DESIGN PATTERN TERM PAPER

DESIGN PATTERN IN CLOUD-NATIVE APPLICATION

VIVEK MURARKA (22200673)

Term paper submitted in part fulfilment of the degree of

MSc. in Computer Science (Negotiated Learning)

Supervisor: Mel Ó Cinnéide

UCD School of Computer Science

University College Dublin

2022-2023

.

Abstract

Microservice is becoming a popular way for developing web services. As the organization moves towards microservice architecture, it is important to understand how to design microservices. This paper will discuss 6 major design patterns observed in a microservice architecture and will reflect upon real-world use cases of how this design pattern helped the organization to build its services.

Table of Contents

1 Introduction

The field of software development strategy has been ever-evolving and has seen significant changes in how systems are designed, developed, deployed, and managed. Not long back application was developed as a monolithic application which was then replaced by Service-oriented architecture (SOA). And now we see a fundamental shift towards microservice architecture which are developed and deployed as cloud-native application.

Each architecture has defined different standards for designing applications which were considered best practices at that time. As the application grew bigger, the design became more complex, and it increased the development time, as it includes a lethargic debugging and refactoring process to do a small fix. As more functionality was included maintaining this application became a challenge. Thus, most of these applications were scrapped and redesigned or repurposed with different software development strategies.

Now major organizations have shifted towards cloud-native applications as it guarantees faster development, deployment, and maintenance strategy. But, if we don't analyze the best design strategy for developing this application, we will end up again refactoring our application. Since, the cloud-native application is a vast topic, which involves multiple stages of design and development, this paper will focus on different design principles associated with microservice architecture.

2 Terminologies

2.1 Cloud Native Application

Although Cloud Native Application has been defined differently by different people and there is no proper general definition, I find below the definition which covers most aspects of it.

Cloud-native is building software applications as a collection of independent, loosely coupled, business-capability-oriented services (microservices) that can run on dynamic environments (public, private, hybrid, multi-cloud) in an automated, scalable, resilient, manageable, and observable way. [1]

2.2 Microservice

Microservice is the key to building cloud-native applications.

Microservices are a way of designing and building an application as a set of smaller, independent services that communicate with each other using APIs. This approach allows for greater flexibility and scalability, as each service can be developed and deployed separately and the overall application can be easily modified to meet changing needs. Microservices are commonly used in the creation of modern web-based applications to divide a large, complex codebase into more manageable pieces.

3 Microservice Design Pattern

After we have understood basic terminology, let's discuss the most popular microservice design pattern in detail.

As with any other design pattern, microservices design patterns are no silver bullet. Each of these design patterns has its merits and demerits. Let’s discuss each one in detail.

3.1 Strangler Pattern

The Strangler pattern is a method for gradually migrating a monolithic application to a microservices architecture. It involves creating new microservices to replace individual features or functionality of the monolithic application while leaving the rest of the system intact. As more features are migrated, the monolithic application becomes a "strangler" of the new microservices, eventually becoming redundant as all its functionality has been migrated.

The Strangler pattern allows for a more incremental and incremental approach to migrating to a microservices architecture, as it allows the organization to gradually decompose the monolithic application into smaller, more manageable pieces. This can be less risky and disruptive than attempting to overhaul the entire system at once.

