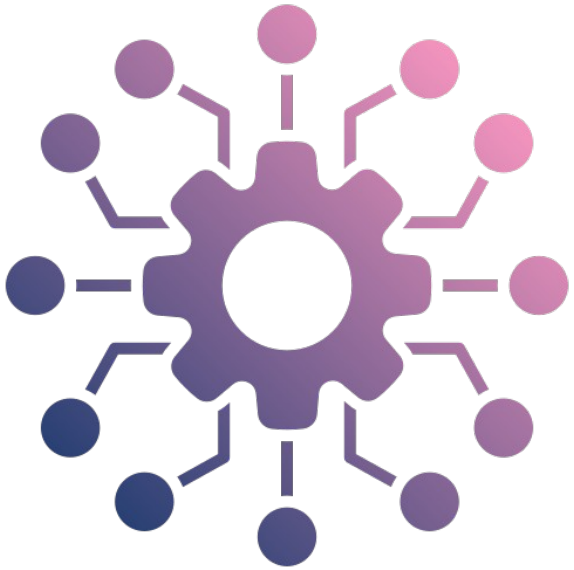


Microservice Design Patterns

Ravindu Nirmal Fernando

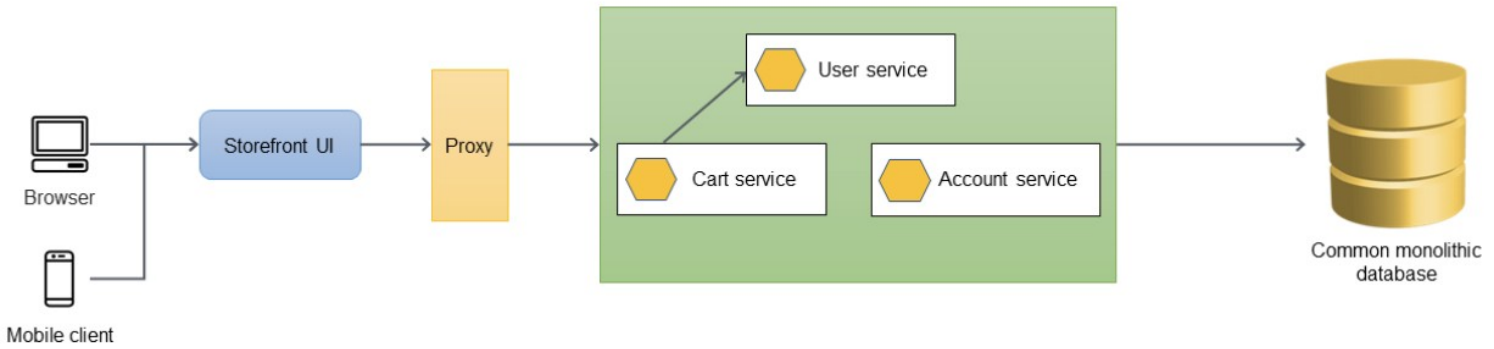
SLIIT | March 2025

Design Patterns for migrating Monoliths to Microservices

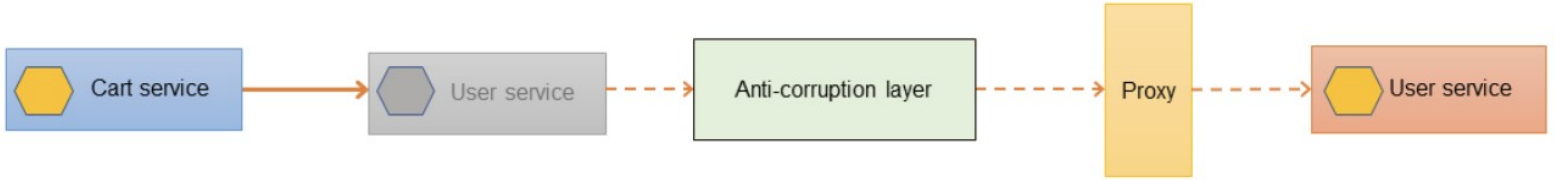
- Anti-Corruption Layer (ACL) pattern
- Strangler fig pattern

