ง

Pros

1. update logic is on application level, easy to implement
2. cache only contains what the application requests for

Cons

1. each cache miss results in 3 trips
2. data may be stale if DB is updated directly



Read Through

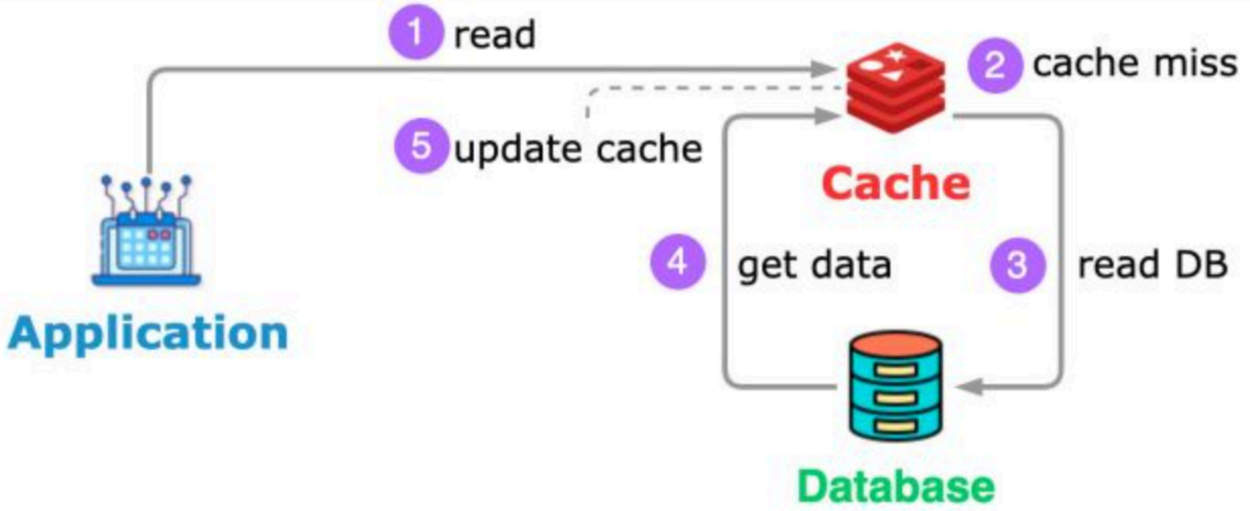
ผ่านlogic ในserver cache

Pros

1. application logic is simple
2. can easily scale the reads and only one query hits the DB

Cons

data access logic is in the cache, needs to write a plugin to access DB



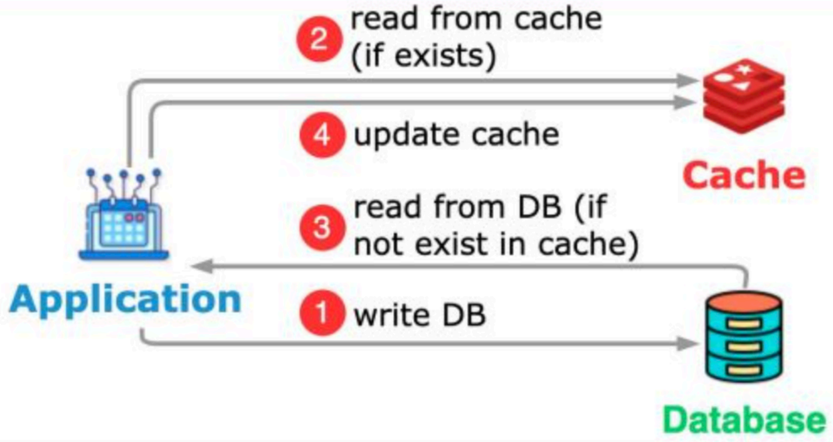
Write Around

Pros

1. the DB is the source of truth
2. lower read latency

Cons

1. higher write latency because data is written to DB first
2. the data in the cache may be stale



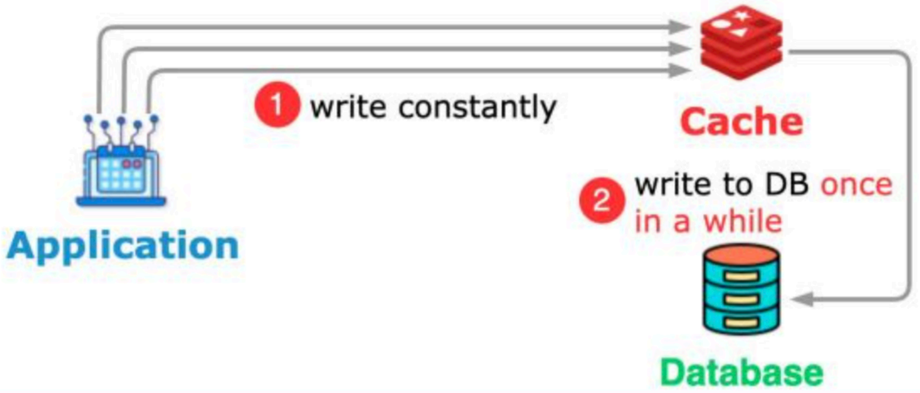
Write Back

Pros

1. lower write latency
2. lower read latency
3. the cache and DB are eventually consistent

Cons

1. there can be data loss if the cache is down
2. infrequent data is also stored in the cache



Write Through

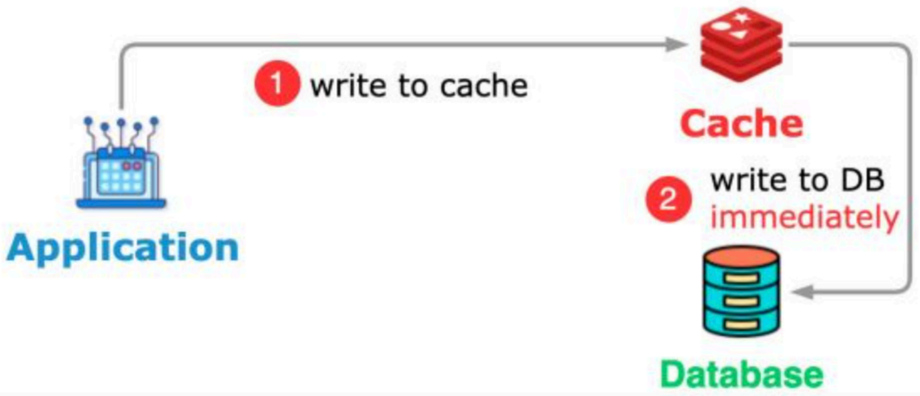
เปลี่ยนในdatabaseเลย

Pros

1. reads have lower latency
2. the cache and DB are in sync

Cons

1. writes have higher latency because they need to wait for the DB writes to finish
2. infrequent data is also stored in the cache



