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

-
  -
    - 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**

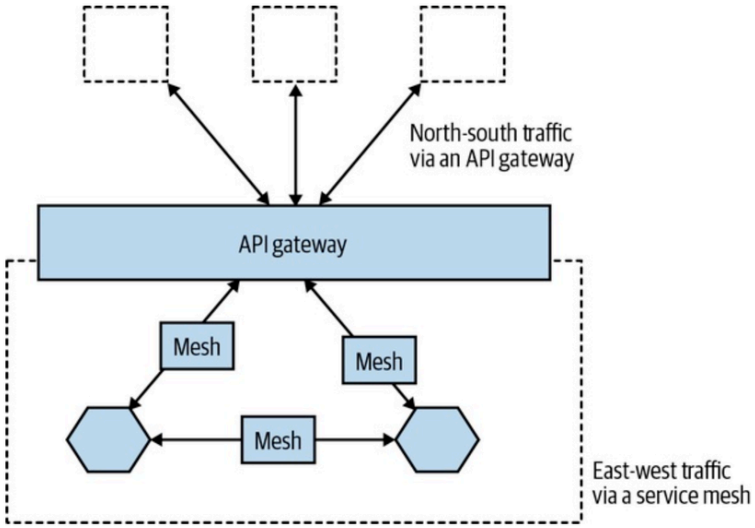

Figure 5-6. An overview of where API gateways and service meshes are used

- 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

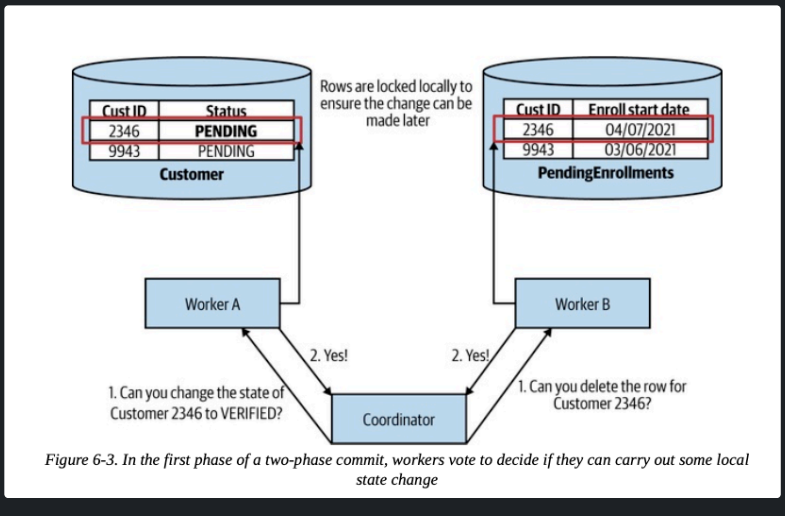

Figure 6-3. In the first phase of a two-phase commit, workers vote to decide if they can carry out some local state change

        - 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

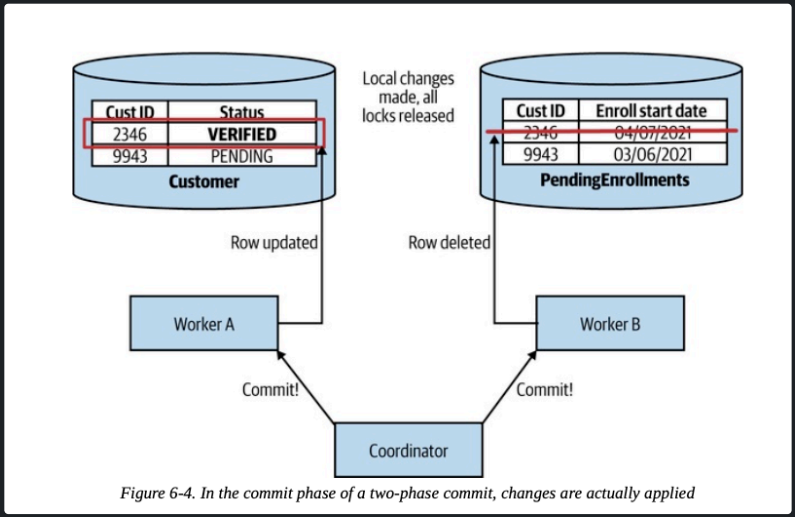

Figure 6-4. In the commit phase of a two-phase commit, changes are actually applied

        - 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

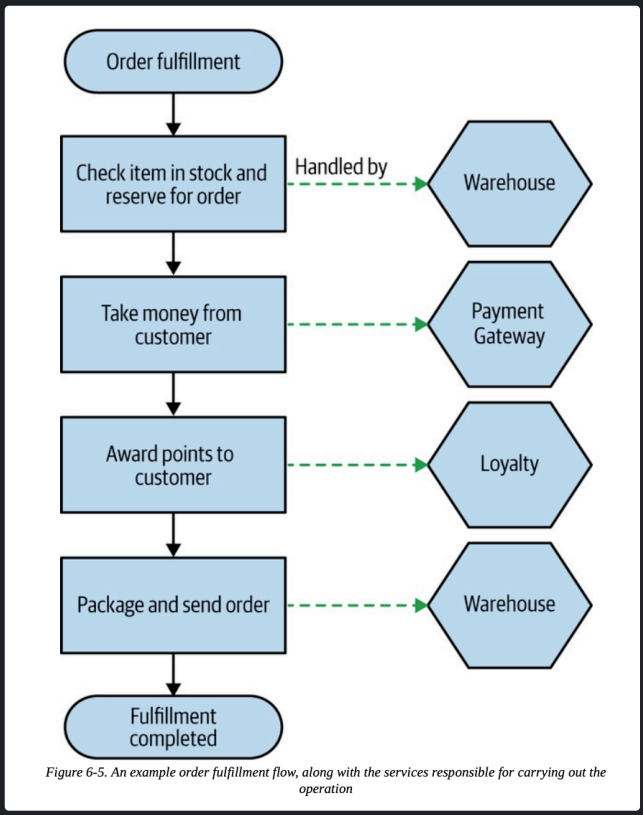

Figure 6-5. An example order fulfillment flow, along with the services responsible for carrying out the operation