Anti-Corruption Layer (ACL) pattern

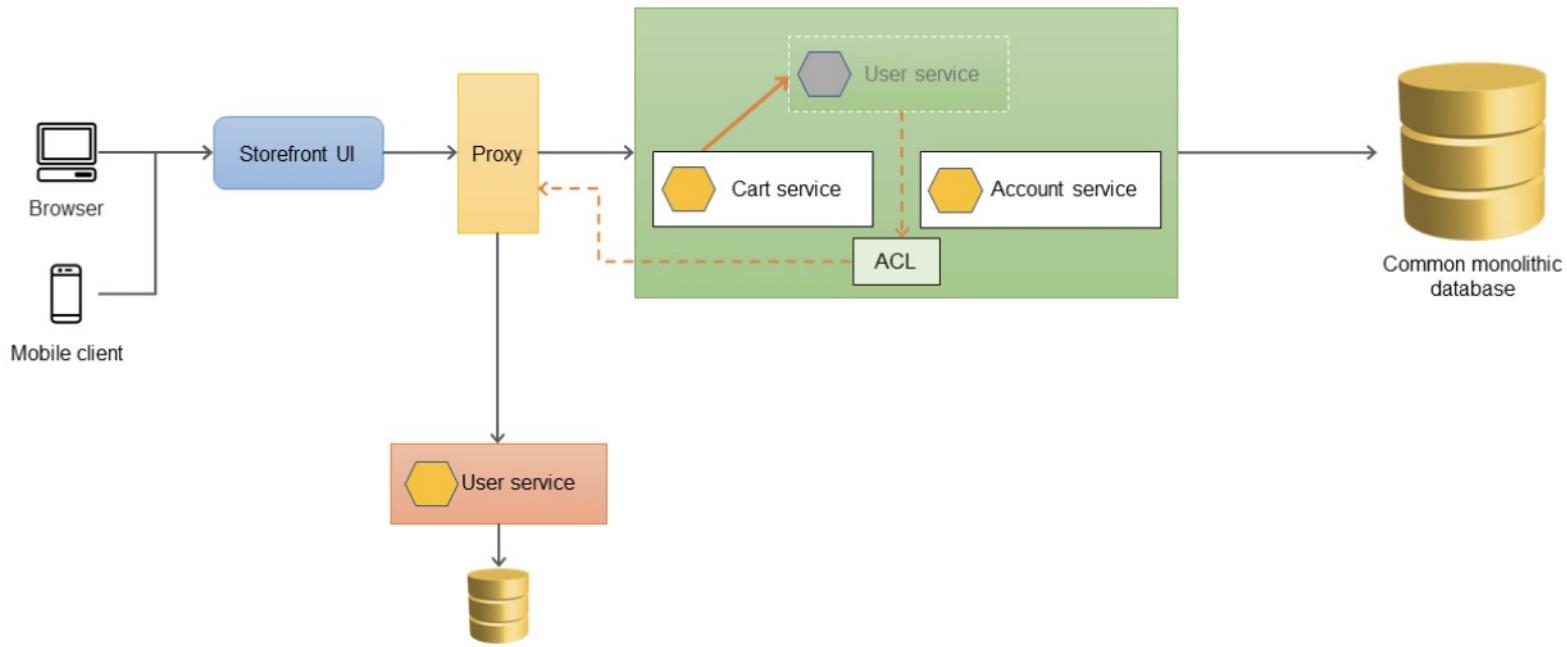
- Allows gradual translation from monoliths to microservices architecture.
- Allows legacy systems to communicate with modern services without internal changes and with minimal impact.
- Goal of ACLs is to minimize changes to existing functionality in monoliths and reduce business disruptions.
- Acts as an adapter or facade, converting calls to the new interface.
- ACL is also consumed as a part of Strangler-Fig pattern which we are going to talk next…



Existing Monolithic App



Introduction of ACL layer when User service is migrated

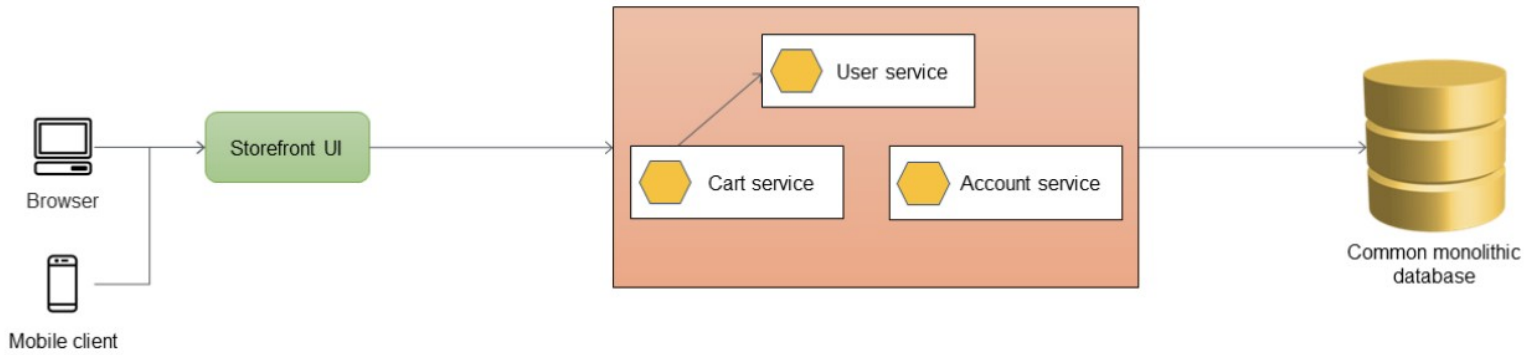

Browser
Mobile client
Storefront UI
Proxy
User service
Cart service
Account service
ACL
Common monolithic database
User service

Strangler fig Pattern

- Helps migrating a monolithic application to a microservices architecture incrementally, with reduced transformation risk and business disruption.
- Migrating a monolithic application to microservices based one requires rewriting and refactoring the code base and doing it once will be a huge risk.
- Hence Strangler fig patterns allows teams to focus on doing this migration incrementally and gradually while allowing app users to use the newly migrated features progressively.