Cache eviction policies

- Cache eviction policies are algorithms that determine how to manage data in a cache, a layer of fast and temporary storage that improves performance by keeping recent or often-used data items in memory locations that are faster or computationally cheaper to access than normal memory stores. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.
- Redis default policy is volatile-LRU

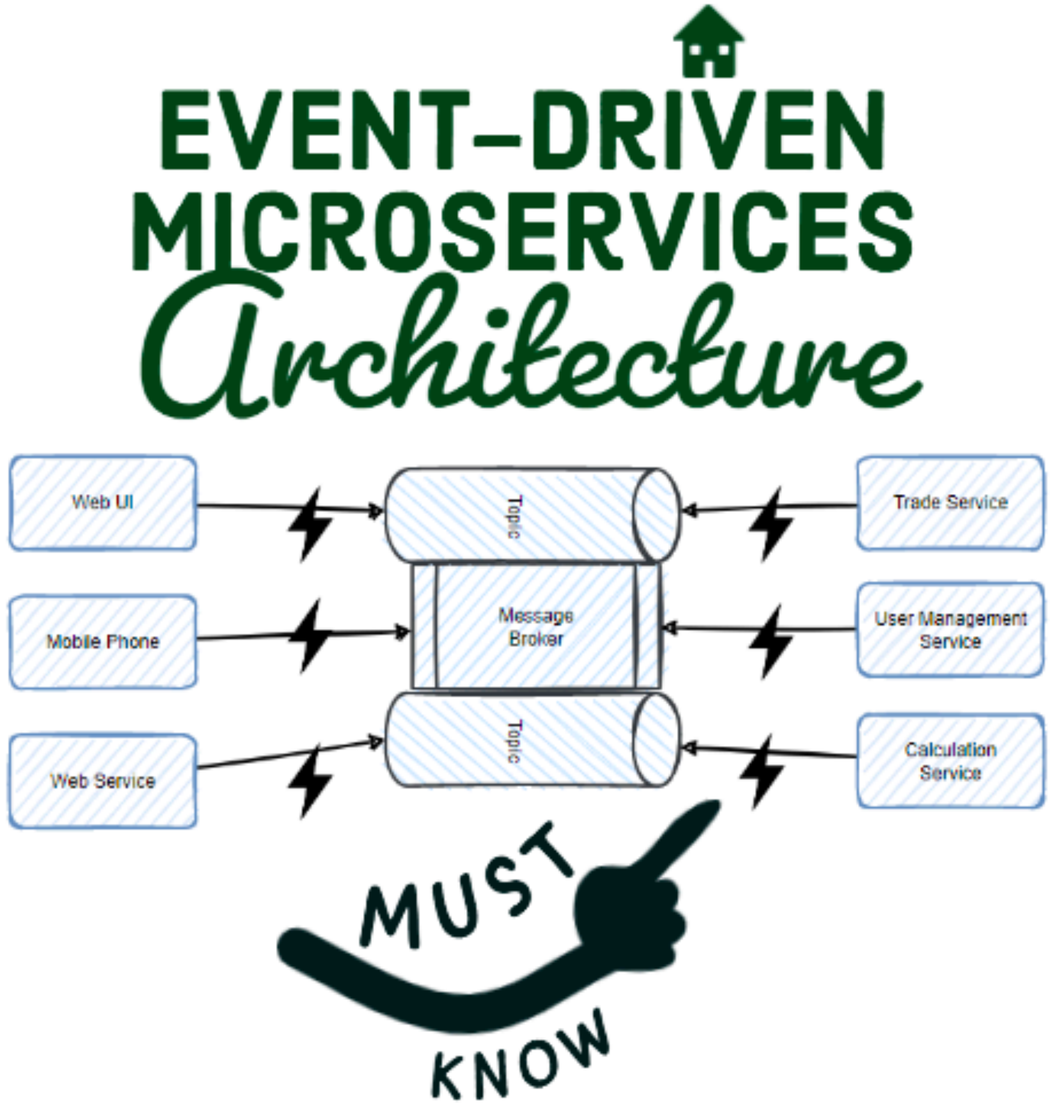
Redis cache eviction policies

- allkeys-lru:
  - Removes least recently used cache entries, regardless of expiration time.
  - Redis tracks access time for each key, evicting keys not accessed recently when reaching the memory limit.
  - Applies to all keys in the database.
- volatile-lru:
  - Removes least recently used cache entries with an expiration time set.
  - Suitable for scenarios where data needs periodic refreshing.
- allkeys-lfu:
  - Removes least frequently used keys when making room for new data.
  - Redis tracks access frequency by incrementing counters associated with each key.
- volatile-lfu:
  - Similar to allkeys-lfu but applies only to keys with an expiration time set.
- volatile-ttl:
  - Removes keys with the shortest TTL (Time To Live) first.
  - TTL is the duration after which the key is automatically deleted.
- noeviction:
  - Does not evict any keys when the memory limit is reached.
  - Returns an error on write commands, leaving it to the application code to handle the error condition.

ครา่วๆ ก้คือ redis มีpolicies ในการmanage cache ที่เขาคิดมาแล้ว

Event-Driven Architecture

A software design pattern where the flow of the system is determined by events such as user actions, sensor outputs, or messages from other programs.



Event-Driven Architecture

Key Characteristics

- Asynchronous communication
- Loose coupling between components
- Scalability and responsiveness

Event-Driven Architecture

Components of EDA

- Event Producers:
  - Systems or components that generate events.
- Event Consumers:
  - Systems or components that respond to events.
- Event Bus:
  - Communication channel that allows events to be broadcast to multiple consumers.