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

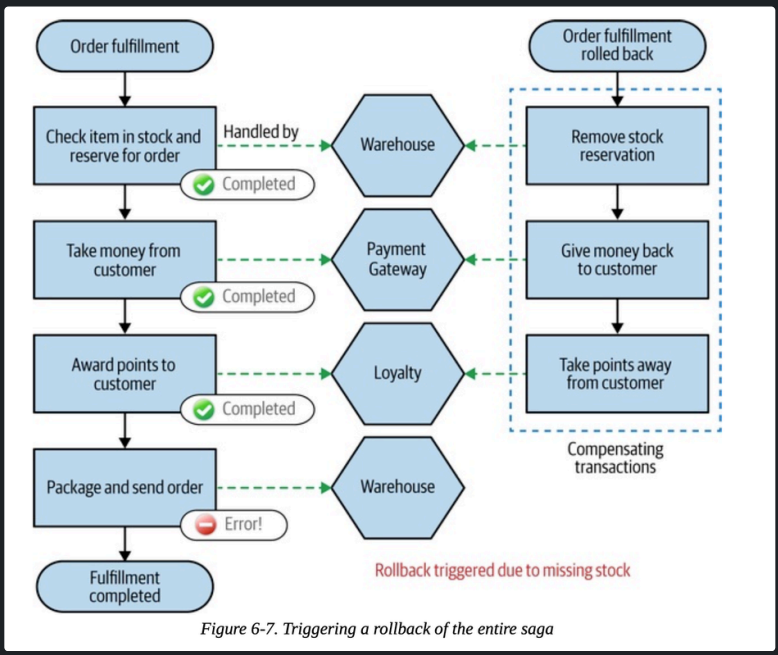

Figure 6-7. Triggering a rollback of the entire saga

    - 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

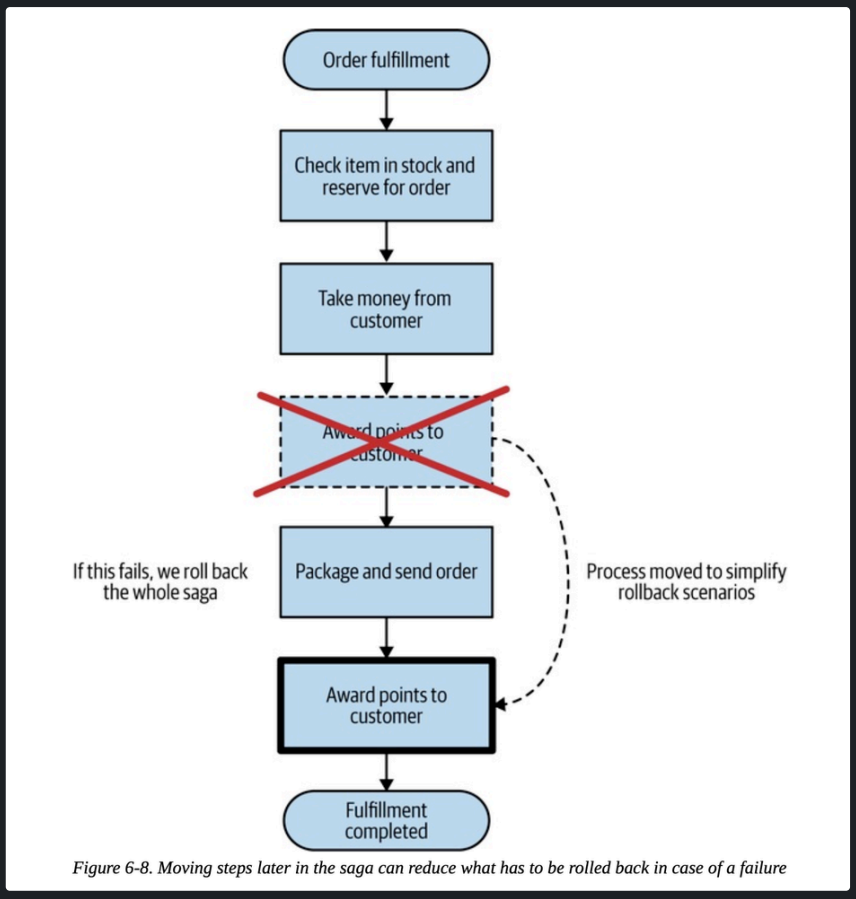

Figure 6-8. Moving steps later in the saga can reduce what has to be rolled back in case of a failure

- Implementation Styles
  - Orchestrated Sagas

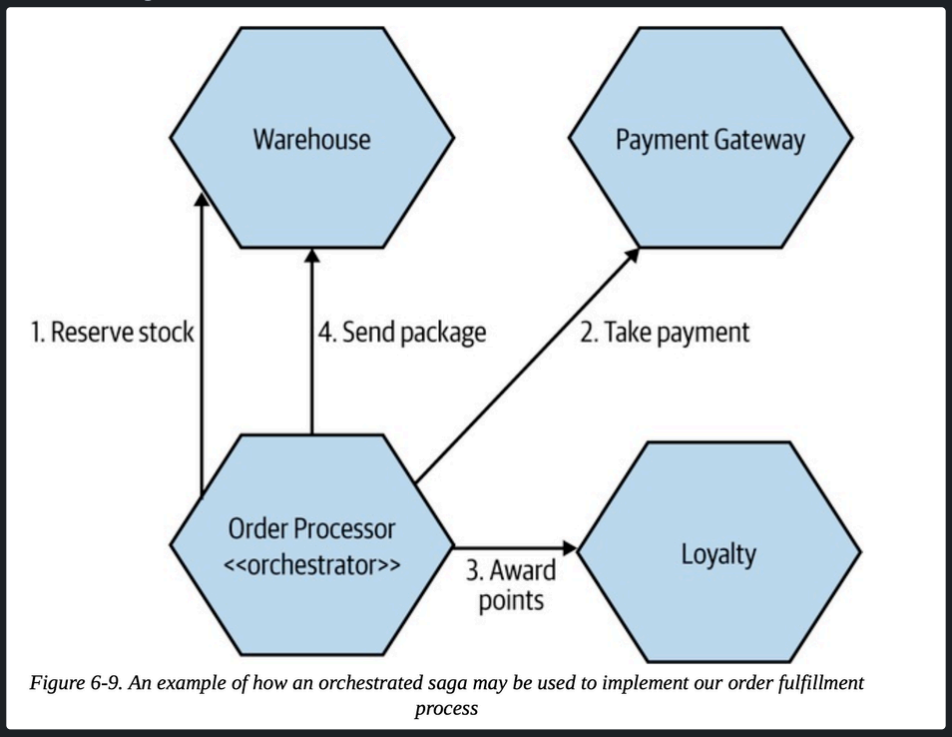

Figure 6-9. An example of how an orchestrated saga may be used to implement our order fulfillment process

        - 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

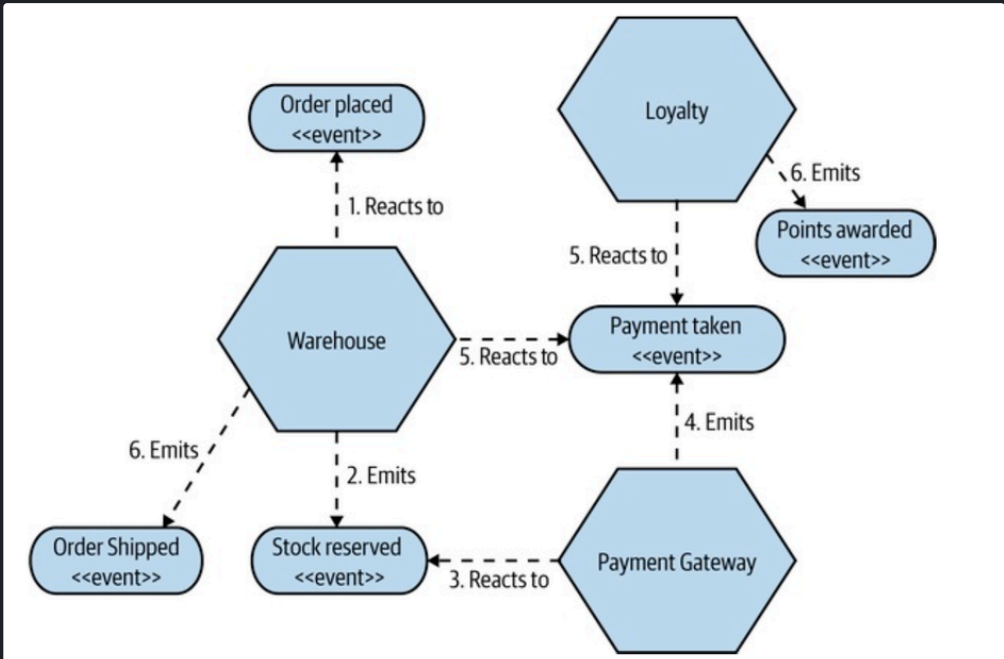

Figure 6-10. An example of a choreographed saga for implementing order fulfillment

        - 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

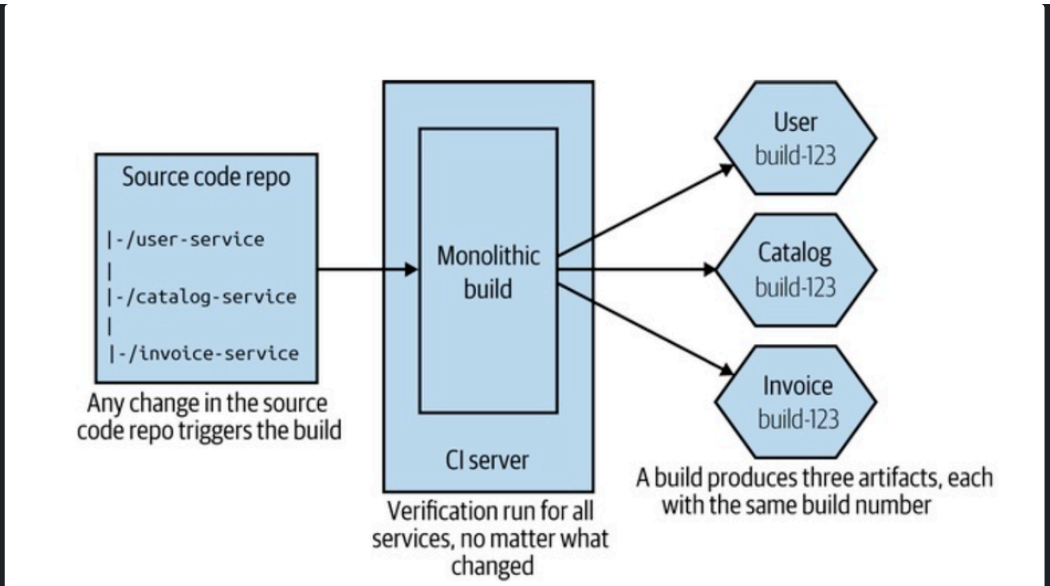

Figure 7-5. Using a single source code repository and CI build for all microservices