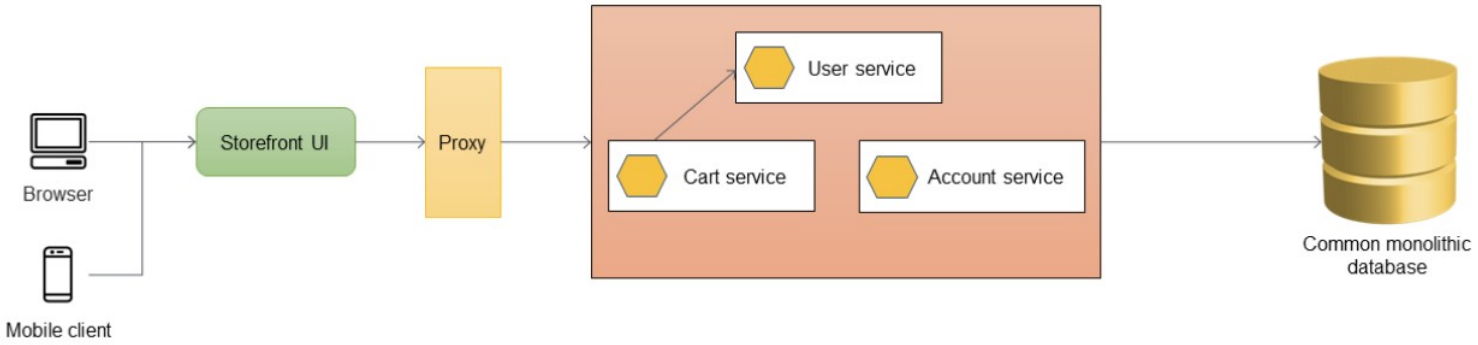
Process Involved in Strangler fig

- **Identify Replaceable Components**: Start with parts of the system that are easiest to replace or most in need of an upgrade.
- **Build New Features as Services**: Develop new functionalities as separate services outside the legacy system.
- **Reroute Traffic**: Gradually reroute user traffic from the old system to the new services.



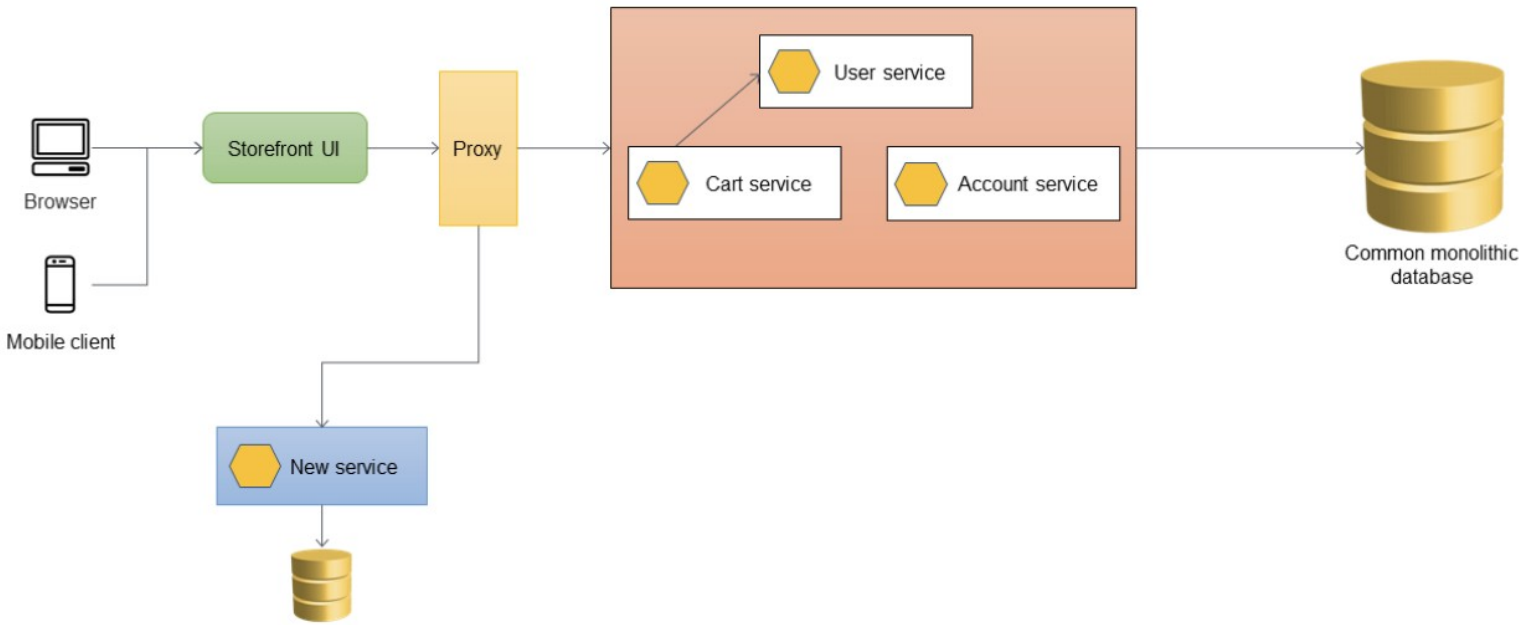
A monolithic application has three services: user service, cart service, and account service. The cart service depends on the user service, and the application uses a monolithic relational database.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