Event-Driven Architecture

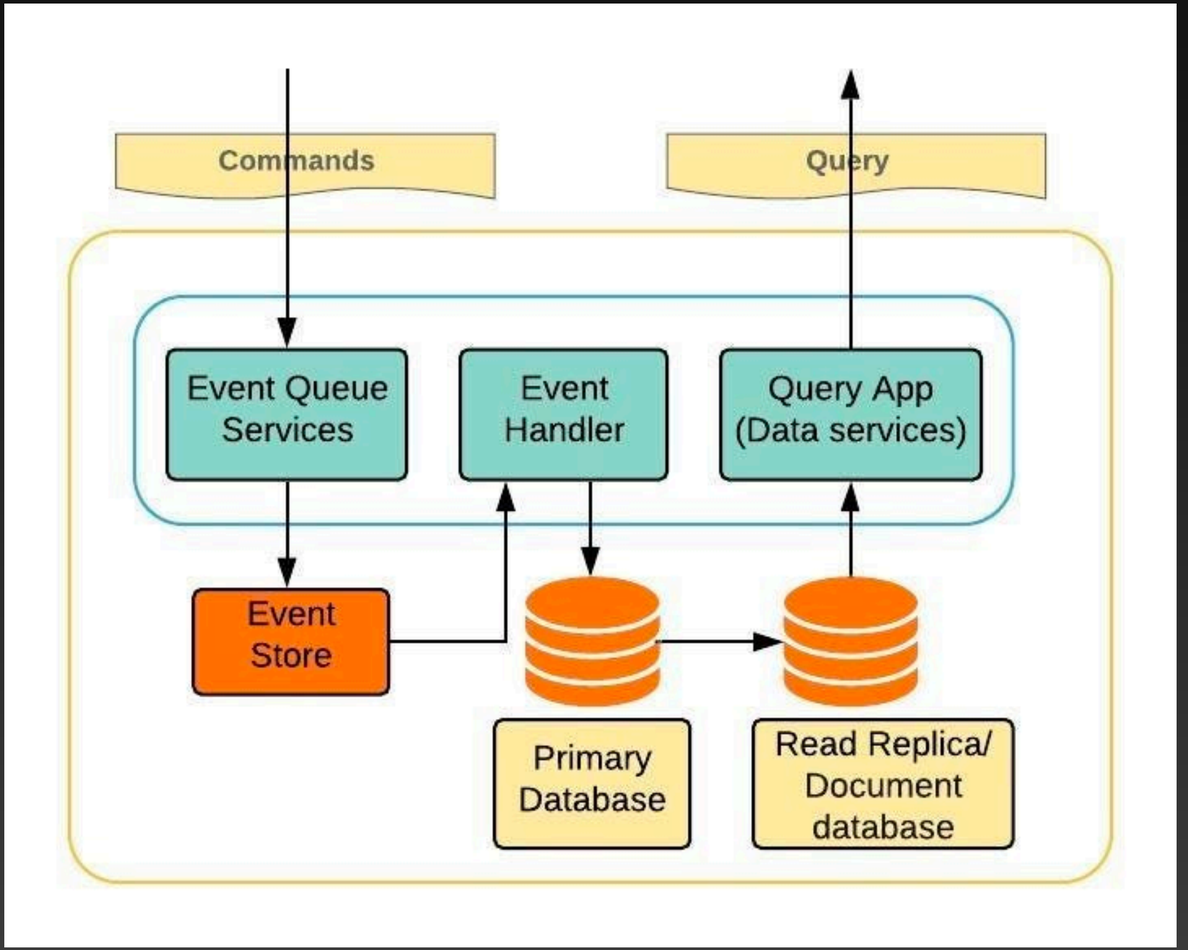
Real-world Application of EDA

- Example scenario of an e-commerce platform:
  - Event: "New Order Placed"
  - Event Producer: Order Service
  - Event Consumers: Inventory Service, Shipping Service
- Benefits:
  - Scalability: Easily add new services without affecting existing ones.
  - Flexibility: Components can evolve independently.

Command Query Responsibility Segregation (CQRS)

CQRS is a pattern that separates the read and write operations of a data store. It uses separate models for reading and writing data.

- Commands: Requests to change state (e.g., Create, Update, Delete).
- Queries: Requests to retrieve data.
- How EDA and CQRS work together:
  - Commands in CQRS can trigger events in EDA.
  - Events in EDA can be used to update read models in CQRS.



Benefit of CQRS

CQRS (Command Query Responsibility Segregation) is a design pattern that separates reading data from writing data in a system. This separation has several benefits:

- Scalability: You can scale the read and write parts independently, which is useful if your system has more reads than writes or vice versa.
- Performance: Each part can be optimized for its specific task, leading to faster reads and writes.
- Simplicity: The read side can use simpler models, making it easier to query data.
- Security: It allows for better control over who can read and who can write data.
- Flexibility: You can use different technologies for the read and write sides to suit their needs.
- Maintainability: Changes to the read or write side can be made independently, making the system easier to manage.

Inside event-driven architectures

What patterns will you come across when building event-driven architectures?

Process Manager

Central unit that orchestrates communication
Process a workflow in a single unit

Event sourcing

Store events and consume for state
Build current state using events

Event streaming

Stream of events usually used for real-time processing

Point-to-point messaging (pull)

Messages put onto queue by sender
Receiver consumes messages from queue

Change data capture

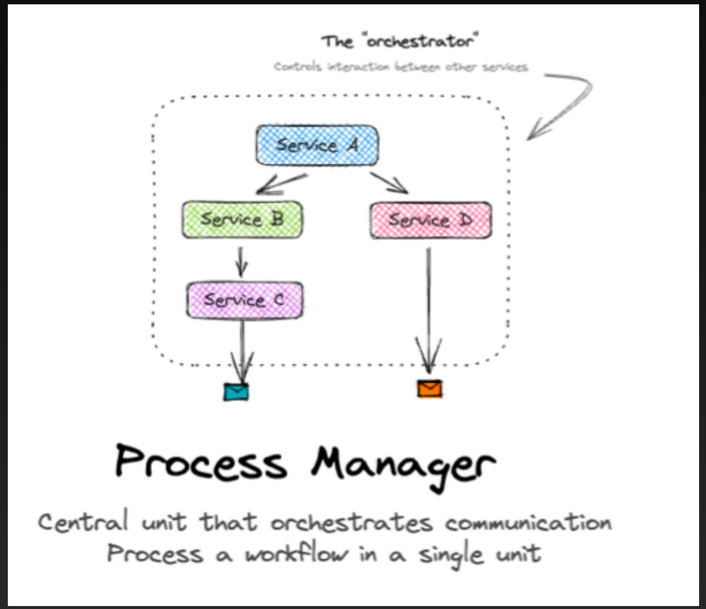
Listening to events from changes in data
Consume events directly from DB

