Part II: Implementation

- Chapter 5: Implementing Microservice Communication
  - Technology Choices:

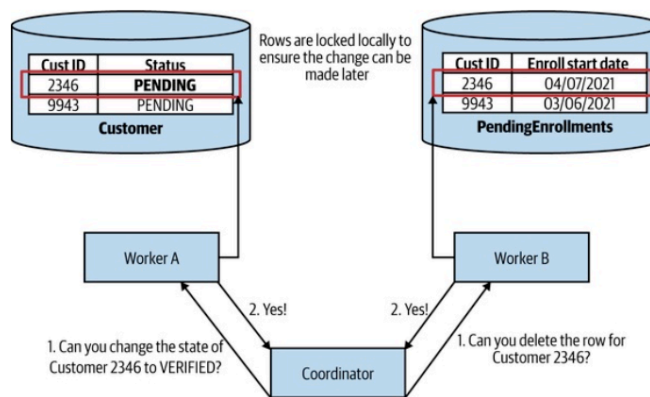|	rpc	rest	graphql	message broker
pros	Simple to implement and use for straightforward service calls Direct method invocation, resembling local calls Supports complex data types and operations Generally lower latency due to binary protocols and HTTP/2 Higher performance due to compact payloads and efficient serialization	Stateless, allowing scalability and reliability Leverages HTTP, making it easy to use and widely supported Flexible and decoupled architecture Generally higher due to text-based protocols and HTTP/1.1 Lower performance due to larger payloads and slower parsing	Clients can request exactly the data they need Reduces over-fetching and under-fetching of data Strongly typed schema enhances query validation and introspection	Decouples producers and consumers, enabling asynchronous communication Scalability and reliability with message persistence and retry mechanisms Supports various communication patterns (pub/sub, queueing)
cons	Tight coupling between client and server Harder to scale due to synchronous nature Poor error handling and limited flexibility	Overhead of HTTP can affect performance No built-in support for real-time communication Fixed data structure, hard to change without versioning	More complex to implement and optimize Potential for complex queries impacting server performance Requires a shift in thinking from traditional REST paradigms	Additional complexity in architecture and maintenance Potential latency due to message queuing Requires careful design to handle message ordering and duplication
mechanism	The client makes a request to execute a method on the server. The request is serialized, sent over the network, and deserialized by the server. The server processes the request, serializes the response, and sends it back to the client supports HTTP/2 multiplexing Persistent connections, multiplexing	Client sends HTTP requests to RESTful endpoints Server processes requests and returns responses, typically in JSON or XML format Each resource is identified by a URI, and operations are performed using HTTP methods typically uses HTTP/1.1 No multiplexing, separate connections for each request	Clients send queries to a single endpoint Server parses and validates the query against the schema Resolvers fetch and return the requested data	Producers send messages to a message broker Broker stores, routes, and delivers messages to consumers Consumers process messages, with acknowledgment mechanisms ensuring reliable delivery

use cases	Microservices requiring high-performance and low-latency communication	Public APIs and web services	APIs requiring flexibility and efficiency in data retrieval	Event-driven architectures
	Internal service-to-service communication	CRUD operations on resources	Applications with complex and varied front-end requirements	Microservices communication requiring decoupling and resilience
	High-throughput microservices in a data center	Services that need to be consumed by different clients (web, mobile, etc.).	Mobile applications where bandwidth optimization is crucial	Real-time data streaming and processing
		Public APIs with broad client compatibility		

-
  -
    - Serialization Formats
      - Textual Formats: json(more compact and simple), xml (more robust schemas and xpath for data extraction)
      - Binary Formats: protocol buffer, optimized payload size and read/write efficiencies
    - Schemas: defining what endpoints expose and accept
      - Advantages of Schemas
        - Explicit Representation
          - Schemas provide a clear definition of what a microservice endpoint exposes and accepts, facilitating easier development and consumption.
          - They reduce the need for extensive documentation
        - Preventing Breakages
          - Schemas help catch structural breakages (changes in endpoint structure) by comparing schema versions
          - They offer a safeguard against accidental changes that could break consumer expectations
      - Types of Contract Breakages
        - Structural Breakages: Changes in the structure that make a consumer incompatible, such as removing fields or adding required fields
        - Semantic Breakages: Changes in behavior without altering structure, like changing the logic of a method, which can break consumer expectations
      - Schemas vs. Schemaless Communication
        - With Schemas:
          - Provide compile-time safety similar to statically typed languages
          - Help in catching structural issues early, allowing testing to focus on other problems
        - Without Schemas
          - Implicit schemas exist through consumer assumptions about data structure
          - Testing must catch both structural and semantic issues, akin to dynamically typed languages
      - Arguments and Recommendations
        - Explicit Schemas
          - Explicit schemas are preferred for clarity and reducing misunderstandings between teams
          - They offer a significant advantage in managing changes and ensuring compatibility
        - Schemaless Endpoints
          - Argued to require less work, but this view may overlook the benefits of explicit schemas in preventing breakages and aiding communication
          - Better tooling and imagination in schema use can enhance their value
      - Contextual Flexibility: In environments with low change costs or where the same team owns both client and server, the necessity for schemas can be relaxed