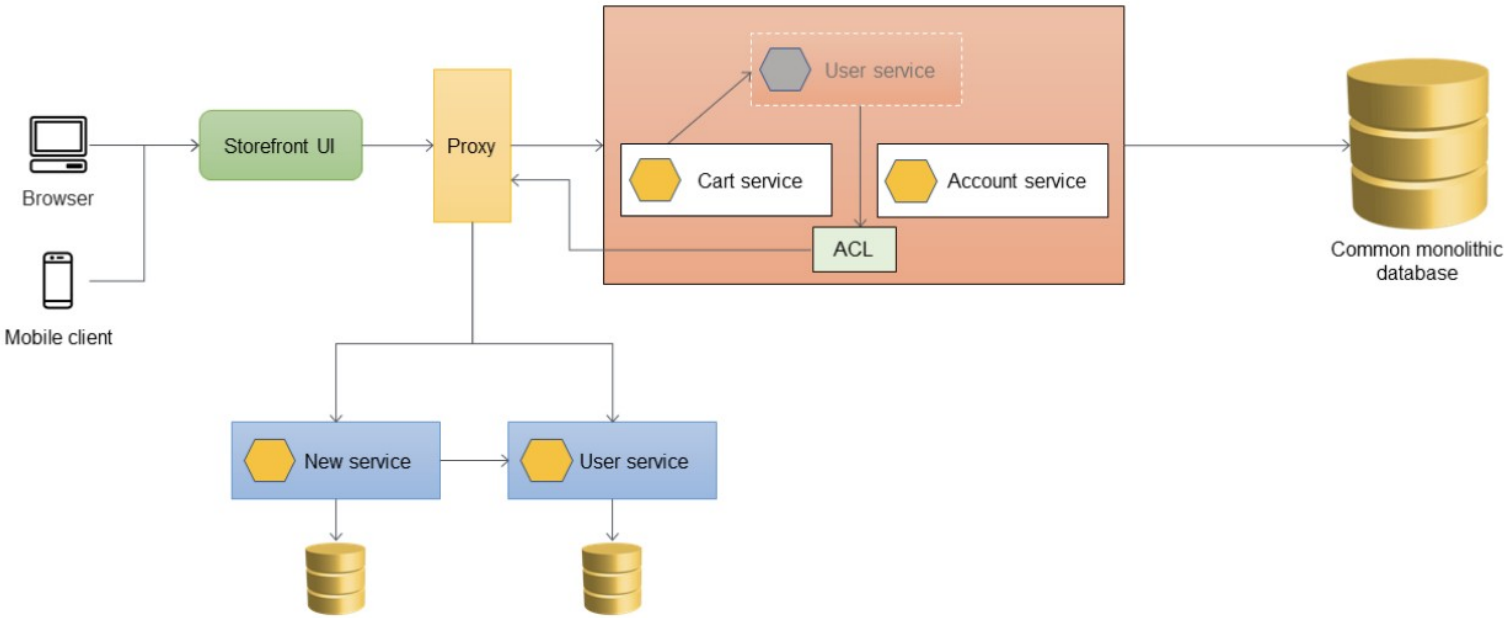
First step is to add a proxy layer between the storefront UI and the monolithic application. At the start, the proxy routes all traffic to the monolithic application.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



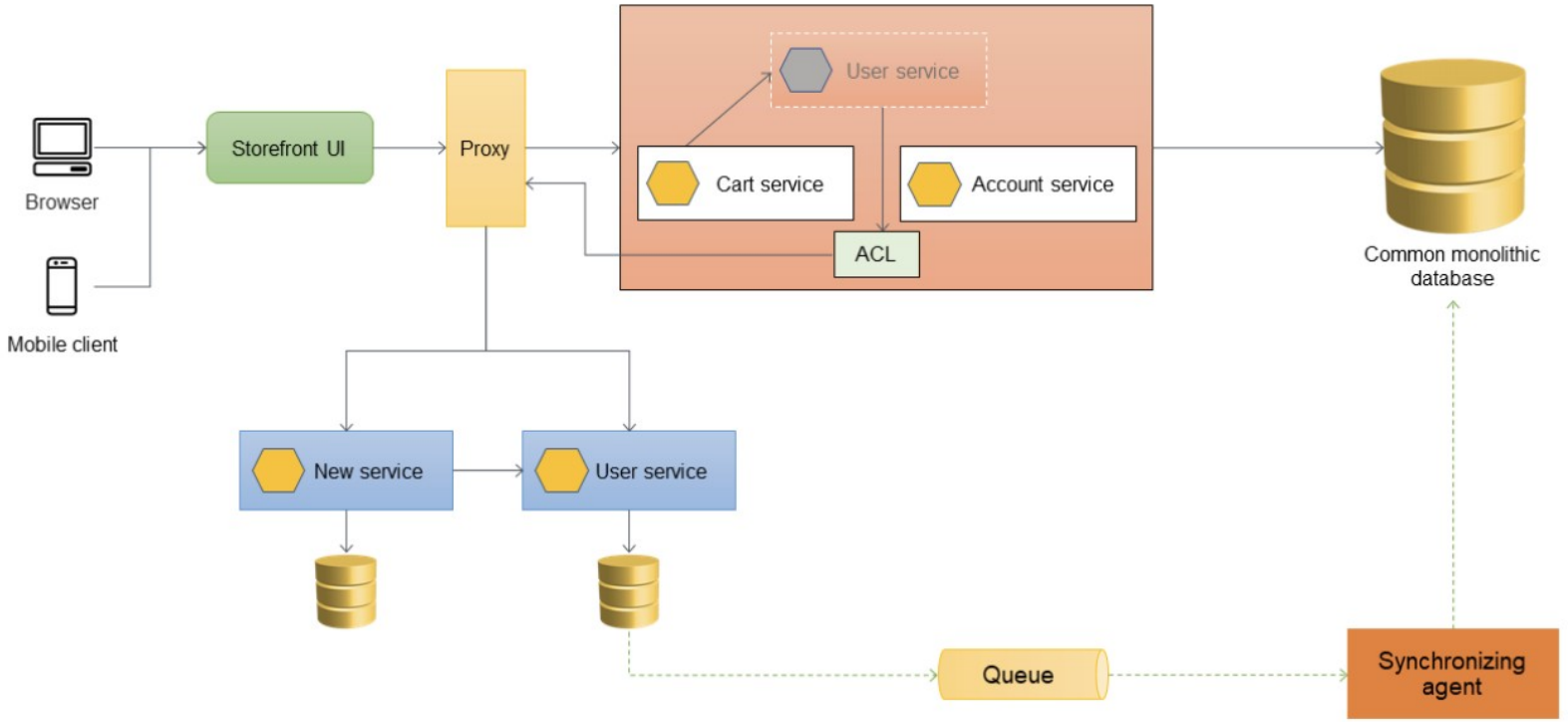
New services are implemented as microservices instead of adding features to the existing monolith. However, you continue to fix bugs in the monolith to ensure application stability. The proxy layer routes the calls to the monolith or to the new microservice based on the API URL.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



