DevOps

Microservices Architecture

Chandara Chea

DD/MM/YYYY
Week IV

What is Software Architecture?

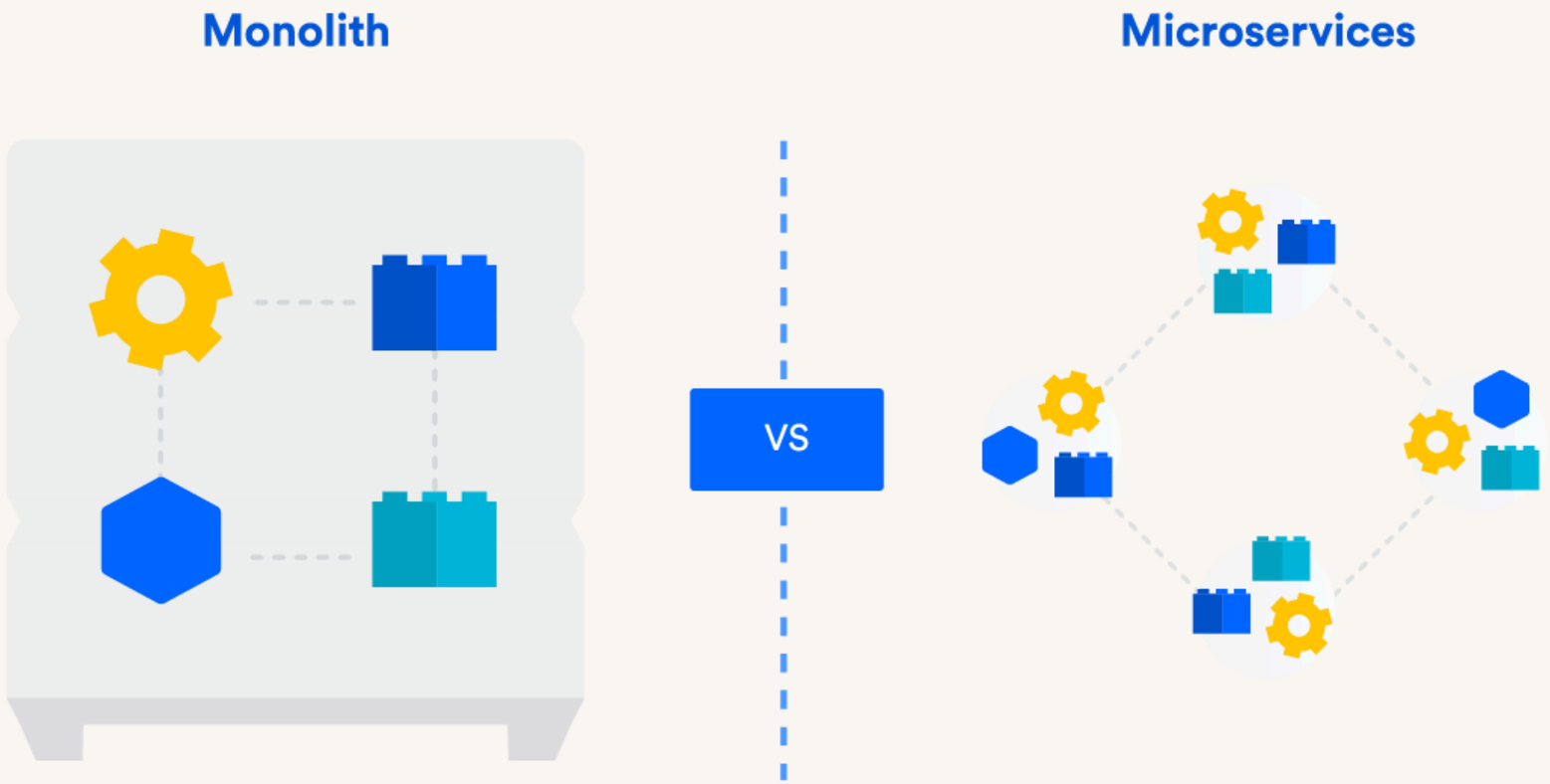
In software engineering, architecture refers to the fundamental structures of a software system and the discipline of creating such structures. A system's software architecture describes its components and how they interact.

In software, architecture defines:

- How application components are organized.
- How components communicate.
- How data is stored and managed.
- How the application is deployed.
- How the application scales.

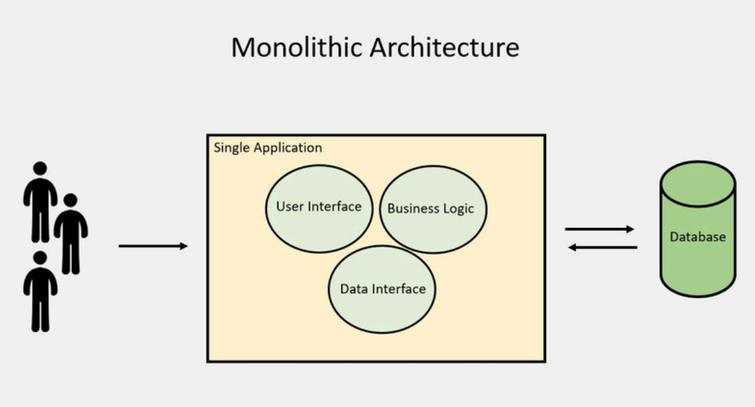
Which software architecture model is best for you in the cloud era?

Monolithic vs Microservices



Monolithic Application

- A monolithic application is a traditional model of a software program, which is built as one unified application and different functionalities are packaged and deployed together.
- All components are tightly coupled, sharing resources and data.
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface. This makes updates restrictive and time-consuming.
- Monoliths can be convenient early on in a project's life for ease of code management, cognitive overhead, and deployment. This allows everything in the monolith to be released at once.