Pub/Sub (push)

Publish messages/events to many subscribers
Each subscriber gets copy of event to process

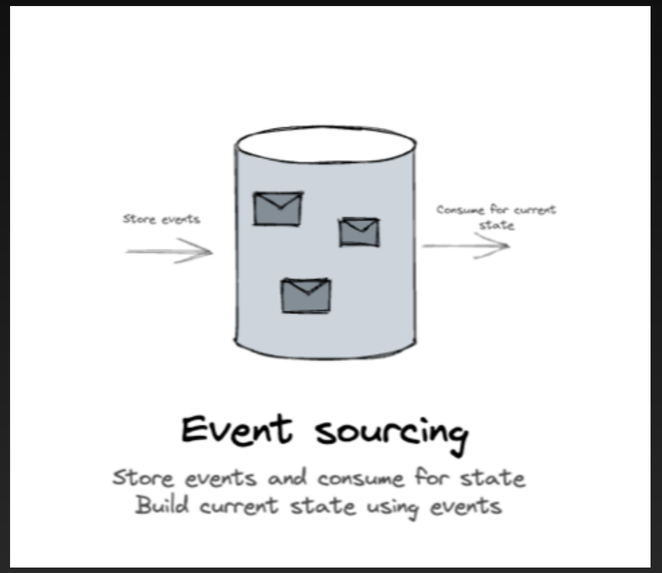
Process Manager

- Definition: Coordinates complex processes involving multiple steps or interactions.
- Use Cases: Workflow management, order processing.
- Example Use Case: E-commerce Order Processing
  - Scenario: Managing the steps of an order process, such as payment, inventory check, and shipping.
  - Role: Ensures that each step is completed in the correct order and handles any failures or retries.



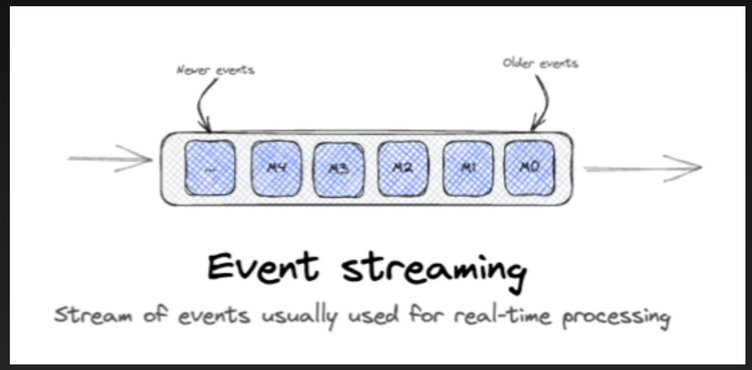
Event Sourcing

- Event Sourcing:
- Definition: Storing changes to the state of an application as a sequence of events.
- Use Cases: Audit trails, historical data analysis.
- Example Use Case: Banking Transactions
  - Scenario: Tracking transactions (deposits, withdrawals) in a bank account.
  - Role: Reconstructs account balances by replaying events, providing an audit trail of transactions.



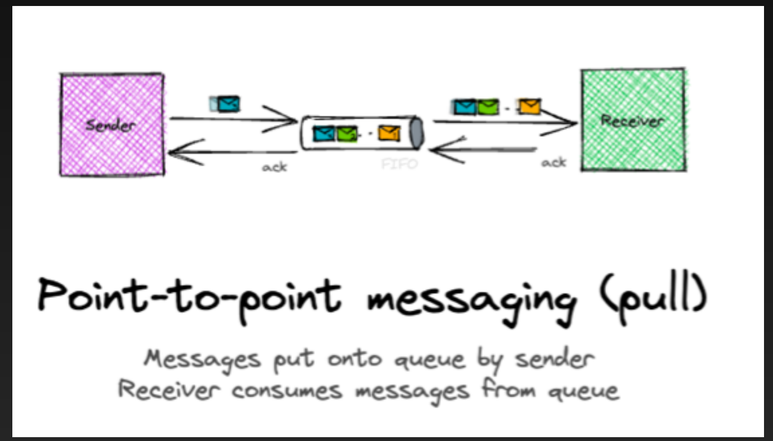
Event Streaming

- Definition: Continuously processing and transferring data as a stream of events.
- Use Cases: Real-time data analytics, monitoring systems.
- Example Use Case: Stock Market Tick Data Processing
  - Scenario: Analyzing real-time stock market data to provide insights and trigger alerts.
  - Role: Streams stock market tick data (price changes, trades) to analyze trends, detect anomalies, and generate real-time alerts for traders and analysts.



Point-to-Point Messaging

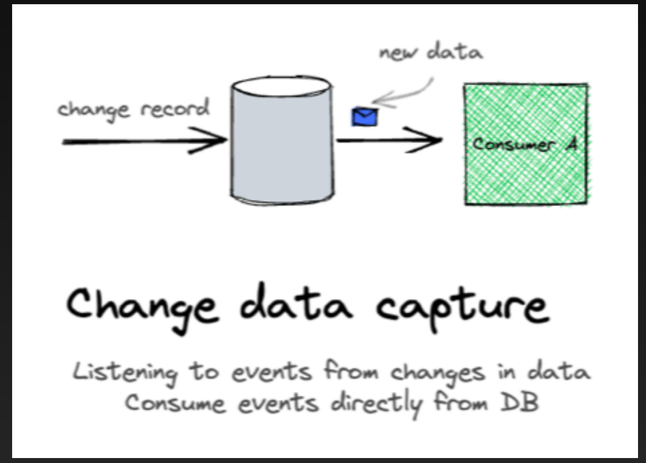
- Definition: Direct communication between participants without a central server.
- Use Cases: Inter-system communication, workflow automation.
- Example Use Case: Inter-System Workflow Automation
  - Scenario: A system that processes transactions and, upon successful completion, sends notifications or data to another system for further processing or reporting.
  - Role: Enables the first system to send messages directly to the second system, informing it of successful transactions or providing data for the next step in the workflow. This can be used in scenarios like order processing systems, where one system handles order fulfillment and then notifies an accounting system to update financial records.