- DRY in microservice
  - Challenges of Code Reuse in Microservices
    - Coupling Issues:
      - Sharing code via libraries can lead to tight coupling between microservices. Changes in shared libraries necessitate updates across all dependent services, which can be cumbersome and error-prone
      - Internal shared code (e.g., logging libraries) is generally safe, but domain-specific shared libraries can create external coupling.
    - Updating Shared Libraries
      - Updating shared libraries across microservices requires redeployment of each service using the library
      - Synchronized updates can lead to widespread deployment challenges
  - Client Libraries
    - Client libraries simplify service usage but risk leaking server logic into clients, breaking cohesion
    - AWS SDK model: Community or separate team-developed SDKs, allowing clients to choose when to upgrade, avoiding direct server-client coupling
    - Netflix's client libraries: Ensure reliability and scalability, though can lead to coupling issues over time.
  - Best Practices for Client Libraries
    - Separate client code handling transport protocols (e.g., service discovery, failure handling) from service-specific logic
    - Allow flexibility in technology stacks, and ensure clients control their upgrade timing to maintain independent service releases
- Services discovery
  - Overview: Service discovery helps in identifying what services are running and where they are located
  - Key Components of Service Discovery
    - Service Registration: A mechanism for a service instance to register itself and announce its presence
    - A method to find the registered service when needed
  - most common solutions to service delivery
    - DNS
    - Dynamic Service Registries
      - overview: Dynamic service registries address the limitations of DNS in dynamic environments by allowing services to register themselves with a central registry, making it easier to locate them
      - ZooKeeper
      - Consul
      - etcd and Kubernetes: etcd is a configuration management store bundled with Kubernetes, offering similar capabilities to Consul. In Kubernetes, service discovery identifies which pods belong to a service by pattern matching on pod metadata. Requests to a service are routed to one of the pods in the service
- **Service Meshes and API Gateways**

Figure 5-6. An overview of where API gateways and service meshes are used
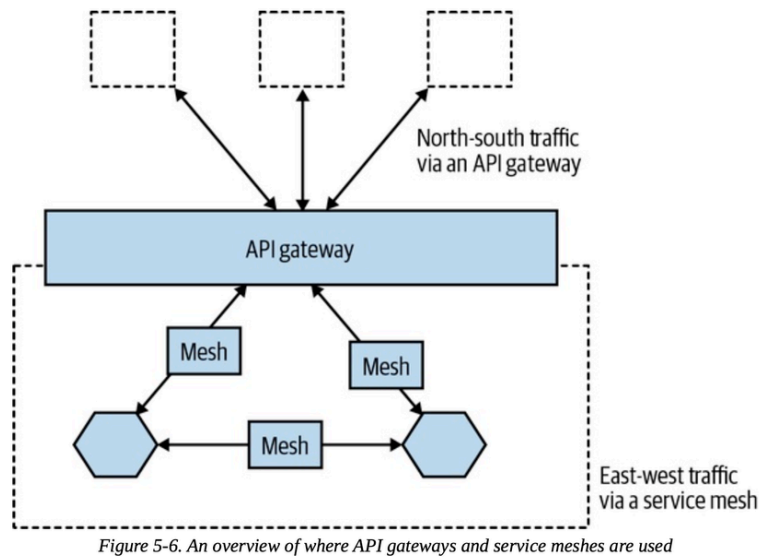
- API Gateway
  - **Location**: Positioned at the perimeter of the system
  - **Function**: Manages north-south traffic, which is the traffic entering or leaving the system from the outside world
  - **Role**: Acts as a reverse proxy, handling requests from external clients and directing them to the appropriate internal microservices
- **Service Meshes**
  - **Location**: Within the internal network of the system
  - **Function**: Manages east-west traffic, which is the traffic between microservices within the network
  - **Role**: Handles inter-microservice communication, providing features like load balancing, service discovery, and mutual TLS
- Chapter 6. Workflow
  - **Database Transactions**:
    - ACID in Microservices: Microservices can still use ACID transactions, but their scope is limited to changes within a single microservice's database. The challenge arises when operations span multiple databases or microservices, as seen in the example of onboarding a new customer where changes are made in two separate databases
  - **Distributed Transactions—Two-Phase Commits**
    - Overview: The two-phase commit (2PC) algorithm is used to handle distributed transactions, where multiple processes need to be updated as part of a single operation. It consists of two phases: the voting phase and the commit phase.while 2PC can coordinate distributed transactions, it introduces complexities and potential inefficiencies, making it less ideal for high-latency or long-running operations
    - Phases:
      - Voting Phase

Figure 6-3. In the first phase of a two-phase commit, workers vote to decide if they can carry out some local state change