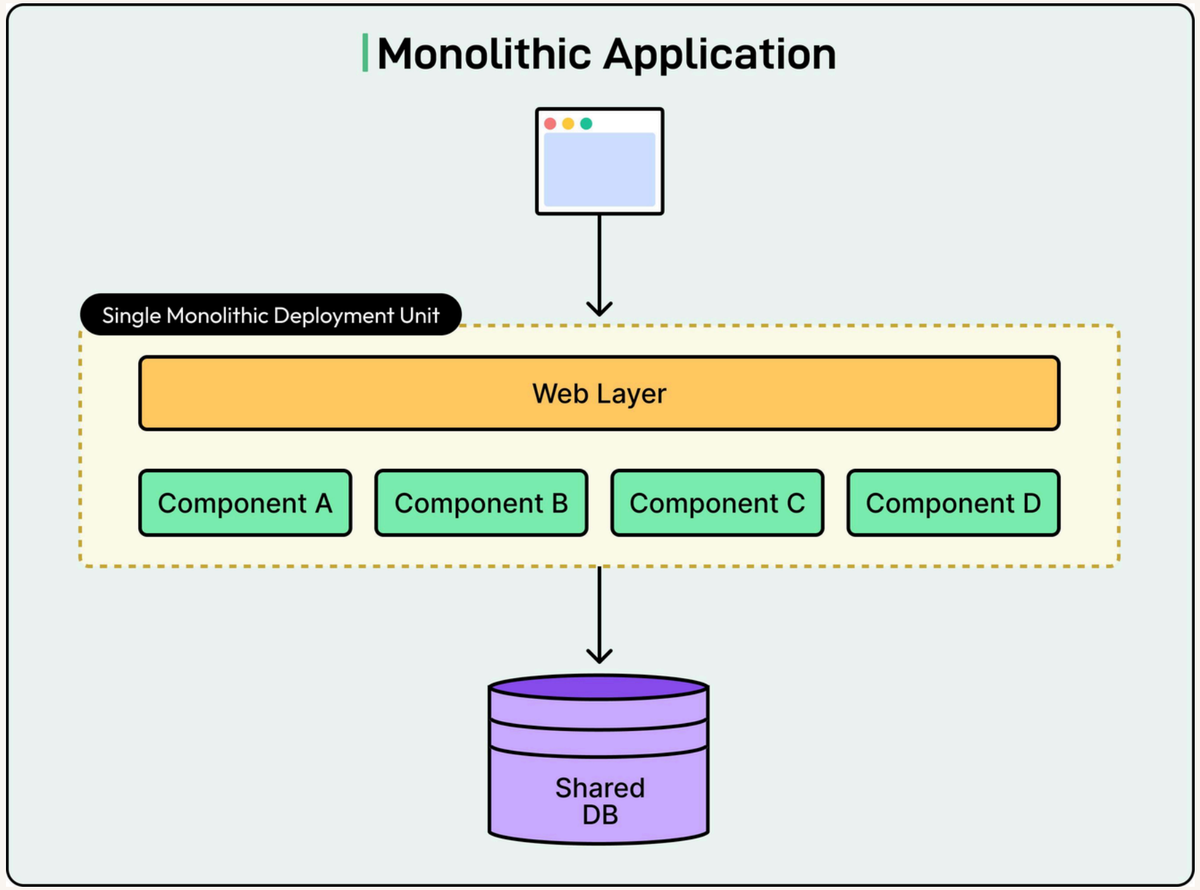
Monolithic Application

1. User sends a request.
2. Application receives the request.
3. The relevant module processes it.
4. Application interacts with the database.
5. Response is returned to the user.

All modules usually run within the same deployable application. Example:

A user places an order:

- Order module creates an order.
- Inventory module reduces stock.
- Payment module processes payment.
- Notification module sends confirmation.



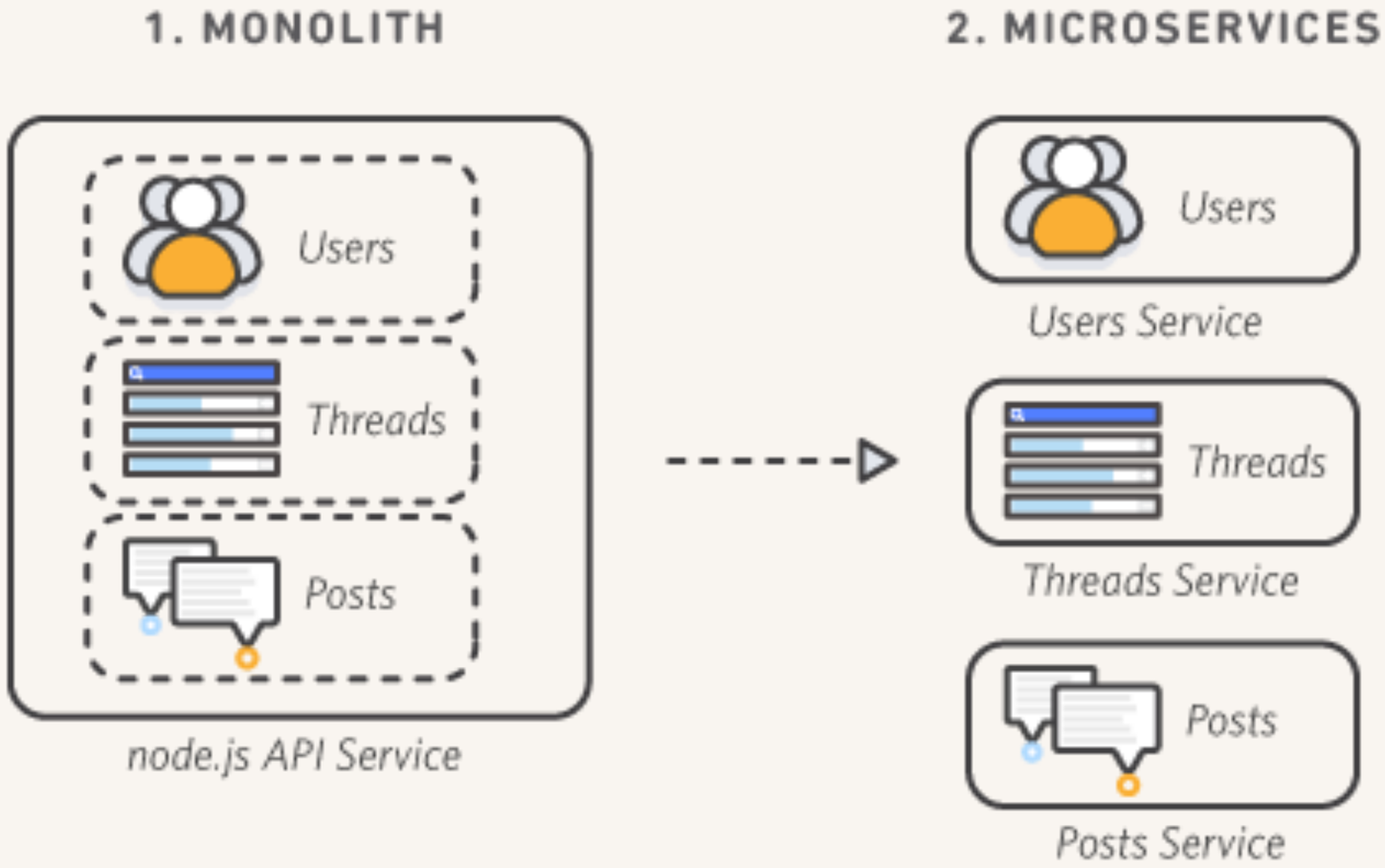
Advantages of A Monolithic Architecture

- 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.