Change Data Capture

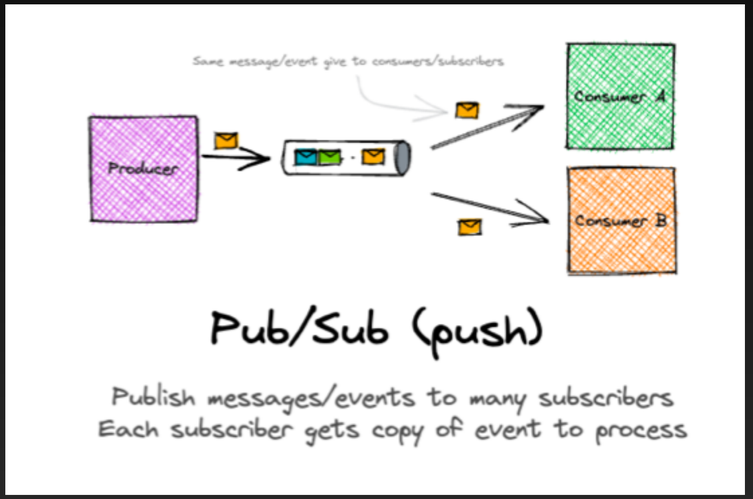
4

- Definition: Capturing changes made to data in a database and applying them elsewhere.
- Use Cases: Data synchronization, real-time data warehousing.
- Example Use Case: Data Warehousing
  - Scenario: Updating a data warehouse with changes from operational databases.
  - Role: Captures incremental changes and updates the warehouse, ensuring it stays current.



Pub/Sub (Publish/Subscribe)

- Definition: A messaging pattern where publishers send messages without knowing the subscribers, and subscribers receive messages based on their interests.
- Use Cases: Notification systems, event-driven architectures.
- Example Use Case: Notification System
  - Scenario: Sending notifications about various events (new emails, calendar reminders).
  - Role: Allows users to subscribe to relevant notifications and receive them as they are published.



Database Transaction

TRANSACTION

bank transfer transaction

queries + commit and rollback

debit the sender account

credit the beneficiary's account

single unit of work

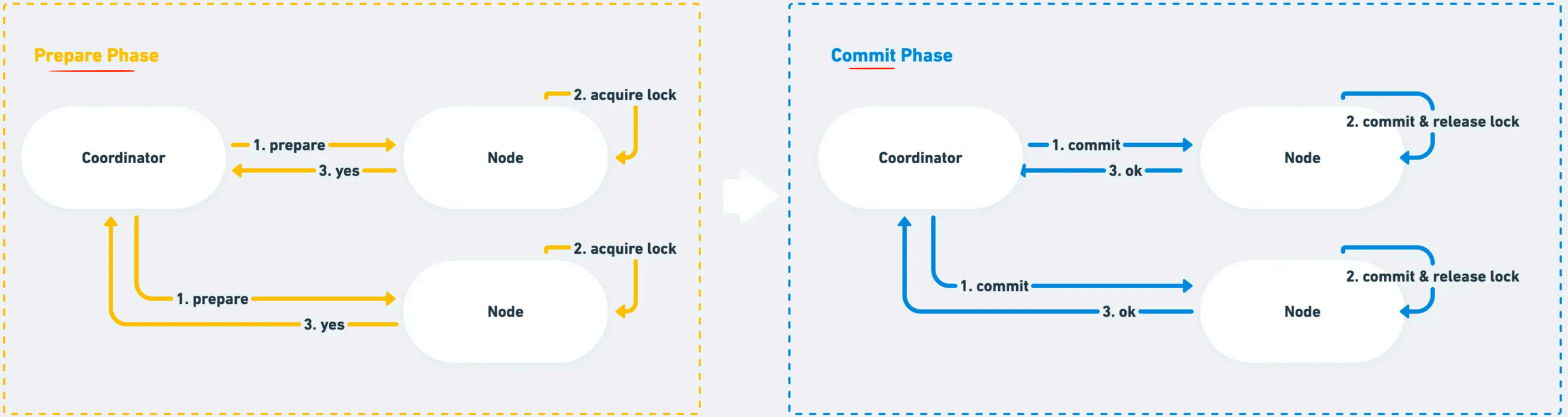
series of actions that fail as a group or complete entirely as a group

Commit

all actions should be rollback in case of any failure

Distributed Transaction for Microservices

Two phase commit (2PC)



Two phase commit
Pros/Cons

- Pros:
  - Atomicity: 2PC ensures ~~atomicity~~ in distributed transactions. Either all participants commit or all participants abort, preventing partial or inconsistent updates across the distributed system.
  - Coordination: It provides a centralized coordinator to manage the distributed transaction. The coordinator plays a crucial role in ensuring that all participants agree on whether to commit or abort.
  - Consistency: 2PC maintains consistency by ensuring that all nodes involved in the transaction reach a consensus before committing or aborting. This prevents situations where some nodes commit changes while others do not.
  - Durability: The protocol guarantees durability in the sense that once a decision is made and communicated, it is guaranteed to persist, even in the face of failures.