- - - A central coordinator asks all participating processes (workers) if they can make a specific state change
    - Each worker evaluates the request and responds whether it can guarantee the change
    - If any worker votes "no," the transaction is aborted and all workers roll back any preparations they made
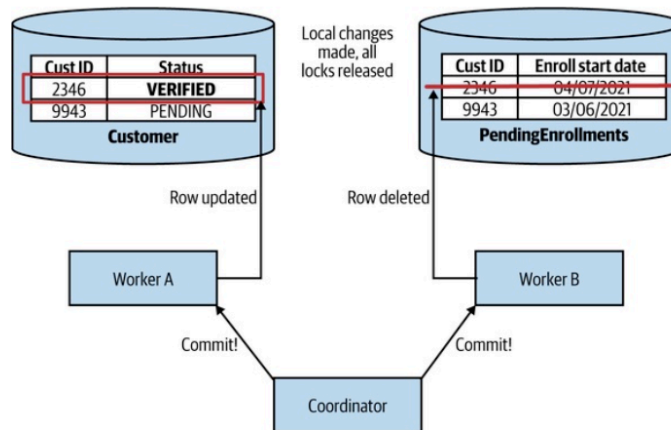 - Commit Phase

Figure 6-4. In the commit phase of a two-phase commit, changes are actually applied

- - - If all workers vote "yes," the coordinator sends a commit message
    - Workers then apply the state changes and release any locks they were holding
  - Challenges
    - Intermediate States:There is no guarantee that all commits happen simultaneously, potentially leading to temporary inconsistency
    - Lock Management: Workers may need to lock resources to guarantee future changes, increasing the complexity of managing locks and avoiding deadlocks
    - Failure Modes: Various failure scenarios can occur, such as a worker agreeing to a change but then failing to respond during the commit phase, sometimes requiring manual intervention
    - Latency and Scalability: Increased latency and the number of participants can exacerbate issues, making 2PC more suitable for short-lived operations due to the resource locks it requires.
  - **Distributed Transactions—Just Say No**
    - Key Advice: Avoid using distributed transactions like the two-phase commit (2PC) to coordinate state changes across microservices. The complexities and inefficiencies they introduce are often not worth the trouble
    - Alternatives
      - Do Not Split the Data:
        - If you require atomic and consistent state management, keep the data in a single database
        - Maintain the functionality that manages this state within a single service or monolith
      - Sagas
        - For operations spanning multiple services that need to avoid locking and might take a long time (minutes to months), consider using sagas.
        - Sagas provide a way to manage long-lived transactions by breaking them into smaller, manageable steps with compensating actions for rollback
    - Note on Database Distributed Transactions
      - Some large-scale databases, like Google's Spanner, use distributed transactions successfully within a single logical database
      - These distributed transactions are applied transparently by the underlying database and not managed at the application level
      - Spanner's success required significant infrastructure, including expensive data centers and satellite-based atomic clocks
  - Sagas

- Overview: Sagas coordinate multiple state changes without long-term locking by modeling steps as discrete, independently executable activities
- Order Fulfillment Example: An order fulfillment saga involves multiple services (e.g., Warehouse, Payment Gateway) each handling different steps within their local transactional scope
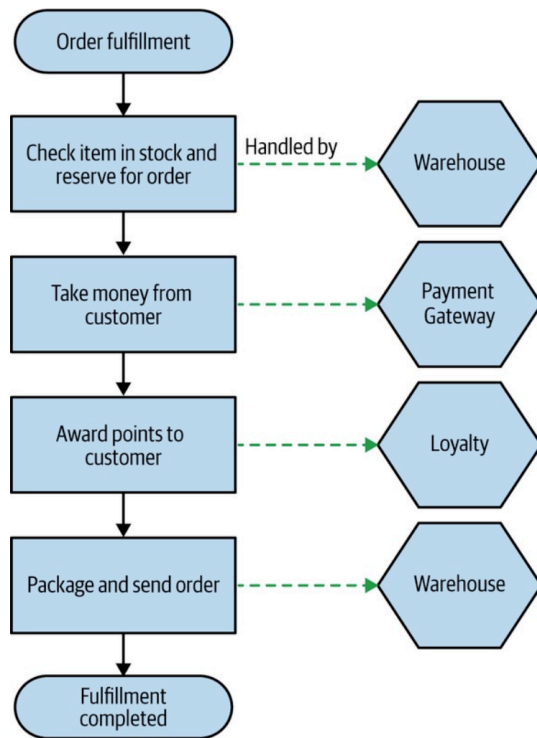

Figure 6-5. An example order fulfillment flow, along with the services responsible for carrying out the operation

- Failure Modes
  - Backward Recovery: Rolling back by undoing previously committed transactions using compensating actions.
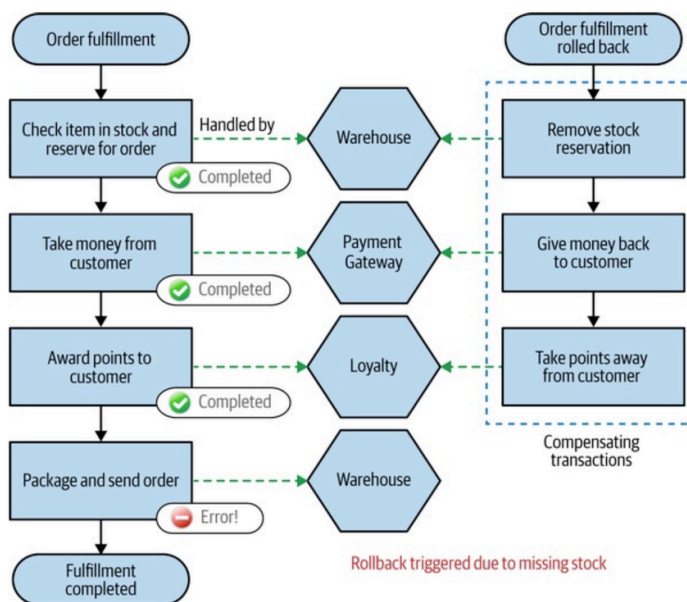