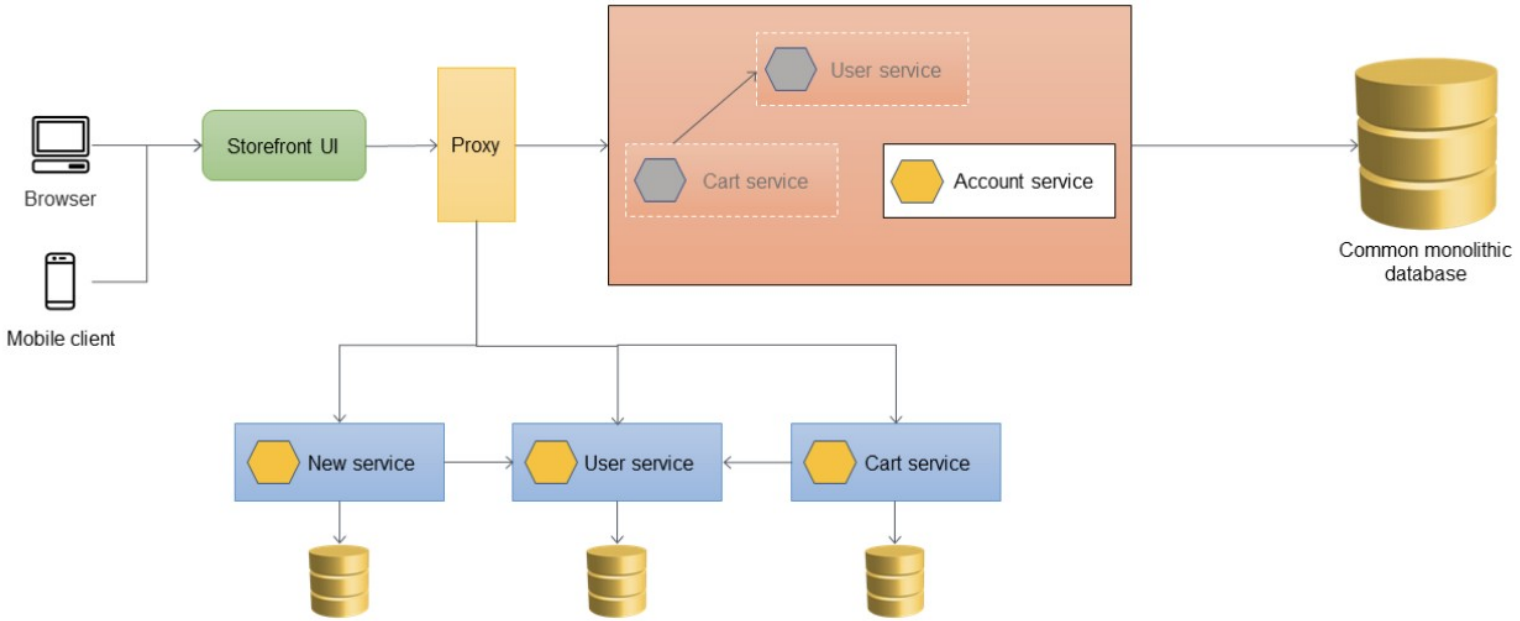
During the migration process, when the features within the monolith need to call the features that were migrated as microservices, the ACL converts the calls to the new interface and routes them to the appropriate microservice.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