Two phase commit
Pros/Cons

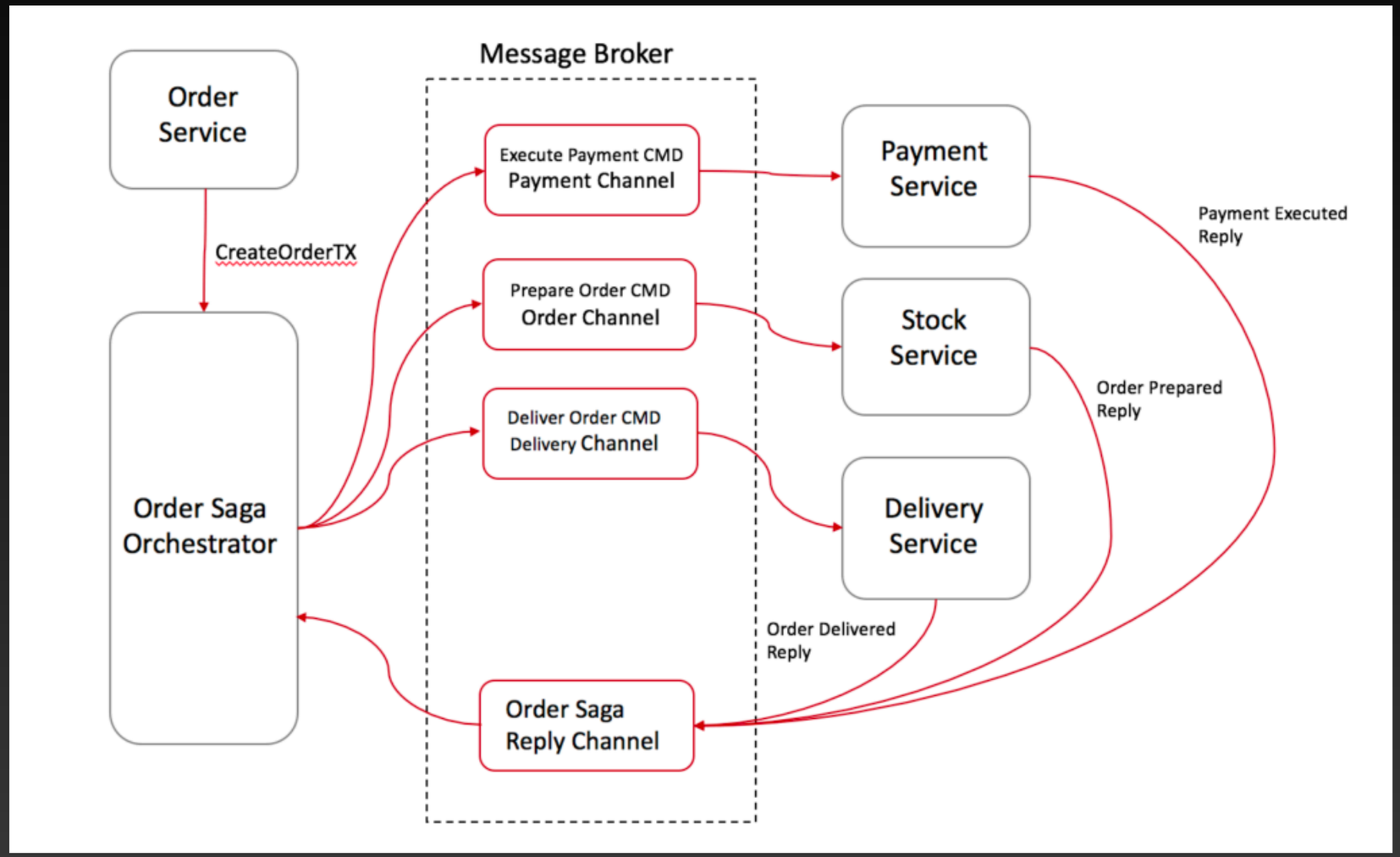
- Cons:
  - Blocking Issues: The protocol can introduce blocking issues. If a participant or the coordinator fails during the process, it can lead to blocking and waiting for a decision, potentially causing system-wide delays.
  - Single Point of Failure: The coordinator can become a single point of failure. If the coordinator fails before reaching a decision, the entire transaction may be in an uncertain state, requiring manual intervention to resolve.
  - Performance Overhead: 2PC introduces additional messages and coordination overhead, which can impact performance. The need for multiple rounds of communication can lead to latency in completing transactions.
  - Concurrency Issues: The locking mechanism used in 2PC can lead to concurrency issues. During the preparation phase, locks are held until a decision is made, potentially impacting the system's ability to handle concurrent transactions.
  - Blocking Resources: Resources held by a transaction may be blocked for an extended period during the protocol execution. This can lead to decreased system throughput and efficiency.
  - Not Suitable for Unreliable Networks: In distributed systems with unreliable communication, the blocking nature of 2PC can become problematic. If messages are lost or delayed, it might lead to uncertainties in the state of the transaction.
  - Lack of Flexibility: 2PC may not be well-suited for systems with dynamic membership or where participants can join or leave during the transaction. It assumes a fixed set of participants throughout the process.

Saga Pattern

Saga Pattern is a design pattern for managing distributed transactions and long-lived business processes in a microservices architecture.

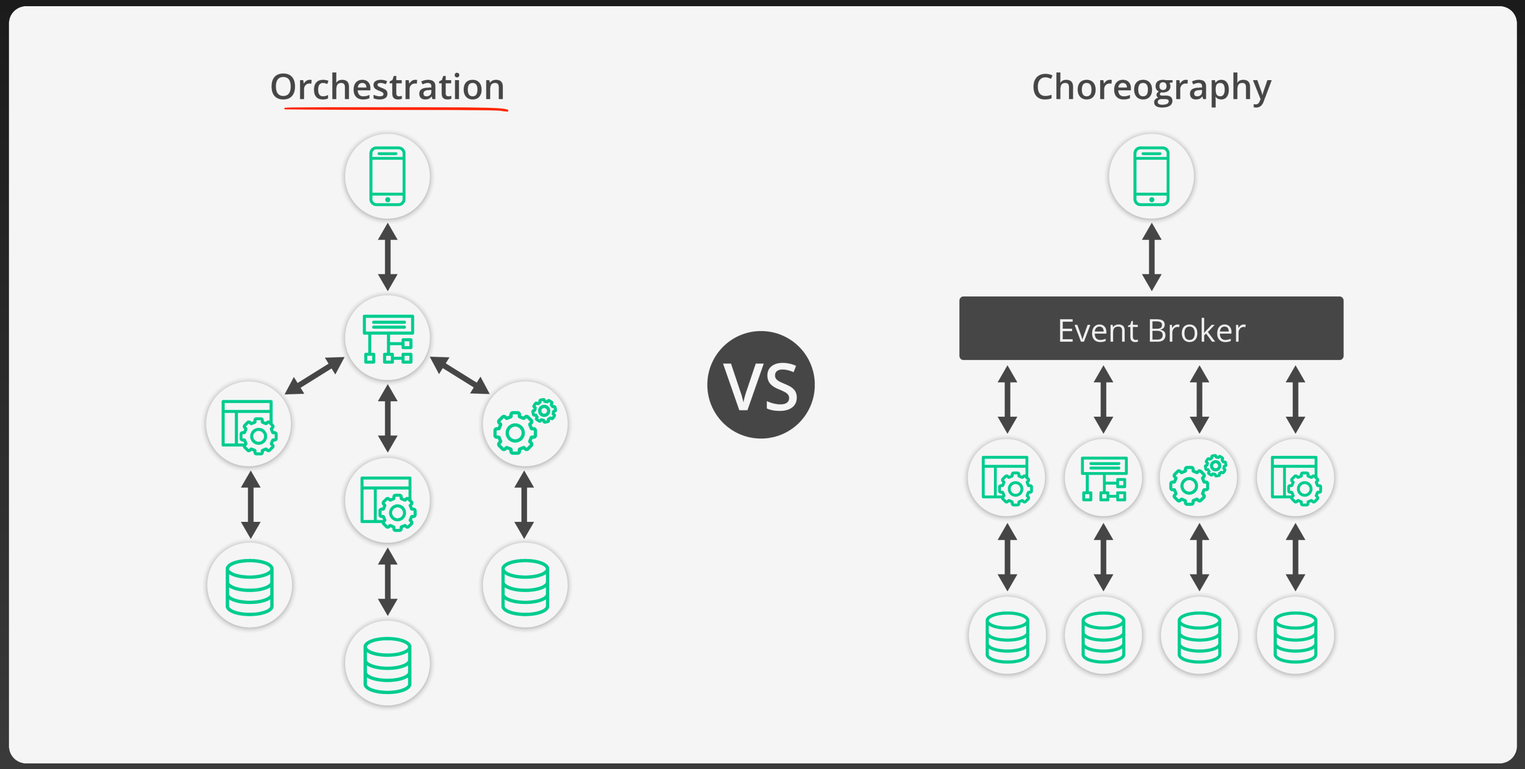
Goal: Ensure consistency and reliability in microservices communication.