Figure 6-7. Triggering a rollback of the entire saga

  - Compensating Transactions: New transactions to revert changes made by committed transactions.
  - **Example**: If an order cannot be fulfilled, compensating transactions might include refunding payment and canceling loyalty points

- Forward Recovery: Continuing from the failure point by retrying transactions, requiring sufficient information persistence for retries
  - Workflow Optimization: **Reordering Steps**: Placing failure-prone steps earlier in the process to reduce the need for rollbacks
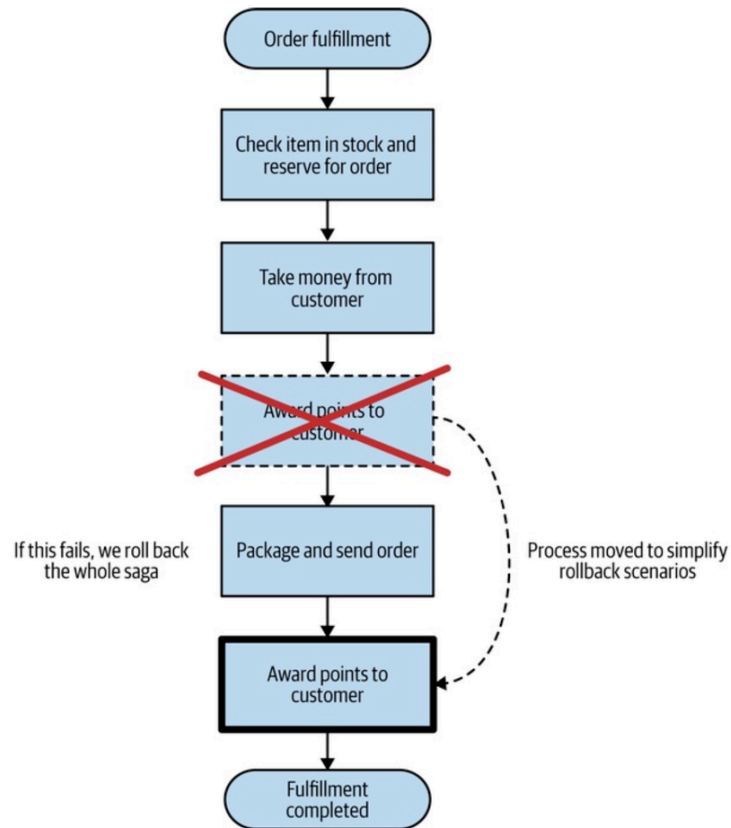

Figure 6-8. Moving steps later in the saga can reduce what has to be rolled back in case of a failure

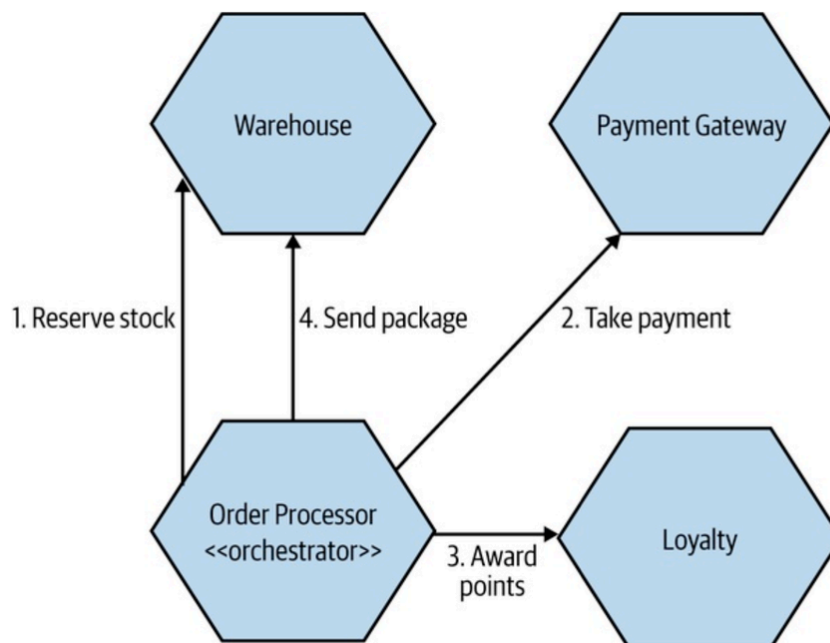 - Implementation Styles
    - Orchestrated Sagas