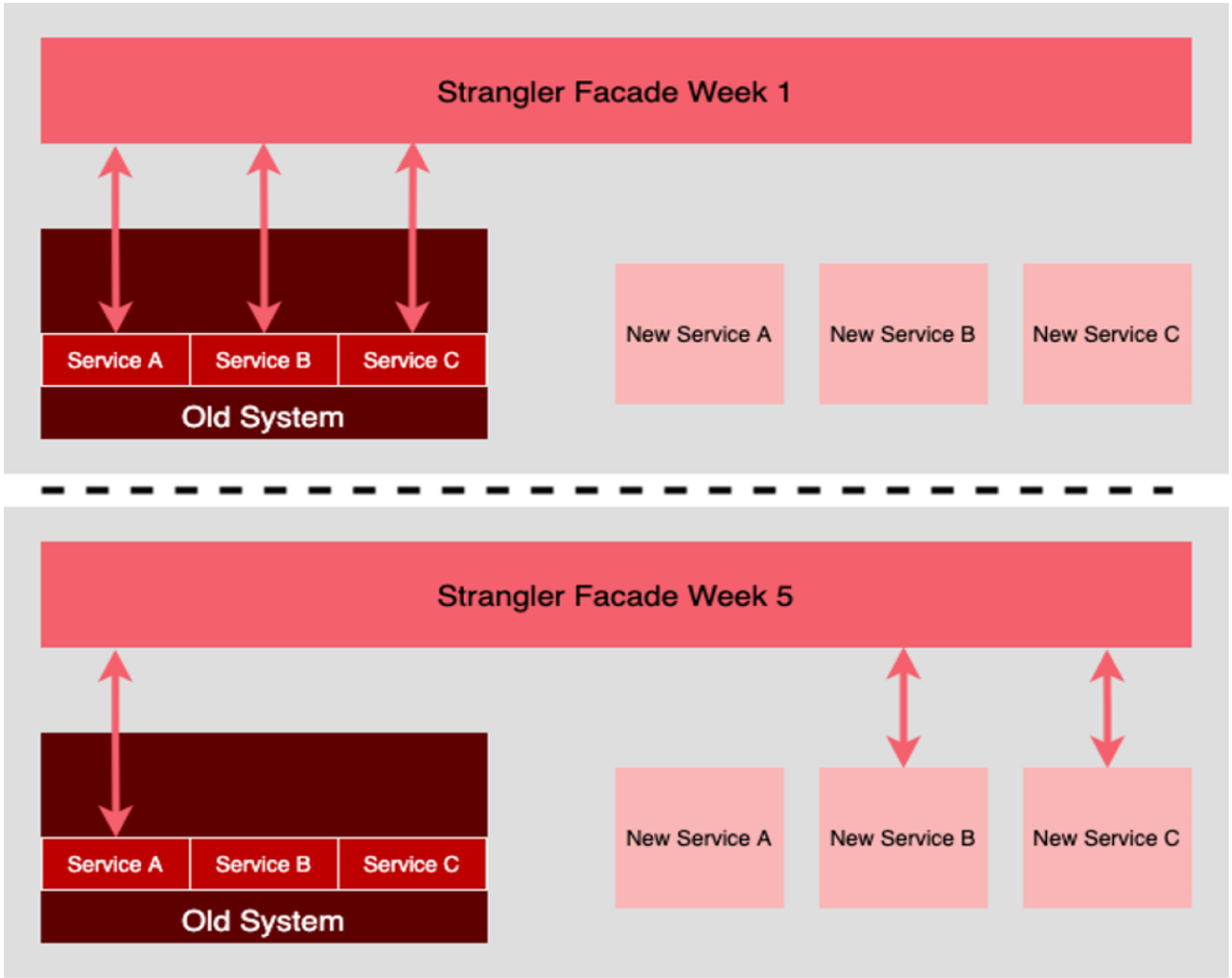

Figure 1 The Strangler architecture pattern [2]

The Strangler pattern involves using a facade to cover an "old" system while adding new replacement services behind it over time. The facade acts as the entry point to the old system, with requests passing through it. The services within the old system are then gradually refactored into a new set of services. As each new service becomes operational, the facade is updated to redirect requests that used to go to the corresponding service in the old system to the new service. This process continues until all the services in the old system have been replaced with the new ones, effectively "strangling" the old system.

This way we could test the new system, with the old system still doing the heavy lifting, and when the new system is reliable and big enough it can take the workload of the old system until the old system becomes obsolete.

Shopify used a strangler pattern to refactor their entire code base which was initially messy, and complex, with no clear boundary. Refactoring was a daunting task and the objective of the company was clear, that refactoring must not cause any downtime. This is when they decided to use the Strangler pattern and they gave a 7-step process for this migration.

1. Define an interface for the thing that needs to be extracted [3]

   To start the process of restructuring, the first step is to establish the public interface for the object being separated. This can involve creating new methods in an existing class or defining a new model altogether. The purpose of this step is simply to establish the new interface; we will still use the existing interface to access data during this phase. In their case, they use an existing Shop object and continue to retrieve data from the shop's database table.

2. Divert calls from the old system to a new system

   Now we divert all the object call to a new system. This new change will allow the controller to make use of the new interface which we defined in step 1.

3. If writing is required then construct a new data source for the new system

   This step entirely depends on our interface and how it processes data. This change might need to modify and create a new column or create a new table entirely.