Disdvantages of A Monolithic Architecture

- 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

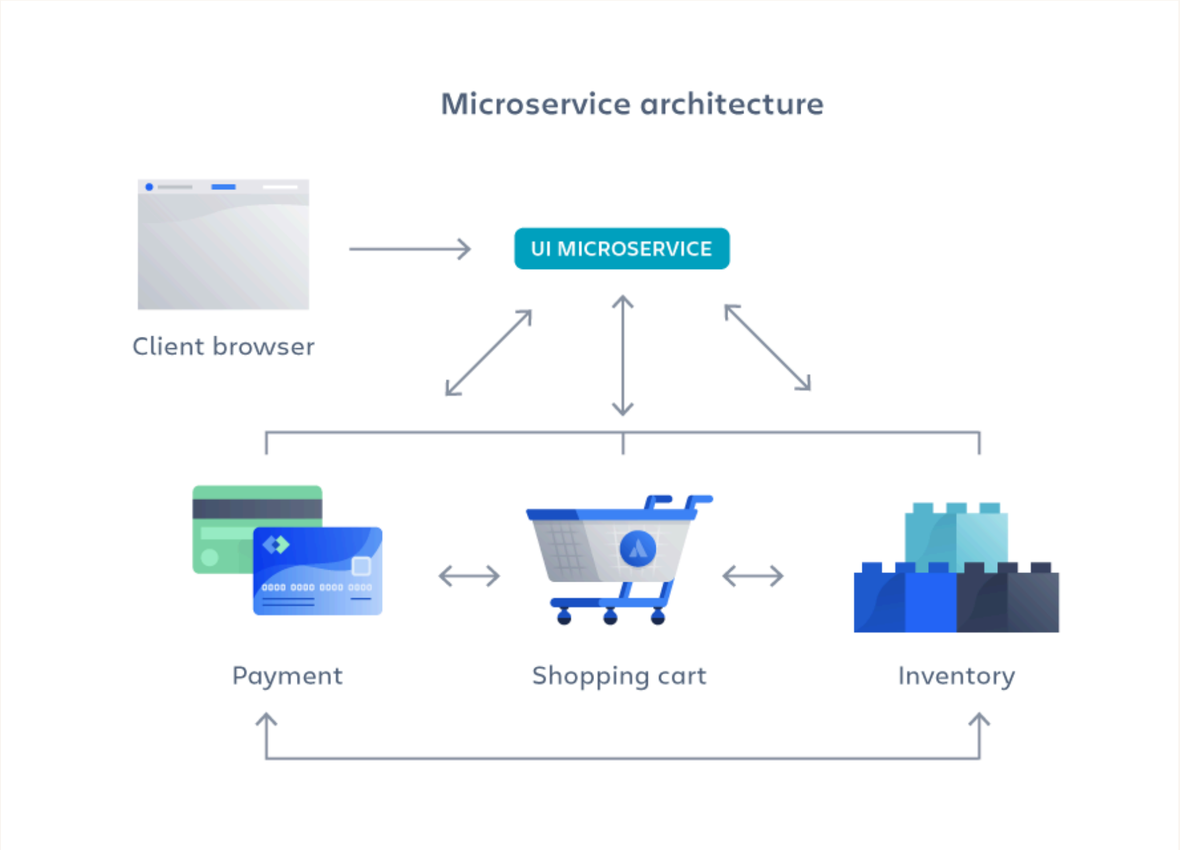


Microservices

- A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services.
- These services have their **own business logic and database with a specific goal**. Updating, testing, deployment, and scaling occur within each service.
- Microservices decouple major business, domain-specific concerns into separate, independent code bases.
- Microservices don’t reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole.
- Each microservice communicates with other services through simple interfaces to solve business problems.
- Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements.

Microservices

- Containers are a well-suited microservices architecture example, since they let you focus on developing the services without worrying about the dependencies. Modern cloud-native applications are usually built as microservices using containers.



Advantages of Microservices

1. **Agility:** Independent teams can develop and maintain services faster.
2. **Flexible Scaling:** Scale individual services based on demand.
3. **Easy Deployment:** Deploy updates independently and roll back changes quickly.
4. **Technological Freedom:** Use the best technology for each specific requirement.
5. **Resilience:** Failure in one service does not necessarily crash the entire application.

Disdvantages of Microservices

1. **Development Sprawl:** Microservices increase system complexity as more services are developed and maintained across multiple teams.
2. **Higher Infrastructure Costs:** Each service requires its own deployment, testing, hosting, and monitoring resources.
3. **Organizational Overhead:** Teams need additional communication and coordination to manage service updates and interfaces.
4. **Debugging Challenges:** Distributed services generate separate logs, making it harder to trace and troubleshoot issues across multiple services.
5. **Lack of Standardization:** Without common standards, teams may use different programming languages, logging formats, and monitoring tools.

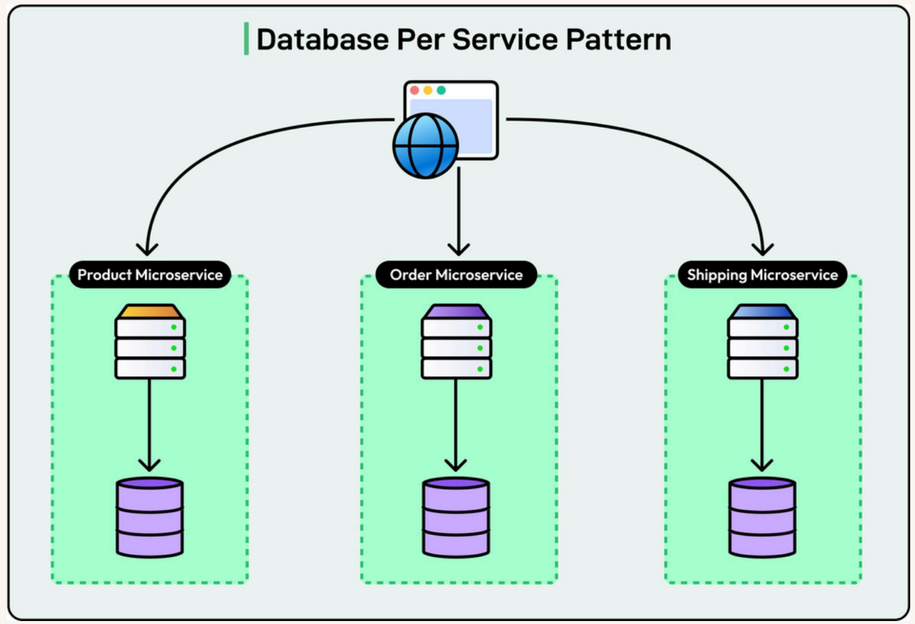
Monolith vs Microservices

- **Scenario A:** A university website with 500 students accessing course information.
- **Scenario B:** An online shopping platform with millions of customers.
- **Scenario C:** A startup building its first application with three developers.

    - Which architecture would you choose?
    - Why?
    - What problems might you face if you choose the wrong architecture?