Figure 6-9. An example of how an orchestrated saga may be used to implement our order fulfillment process

- Central coordinator (orchestrator) controls the process, triggering compensating actions if needed
- Pros: Clear visibility and explicit modeling of processes
- Cons: Higher domain coupling and risk of centralized logic
- Orchestration is easier for single-team ownership
- orchestration relies more on request-response interactions
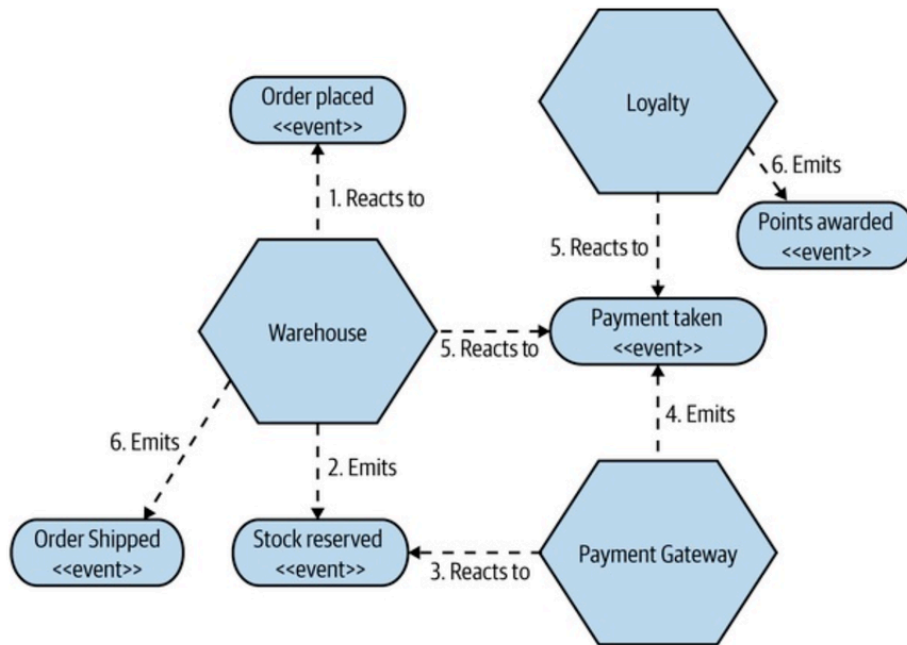- Choreographed Sagas

Figure 6-10. An example of a choreographed saga for implementing order fulfillment

- Distributed responsibility among services, often using events for coordination
- Pros: Reduced domain coupling and decentralized logic
- Cons: Harder to track the overall process and state of the saga
- choreography is better for distributed teams.
- Choreography typically involves heavier use of events
- Mixing Approaches: Use orchestration for some processes and choreography for others, or mix within a single saga for different parts of the process
- Chapter 7. Build
  - Mapping Source Code and Builds to Microservices
    - Overview: When organizing code for microservices, there are different approaches, each with its own pros and cons. Here's a summary of the key patterns and their implications:
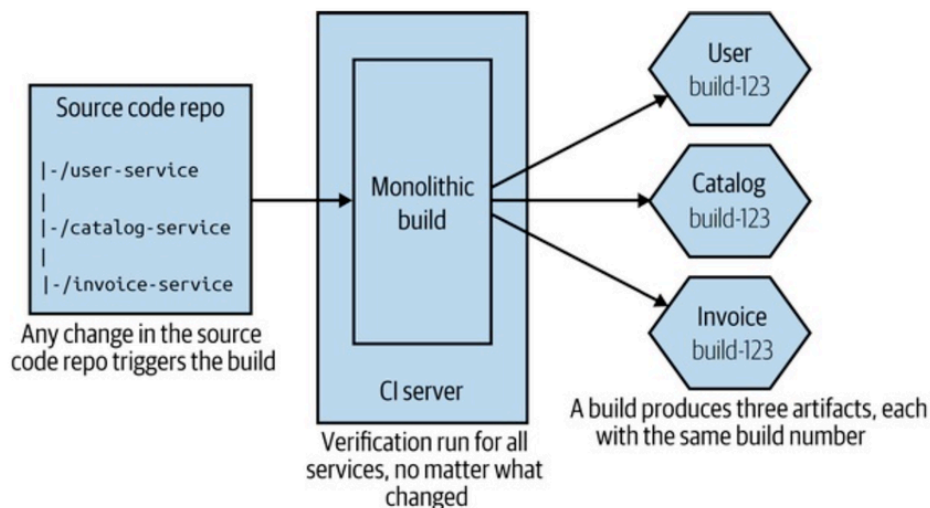   - One Giant Repo, One Giant Build

Figure 7-5. Using a single source code repository and CI build for all microservices