4. Create new writers to write into a new data source

   If our existing business logic is less complicated we can continue using the current writer. But, it is suggested to create a new writer which is capable enough to continue writing in existing data sources. Then we add some test cases to ensure that all our validation work as intended.

5. Migrating data from old data sources to new

   In this step, we must implement an iterator that iterates over all records and stores them into a new data source. We must ensure data consistency between both new and old sources. It is recommended to implement a proper logging mechanism as it is very handy in finding the persistent failure during migration. Finally, we do run a comparative test between new and old sources to check if data are truly in-sync.

6. Enforce method in newly defined interface to read from a newer data source

   As we have removed dependency from the old data source, we need not continue reading from it, as our new data source has all the data from the old ones.

7. Finally, get rid of old source and legacy code.

3.1.1 Advantages

- Mitigates risk when updating or transforming a system
- Older systems continue working until newer systems are reliable.
- We can add new unique services or functionality during refactoring

3.1.2 Disadvantages

- Refactoring a system requires constant attention to routing and network management.
- It can also be challenging to transition from older services to new ones, as each instance may require special logic to accommodate the rerouting. This process, known as “adapter hell,” can be particularly time-consuming when there are many services involved.
- It is important to have a rollback plan in place in case anything goes wrong during the refactoring process, as it allows for a quick and safe return to the previous system.

3.2 Saga Pattern

In a case where each Microservice has there own database, which works perfectly for individual functionality, but when a transaction requires access to multiple services this creates an issue. To resolve this we use Saga Pattern.

In the saga pattern, a business transaction that involves multiple services is implemented as a sequence of local transactions, where each local transaction updates the database and triggers the next one through the publication of a message or event. If a local transaction fails due to a violation of a business rule, the saga executes compensating transactions to reverse the changes made by the preceding local transactions. This helps to maintain data consistency and handle errors in a distributed system.

There are two popular ways to implement the Saga pattern.

To best understand these two patterns we will consider a business use case, where an Online Retail system allows users to place orders only if they have sufficient credit. For this, it would need access to two services, one CreateOrderService and CustomerCreditService. Only if both services return a positive response then our call is successful. Let’s see how we will solve this with a different Saga approach.

1. **Choreography** – each local transaction publishes domain events that trigger local transactions in other services [4]

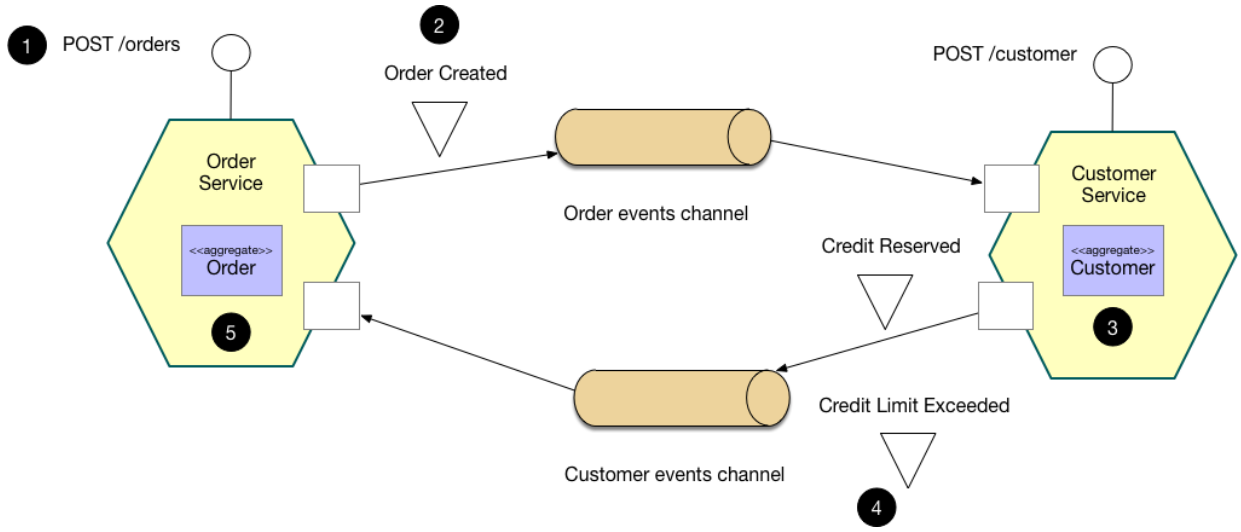

Figure 2 Choreography-based saga [4]

The CreateOrderService receives a request to create a new order and puts it in a PENDING state. It then sends a notification that an order has been created. The CustomerCreditService tries to set aside credit for the order and sends a message about the result. The Order Service then either approves or rejects the order based on this information.