- **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)

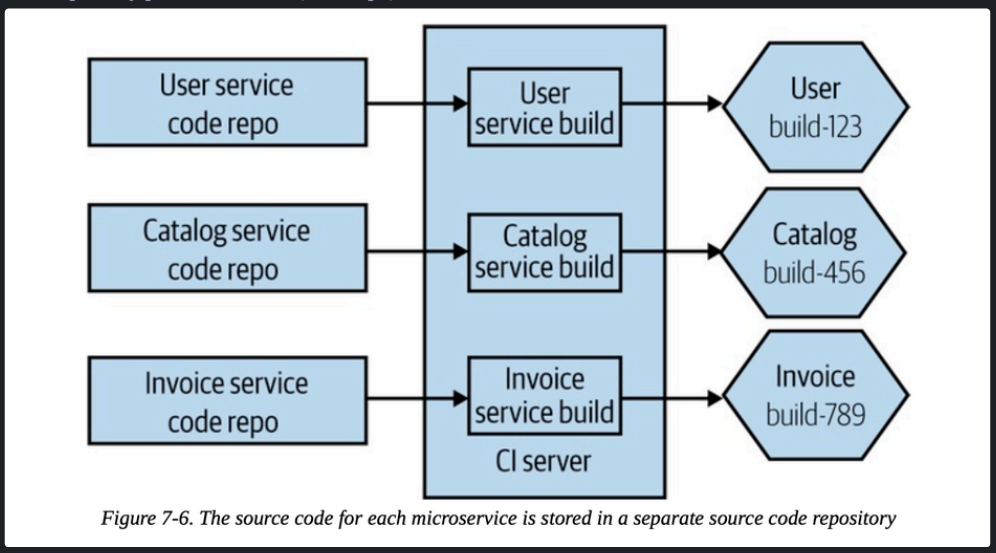

Figure 7-6. The source code for each microservice is stored in a separate source code repository

- **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

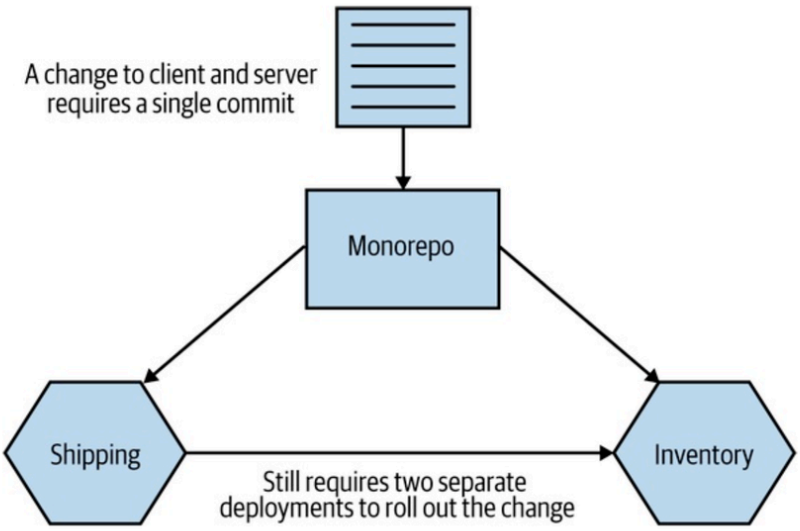

Figure 7-9. Using a single commit to make changes across two microservices using a monorepo