- **Concept:** Store all microservice code in a single repository and use a single build for all services
- **Pros:** Simplifies the repository and build process, straightforward for developers
- **Cons:** Inefficient for large projects, as a small change in one service triggers the build for all services, affecting cycle time and potentially causing unnecessary redeployments. It complicates artifact management and ownership responsibilities
- **Usage:** Rarely used except in early stages of a project
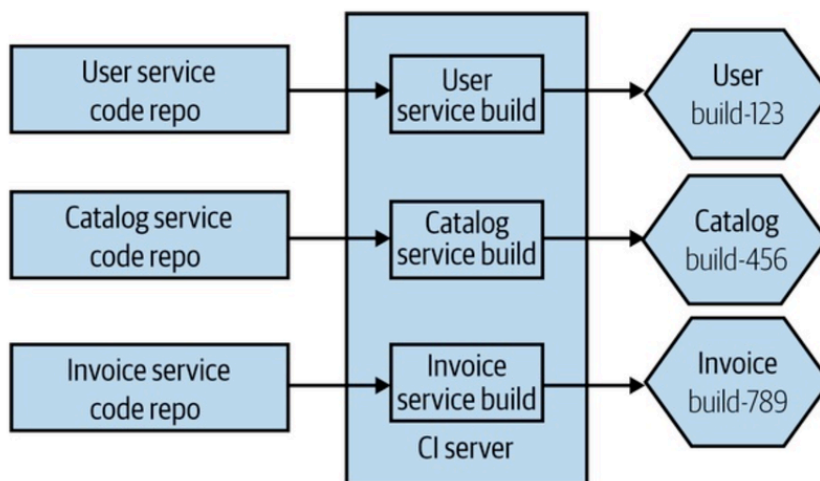
- One Repository per Microservice (Multirepo)

Figure 7-6. The source code for each microservice is stored in a separate source code repository

- **Concept:** Each microservice has its own source code repository
- **Pros:** Direct mapping between source code changes and CI builds, easier ownership management. Changes are isolated to individual services, facilitating independent deployability
- **Cons:** Developers may find it cumbersome to work across multiple repositories, especially when changes span several services. Managing code reuse can be challenging
- **Usage:** Suitable for small to large teams, but can be problematic if frequent cross-service changes are needed, which suggests that service boundaries might need reevaluation
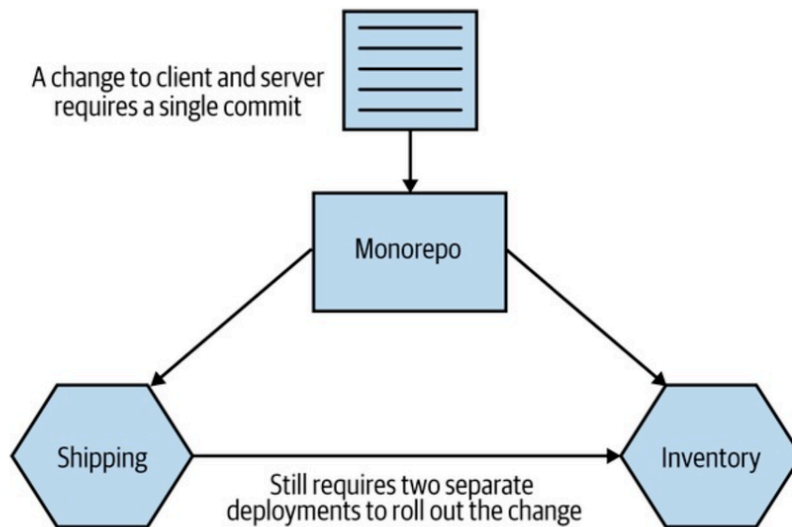
- Monorepo

Figure 7-9. Using a single commit to make changes across two microservices using a monorepo

- **Concept:** All microservice code is stored in a single repository, but with separate build processes for different parts of the codebase
- **Pros:** Facilitates atomic commits across services, better visibility of code, easier code reuse. Allows for finer-grained code reuse and collaboration
- **Cons:** Can become complex at scale, requiring sophisticated tools for dependency management and build optimization. Ownership and changes become harder to manage as the number of developers grows
- **Usage:** Works well for small teams or very large, tech-focused companies that can invest in necessary tooling. Mid-sized organizations may struggle due to the complexity and resource requirements
  - Considerations for Code Reuse and Cross-Service Changes
    - **Libraries for Shared Code:** Code can be packaged into libraries that microservices can depend on. Changes in these libraries require updating and redeploying dependent services
    - **Tools for Managing Monorepos:** Tools like Google's Bazel or Microsoft's VFS for Git help manage dependencies and build processes in large monorepos, but they require significant investment and resources to implement and maintain
- Chapter 8. Deployment
  - **Principles of Microservice Deployment**
    - Zero-Downtime Deployment
      - rolling upgrades: Rolling upgrades allow you to update your software without downtime by upgrading instances one at a time. During the process, only one instance is taken offline and updated while the others continue to serve users
      - blue-green deployments: Blue-green deployments are a technique to update software with minimal downtime and risk
        - **Two Environments:** You have two identical environments, called "Blue" and "Green." Let's say the "Blue" environment is currently live, serving users
        - **Deploy New Version:** You deploy the new version of your software to the "Green" environment, while "Blue" continues to handle all user traffic
        - **Testing:** You thoroughly test the "Green" environment to ensure the new version works correctly and is ready for production
        - **Switch Traffic:** Once you're confident the new version is stable, you switch user traffic from the "Blue" environment to the "Green" environment. Now, "Green" is live, and "Blue" is on standby
        - **Rollback:** If any issues arise, you can quickly switch back to the "Blue" environment, ensuring minimal disruption for users
    - GitOps: GitOps is a way to manage and deploy applications using Git as the single source of truth for your system's desired state
      - **Version Control with Git:** Just like how you use Git to track changes in your code, with GitOps, you also track changes in your infrastructure and application configurations using Git
      - **Define Desired State:** You define how your system should look (the desired state) in a Git repository. This includes configuration files, deployment scripts, and anything else that describes your infrastructure and applications