Microservices: Design Principles

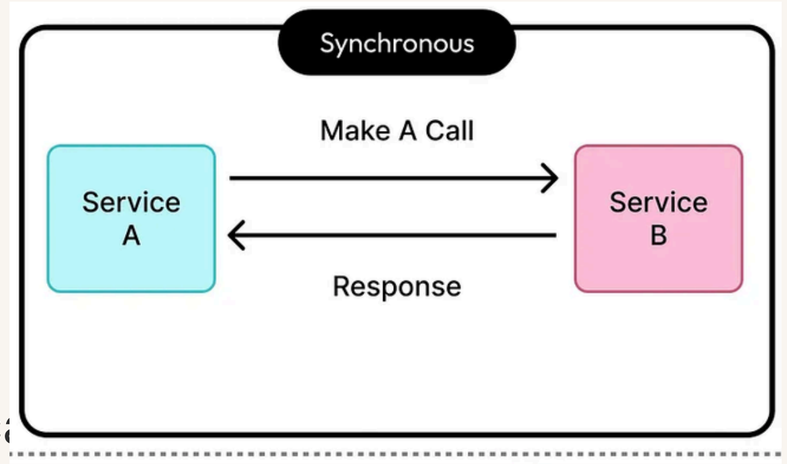
1. **Single Responsibility:** Each service focuses on one specific business function and does it well.
2. **Loose Coupling:** Services communicate through APIs and can be changed independently without affecting other services.
3. **Decentralised Data Management:** Each service owns and manages its own database, choosing the technology that best fits its needs.
4. **Fault Isolation:** A failure in one service should not crash the entire application. Use timeouts, retries, and circuit breakers.
5. **API-First Design:** Define clear and standard APIs so services can communicate and integrate easily, regardless of their programming languages or technologies.



Database per service means logical ownership, not necessarily a physically separate database server. Several databases may run on the same database infrastructure.

Fundamentals of Microservices Communication

- **Synchronous Communication**
  - The caller service sends a request and waits for a response from the receiver service.
  - Pros:
    - Simple to implement
    - Easy to debug and trace
    - Immediate response
  - Cons:
    - Tightly coupled; if the receiver is slow or down, the ca
    - Less resilient under high load
  - E.g.
    - Order Service requests payment from Payment Service.
    - Payment Service processes the request.
    - Payment Service returns success or failure.
    - Order Service continues.

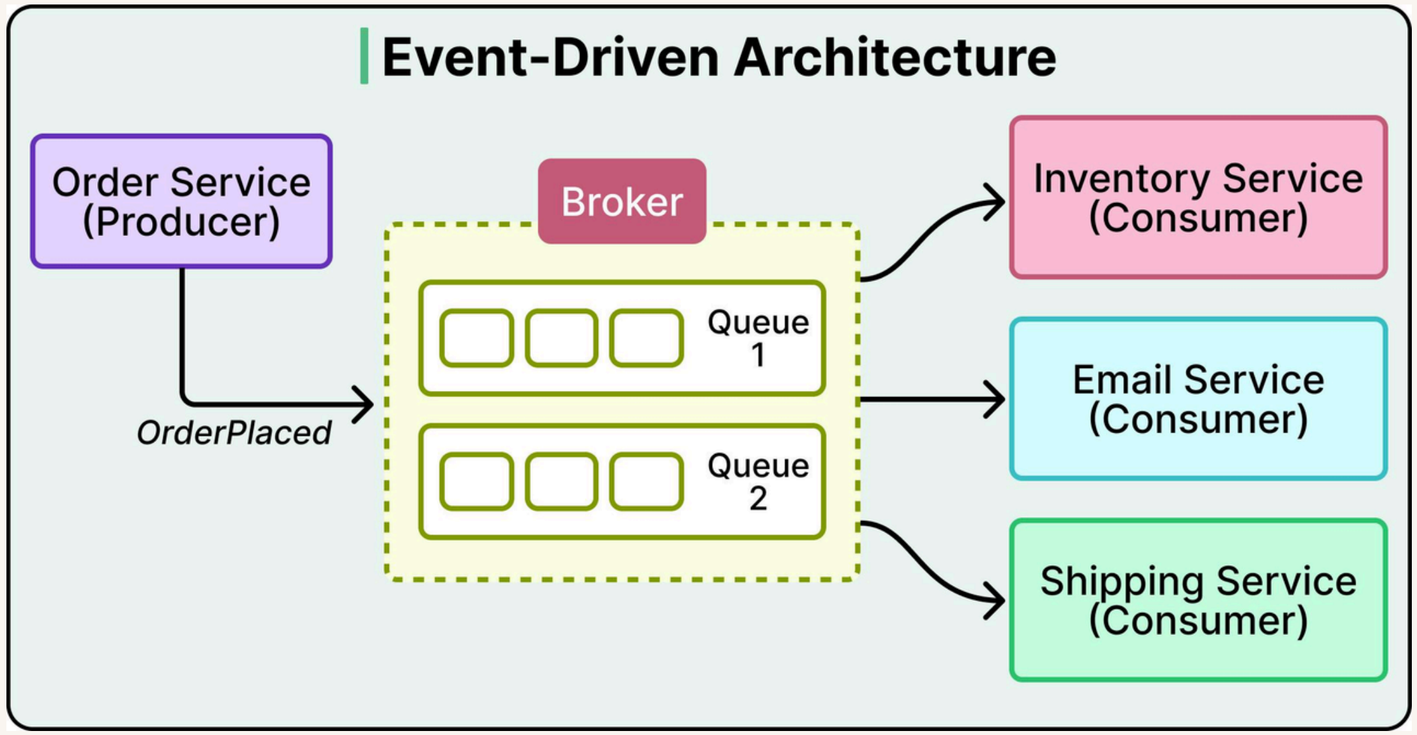


Fundamentals of Microservices Communication

- **Asynchronous Communication**
  - Definition: The caller service sends a message/event and continues its work without waiting for a response. E.g: Kafka, RabbitMQ, AWS SNS/SQS
  - Pros:
    - Loosely coupled
    - More resilient; the system continues even if some services are down
    - Scalable; can handle high volume of events
  - Cons:
    - Harder to debug
    - Eventual consistency instead of immediate consistency