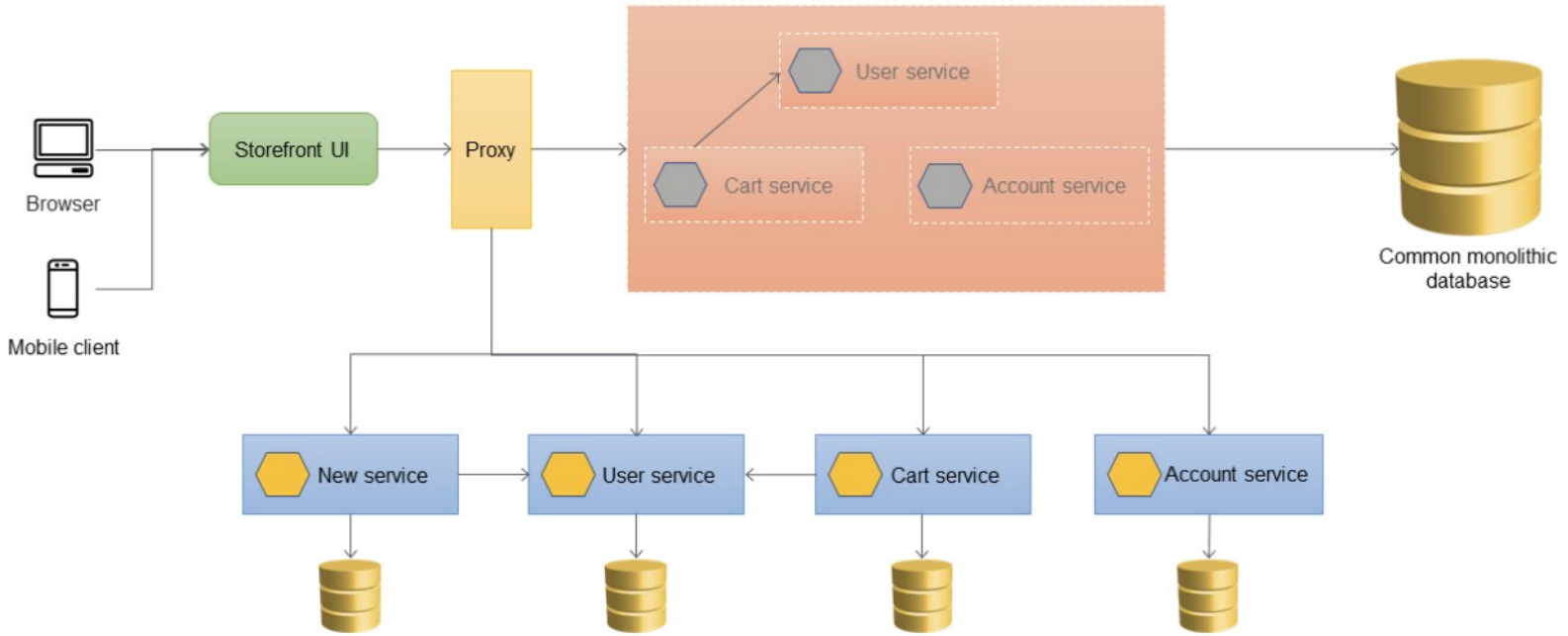
Data synchronization is crucial when downstream services, rely on a monolithic architecture, use data from a microservice. In this scenario, a User microservice, which has its own data layer, requires synchronization with the monolith. To facilitate this, a synchronization agent can be introduced. Update events from the microservice's database are sent to a queue. The agent then reads these events from the queue and synchronizes them with the monolithic database, ensuring eventual data consistency between the microservice and the monolith.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



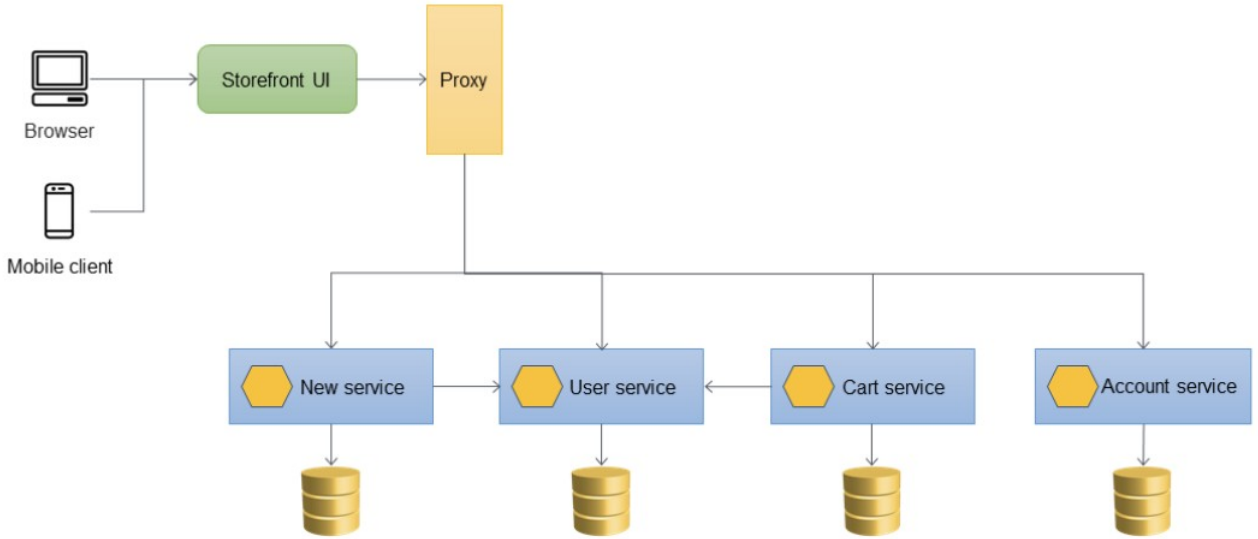
Once all interdependent components have been fully migrated to microservices, it becomes feasible to refactor the code to eliminate the Anti-Corruption Layer (ACL) components.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



The final strangled state where all services have been migrated out of the monolith and only the skeleton of the monolith remains. Historical data can be migrated to data stores owned by individual services. The ACL can be removed, and the monolith is ready to be decommissioned at this stage.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html



The final architecture after the monolithic application has been decommissioned.