- **Automate Changes:** Whenever you make changes to the configuration in Git, automated tools (like Flux or Argo CD) detect these changes. These tools then apply the changes to your actual running system to match the desired state
- **Continuous Monitoring:** The GitOps tools continuously monitor the state of your running system. If it detects any deviation from the desired state (as defined in Git), it will automatically correct it, ensuring your system is always in sync with what's described in Git
- **Easy Rollbacks:** If something goes wrong, you can easily revert to a previous state by rolling back the changes in Git. The tools will then automatically apply this rollback to your system

- **Deployment Options**
  - **Function as a Service (FaaS)**
    - Limitations:
      - **Runtime Control:** Limited control over the underlying runtime, which might not support all programming languages
      - **Resource Management:** Limited control over CPU and I/O resources; primarily, only memory can be configured
      - **Execution Time:** Functions have execution time limits (e.g., AWS Lambda: 15 minutes, Google Cloud Functions: 9 minutes).
      - **Statelessness:** Functions are stateless by default, making it difficult to maintain state across invocations
    - Challenges
      - **Spin-Up Time:** Initial "cold start" times can be significant for some runtimes, although optimizations often mitigate this
      - **Dynamic Scaling:** Can overwhelm other parts of the system that don't scale as easily, causing bottlenecks or failures
    - Mapping to Microservices
      - **Single Function per Microservice:** Each microservice instance can be deployed as a single function
      - **Function per Aggregate:** Break down microservices into functions for each aggregate (a collection of related objects), maintaining a coarse-grained external interface and potentially sharing databases within the same microservice
    - Future and Adoption
      - FaaS is expected to become a more common platform for developers, providing significant operational benefits while simplifying deployment
      - Organizations are increasingly adopting FaaS alongside other solutions, leveraging its strengths for specific use cases where it fits well
  - WebAssembly (Wasm):
    - Overview: WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine. It is designed to be a portable compilation target for high-level programming languages like C, C++, and Rust, enabling these languages to run on the web at near-native speed. Wasm is an official standard supported by all major browsers, allowing developers to run complex applications in a web environment with high performance
    - Key Features of WebAssembly
      - **Portability:** Wasm can run on any platform that supports a compatible runtime environment
      - **Performance:** Wasm code executes at near-native speed by leveraging common hardware capabilities.
      - **Security:** Wasm provides a sandboxed execution environment, ensuring that code runs safely within the browser
    - Example: Barcode Scanner with Wasm. Let's walk through an example of how eBay used WebAssembly to bring a barcode scanner application to the web
      - Scenario: eBay had a barcode scanner application written in C++ that was originally available only for Android and iOS native applications. They wanted to make this barcode scanner available on the web to improve user experience and expand accessibility
      - Solution:
        - Compile C++ Code to WebAssembly: eBay took the existing C++ barcode scanner code and compiled it to WebAssembly using tools like Emscripten, which translates C++ code into Wasm bytecode
        - Integrate Wasm into the Web Application: The `.wasm` file is then included in the web application, where it can be instantiated and executed using JavaScript. JavaScript acts as the bridge between the Wasm module and the web environment, handling user interactions and displaying the results
      - Example Code:

```
1  <html>
2  <body>
```

```
3    <h1>eBay Barcode Scanner</h1>
4    <video id="video" width="300" height="200"></video>
5    <script>
6      // Load the WebAssembly module
7      fetch('barcode_scanner.wasm')
8        .then(response => response.arrayBuffer())
9        .then(bytes => WebAssembly.instantiate(bytes, {}))
10       .then(results => {
11         const instance = results.instance;
12         const scanBarcode = instance.exports.scanBarcode;
13
14         // Access the webcam and pass the video stream to the Wasm module
15         navigator.mediaDevices.getUserMedia({ video: true })
16           .then(stream => {
17             document.getElementById('video').srcObject = stream;
18             const video = document.getElementById('video');
19
20             video.addEventListener('play', () => {
21               const barcodeData = scanBarcode(video);
22               console.log('Scanned Barcode:', barcodeData);
23             });
24           })
25           .catch(error => console.error('Error accessing webcam:', error));
26       });
27   </script>
28   </body>
29   </html>
```