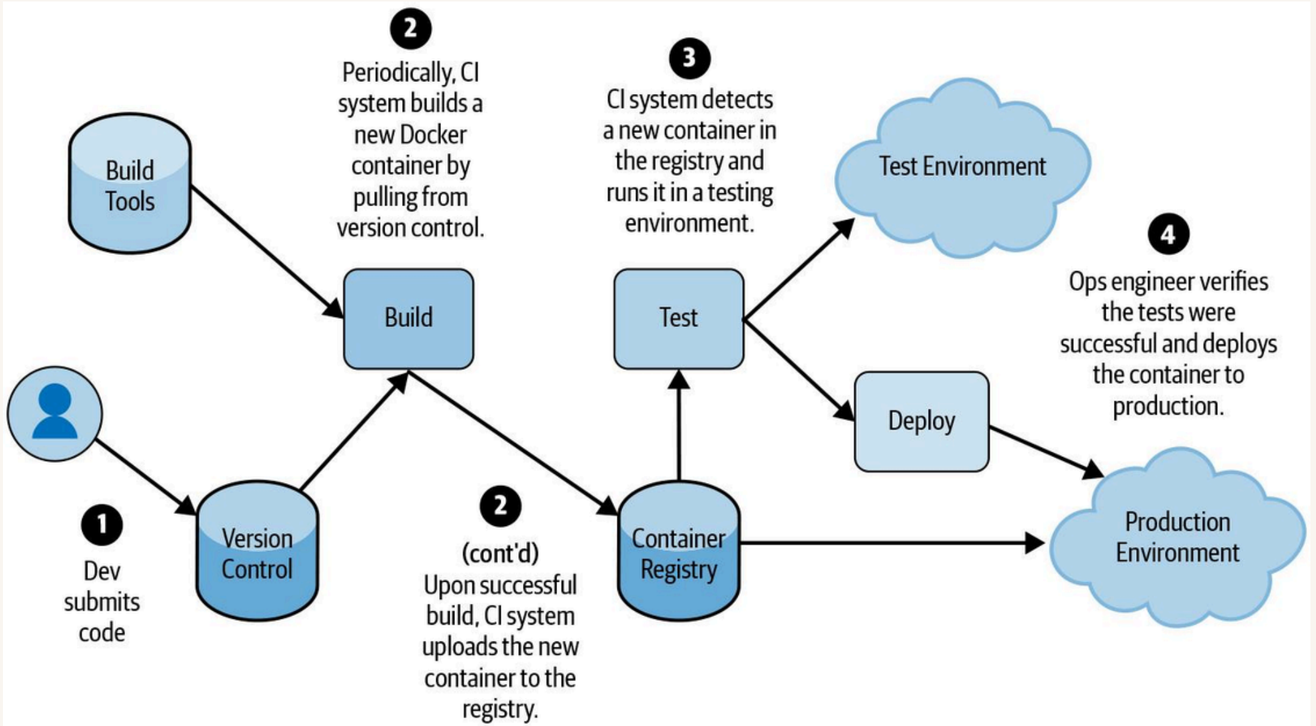
Fundamentals of Microservices Communication

- **Asynchronous Communication**
  - Example: All happen independently and asynchronously)
    - Order Service publishes OrderCreated.
    - Message Broker / Queue receives and distributes the event.
    - Inventory Service receives the event.
    - Other interested services process it independently.



Microservices Both Enable and Require DevOps

- Microservices and DevOps complement each other.
- Microservices allow teams to develop and deploy services independently.
- DevOps provides the practices and automation needed to build, test, deploy, monitor, and operate those services reliably.
- Without DevOps automation, managing 20 or 50 microservices can become extremely difficult.



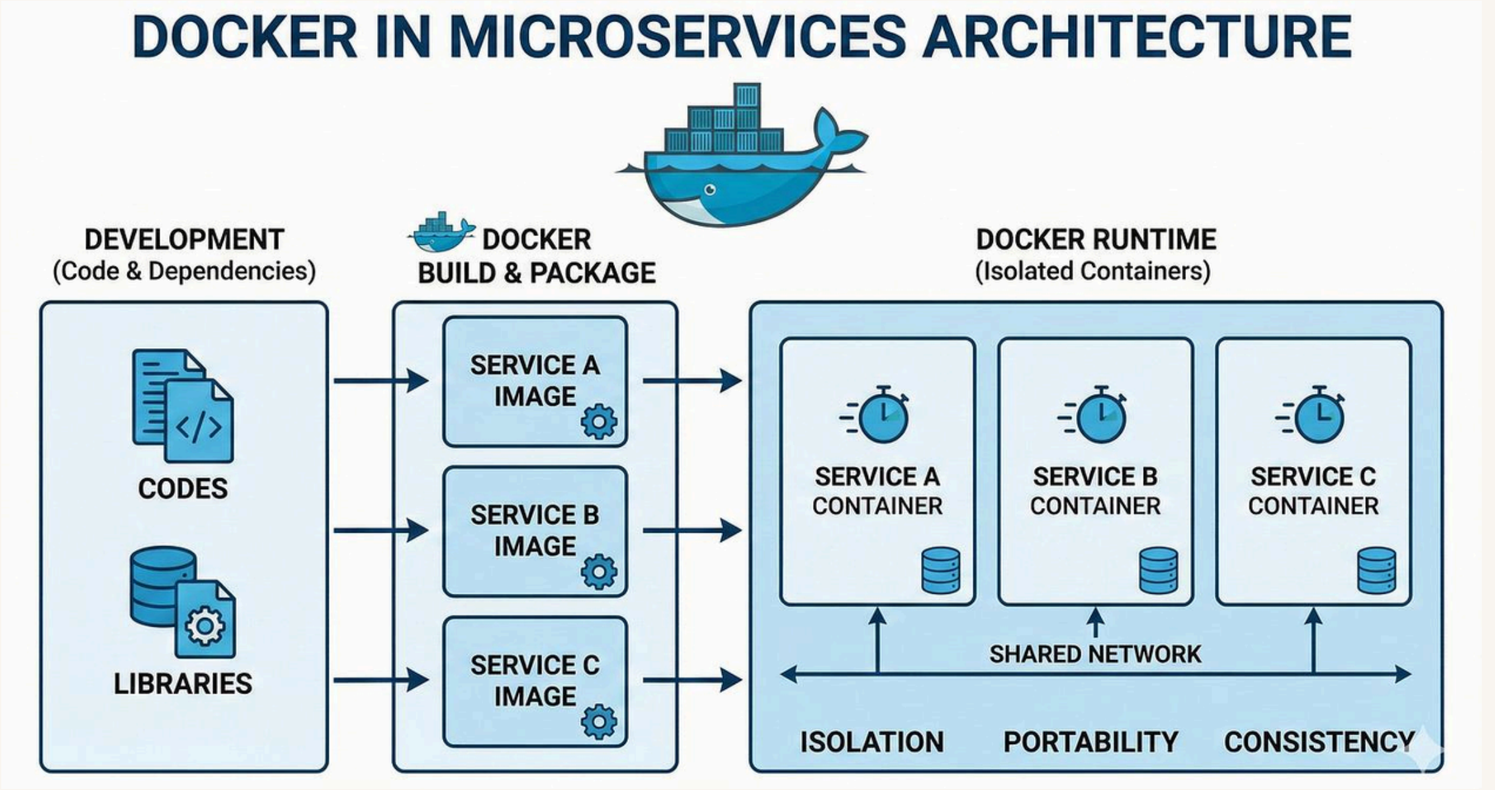
Key Enabling Tools and Technologies

These technologies help microservices communicate, run, and work independently.

1. Containers (Docker)

- Package each microservice with its application code and dependencies.
- Lighter and smaller than virtual machines because they share the host operating system.
- Start and stop quickly.

Why? Run each microservice independently in its own container.