        - **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:

```
<html>
<body>
```

```
  <h1>eBay Barcode Scanner</h1>
  <video id="video" width="300" height="200"></video>
  <script>
    // Load the WebAssembly module
    fetch('barcode_scanner.wasm')
      .then(response => response.arrayBuffer())
      .then(bytes => WebAssembly.instantiate(bytes, {}))
      .then(results => {
        const instance = results.instance;
        const scanBarcode = instance.exports.scanBarcode;

        // Access the webcam and pass the video stream to the Wasm module
        navigator.mediaDevices.getUserMedia({ video: true })
          .then(stream => {
            document.getElementById('video').srcObject = stream;
            const video = document.getElementById('video');

            video.addEventListener('play', () => {
              const barcodeData = scanBarcode(video);
              console.log('Scanned Barcode:', barcodeData);
            });
          })
          .catch(error => console.error('Error accessing webcam:', error));
      });
  </script>
</body>
</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**

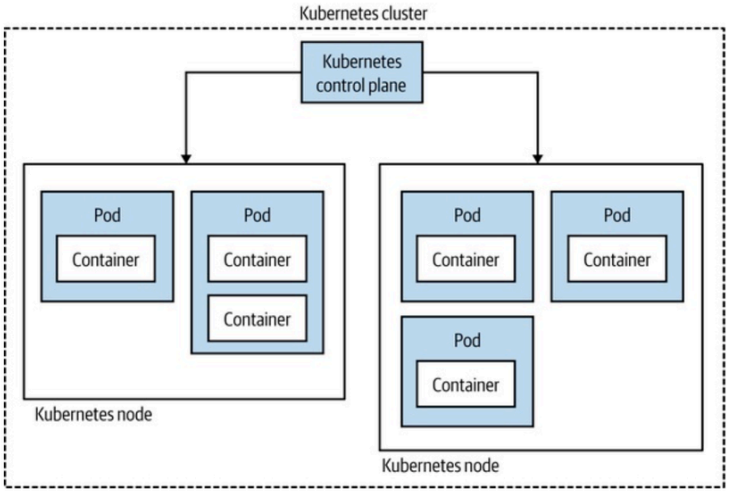

Figure 8-22. A simple overview of Kubernetes topology

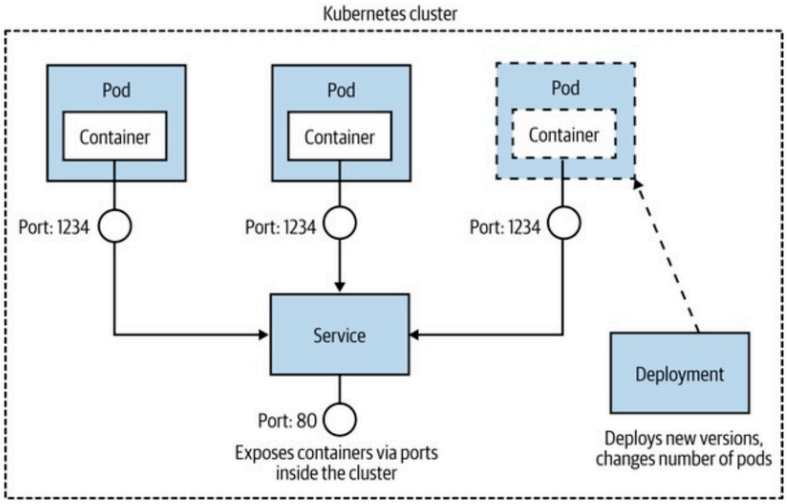

Figure 8-23. How a pod, a service, and a deployment work together

- **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
    - Key Stability Patterns
        - Time-Outs
            - Importance of Time-Outs:
                - setting appropriate time-outs for calls to downstream services is crucial to prevent system slowdowns or hangs
                - **AdvertCorp Case Study:** Two main issues were identified

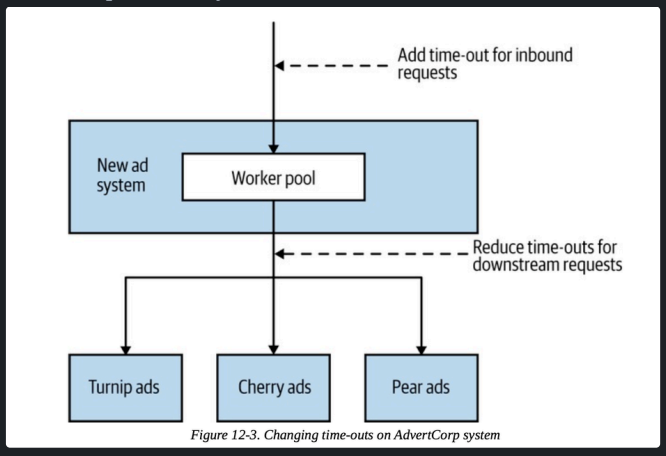

Figure 12-3. Changing time-outs on AdvertCorp system

                    - Missing time-out on the HTTP request pool, causing indefinite blocking
                    - Excessive waiting time for HTTP requests to the turnip ad system, causing unnecessary delays.
            - Effective use of time-outs helps maintain system performance and prevent cascading failures by ensuring that operations do not wait excessively for unresponsive services
        - **Retries**
            - When to Retry
                - **HTTP Response Codes:** Use response codes to determine if a retry is warranted
                    - **Do not retry:** 404 Not Found
                    - **Retry:** 503 Service Unavailable or 504 Gateway Time-out
            - Implementing Retries
                - **Delay Between Retries:** Add delays to avoid overwhelming an already overloaded service
                - Consider Time-Out Thresholds
                    - Ensure retries are within the overall time budget for the operation
                    - Example: If the initial call has a 500 ms time-out and you allow three retries with 1-second intervals, the total wait time could be 3.5 seconds
                    - For user-facing operations, ensure retries do not exceed the time users are willing to wait
                    - For non-user-facing operations, longer wait times might be acceptable
        - **Bulkheads**
            - concept: Inspired by ship design, where bulkheads isolate compartments to contain damage and protect the rest of the ship
            - Purpose: **Isolation:** Prevents failure in one part of the system from affecting other parts
            - Example - AdvertCorp

        - **Problem:** Single connection pool for all downstream calls caused system-wide failures when one service was slow
        - **Solution:** Use separate connection pools for each downstream service to isolate issues
    - Implementation
        - **Connection Pools:** Separate pools for each downstream service
        - **Separation of Concerns:** Break functionality into distinct microservices
        - **Load Shedding:** Reject requests to prevent resource saturation and ensure critical system functions remain operational
    - **Key Pattern for Resiliency:** Bulkheads are essential for maintaining stability by isolating failures and ensuring the system remains functional even when parts of it fail. They work alongside other patterns like time-outs and circuit breakers to effectively manage and mitigate failures.
- Circuit Breakers
    - Concept

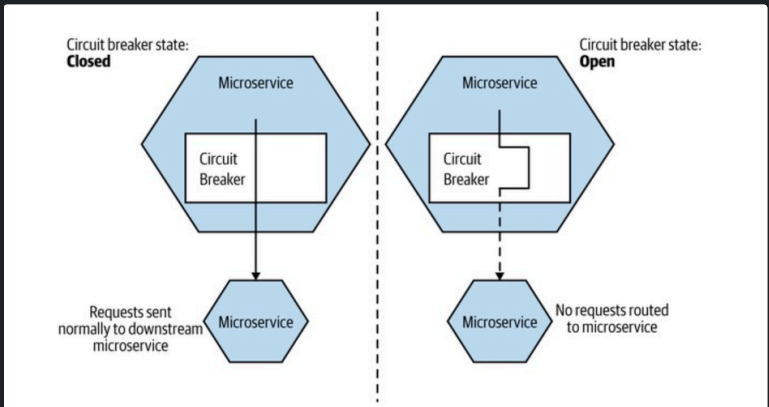

Figure 12-5. An overview of circuit breakers

        - Inspired by electrical circuit breakers, which protect devices from power spikes
        - Software circuit breakers automatically stop requests to failing services to prevent system-wide issues
    - Function
        - **Protection Mechanism:** Acts like a bulkhead to prevent cascading failures
        - **Fail Fast:** Stops traffic to a failing service after a certain number of failures, quickly returning an error instead of waiting
    - Implementation
        - **Trigger:** Activated by errors or timeouts in downstream calls
        - **Blown State:** When activated, further requests fail immediately until the service is healthy again.
        - **Reset:** Periodically sends test requests to check service health and reset if healthy
    - Example - AdvertCorp

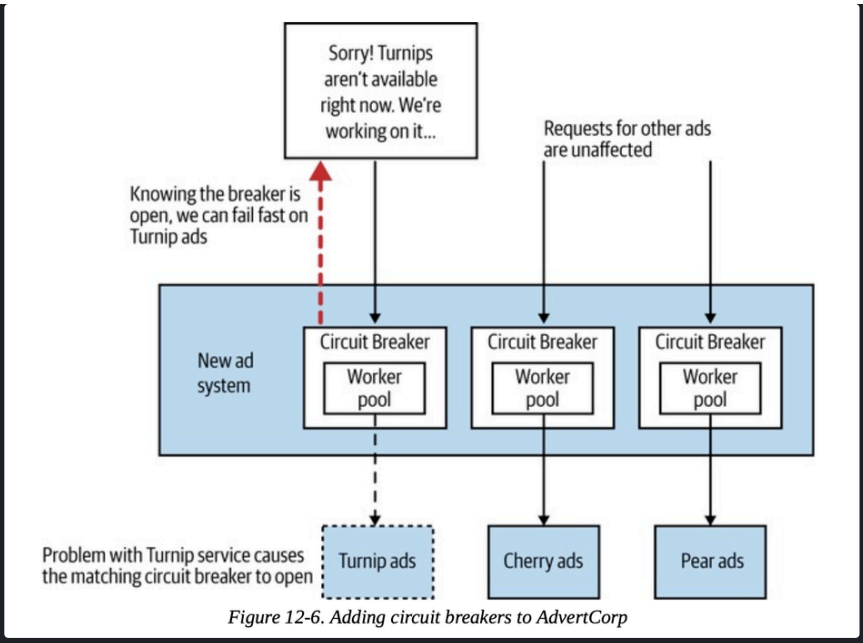

Figure 12-6. Adding circuit breakers to AdvertCorp

    - **Problem:** Slow responses from the turnip system causing system-wide slowdowns
    - **Solution:** Implement circuit breakers for each downstream service
    - **Outcome:** When a service fails, updates the website to notify users of the issue with specific parts while keeping the rest operational
- Isolation
- Redundancy
    - Redundancy involves having multiple instances or backups of critical components to improve system robustness
    - Implementation
        - **Multiple Instances:** Running more than one instance of a microservice to tolerate failures
        - Availability Zones: Deploy instances across different availability zones to mitigate the risk of a single zone failure (e.g., AWS EC2 instances).
- Middleware
    - Role
        - Middleware, particularly message brokers, facilitates both request-response and event-based interactions between services