2. **Orchestration** – an orchestrator (object) tells the participants what local transactions to execute [4]

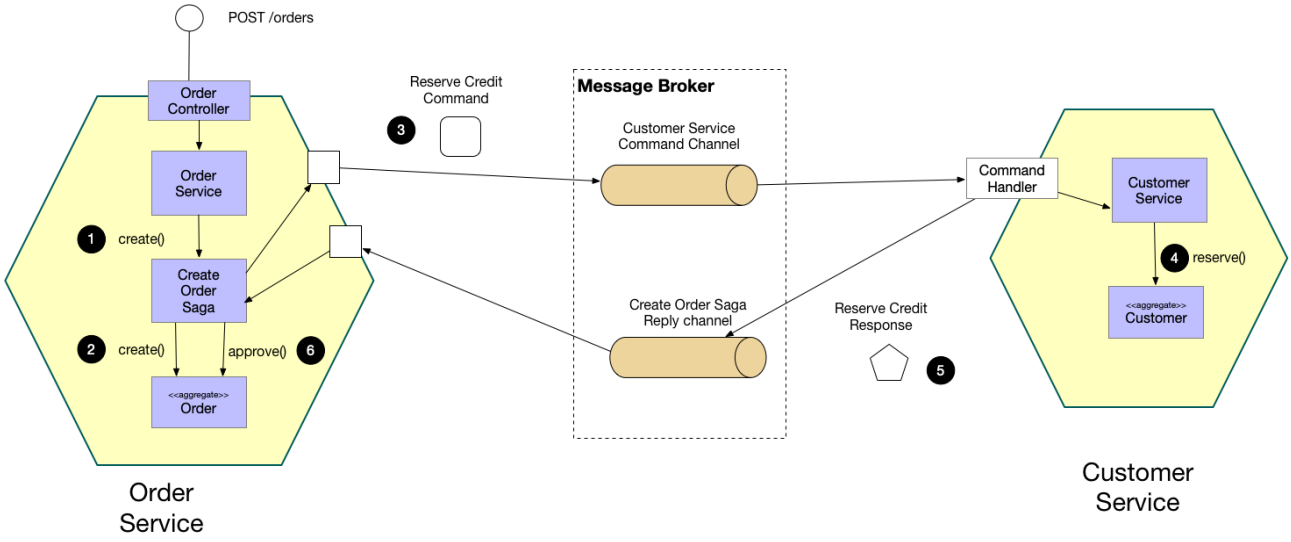

Figure 3 Orchestration-based saga [4]

The CreateOrderService receives a request to create a new order and initiates a process (called a "saga orchestrator") to handle the request. The process creates the order in a PENDING state and sends a request to the CustomerCreditService to set aside credit for the order. The CustomerCreditService attempts to reserve the credit and sends a message with the result. The process overseeing the request then either approves or rejects the order based on the credit reservation outcome.

3.2.1 Advantages

- It ensures data consistency and handles failure in a distributed environment.

- It allows for more flexibility and scalability in a microservices architecture. Since each step in the process is handled by a separate service, it is easier to add or remove services without affecting the overall process.
- Better failure handling, as each step in the process, can be compensated for if necessary.

3.2.2 Disadvantages

- A major drawback is that it makes debugging complex to pinpoint where the transaction initially failed.
- More resource-intensive, as it requires the creation and management of additional processes.

3.3 Aggregator Pattern

The Aggregator design pattern is often used in microservice architectures to combine data from multiple microservices into a single response to a client request. It can be used to improve the performance and scalability of a system by reducing the number of requests that a client needs to make to different services.