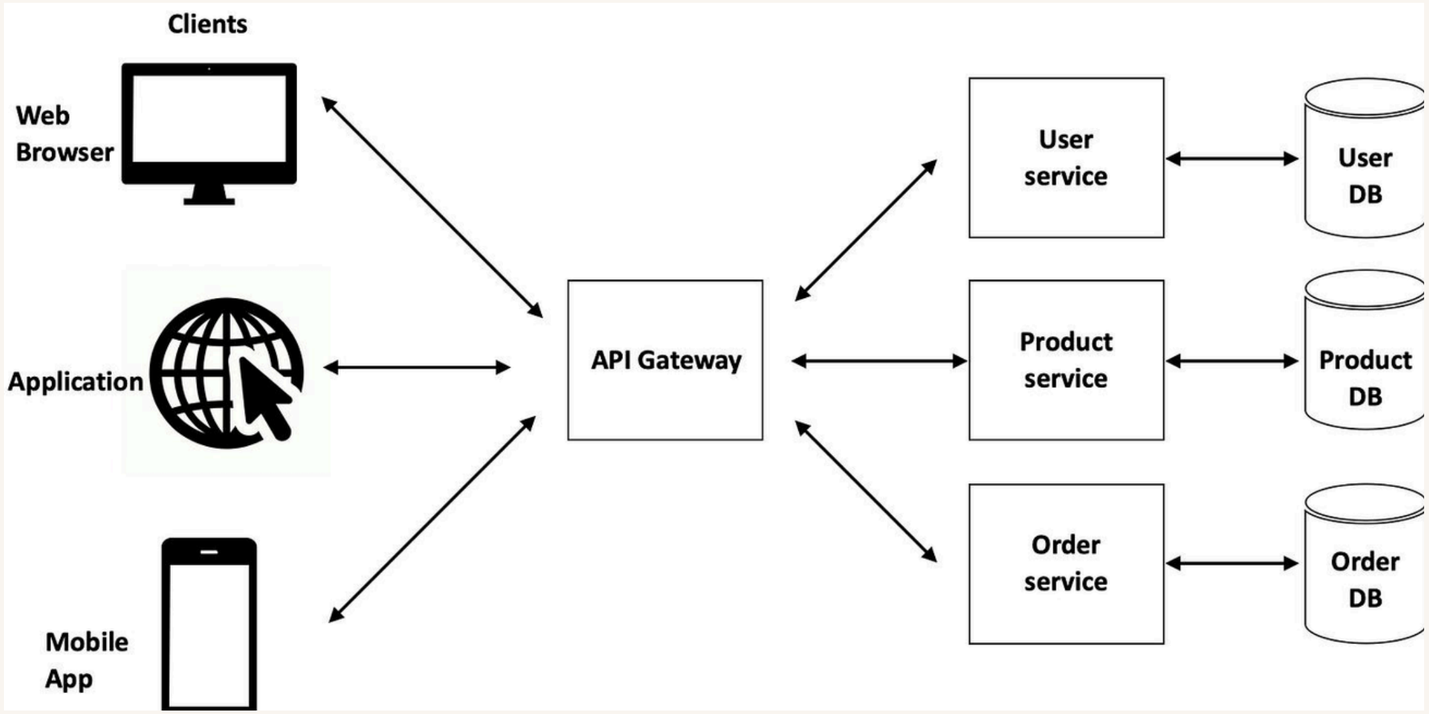


Key Enabling Tools and Technologies

2. API Gateway

- Acts as a single entry point for client requests.
- Routes requests to the correct microservice.
- Can handle authentication, security, and request limits.
- Can combine responses from multiple services.

<u>Why? Clients do not need to communicate with every microservice directly.</u>

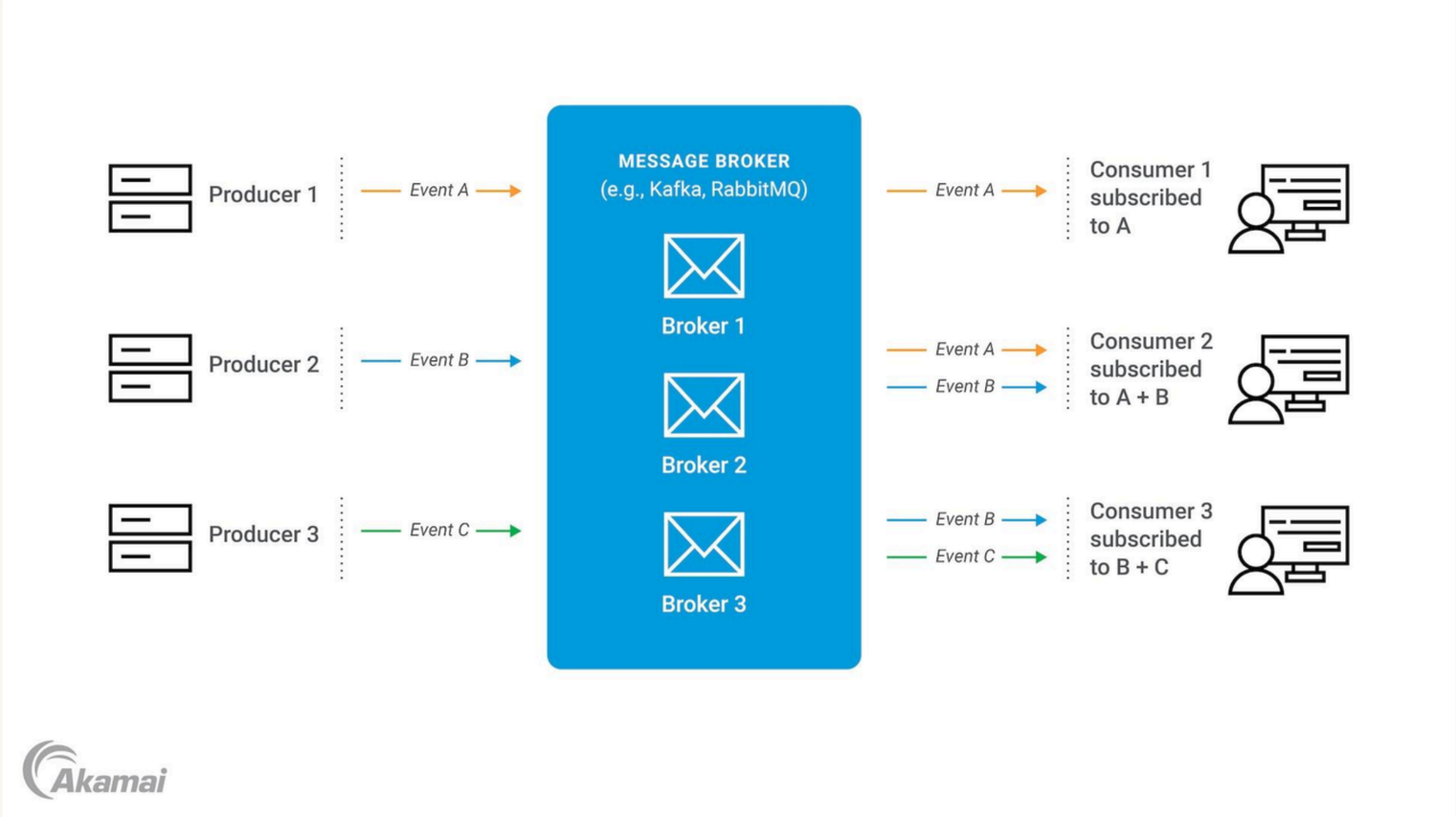


Key Enabling Tools and Technologies

3. Messaging and Event Streaming

- Allow services to communicate without waiting for a response.
- Services publish messages or events to a message broker.
- Other services receive and process messages when ready.
- Reduce direct dependencies between services.