https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/strangler-fig.html

Microservice Design Patterns

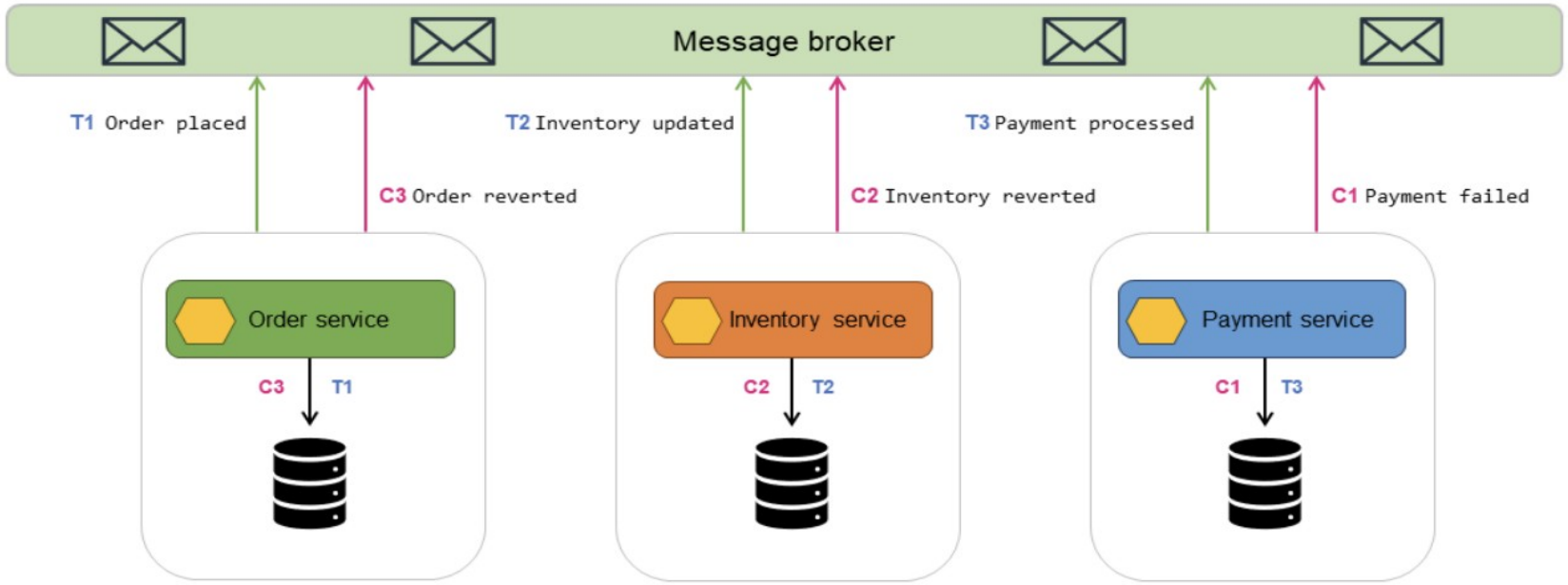
In a distributed transaction, multiple services can be called before a transaction is completed. When the services store data in different data stores, it can be challenging to maintain data consistency across these data stores.

Saga Pattern - A *saga* consists of a sequence of local transactions. Each local transaction in a saga updates the database and triggers the next local transaction. If a transaction fails, the saga runs compensating transactions to revert the database changes made by the previous transactions.

- Manages Distributed Transactions Across Multiple Microservices and Databases.
- Breaks Down a Transaction into a Series of Local Transactions for Each Service.
- Addresses the challenge of maintaining atomicity, consistency, isolation, and durability in microservices.
- Utilizes Compensating Transactions or Actions in Case of Local Transaction Failures.
- Avoids Using Two-Phase Commit Protocols for Transaction Management.
- Maintains Overall System Consistency Through Compensatory Mechanisms.

Saga Choreography

- Ensures data integrity in distributed transactions across multiple services using event subscriptions.
- Depends on the events published by the microservices.
- The saga participants (microservices) subscribe to the events and act based on the event triggers.



https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/saga-choreography.html#saga-choreography-implementation

Use the saga choreography pattern when:

• Your system requires data integrity and consistency in distributed transactions that span multiple data stores.

• The data store (for example, a NoSQL database) doesn't provide 2PC to provide ACID transactions, you need to update multiple tables within a single transaction, and implementing 2PC within the application boundaries would be a complex task.

• A central controlling process that manages the participant transactions might become a single point of failure.