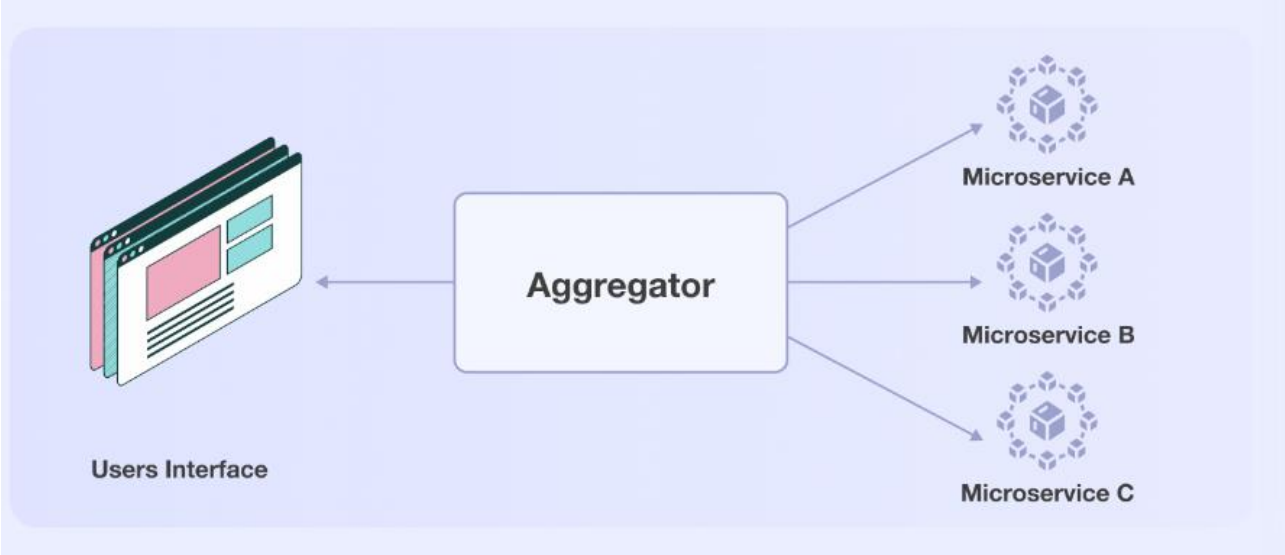

Figure 4 Aggregator Pattern [5]

In an Aggregator pattern, a client sends a request to an Aggregator service, which then makes requests to the relevant microservices to retrieve the necessary data. The Aggregator service combines the data from these different services and returns a single, comprehensive response to the client.

The Aggregator pattern allows you to gather the business logic from multiple microservices into a single microservice, which is then exposed to the user interface (UI). This means that the UI only communicates with a single microservice, rather than multiple services.

3.3.1 Advantages

- Improve the performance of a system by reducing the number of round trips that a client needs to make to different services.
- Simplify the client code, by creating abstraction, as the client only needs to make a single request to the Aggregator service rather than multiple requests to different services.

3.3.2 Disadvantages

- Increase latency in response as the aggregator wait for the response from multiple microservice

- Single point of failure: If the Aggregator class goes down, the entire system may be affected.
- Tight coupling: The Aggregator class and the classes it aggregates are tightly coupled

3.4 Event Sourcing

The Event Sourcing design pattern is a way of storing and managing the state of a system by recording the events that occur within it, rather than storing the current state directly. In a microservice architecture, Event Sourcing can be used to maintain the state of individual microservices, as well as the relationships between them.