- Benefits
    - **Performance:** The barcode scanner runs at near-native speed in the browser, providing a fast and smooth user experience
    - **Portability:** Users can access the barcode scanner directly from their web browser without needing to install a native app
    - **Security:** The Wasm module operates within a secure sandboxed environment, ensuring safe execution of the code
  - WebAssembly allows developers to leverage existing codebases and bring high-performance applications to the web. By compiling native code to Wasm, eBay was able to provide a barcode scanning feature on their website, enhancing user accessibility and maintaining performance and security standards
- When choosing the right deployment option for your microservices
  - **Stick with what works**: If your current deployment method is effective, there's no need to change it just for the sake of change
  - **Offload control**: Use a Platform as a Service (PaaS) like Heroku or a Function as a Service (FaaS) platform if possible. These platforms handle much of the deployment complexity, allowing you to focus more on your application
  - Role of Puppet, Chef, and Similar Tools
    - These tools are still useful, but their role has shifted. They are now more often used for managing legacy applications and infrastructure or for setting up clusters that run container workloads
    - Tools like Terraform and Pulumi are becoming more popular for provisioning cloud infrastructure, moving away from domain-specific languages to using standard programming languages
- **Kubernetes and Container Orchestration**
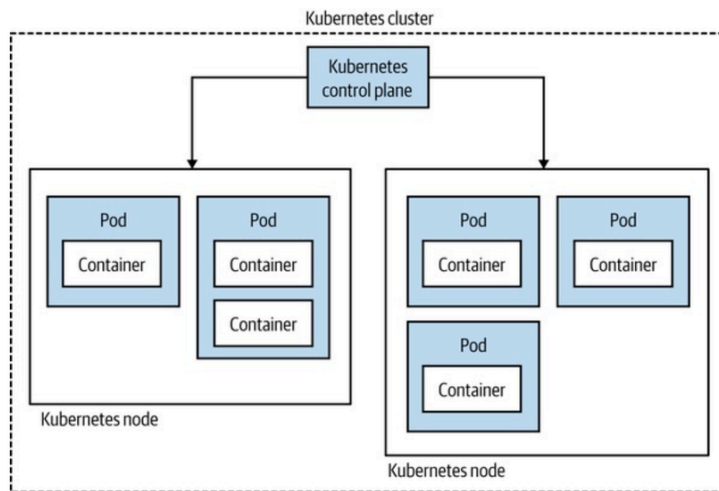 - **A Simplified View of Kubernetes Concepts**

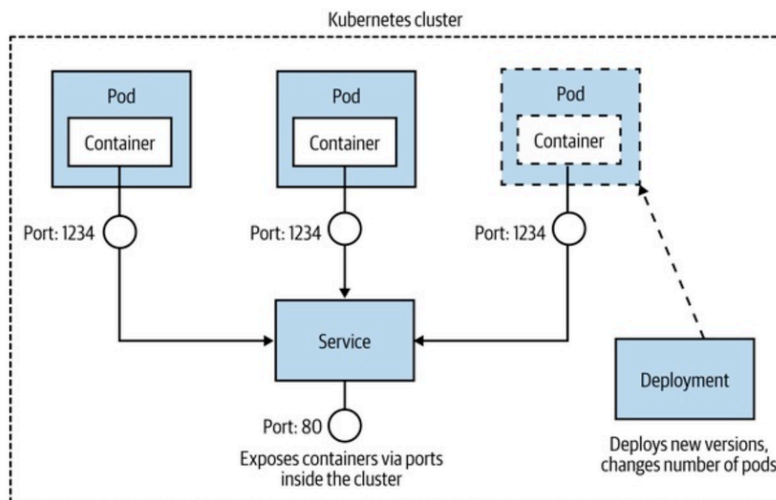*Figure 8-22. A simple overview of Kubernetes topology*

Figure 8-23. How a pod, a service, and a deployment work together

- **Cluster**: Consists of nodes (machines) running workloads and a control plane managing these nodes.
- **Pod**: The basic unit in Kubernetes, usually containing a single container
- **Service**: A stable routing endpoint that maps to pods, handling network traffic within the cluster
- **Replica Set**: Defines the desired state of a set of pods
- **Deployment**: Manages updates to pods and replica sets, enabling rolling upgrades and rollbacks
  - **Progressive Delivery**
    - Feature Toggles
      - Allows functionality to be switched on or off, enabling partial rollouts or turning off problematic features without redeployment
      - **Example:** A new search feature is deployed but hidden behind a feature toggle. Initially, only internal employees can access it. Once confirmed to work well, it's gradually enabled for all users
    - Canary Release
      - New features are initially rolled out to a small subset of users, expanding to more users if no issues are found
      - **Example:** A social media platform releases a new messaging feature to 5% of its users. Monitoring shows it works well, so the feature is gradually made available to everyone
    - Parallel Run

- Running two versions of the same functionality side by side to compare results and ensure new versions work as expected without impacting users
  - **Example:** An e-commerce site introduces a new payment processing system. Both the old and new systems run simultaneously, processing identical transactions. Results are compared to ensure the new system's accuracy before fully switching over
- Chapter 9. Testing
- Chapter 10. From Monitoring to Observability
- Chapter 11. Security
- Chapter 12. Resiliency
- Chapter 13. Scaling