Key Concept: Breaking down a transaction into a series of smaller, more manageable steps.

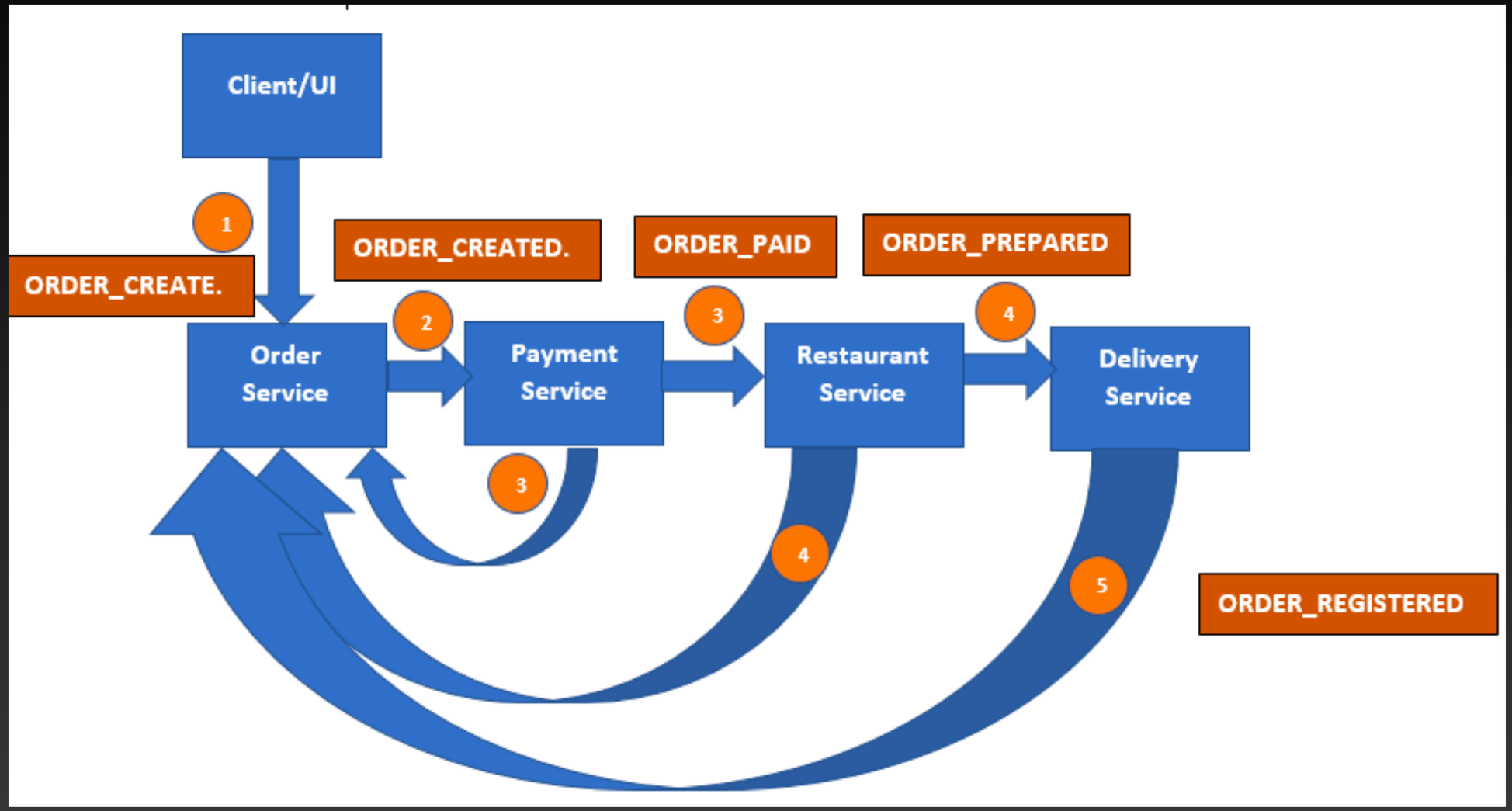


Orchestrator vs. Choreography

- Orchestrator: A central component that coordinates and manages the flow of the entire saga. Initiates and controls the sequence of services.
- Choreography: Each service in the saga is responsible for deciding its own behavior. Communication happens through events.



Choreography Example



Orchestration

vs.