        - Provides **guaranteed delivery** by implementing retries and time-outs
    - Case Study - AdvertCorp
        - Using middleware for request-response communication with the turnip system had limited benefits
            - Resource contention was reduced, but pending requests increased
            - Requests might become invalid by the time they were processed
        - Alternative approach: Use middleware to broadcast updates (e.g., turnip ads), which could be consumed by other services
- **Idempotency**
    - Idempotent operations produce the same result regardless of how many times they are applied. This ensures that repeated calls have no adverse effects
    - Useful for replaying messages to recover from errors without duplicating results
    - Example
        - A call to credit points to a customer's account
            - **Non-idempotent call:** `<credit><amount>100</amount><forAccount>1234</account></credit>` would add 100 points each time it's received
            - **Idempotent call:** Adding a reason makes it idempotent: `<credit><amount>100</amount><forAccount>1234</account> <reason><forPurchase>4567</forPurchase></reason></credit>`. This ensures points are added only once per purchase

        - Application: Works well with event-based systems, ensuring multiple instances processing the same event don't cause issues
        - Idempotency ensures reliable operations in distributed systems by preventing repeated execution from causing errors or unintended effects. It's crucial for handling retries and ensuring system robustness
- Spreading Your Risk
    - Ensuring resilience by not concentrating your services in a single location or on a single platform
    - Examples
        - Avoid hosting multiple services on one physical host or virtual hosts running on the same physical machine to prevent a single point of failure
        - Virtualization platforms can distribute hosts across different physical boxes to mitigate risk
    - Considerations
        - SANs (Storage Area Networks) can be a risk as multiple VMs mapped to one SAN can all go down if the SAN fails
        - Distributing services across different racks in a data center or across multiple data centers can reduce risk
        - For cloud services like AWS, distribute workloads across multiple availability zones within a region to achieve higher resilience
    - AWS Specifics
        - AWS regions are split into availability zones (data centers).
        - AWS offers a 99.95% uptime guarantee for a region over a month, not for individual nodes or zones, making it essential to distribute services across zones
    - Alternative Plans
        - Have backup plans (Plan B or C) to cover scenarios where a primary supplier fails to meet obligations
        - Consider disaster recovery hosting with a different supplier to reduce vulnerability
- CAP Theorem
    - Definition: The CAP theorem states that in a distributed system, you can only achieve two out of three characteristics: Consistency, Availability, and Partition Tolerance
    - example:

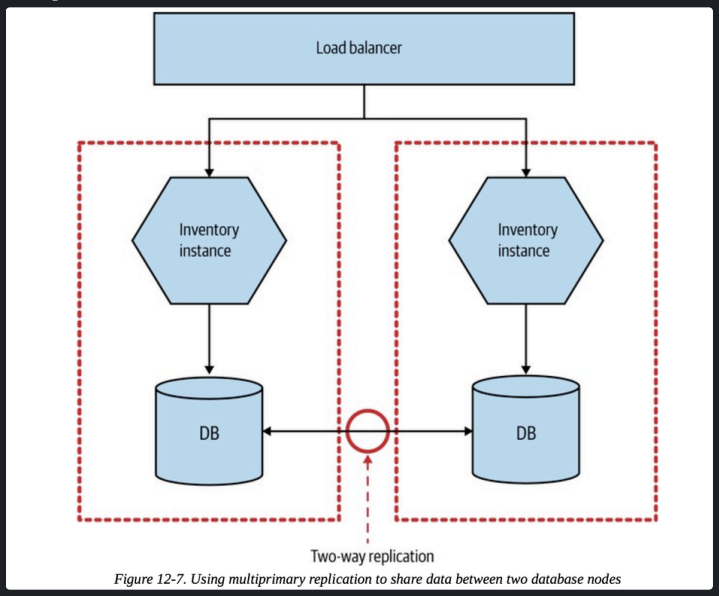

Figure 12-7. Using multiprimary replication to share data between two database nodes

        - Inventory service is deployed across two separate data centers. Each data center contains an instance of the inventory service and a database. The databases are synchronized using a two-way replication mechanism. A load balancer distributes incoming requests between the two inventory instances
        - In the context of the CAP theorem, this setup can be analyzed as follows
            - **Consistency (C):** The goal of two-way replication is to ensure that both database nodes have the same data, providing consistency. However, during network partition or communication failure between the two data centers, the databases cannot synchronize, potentially leading to inconsistent data between the two nodes

        - **Availability (A):** Both inventory instances continue to serve requests even if they cannot synchronize their databases due to a partition. This ensures the system remains available, as each data center can independently handle requests
        - **Partition Tolerance (P):** The system can handle the network partition by continuing to operate, but it must choose between maintaining consistency or availability. In the event of a partition, if both nodes keep accepting writes, the system sacrifices consistency (AP system). If one or both nodes stop accepting writes to maintain consistency, the system sacrifices availability (CP system).
    - Scenario Analysis
        - **Partition Tolerance with Consistency (CP):** If the network link between the two data centers fails, to maintain consistency, the system could stop accepting writes or reads in one or both data centers until synchronization is restored. This ensures that any data read is consistent, but it reduces availability because not all requests can be served during the partition
        - **Partition Tolerance with Availability (AP):** If the network link fails, both data centers continue to accept and process requests independently. This ensures the system remains available, but data may become inconsistent between the two data centers during the partition. Once the network link is restored, the system must reconcile any differences between the databases, which can be complex and may lead to temporary inconsistencies
  - Key Concepts
    - **Consistency (C):** Every read receives the most recent write
    - **Availability (A):** Every request receives a response (success/failure).
    - **Partition Tolerance (P):** The system continues to operate despite network partitions
  - Trade-Offs
    - **AP Systems (Availability & Partition Tolerance):** Systems remain available and partition-tolerant but sacrifice consistency. Example: During a network failure, different nodes might return stale data
    - **CP Systems (Consistency & Partition Tolerance):** Systems remain consistent and partition-tolerant but sacrifice availability. Example: If nodes cannot communicate, they refuse requests to maintain consistency