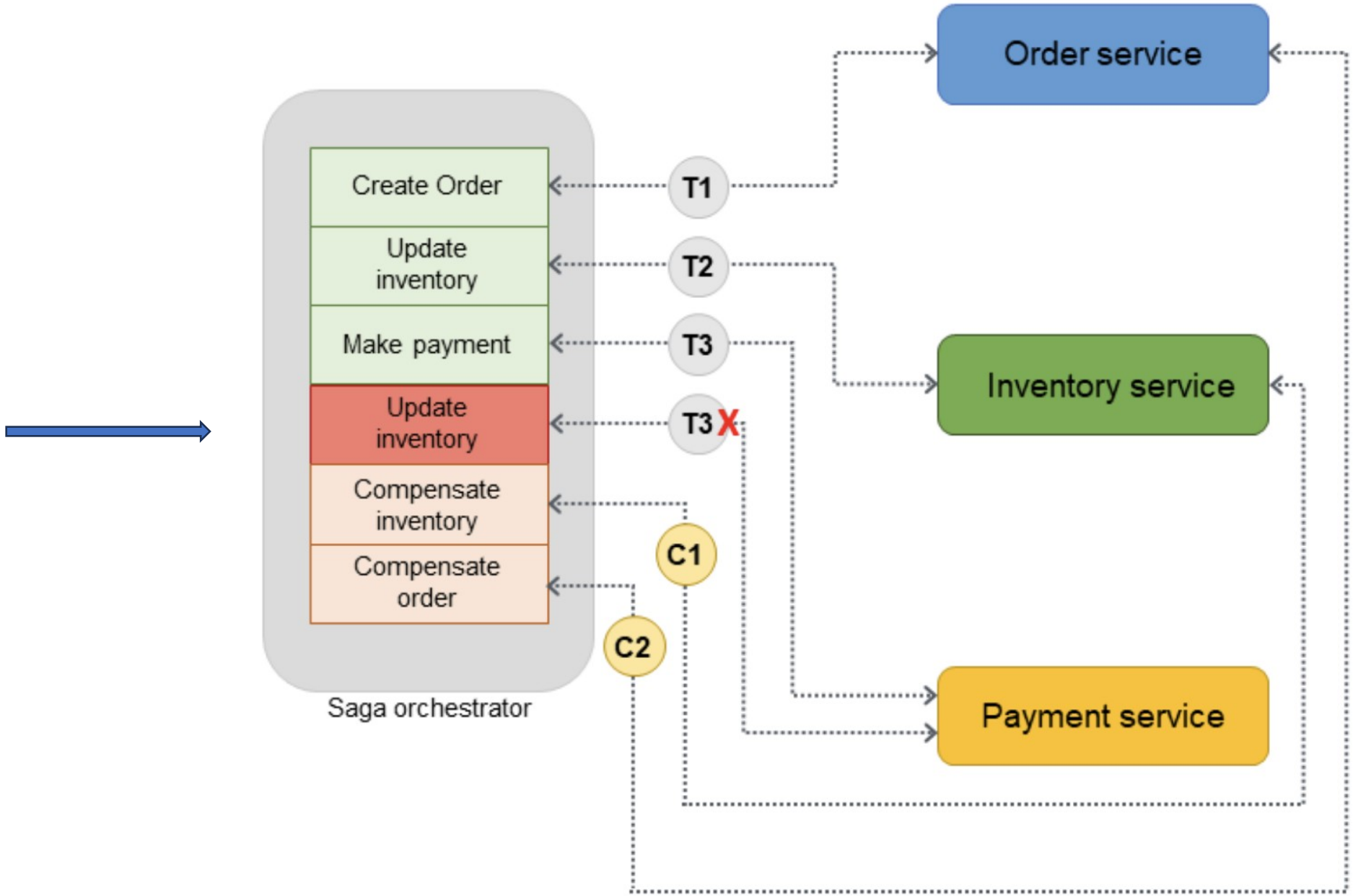
• When there are small number of participants (microservices) involved.

• The saga participants are independent services and need to be loosely coupled.

• There is communication between bounded contexts in a business domain.

Saga Orchestrator

- Uses a central coordinator (*orchestrator*) to help preserve data integrity in distributed transactions that span multiple services.
- Utilizes the orchestrator to manage the transaction lifecycle. Orchestrator is aware of all the steps required for transaction.
- Orchestrator sends messages to to participant microservices to initiate operations. These participant microservices report back to the orchestrator.
- Orchestrator acts as the decision maker and determines the next microservice to engage based on received messages.


Order service
Create Order
Update inventory
Make payment
Update inventory
Compensate inventory
Compensate order
T1
T2
T3
T3 X
C1
C2
Inventory service
Payment service
Saga orchestrator

Use the saga orchestration pattern when:

• Your system requires data integrity and consistency in distributed transactions that span multiple data stores.

• The data store doesn't provide 2PC to provide ACID transactions, and implementing 2PC within the application boundaries is a complex task.

• You have NoSQL databases, which do not provide ACID transactions, and you need to update multiple tables within a single transaction.

• Your system has many saga participants (microservices) involved and loose coupling between participants are required.

Developing and deploying Microservices with K8s

- **Traffic Management with Ingress**:
  - Using Kubernetes Ingress for efficient HTTP/HTTPS routing to services.
  - Ingress acts as a reverse proxy, simplifying routing and providing SSL/TLS termination and load balancing.
- **Scaling Microservices**:
  - Leveraging tools like Horizontal Pod Autoscaler (HPA) for automatic scaling based on CPU usage or custom metrics.
  - Kubernetes also supports manual scaling for handling varying loads effectively.
- **Using Namespaces for Organization**:
  - Utilizing Kubernetes namespaces to divide cluster resources among multiple users or teams.
  - Grouping related services in the same namespace simplifies management and applies policies at the namespace level.
- **Implementing Health Checks**:
  - Essential for monitoring the status of services using readiness and liveness probes.
  - Health checks allow Kubernetes to replace non-functioning pods automatically.
- **Service Mesh for Advanced Traffic Management**:
  - Implementing a service mesh for handling service-to-service communication.
  - Provides traffic management, service discovery, load balancing, and failure recovery.

- **Single Responsibility Principle for Microservices**:
  - Design each microservice with a single responsibility for easier scaling, monitoring, and management.
  - Tailor scaling policies, resource quotas, and security configurations to the specific needs of each service.
- **Continuous Delivery/Deployment (CD)**:
  - Utilizing Kubernetes Deployment objects for a declarative management of microservices.
  - Implement rolling updates for gradual change rollout and use tools like Argo Rollouts for more reliable rollback and progressive deployment strategies.
- **Monitoring and Debugging**:
  - Collect and visualize metrics using tools like Prometheus and Grafana.
  - Use application performance monitoring (APM) tools for detailed performance data of microservices.

References

- https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/introduction.html
- https://learn.microsoft.com/en-us/azure/architecture/patterns/