Choreography

Pros

- Simplicity
- Centralized control
- Visibility
- Ease of troubleshooting

Cons

- Tight coupling
- Single point of failure
- Difficulty adding, removing, or replacing microservices
- Overhead

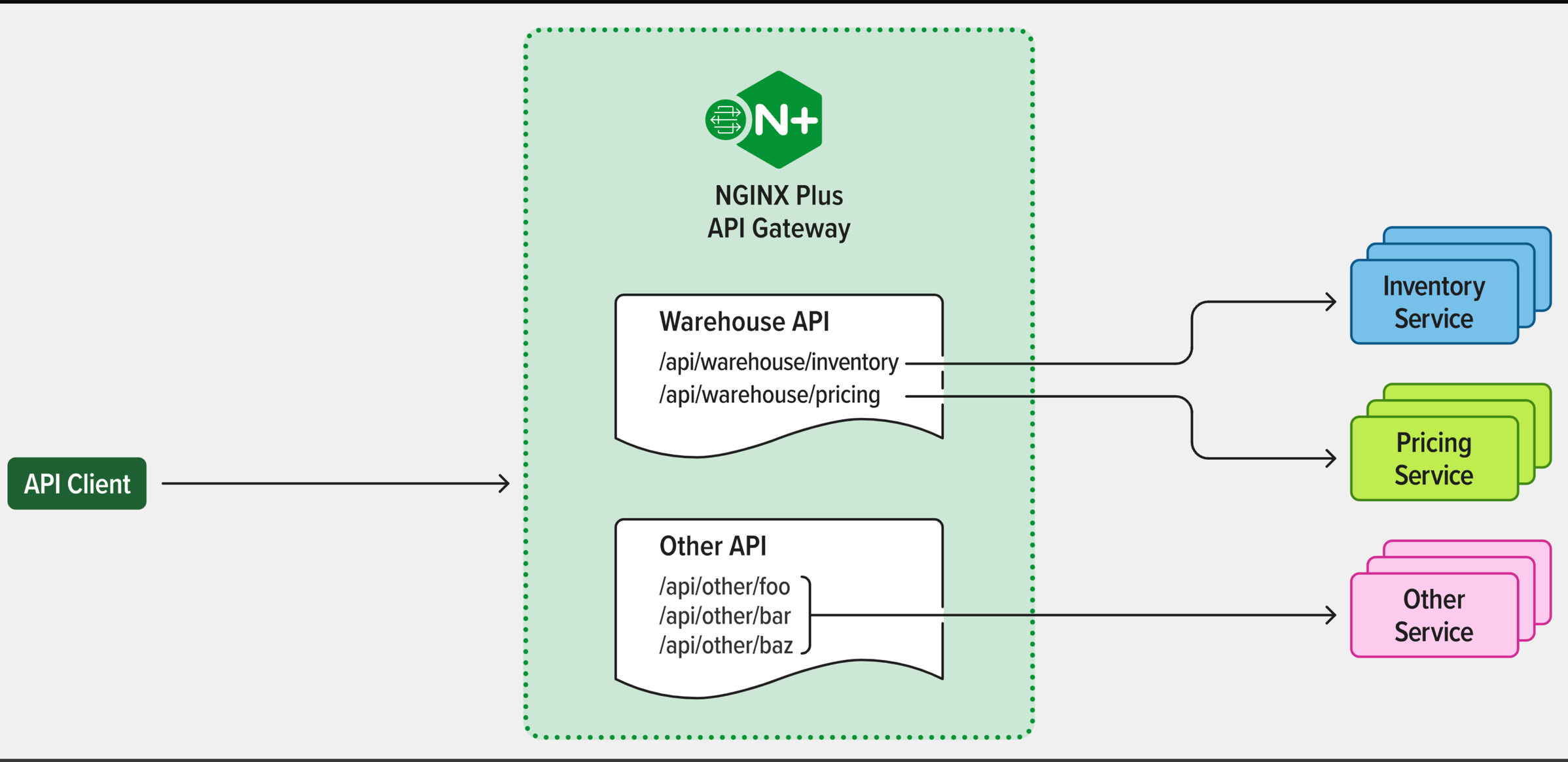
Pros

- Loose coupling
- Ease of maintenance
- Decentralized control
- Asynchronous communication

Cons

- Complexity
- Lack of central control
- Lack of visibility
- Difficulty troubleshooting

API Gateway


NGINX Plus
API Gateway
Warehouse API
/api/warehouse/inventory
/api/warehouse/pricing
Other API
/api/other/foo
/api/other/bar
/api/other/baz
API Client
Inventory Service
Pricing Service
Other Service

API Gateway Capabilities

- Centralized Management:
  - Aggregation: API Gateway can aggregate multiple microservices into a single API endpoint. This simplifies the client-side experience by providing a unified interface and reduces the number of requests needed.
- Security:
  - Authentication and Authorization: API Gateways handle authentication and authorization, ensuring that only authorized users and applications can access the underlying microservices. This centralizes security measures, making it easier to enforce and manage.
- Rate Limiting:
  - Throttling: API Gateways can enforce rate limits to prevent abuse or overuse of the services. This is crucial for controlling traffic and ensuring fair usage of resources.

API Gateway Capabilities

- Load Balancing:
  - Distribution of Requests: API Gateways can distribute incoming requests among multiple instances of microservices, balancing the load and preventing any one instance from being overwhelmed.
- Logging and Monitoring:
  - Centralized Logging: API Gateways provide a centralized point for logging and monitoring API requests. This facilitates easier troubleshooting, debugging, and performance monitoring.
- Transformation and Enrichment:
  - Request/Response Transformation: API Gateways can transform requests and responses to match the expected formats, which can be especially useful when dealing with diverse clients or microservices.
- Caching:
  - Response Caching: API Gateways can implement caching mechanisms to store frequently requested data, reducing the load on microservices and improving response times.

API Gateway Capabilities

- Routing:
  - Dynamic Routing: API Gateways can dynamically route requests to the appropriate microservices based on various criteria, such as the request URL or headers.
- Simplifying Client-Side Logic:
  - Client Agnostic: API Gateways abstract the complexities of microservices architecture from clients. Clients interact with the API Gateway, which then manages the communication with the underlying microservices.
- Scalability:
  - Elasticity: API Gateways contribute to the scalability of a microservices system by efficiently distributing and managing the traffic load.

Q&A