Examples: Apache Kafka, RabbitMQ, Amazon SNS and SQS.

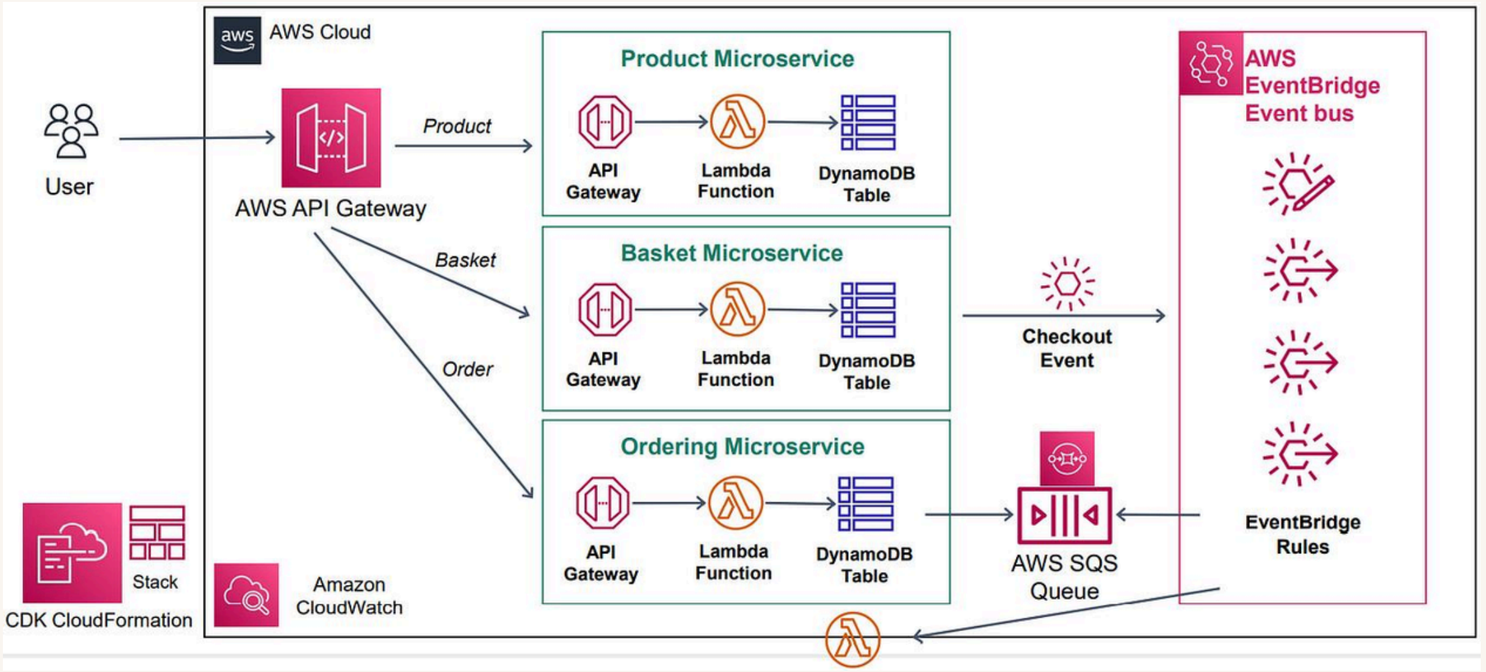


Key Enabling Tools and Technologies

4. Serverless Computing

- Run small pieces of code (functions) without managing servers.
- Functions execute when triggered by an event or request.
- Cloud providers manage the underlying infrastructure.
- Often used for small tasks that do not need a continuously running service.

Example: AWS Lambda. A function can perform a small task, such as sending an order confirmation email.



Best Practices

1. Don't Start with Microservices

- Start with a monolithic application.
- Move to microservices only when the application becomes difficult to maintain, update, or scale.
- Split the application into smaller services when there is a clear need.

2. Don't Use Microservices Without DevOps

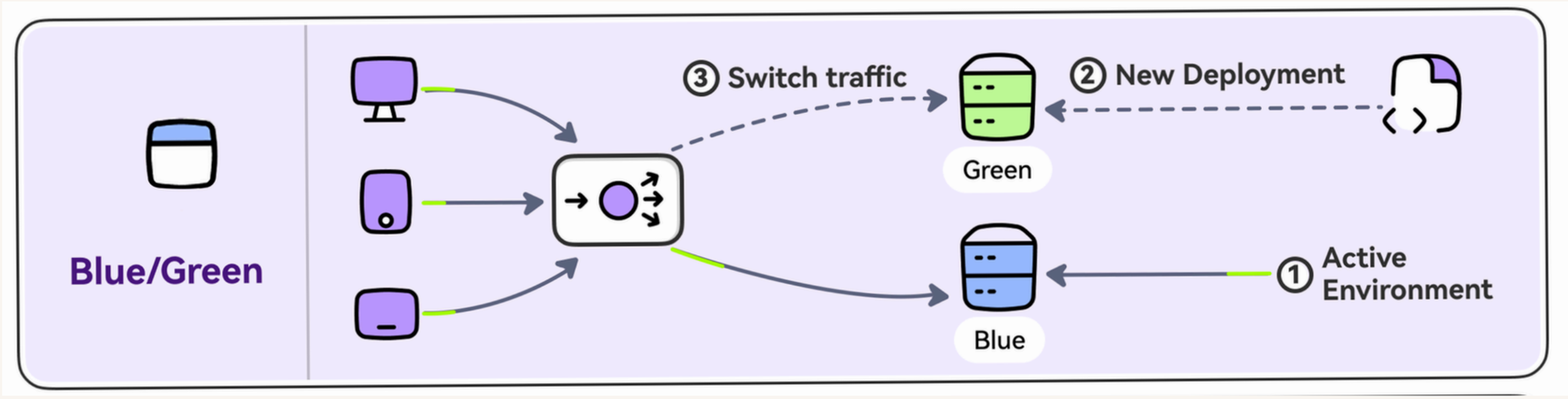
- Microservices create many services that need to be deployed and managed.
- Use CI/CD pipelines to automate building, testing, and deployment.
- Use monitoring and logging to detect problems.

3. Don't Create Too Many Small Services

- Avoid splitting an application into too many tiny services.
- Start with services that handle related tasks together.
- Split a service only when it becomes difficult to maintain, deploy, or scale.
- Each additional service adds communication and management complexity.