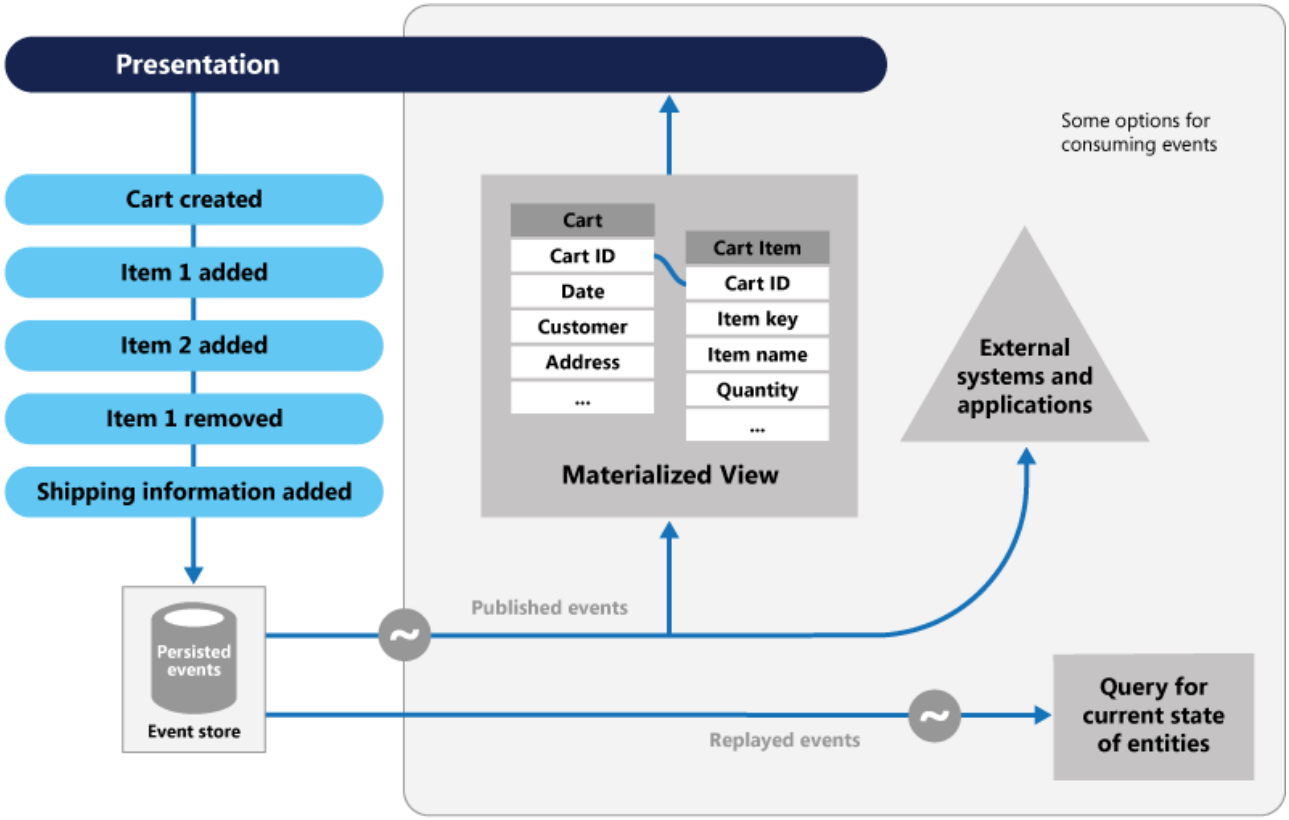

Figure 5 Event Sourcing pattern [6]

In an eCommerce app, actions such as creating a shopping cart or adding or removing items are considered events. These events are recorded in an event store and added to an event bus, which acts as a message queue. The event store acts as a message broker, forwarding the events to the event bus. These events are then summarized in a materialized view database, which allows users to view the status of the shopping cart in real-time. The event store, or write database, maintains a historical record of all events, or changes in state, so that the system can replay the events at any point in history if a user requests the status of the shopping cart.

3.4.1 Advantages

- Decouple System: Microservice doesn't need to query or update each other's state.
- Separation of Concern: Each microservice is only responsible for storing its own event.
- Simplified Testing: Allow us to replay the events that have occurred in the past to verify their behavior

3.4.2 Disadvantages

- Requires additional resources to store and process the events
- Increase complexity: Event Sourcing often requires the use of materialized views, which can be complex to implement and maintain.

3.5 Command Query Responsibility Segregation (CQRS)

When an application becomes large, it can be challenging to efficiently read data from an event store, especially when using the Event Sourcing pattern, as it requires processing all the entities to retrieve one. In these situations, the Command Query Responsibility Segregation (CQRS) pattern can be a useful solution, as it separates read and update operations. This can improve the performance and scalability of the system.

CQRS is a design pattern that separates the responsibilities of querying data (reading) from modifying data (writing). In a microservice architecture, CQRS can be used to decouple the different concerns of a system and improve scalability and performance.