    - **CA Systems (Consistency & Availability):** These systems cannot exist in distributed environments as they lack partition tolerance
  - Practical Implications
    - **Eventual Consistency:** AP systems are often eventually consistent, meaning all updates will propagate eventually, but temporary inconsistencies are possible
    - **Contextual Decisions:** The choice between AP and CP depends on the specific needs of the application. For example, stale data might be acceptable for inventory systems but not for financial transactions
  - System Design
    - Systems do not need to be entirely AP or CP. Different parts can follow different models based on requirements
    - Individual service capabilities can also follow different models, such as an idempotent read that is eventually consistent and a strict consistent write
  - Real-World Considerations: In real-world applications, even highly consistent systems cannot account for every physical event, like a damaged inventory item, reinforcing the practical appeal of eventual consistency. The trade-off decision (AP vs. CP) depends on the specific requirements and tolerances of the application in question. For instance, a banking system might prioritize consistency (CP), whereas a social media platform might prioritize availability (AP).
- Chapter 13. Scaling
  - Purpose of Scaling
    - **Performance Improvement:** Enhance system capability to handle more load and improve latency.
    - **Robustness Improvement:** Strengthen the system's reliability and fault tolerance
  - **The Four Axes of Scaling**
    - Vertical Scaling: In a nutshell, this means getting a bigger machine.
    - Horizontal Duplication
      - Concept: Horizontal duplication involves duplicating parts of a system to handle more workloads by distributing the work across these duplicates. It's a straightforward scaling technique often used early on to address load issues
      - Implementations

- Load Balancer: Distributes requests across multiple instances of a service. The load balancer detects when a node is unavailable and removes it from the pool, ensuring transparency to the consumer.

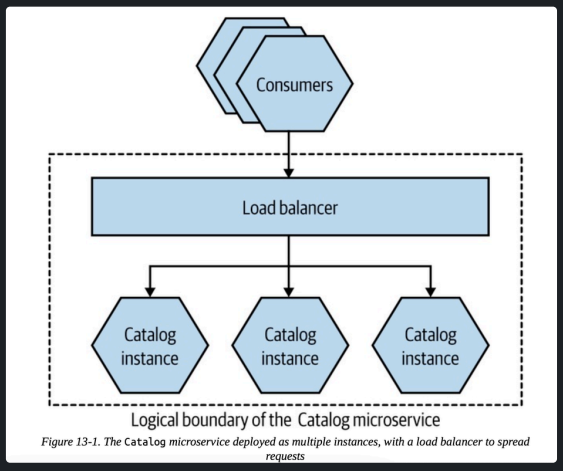

Figure 13-1. The `Catalog` microservice deployed as multiple instances, with a load balancer to spread requests

- **Competing Consumer Pattern:** Multiple instances consume from a common queue, increasing system throughput

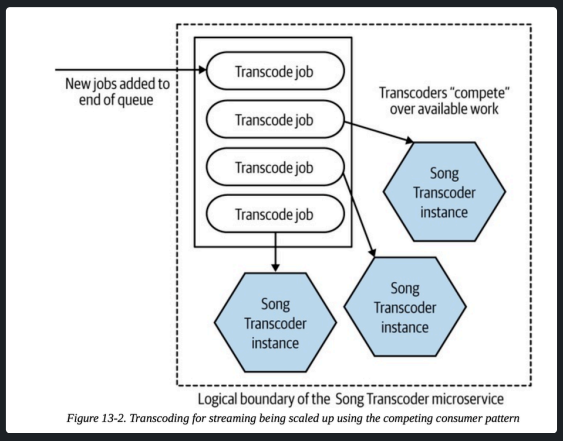

Figure 13-2. Transcoding for streaming being scaled up using the competing consumer pattern

- Read Replicas: Used to reduce read load on the primary database by redirecting read requests to replicas, often managed by a load balancer

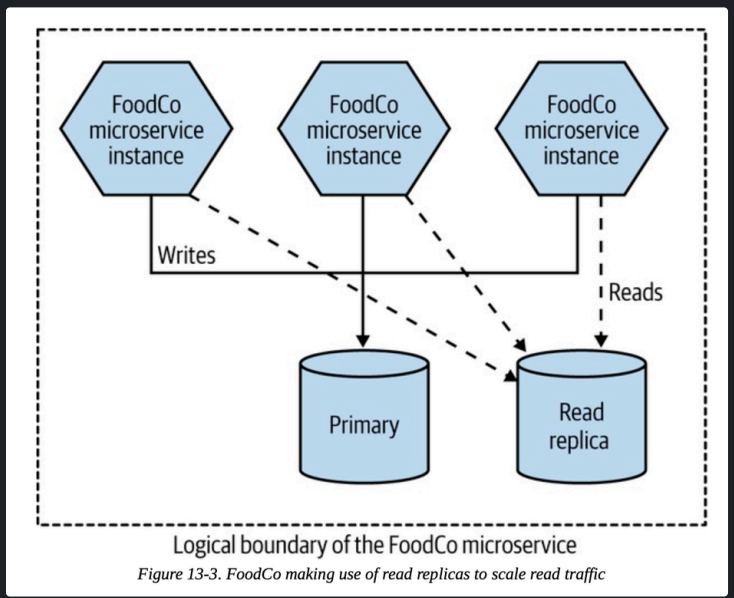

Figure 13-3. FoodCo making use of read replicas to scale read traffic

- Key Benefits
  - **Simplicity:** Often doesn't require application changes, as load distribution can be managed by external tools (e.g., load balancers, message brokers).

        - **Efficiency:** Spreads load evenly, reducing contention for computing resources
        - **Quick Implementation:** Easier to implement compared to more complex scaling techniques
    - Limitations
        - **Infrastructure Cost:** Requires more infrastructure, increasing costs
        - **Blunt Instrument:** Might involve running multiple full copies of an application even if only a part needs scaling
        - **Complex Load Distribution:** Requires reliable load distribution mechanisms. Systems might have special requirements, such as sticky sessions, which can complicate load balancing and are generally best avoided if possible
- **Data Partitioning**
    - Concept: Data partitioning distributes load based on a specific aspect of data, such as user information, to handle scaling more effectively. It involves splitting data into different partitions (or shards) to distribute workloads
    - Implementation
        - Database Level Partitioning: Requests are directed to specific database nodes based on a partition key, like the customer's family name

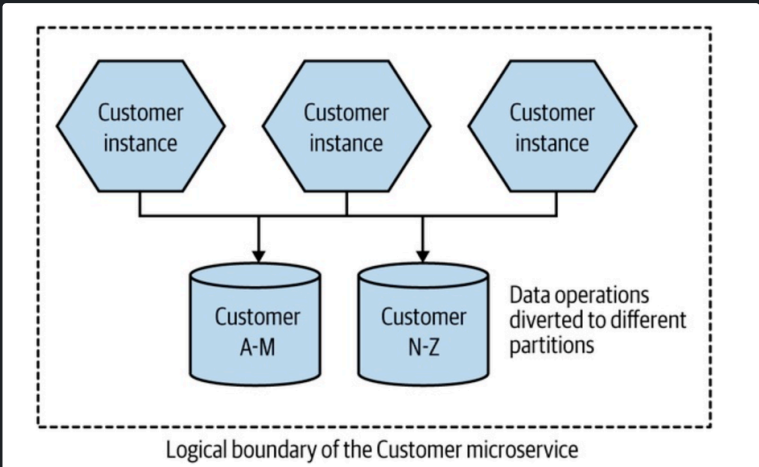

Figure 13-4. Customer data is partitioned across two different databases

        - Microservice Instance Level Partitioning: Requests are routed to the appropriate microservice instance using a proxy based on the partition key. Example: Partitioning requests based on customer information to specific microservice instances.

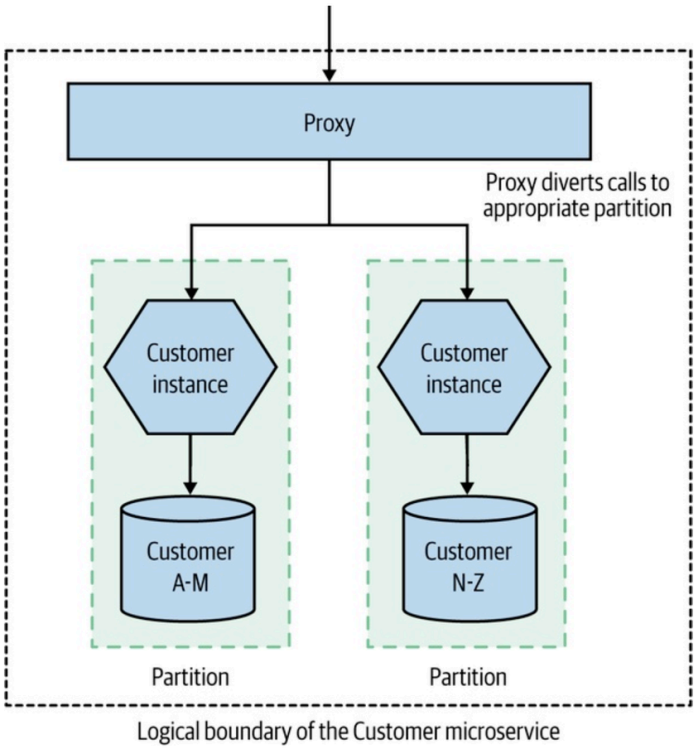

Figure 13-5. Requests are directed to the appropriate microservice instance

        - Geographic Partitioning: Data is partitioned based on geographic regions. Example: Assigning customers in different countries to different databases
    - Key Benefits
        - **Scales Transactional Workloads:** Effective for systems constrained by write operations
        - **Reduced Maintenance Impact:** Maintenance can be done on a per-partition basis, minimizing overall downtime
        - **Geographical Compliance:** Ensures data remains within specific jurisdictions if required
    - Limitations
        - **Robustness Issues:** If a partition fails, requests to that partition will fail
        - **Partition Key Complexity:** Choosing the right partition key is challenging. For example, partitioning by family name may lead to uneven distribution
        - **Querying Across Partitions:** Queries spanning multiple partitions require additional handling, like in-memory joins or alternative read stores
        - **Scalability:** Adding new partitions can be straightforward, but changing an existing partitioning scheme can be complex and time-consuming
    - Examples: **Cassandra and Kafka:** These technologies natively support partitioning to distribute reads and writes across nodes
    - Considerations
        - Ensure even load distribution by selecting appropriate partition keys
        - Combine with horizontal duplication to improve robustness
        - Evaluate the need for different types of databases if current solutions hit scalability limits
- Functional Decomposition
    - Concept: Functional decomposition involves extracting functionality from an existing system and allowing it to be scaled independently, typically by creating new microservices

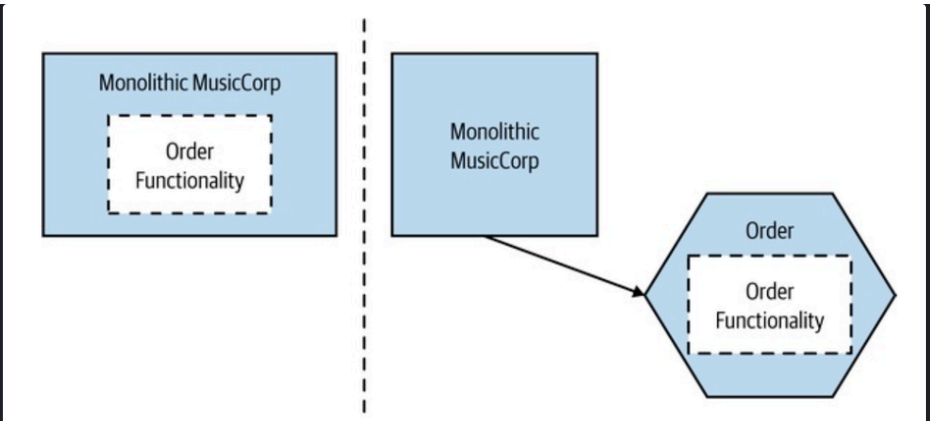

Figure 13-6. The `Order` microservice is extracted from the existing MusicCorp system

- Example
  - **MusicCorp:** Extracted the order functionality from the main system to scale it separately
  - **FoodCo:** Exhausted other scaling options and began moving key data and workloads from the core system to dedicated microservices, such as those for deliveries and menus
- Key Benefits
  - **Rightsizing Infrastructure:** Different workloads can be scaled independently, optimizing cost and performance. For example, lightly used functionality can be deployed on smaller machines, while more constrained functionality can use larger machines or multiple instances
  - **Cost Optimization:** Large SaaS providers use this to balance infrastructure costs and improve profitability
  - **Partial Failure Tolerance:** Opens the opportunity to build systems that can tolerate partial failures.
  - **Technology Flexibility:** Allows using different technologies or databases better suited to specific tasks, improving efficiency
  - **Organizational Scaling:** Facilitates scaling the organization by allowing teams to own different microservices

- Combining Models
  - Overview: Combining scaling models is essential for effectively managing the growth and performance of applications. The original Scale Cube concept encourages thinking beyond a single type of scaling and understanding the benefits of applying multiple scaling axes based on specific needs
  - Example
    - **Order Microservice:** Initially extracted from the main system to run independently (functional decomposition).

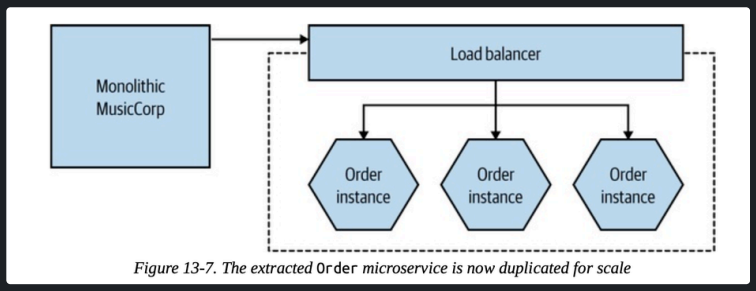

Figure 13-7. The extracted `Order` microservice is now duplicated for scale

    - **Duplication:** Multiple copies of the Order microservice are created to handle increased load
    - **Geographical Partitioning:** The Order microservice is partitioned by geographical regions, with horizontal duplication within each region

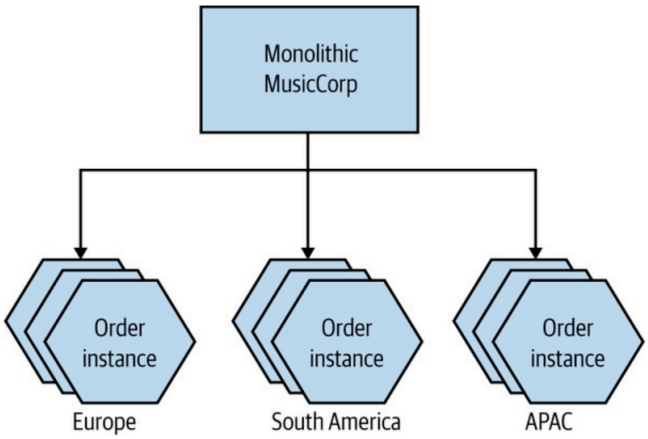

Figure 13-8. MusicCorp's `Order` microservice is now partitioned across geography, with duplication in each group

- Steps
  - **Functional Decomposition:** Break down the monolith into smaller microservices, such as extracting the Order functionality
  - **Horizontal Duplication:** Scale the Order microservice by creating multiple instances
  - **Data Partitioning:** Partition the Order microservice by geographical locales, applying horizontal duplication within each locale
- Start small
  - Identify Real Needs: Only optimize when there's a proven necessity. Adding complexity to handle non-existent problems can waste resources and make the system fragile
  - **Experimentation:** Follow a scientific approach to validate bottlenecks. Use automated load tests to establish baselines, make changes, and observe results
  - CQRS and Event Sourcing
    - **CQRS (Command Query Responsibility Segregation):** Separates read and write responsibilities into different models, allowing independent scaling. It's a powerful pattern but difficult to implement well. Consider simpler solutions like read replicas first if you're read-constrained
    - **Event Sourcing:** Stores the history of events to project the current state, rather than storing the state as a single record. Fits well in some situations but adds cognitive load and complexity for developers.
- Caching: Caching is a performance optimization technique that stores the result of an operation to reuse in subsequent requests, saving time and resources. It is useful for improving latency, scaling applications, and enhancing robustness
  - **For Performance:**
    - Purpose: Caching helps mitigate network latency and reduces the cost of interacting with multiple microservices for data retrieval
    - **Example:** For fetching a list of the most popular items by genre, caching the results of an expensive database join query ensures data regeneration only when the cache is invalidated. This optimizes performance and reduces the computational load
  - **For Scale:**
    - **Purpose:** Diverting read operations to caches helps avoid system contention and improves scalability.
    - **Example:** Using database read replicas to handle read traffic reduces the load on the primary database. Although replicas might serve slightly stale data, they are eventually updated through replication
  - **For Robustness**
    - **Purpose:** Using local caches allows operation even if the origin is unavailable, enhancing system robustness
    - **Cache Invalidation:** Configure cache invalidation to prevent automatic eviction of stale data, ensuring data remains in the cache until it can be updated. This avoids cache misses and failure to retrieve data when the origin is offline
    - **Example:** The Guardian periodically crawls their live site to generate a static version that can be served during outages, ensuring a version of the site remains accessible even if the live content isn't fresh
  - **Where to Cache**
    - Client-side Caching

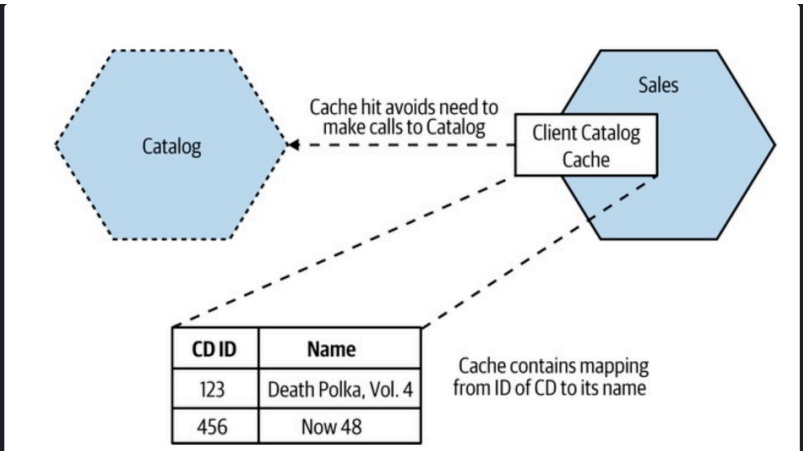

Figure 13-10. `Sales` *holds a local copy of* `Catalog` *data*

- **Example:** Sales microservice maintains a local cache of Catalog data
- **Benefits:** Reduces network calls, improves latency, and enhances robustness
- **Downsides:** Limited invalidation options, potential inconsistency between clients, and risk of stale data

- Shared Client-side Cache

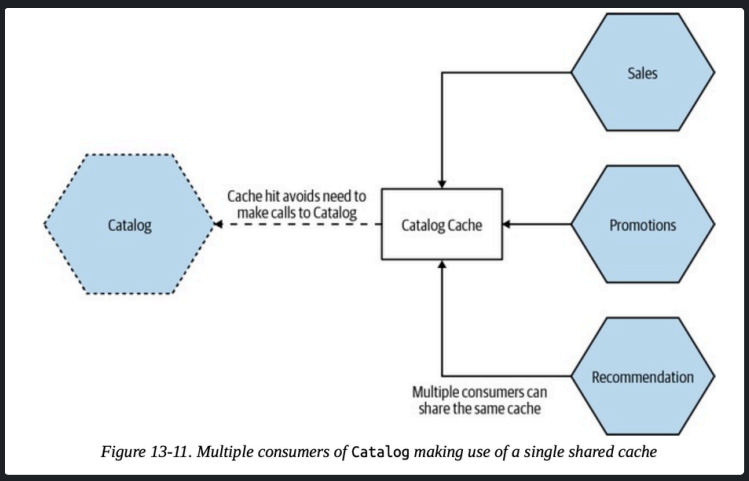

Figure 13-11. Multiple consumers of `Catalog` *making use of a single shared cache*

- **Example:** Using Redis or memcached to share cache across multiple consumers
- **Benefits:** Avoids inconsistency between clients, more efficient resource use
- **Considerations:** Clients need to make a round trip to the shared cache, blurring lines between client-side and server-side caching

- Server-side Caching

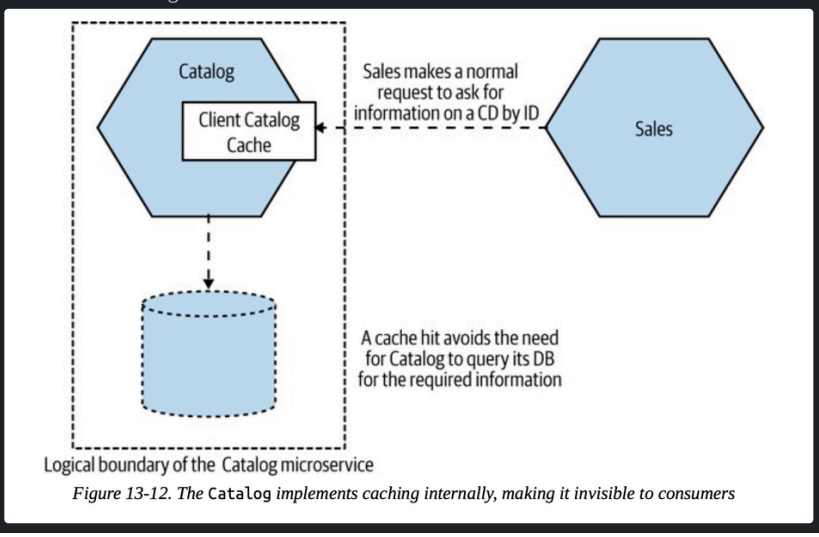

Figure 13-12. The `Catalog` *implements caching internally, making it invisible to consumers*

- **Example:** Catalog microservice maintains its own cache
- **Benefits:** Easier to implement sophisticated invalidation mechanisms, avoids inconsistencies seen in client-side caching
- **Downsides:** Less effective for optimizing latency and robustness compared to client-side caching.

- Request Cache

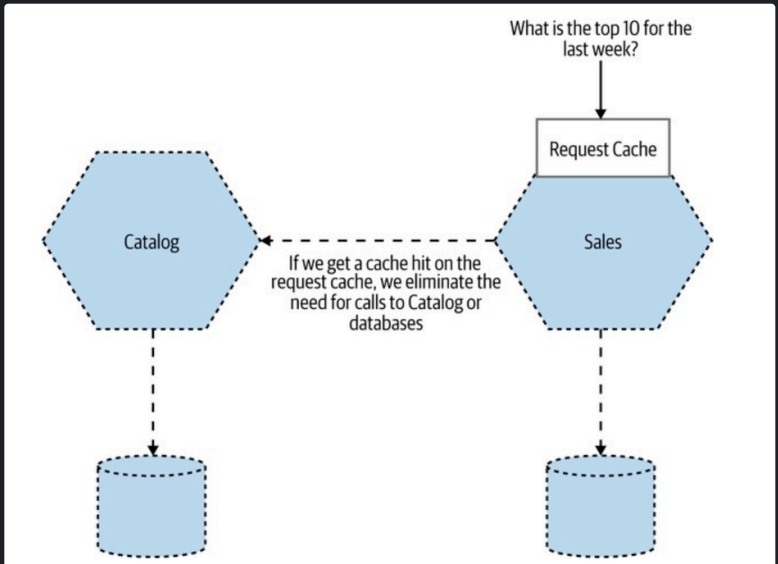

Figure 13-13. Caching the results of the top 10 request

- **Example:** Caching the result of specific queries like the top ten best sellers
- **Benefits:** Highly efficient for specific queries
- **Downsides:** Limited scope, benefits only specific requests

- **Invalidation:** Invalidation is the process of evicting outdated data from a cache, a concept that is simple in theory but complex in execution. Here are several mechanisms for cache invalidation
  - Time to Live (TTL):
    - **Mechanism:** Each cache entry has a duration of validity (TTL). Once this time expires, the entry is invalidated
    - **Example:** A five-minute TTL means data is considered valid for five minutes
    - **Benefits:** Simple to implement
    - **Challenges:** Data might become outdated within the TTL period
  - Conditional GETs
    - **Mechanism:** HTTP conditional GET requests use ETags to check if the resource has changed.
    - **Example:** An ETag is sent with a GET request, and the server responds only if the resource has changed
    - **Benefits:** Avoids unnecessary resource regeneration
    - **Challenges:** Still involves a network request, so it might not help with reducing round trips.
  - Notification-based Invalidation

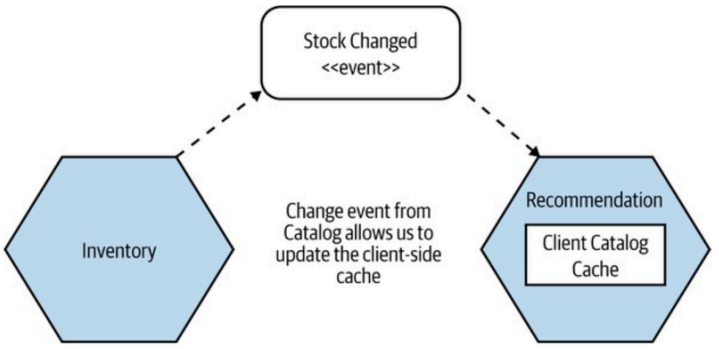

Figure 13-14. `Inventory` *fires* `Stock Change` *events, which* `Recommendation` *can use to update its local cache*

    - **Mechanism:** Events are used to notify subscribers about data changes
    - **Example:** An Inventory microservice fires Stock Change events to update the Recommendation microservice's cache
    - **Benefits:** Reduces the window for stale data
    - **Challenges:** More complex to implement, relies on a reliable notification mechanism
  - Write-through Caching
    - **Mechanism:** Updates to the cache and the origin are made simultaneously
    - **Example:** Updating a database and an in-memory cache within the same transaction
    - **Benefits:** Minimizes the window for stale data
    - **Challenges:** Typically used on the server side, less useful for broader client-side caching
  - Write-behind Caching
    - **Mechanism:** The cache is updated first, followed by the origin
    - **Example:** Writing data to the cache and then asynchronously updating the origin
    - **Benefits:** Faster writes to the cache
    - **Challenges:** Risk of data loss if the cache is not durable, making it difficult to determine the true source of truth
- **The Golden Rule of Caching**
  - Be cautious about implementing caching in too many places. Multiple caches between the client and the source of fresh data can result in stale data and complicate the process of determining data freshness. This complexity arises from the need to balance data freshness with system optimization for load or latency.
  - Example Scenario
    - **Inventory Microservice:** Caches stock levels with a one-minute TTL
    - **Recommendation Microservice:** Also caches stock levels with a one-minute TTL
    - When the client-side cache in Recommendation expires, it fetches data from Inventory's cache, which could be up to one minute old. This nesting can result in data being up to two minutes stale, even if each cache individually considers the data to be only one minute old
  - Tips to Avoid Issues
    - **Timestamp-based expiration:** Preferable over TTLs for better freshness control
    - **Minimize Cache Locations:** Aim to cache in as few places as possible to simplify reasoning about data freshness
    - **Premature Optimization:** Avoid adding unnecessary complexity. Use caching primarily as a performance optimization, and only when needed
- **Freshness Versus Optimization:** TTL (Time-To-Live) invalidation can lead to stale data if the origin changes shortly after a new copy is fetched. Reducing TTL (e.g., from five minutes to one minute) decreases staleness but increases the number of calls to the origin, impacting latency and load. Balancing data freshness and system optimization involves understanding user needs and system capabilities. Users prefer fresh data, but not at the cost of system stability. Simplifying caching strategies minimizes complexity and helps maintain a balanced system. The fewer the caches, the easier it is to manage and reason about data freshness and system performance

- Autoscaling
  - Autoscaling enables the automatic scaling of microservices based on load and instance availability through automated provisioning and deployment of virtual hosts. There are two main approaches
    - **Predictive Scaling:** This method uses historical data to anticipate peak loads and scale resources accordingly. For example, if peak load occurs between 9 a.m. and 5 p.m., additional instances are brought online before the peak and turned off afterward. Businesses with seasonal cycles also benefit from this approach
    - **Reactive Scaling:** This method scales resources in response to real-time load changes or instance failures. It requires quick response times and sufficient buffer capacity to handle sudden load spikes. Load tests are crucial for validating autoscaling rules to avoid production failures.
  - Implementation Tips
    - Collect historical data to understand load patterns
    - Use load tests to simulate different loads and validate autoscaling rules
    - Combine predictive and reactive scaling for optimal performance and cost-effectiveness
  - Use Case Example: A news site that experiences regular daily traffic patterns can use predictive scaling for routine load changes and reactive scaling for unexpected spikes due to big news stories
  - Caution
    - Avoid scaling down too quickly to ensure sufficient capacity
    - Use autoscaling for failure conditions initially while gathering data before scaling based on load.