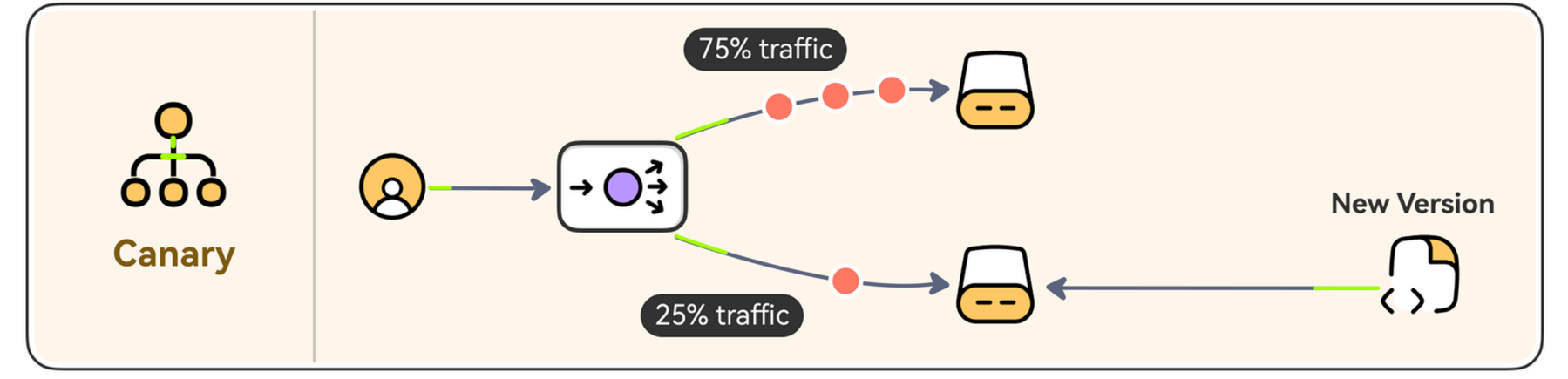
Deployment Strategies

- How it works: Maintain two identical environments:
  - Blue (current live)
  - Green (new version)
  - Switch traffic to green after testing.
- Pros: Instant rollback if issues arise, no downtime.
- Cons: Requires double resources, higher cost.
- Use case: Mission-critical apps needing zero downtime.



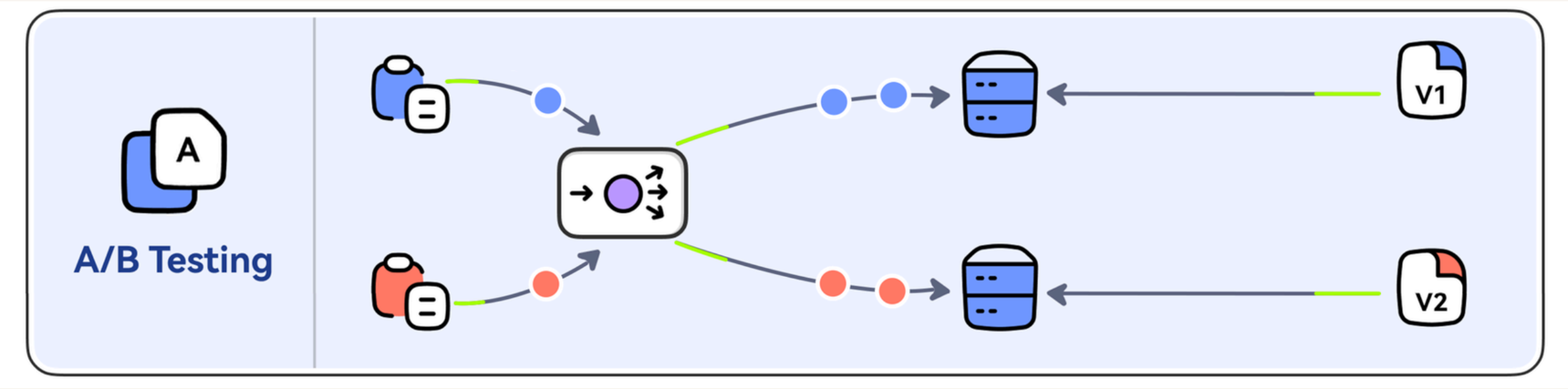
Deployment Strategies

- How it works:
  - Release the new version to a small subset of users first, monitor performance, then gradually expand.
- Pros: Limits exposure to potential issues, safe way to test in production.
- Cons: More complex monitoring needed.
- Use case: Large-scale apps with millions of users.



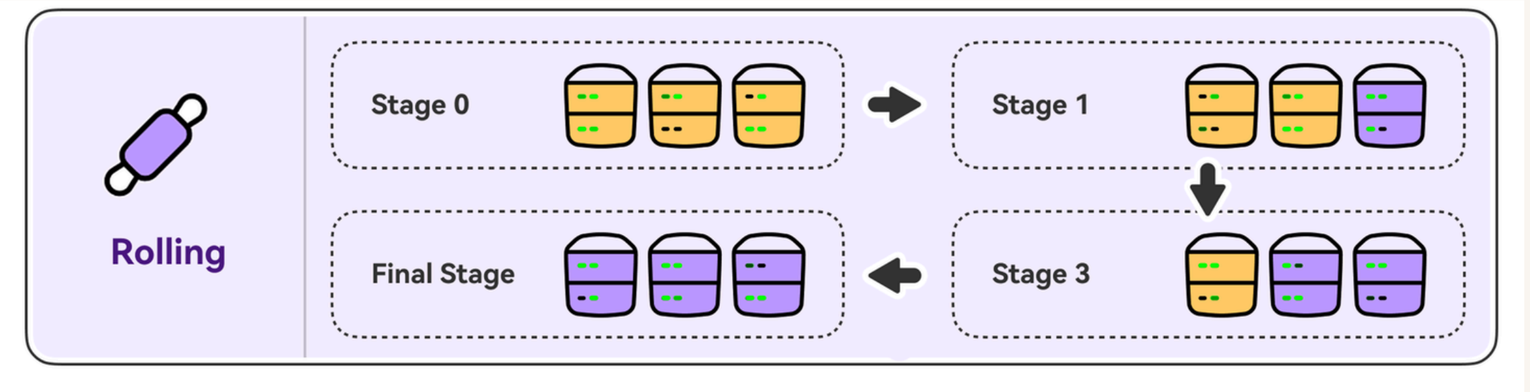
Deployment Strategies

- How it works:
  - Show two versions of a feature or page to different user groups simultaneously; compare performance on key metrics.
- Pros: Identify the better-performing version based on real user behavior.
- Cons: Only a portion of users see the new version; requires careful metric tracking.
- Use case: Testing UI changes, new features, or marketing content to optimize conversions or engagement.



Deployment Strategies

- How it works:
  - Gradually replace old instances with new ones, one batch at a time.
- Pros: Minimal downtime, users gradually transition.
- Cons: Old and new versions coexist → risk of incompatibility.
- Use case: Applications running on multiple servers or containers.



References

https://www.salesforce.com/in/blog/microservice-design-principles/
https://blog.bytebytego.com/p/a-guide-to-microservices-architecture
https://bytebytego.com/guides/what-does-a-typical-microservice-architecture-look-like/
https://circleci.com/blog/deployment-strategies-types-trade-offs-and-how-to-choose/