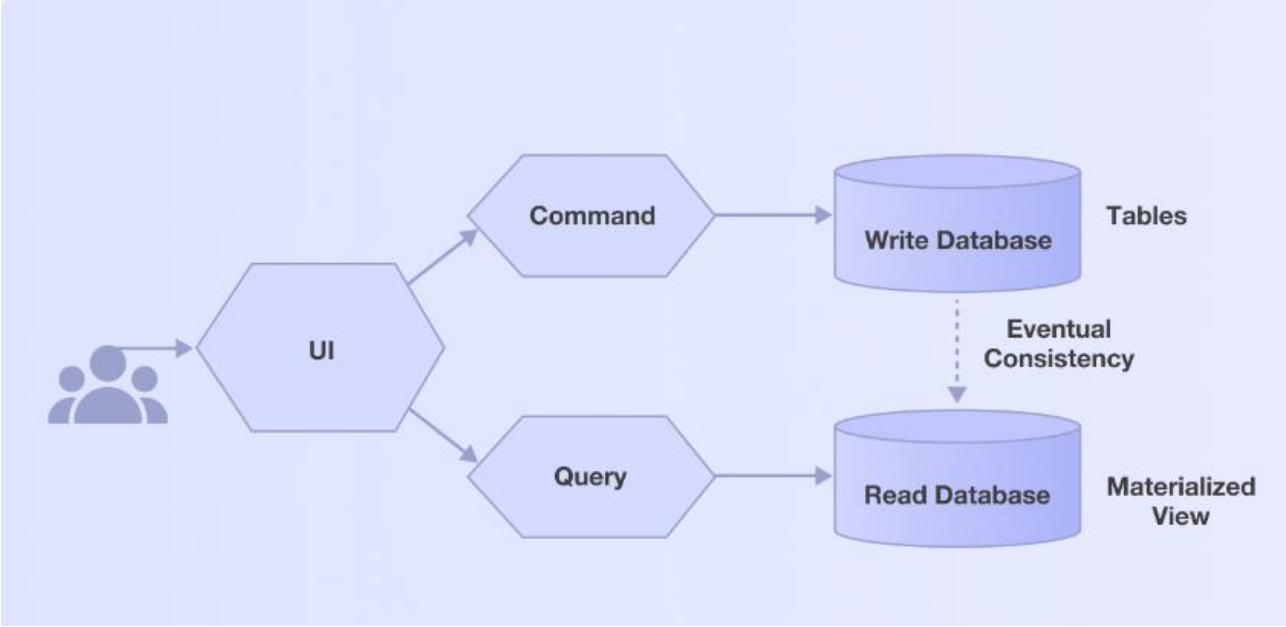

Figure 6 Command Query Responsibility Segregation (CQRS) [5]

In a CQRS-based system, each microservice is responsible for either handling commands (write operations) or handling queries (read operations). Commands are typically sent to a command handler, which updates the system's state and generates events to be stored in an event store. Queries are sent to a query handler, which retrieves data from a separate read store.

In 2013, **Instagram** had 200 million active users per month and was storing over 20 billion photos each month. By 2015, the number of active users had increased to 400 million and the company was storing 40 billion photos per month, serving over a million requests per second. To handle the increased data storage and requests, Instagram decided to scale its infrastructure geographically, but this presented challenges including replicating data. To improve performance, the company implemented the Command Query Responsibility Segregation (CQRS) pattern, which separated the update and reading operations for each new replica. This allowed them to create a separate streamer for new replicas that stored data locally, resulting in faster query execution and reduced replica generation time by half.

3.5.1 Advantages

- Improved Performance: For example, read-only replicas can be used for querying data, which can be faster than querying the primary data store.
- Improved security: For example, write access can be restricted to a small group of users, while read access can be granted to a larger group.
- Improved reliability: For example, if the primary data store goes down, read-only replicas can still be used to serve query requests.

3.5.2 Disadvantages

- Inconsistency: Eventual consistency is based on the BASE principle [7], which means there is a short duration when replicas and primary data are not in-sync.
- Increased testing and maintenance complexity as multiple databases must be tested and maintained.

3.6 Sidecar Pattern

The sidecar design pattern is a way to add functionality to an existing system by running a separate process alongside it. The sidecar process communicates with the main process through some form of inter-process communication (IPC), such as a network socket or shared memory.

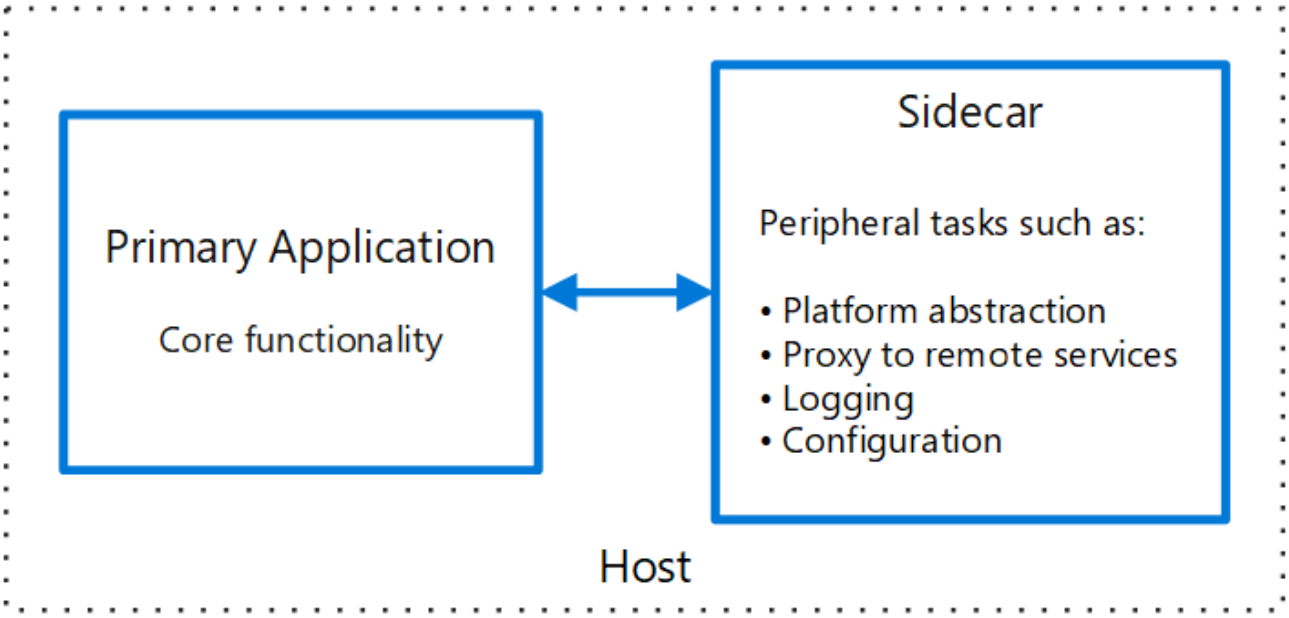

Figure 7 Sidecar pattern [8]

One common use of the sidecar pattern is to add logging or monitoring capabilities to a system. For example, a sidecar process could be used to capture log messages or metrics emitted by the main process and forward them to a centralized logging or monitoring system. This can be useful in situations where adding logging or monitoring directly to the main process is not feasible or desirable.

Another use of the sidecar pattern is to add dynamic behavior to a system. For example, a sidecar process could be used to update the configuration or routing rules of the main process at runtime, without requiring a restart of the main process.

3.6.1 Advantages

- Flexibility: The sidecar pattern can be used to add a wide range of functionality to a system, including logging, monitoring, configuration, and routing.
- Reusability: The sidecar pattern can be used to add functionality to multiple systems, allowing code reuse and reducing development effort.

3.6.2 Disadvantages

- Increased development effort: Require additional development effort, as it requires building and maintaining separate codebases for the main process and the sidecar process.
- Increased testing effort: Requires testing the main process, the sidecar process, and the integration between them.

4 References

[1] S. S. Kasun Indrasiri, Design Pattern for Cloud Native Applications, O'Reilly Media, Inc., 2021.

[2] B. Reselman, “The pros and cons of the Strangler architecture pattern,” [Online]. Available: https://www.redhat.com/architect/pros-and-cons-strangler-architecture-pattern.

[3] A. Chang, “Refactoring Legacy Code with the Strangler Fig Pattern,” [Online]. Available: https://shopify.engineering/refactoring-legacy-code-strangler-fig-pattern.

[4] C. Richardson, “Pattern: Saga,” [Online]. Available: https://microservices.io/patterns/data/saga.html.

[5] H. Dhaduk, “Top 7 Microservice Design Patterns to Use For Your Business in 2023,” [Online]. Available: https://www.simform.com/blog/microservice-design-patterns/.

[6] Microsoft Azure, “Event Sourcing pattern,” [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing.

[7] M. Knight, “What Is BASE?,” 17 March 2021. [Online]. Available: https://www.dataversity.net/what-is-base/.

[8] Microsoft Azure, “Sidecar pattern,” [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar.