ARQSOFT

ISEP – Mestrado em Engenharia Software

Sprint 2 - Software Architecture

2023

Sprint 2 – Software Architecture

Relatório de Desenvolvimento – Versão 1.0

José Carvalho, Luís Serapicos e Telo Gaspar

Project

1. Introduction

1.1 Objective

In the previous sprint, the ACME application was developed, and configuration issues were resolved. The main goal of this sprint is to redesign the ACME application by adopting a decentralized/distributed strategy.

.

1.2 Decentralized/Distributed Strategies

- **Business Domain Segregation:**

  The monolithic application is divided into three separate applications, each focusing on different aspects: Products manage functionalities related to products, Reviews handle customer feedback, and Votes manage voting/polling features.

- **Cloning:**
  - **Multiple Instances:**
    - Each application (Product, Review, Vote) has multiple copies implemented on various virtual machines or components.
  - **Scalability and Redundancy:**
    - Cloning enables scalable implementation, ensuring that if one instance fails, others continue to function. This configuration allows for greater scalability, fault tolerance, and focused development for each application.

NB: The current most updated branch is reviewMicroservice.

ADD

-Attribute Driven Design, is an architecture design driven by quality attributes concerns, and in this method there are a number of iterations, fitting nicely in an agile mindset, meaning we can do a little bit of early design and then start implementing, or we can do more iterations (e.g. three or four iterations), so that the refinement of the system can be enhanced and optimized progressively, to facilitate the development and align closely with desired quality attributes while accommodating changes and improvements throughout the design and implementation phases.

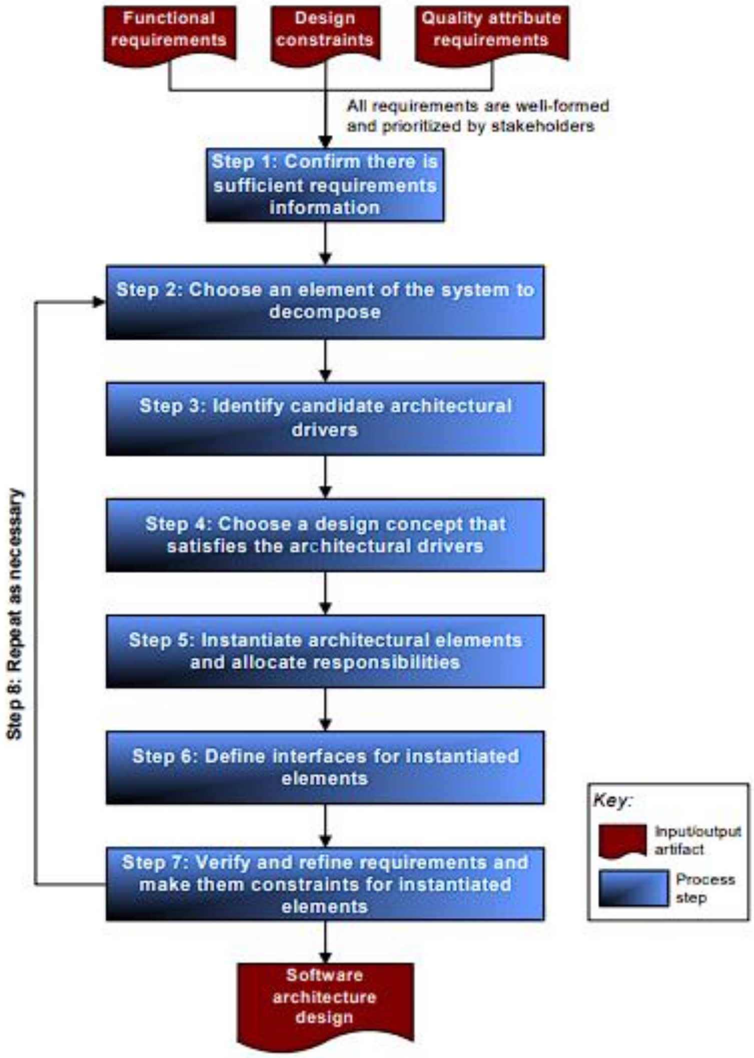

Functional requirements
Design constraints
Quality attribute requirements
All requirements are well-formed and prioritized by stakeholders
Step 1: Confirm there is sufficient requirements information
Step 2: Choose an element of the system to decompose
Step 3: Identify candidate architectural drivers
Step 4: Choose a design concept that satisfies the architectural drivers
Step 5: Instantiate architectural elements and allocate responsibilities
Step 6: Define interfaces for instantiated elements
Step 7: Verify and refine requirements and make them constraints for instantiated elements
Step 8: Repeat as necessary
Software architecture design
Key:
Input/output artifact
Process step

Figure 2: Steps of ADD

Non-functional requirements

- The system must increase the performance when in high demand (i.e. >Y requests/period).
- The system must use hardware parsimoniously, according to the run=me demanding of the system. Demanding peeks of >Y requests/period occur seldom.
- The system must increase releasability frequency.
- The clients should not be affected by the changes in the API (if any).
- The system must adhere to the company's SOA strategy of API-led connec=vity.

maintainability, performance, availability, scalability and elasticity.

Quality attributes:

. Maintainability

. Performance

. Availability

. Scalability

. Elasticity

. Compatibility

. Interoperability

. Releasability

Functional requirements

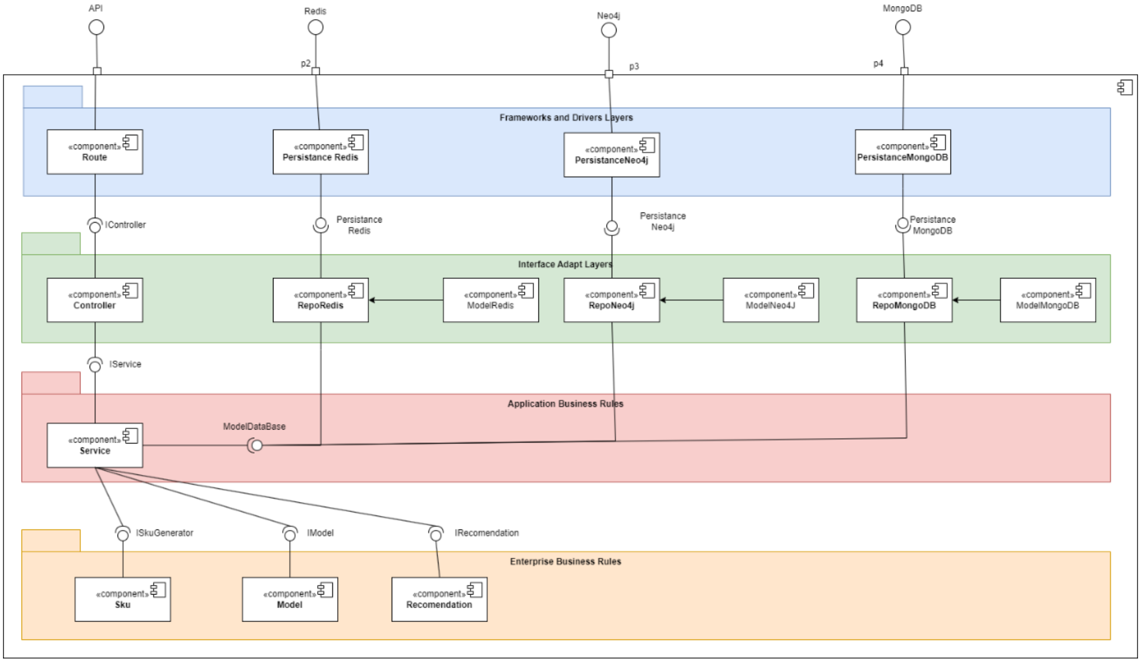
- As product manager, I want to publish a Product (i.e. available for clients, reviewers, voters) only after two other product managers accept it.
- As a reviewer, I want my Review is published (i.e. available voters) only if it is accepted by two other users that are recommended with that review (cf. recommendation algorithm).
- As a voter, I want to create a Vote and a Review in the same process (cf. previous requirement).

Quality Attribute Scenarios

ID	Quality Attribute	Scenario	Associated Use Case	Priority (Importance,Difficulty)
QA-1	Performance, Scalability, Elasticity, Availability	The system must increase the performance when in high demand (i.e. >Y requests/period).	All	(H,H)
QA-2	Scalability, Performance	The system must use hardware parsimoniously, according to the runtime demanding of the system. Demanding peeks of >Y requests/period occur seldom.	All	(H,M)
QA-3	Releasability, Maintainability	The system must increase releasability frequency.	All	(M,M)
QA-4	Compatibility, Maintainability	The clients should not be affected by the changes in the API (if any).	All	(H,H)
QA-5	Maintainability, Interoperability	The system must adhere to the company's SOA strategy of API-led connectivity.	All	(H,M)

Iteration 1

Goal – Create an overall system structure



Pros of Monolithic Architecture:

- **Simplicity** - Monolithic architectures are often simpler to develop, deploy, and manage compared to microservices or other distributed architectures. The codebase is usually contained in a single project, making it easier to understand and work with.
- **Development Speed** - Development can be faster in a monolith since all components are part of a single codebase. Changes can be made more easily without worrying about inter-service communication or coordination.
- **Easier Testing** - Testing is simplified in a monolith because all components are tightly integrated. It's easier to perform end-to-end testing, and the test environment is more straightforward to set up.
- **Deployment** - Deploying a monolith is usually simpler than deploying a distributed system. There's only one application to deploy, and the deployment process is typically more straightforward.
- **Resources** - In some cases, a monolith may use resources more efficiently than a distributed system, as there's less overhead associated with communication between services.

Cons of Monolithic Architecture:

- **Scalability** - Scaling a monolith can be challenging. If a specific component of the application requires more resources, you may need to scale the entire application, leading to inefficient resource usage.
- **Flexibility** - Monolithic architectures may limit the flexibility to choose different technologies for different components. If a technology upgrade is needed, it may require upgrading the entire monolith.
- **Complexity** - As the size of the monolith grows, complexity increases. It can become challenging to maintain, understand, and extend the codebase, leading to slower development cycles and increased risk of errors.
- **Development Bottlenecks** - In large development teams, multiple developers may need to work on the same codebase, leading to potential bottlenecks and conflicts. This can slow down the development process.
- **Limited Technology Diversity** - A monolithic architecture may limit the use of different programming languages or frameworks within a single application. In contrast, microservices can support a polyglot architecture where each service can use the most appropriate technology stack.

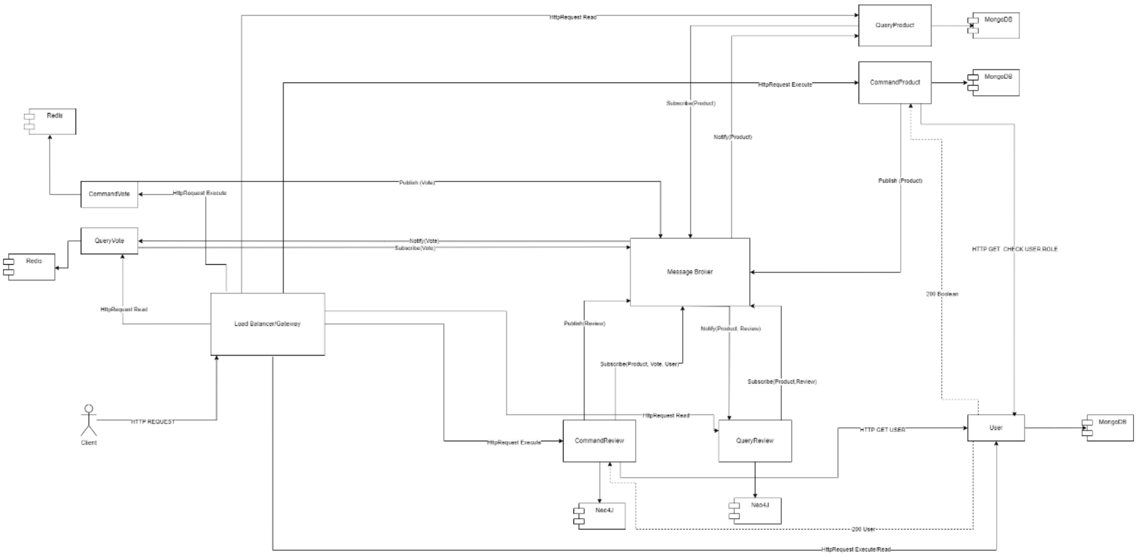


Fig. 2: Main architecture with a single instance of each microservice.

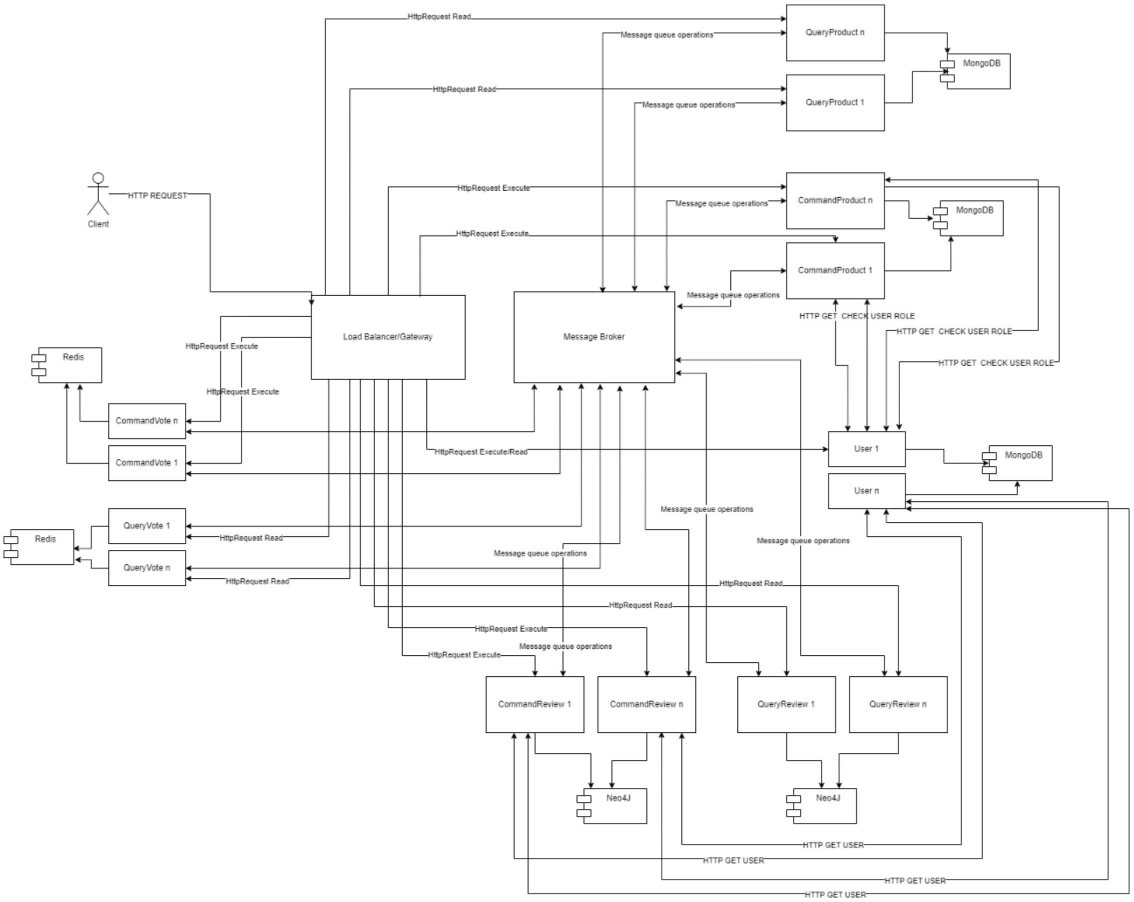


Fig. 3: Main architecture with a multiple instances of each microservice but with a shared database between the same instance type.

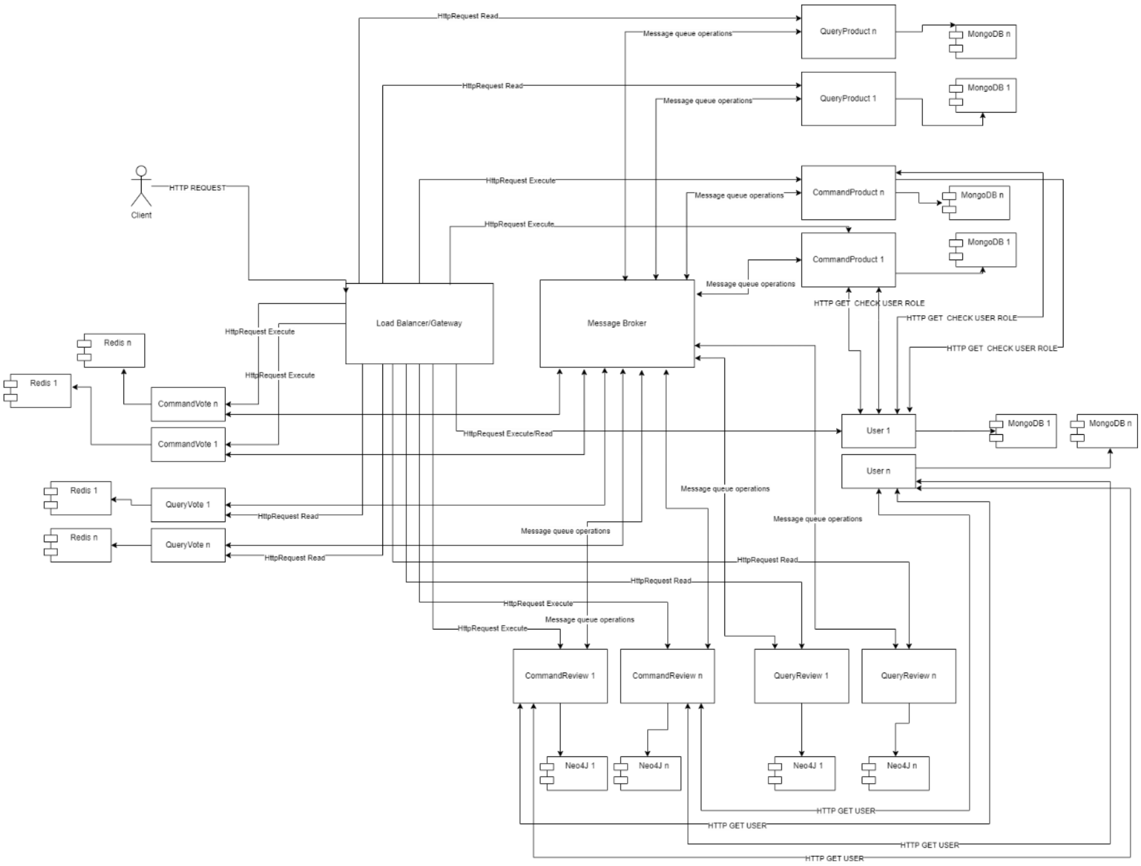


Fig. 4: Main architecture with a multiple instances of each microservice and also it's own databases.

Pros of Microservices Architecture:

- **Scalability** - Microservices enable independent scaling of individual services. This means that only the services that need additional resources can be scaled independently, providing better resource utilization.
- **Flexibility and Technology Diversity** - Microservices allow each service to be developed, deployed, and scaled independently. This flexibility enables the use of different programming languages, frameworks, and databases for different services, known as a polyglot architecture.
- **Easier Maintenance and Updates** - Since services are independent, it's easier to update, deploy, and maintain individual microservices without affecting the entire application. This leads to faster release cycles and reduced downtime.
- **Fault Isolation** - Failures in one microservice do not necessarily affect the entire system. Isolation of services ensures that issues are contained within a specific service and don't cascade to other parts of the application.
- **Team Independence** - Different teams can work on different microservices independently, allowing for parallel development and faster time-to-market. Teams can choose the most suitable technology stack for their specific service.

- **Easy Integration with External Services** - Microservices are well-suited for integrating with external services and third-party APIs. Each microservice can independently interact with external systems without affecting the overall application.

Cons of Microservices Architecture:

- **Complexity** - Microservices introduce a higher level of complexity compared to monolithic architectures. Managing the communication between services, data consistency, and versioning can be challenging.
- **Data Management Challenges** - Maintaining data consistency and managing transactions across multiple services can be complex. Coordinating database updates and ensuring data integrity requires careful design and implementation.
- **Initial Development Cost** - The transition from a monolithic architecture to microservices can be costly and time-consuming. It may involve rewriting or refactoring existing code, establishing new development and deployment processes, and training teams.
- **Testing Complexity** - Testing a microservices architecture involves testing individual services and their interactions. Comprehensive end-to-end testing becomes challenging, and additional effort is required for integration testing.
- **Security** - Securing a microservices architecture involves addressing challenges such as securing communication between services, managing access controls, and dealing with potential vulnerabilities in each service.

Strangler Fig Pattern

The "Strangler Fig Pattern" is a strategic approach to software architecture evolution, particularly when transitioning from a monolithic architecture to a microservices architecture. The pattern is named after the strangler fig tree, which starts as a seed planted in the canopy of a host tree. Over time, the strangler fig sends down roots that envelop and eventually replace the host tree. Similarly, in software development, the Strangler Fig Pattern involves gradually replacing or strangling components of a monolithic application with new microservices.

We encountered a real-world application of the Strangler Fig Pattern in a project at ACME. The initial project, Project 1, was a monolithic architecture. However, as the software requirements for Project 2, the need for a more scalable, flexible, and maintainable architecture became apparent.

Project 2 was initiated with the goal of transitioning the monolithic architecture of Project 1 into a microservices-based architecture. The Strangler Fig Pattern was adopted as the guiding strategy for this transition. The approach involved identifying and isolating specific functionalities or components within the monolith and gradually replacing them with microservices.

In this transition, we started by strangling the "Product" component. The Product-related functionality was extracted from the monolith and implemented as an independent microservice. This microservice could be developed, deployed, and scaled independently, allowing for better flexibility and scalability.

Following the success of the first strangle, subsequent efforts focused on other components like "Review" and "Vote." These components were also extracted from the monolith and implemented as separate microservices. This step-by-step migration helped in minimizing the risk associated with a complete system overhaul, allowing the development team to gradually modernize the architecture.

However, not all components were strangled into microservices. Certain classes, such as "User" and "Rating," were deliberately maintained within the monolith. The approach was pragmatic, ensuring that the benefits of microservices were realized where feasible while maintaining stability in areas where the transition could be more challenging.

The Strangler Fig Pattern provided a structured and iterative approach to modernizing the architecture, allowing the team to transition from a monolith to a microservices-based system incrementally. By adopting this pattern, the project achieved its objectives, gaining the advantages of microservices while minimizing disruption and risk during the migration process.

Command-Query Responsibility Segregation (CQRS)

The Command Query Responsibility Segregation (CQRS) pattern is a software architectural pattern that separates the read and write operations for a data store. This segregation allows for the optimization of each side independently, making it particularly useful in microservices architectures. After successfully adopting the Strangler Fig Pattern to transition from a monolith to a microservices architecture at ACME, the team further refined their design by implementing the CQRS pattern.

Following the strangulation process, where various components of the monolith were incrementally replaced with microservices, the team recognized the need for a more specialized approach to handle the diverse requirements of commands and queries within these microservices. This led to the adoption of the CQRS pattern at the microservices level.

Each microservice, starting with the "Product" microservice, followed the CQRS pattern. The key principle behind CQRS is the separation of the write operations (commands) from the read operations (queries). In the context of microservices, this meant creating two distinct microservices for each domain: one responsible for handling commands and another for handling queries.

For instance, the "Product" microservice was divided into two separate microservices – one dedicated to handling commands related to updating or modifying product data, and another specialized in handling queries for retrieving product information. The same approach was applied to other microservices, such as the "Review" microservice.

The Command microservices were designed to handle operations that modified the state of the system. These could include creating, updating, or deleting entities within the microservice's domain. These microservices were optimized for write-intensive operations and could be scaled independently based on the volume of commands being processed.

On the other hand, the Query microservices were tailored for read-intensive operations. They were responsible for responding to queries by providing read-only access to the system's data. Optimizing these microservices for read operations allowed for efficient and scalable data retrieval to enhance query performance.

By adopting the CQRS pattern at the microservices level, the ACME project achieved several benefits. It enhanced system scalability by allowing independent scaling of read and write operations based on their respective workloads. It also improved maintainability by providing a clear separation of concerns between commands and queries, making the codebase more modular and easier to comprehend.

The CQRS pattern, in conjunction with microservices, demonstrated its effectiveness in creating a more responsive and scalable system, aligning with the evolving needs of the ACME project. The thoughtful application of patterns like CQRS showcased the adaptability of the architecture to accommodate complex requirements while maintaining a high level of flexibility and efficiency.

Database-per-service and Polyglot Persistance

The "Database-per-service" and "Polyglot Persistence" patterns are architectural strategies commonly employed in microservices architectures to enhance flexibility, scalability, and the ability to tailor databases to the specific needs of each microservice. In the context of an ACME project that transitioned from a monolith to microservices, these patterns played a crucial role in optimizing the storage layer.

The "Database-per-service" pattern suggests that each microservice should have its dedicated database, decoupling the data storage concerns of different microservices. This approach aligns with the microservices principle of encapsulation, ensuring that each microservice can independently manage and evolve its data schema without impacting other services. In the ACME project, this pattern was meticulously applied to achieve a high level of autonomy and isolation among microservices.

Moreover, the "Polyglot Persistence" pattern encourages the use of different types of databases based on the unique requirements of each microservice. In essence, it promotes choosing the most suitable database technology for the specific needs of a microservice rather than adhering to a one-size-fits-all approach. This strategy acknowledges that different databases excel in different scenarios, offering optimal performance and features for specific use cases.

In the ACME project, the implementation of these patterns was exemplified by the choices made for the "Product" and "Review" microservices:

Product Microservices:

ProductCommand Microservice: This microservice, responsible for handling commands related to updating or modifying product data, was paired with a MongoDB database. MongoDB, a NoSQL document-oriented database, is well-suited for scenarios requiring flexible schema design and efficient handling of large amounts of semi-structured data.

ProductQuery Microservice: To handle read-intensive operations and queries for retrieving product information, the Polyglot Persistence pattern was applied. This microservice had its dedicated MongoDB database optimized for efficient and scalable data retrieval.

Review Microservices:

ReviewCommand Microservice: Designed to manage commands related to reviews, this microservice was paired with a Neo4j graph database. Neo4j excels in handling relationships between entities, making it a suitable choice for scenarios where the relationships among reviews, users, and products are essential.

ReviewQuery Microservice: For read-intensive operations and queries related to reviews, a separate Neo4j database was employed. This choice aligned with the Polyglot Persistence principle, ensuring that the storage solution matched the specific needs of the microservice.

Architecture Alternatives

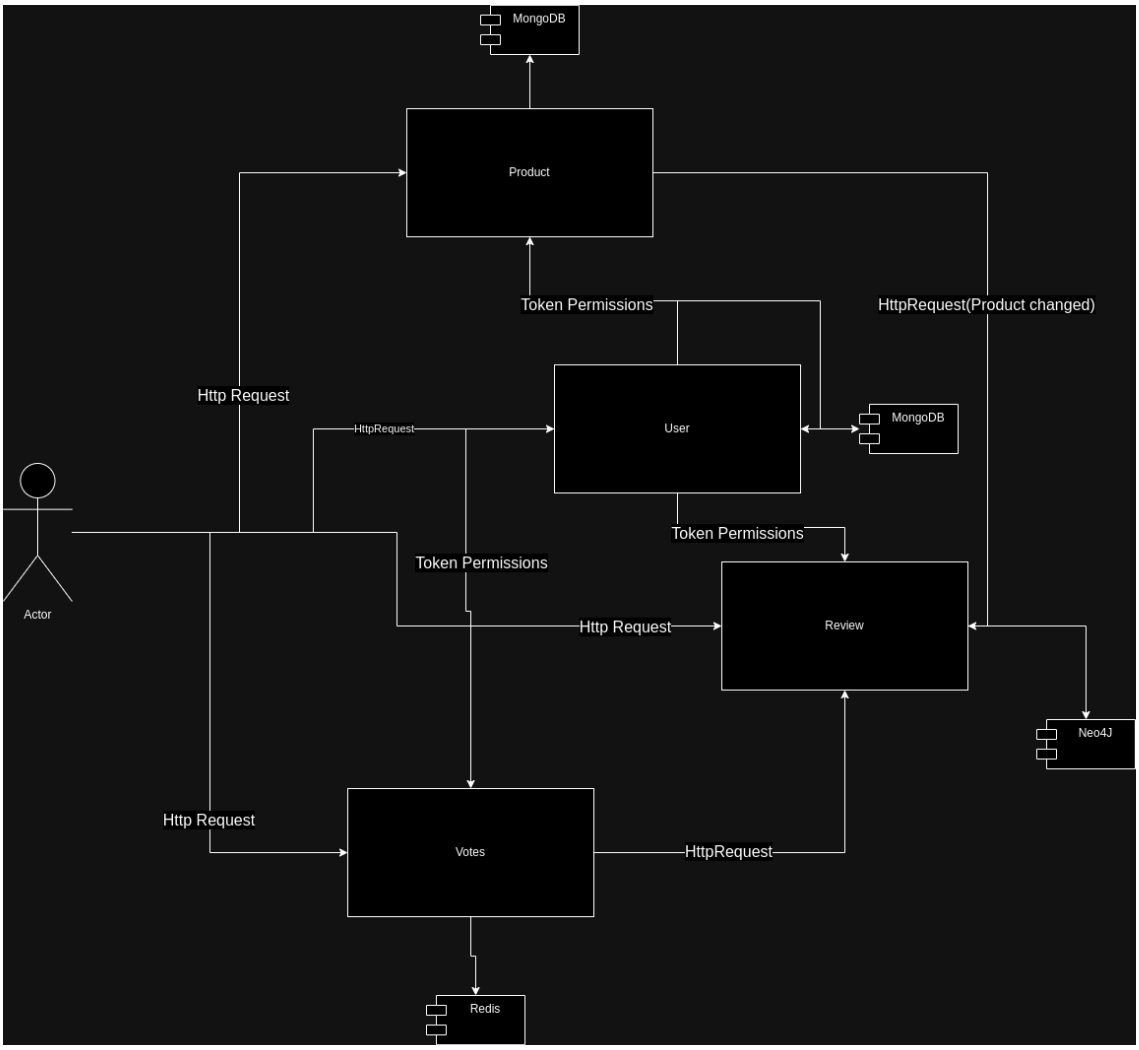


Fig. 5: Alternative microservices architecture based only on HTTP requests.

A microservices architecture is a modern and scalable approach to software development that revolves around the concept of breaking down a complex application into smaller, independent services. Each service, known as a microservice, is designed to handle a specific business capability and operates as a self-contained unit. A key aspect of microservices architecture is how these services communicate with each other, and one prevalent method is through HTTP APIs.

In this architecture, microservices leverage HTTP (Hypertext Transfer Protocol) for communication, establishing a standardized and widely adopted protocol for exchanging data between components. The communication is typically achieved through RESTful (Representational State Transfer) APIs or HTTP-based RPC (Remote Procedure Call) APIs. Let's delve into the significance of APIs and HTTP communication in a microservices context:

1. Service Interaction:

Microservices interact with each other through well-defined APIs. These APIs serve as the contract that specifies how services can communicate, including the data formats, endpoints, and methods.

2. RESTful APIs:

RESTful APIs are a common choice in microservices architecture due to their simplicity, scalability, and statelessness. They rely on standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.

3. Loose Coupling:

The use of APIs promotes loose coupling between microservices. Each microservice is independent and only needs to know the specifications of the APIs it interacts with. This independence enables services to evolve and scale independently.

4. Stateless Communication:

Microservices communication via HTTP is inherently stateless. Each request from one service to another contains all the information needed for the recipient service to understand and fulfill the request. This statelessness simplifies scalability and fault tolerance.

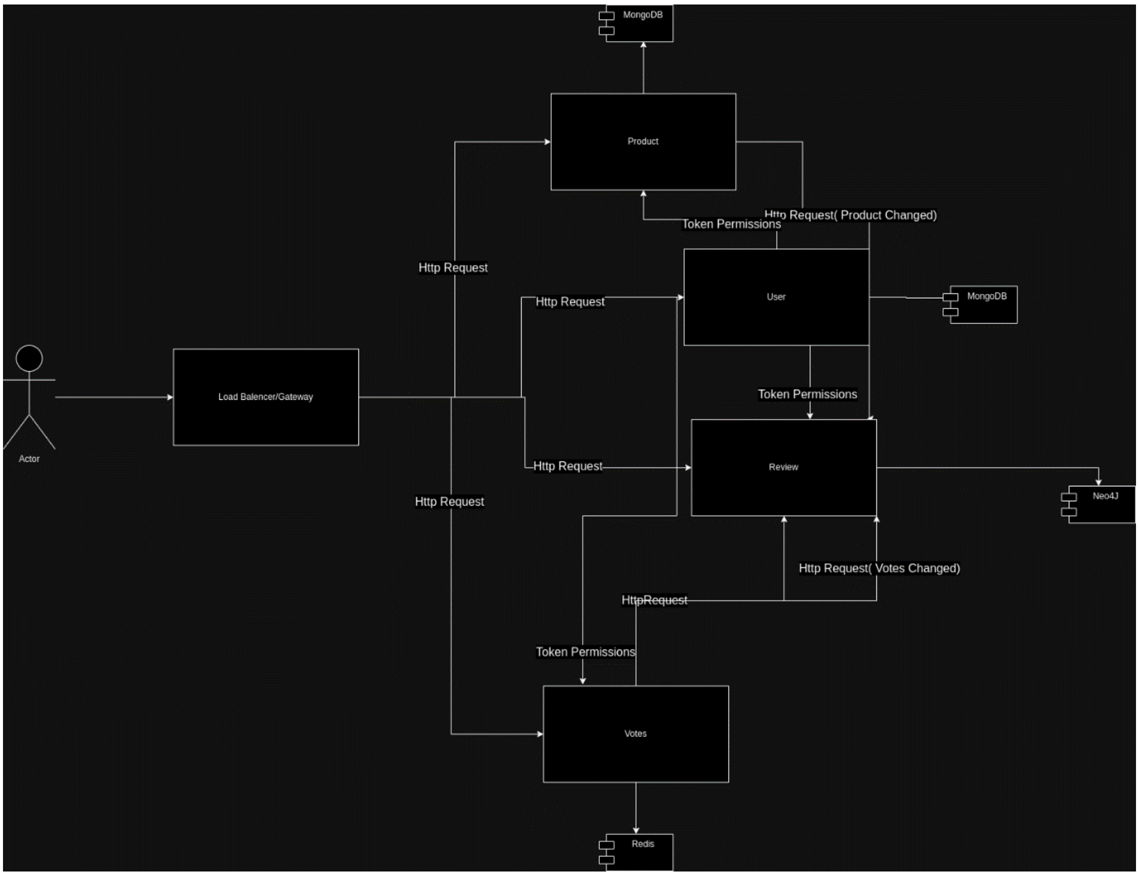
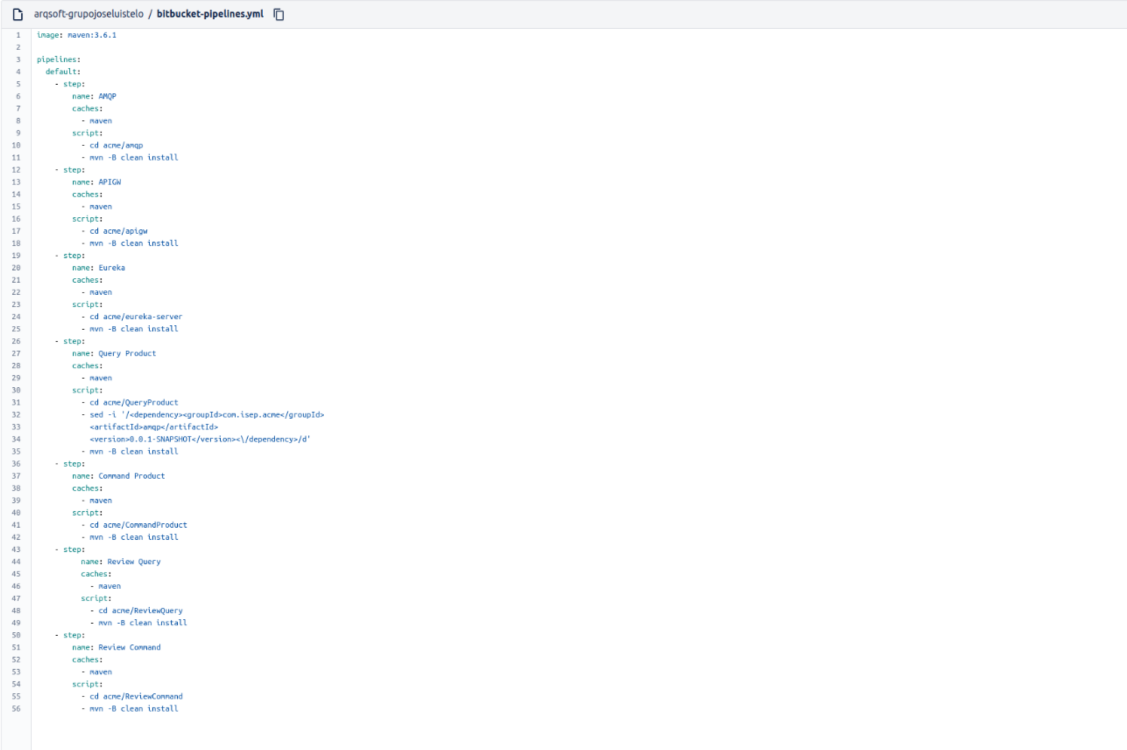


Fig. 6: Alternative microservices architecture based on HTTP requests with a Load Balancer to receive all requests.

CI/CD

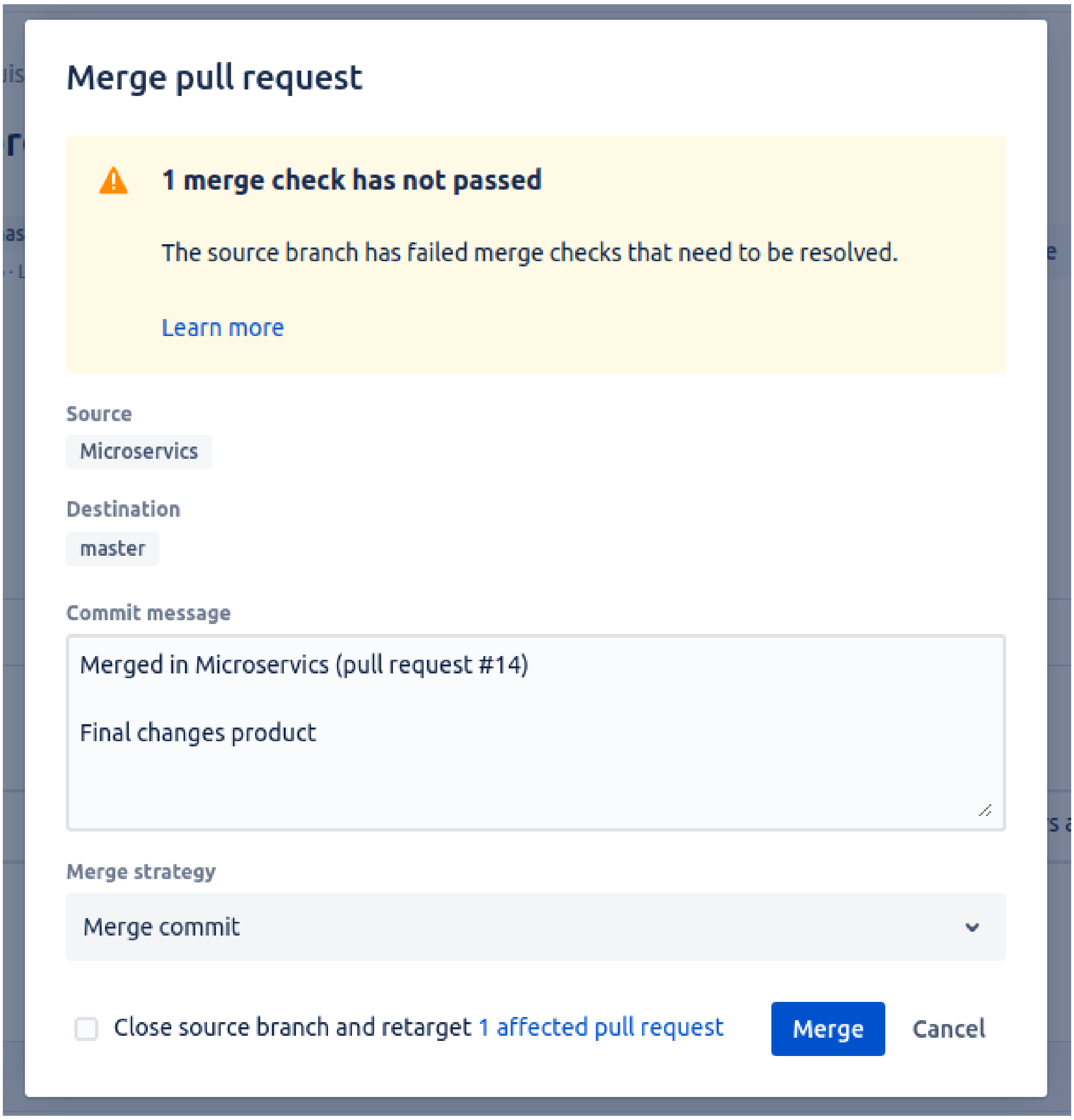


arqsoft-grupojoseluistelo / bitbucket-pipelines.yml

```yaml
image: maven:3.6.1

pipelines:
  default:
    - step:
        name: AMQP
        caches:
          - maven
        script:
          - cd acme/amqp
          - mvn -B clean install
    - step:
        name: APIGW
        caches:
          - maven
        script:
          - cd acme/apigw
          - mvn -B clean install
    - step:
        name: Eureka
        caches:
          - maven
        script:
          - cd acme/eureka-server
          - mvn -B clean install
    - step:
        name: Query Product
        caches:
          - maven
        script:
          - cd acme/QueryProduct
          - sed -i '/<dependency><groupId>com.isep.acme</groupId>
            <artifactId>amqp</artifactId>
            <version>0.0.1-SNAPSHOT</version><\/dependency>/d'
          - mvn -B clean install
    - step:
        name: Command Product
        caches:
          - maven
        script:
          - cd acme/CommandProduct
          - mvn -B clean install
    - step:
          name: Review Query
          caches:
            - maven
          script:
            - cd acme/ReviewQuery
            - mvn -B clean install
    - step:
        name: Review Command
        caches:
          - maven
        script:
          - cd acme/ReviewCommand
          - mvn -B clean install
```

Fig. 7: Bitbucket pipeline.



Merge pull request

1 merge check has not passed

The source branch has failed merge checks that need to be resolved.

Learn more

Source
Microservics

Destination
master

Commit message

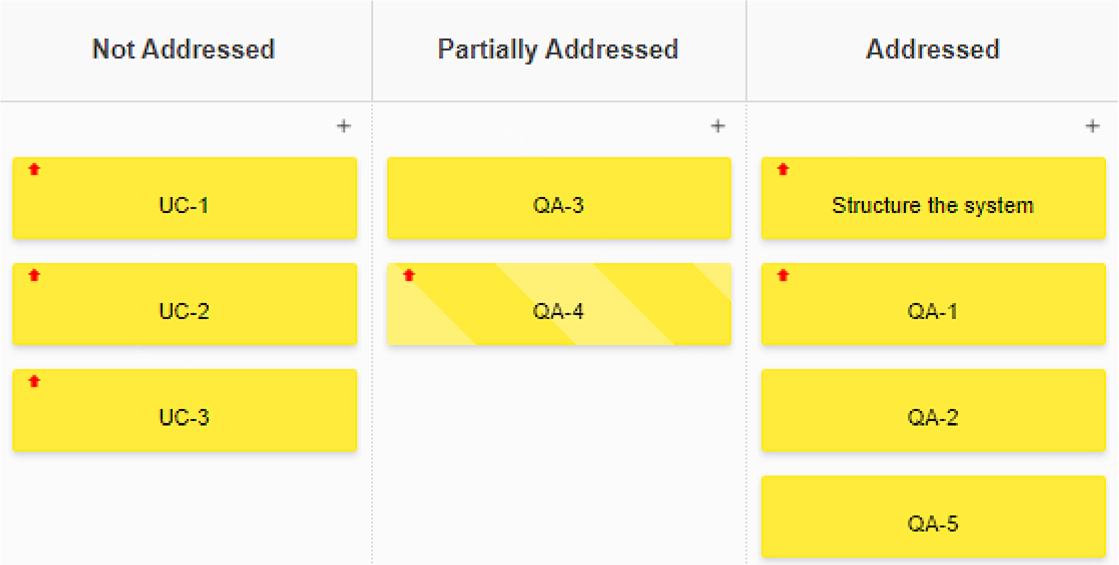
Merged in Microservics (pull request #14)

Final changes product

Merge strategy
Merge commit

□ Close source branch and retarget 1 affected pull request

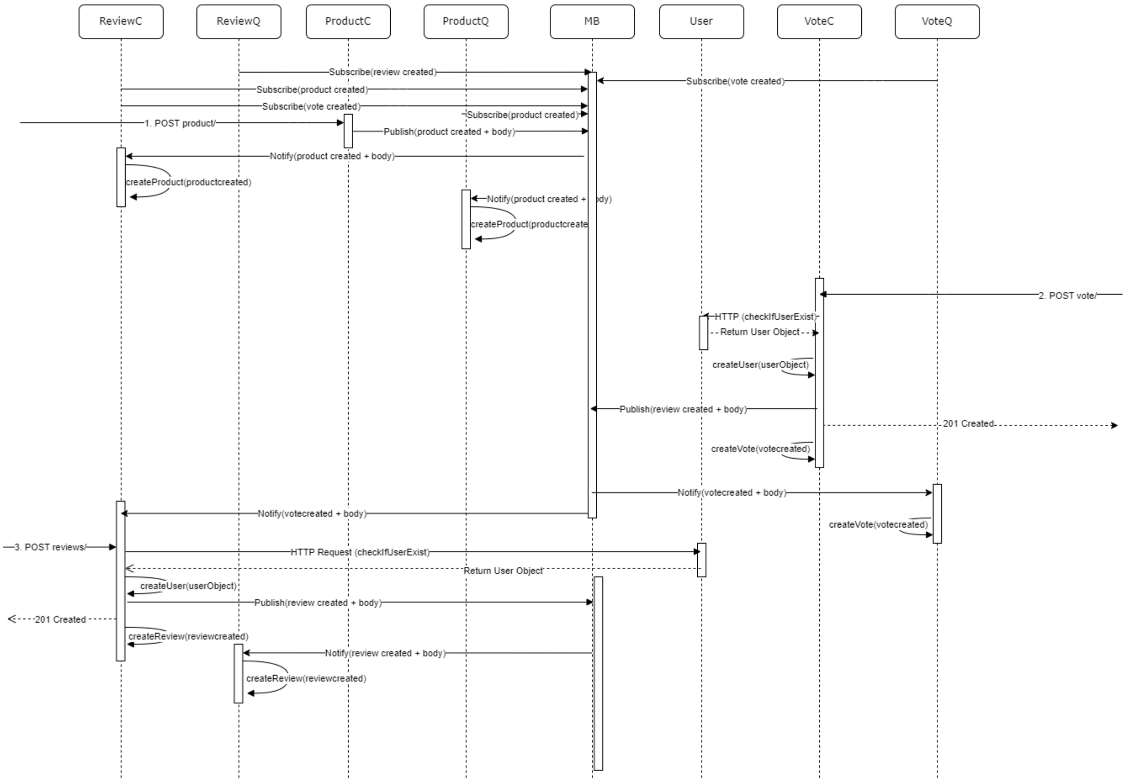
Fig. 8: Merge request.

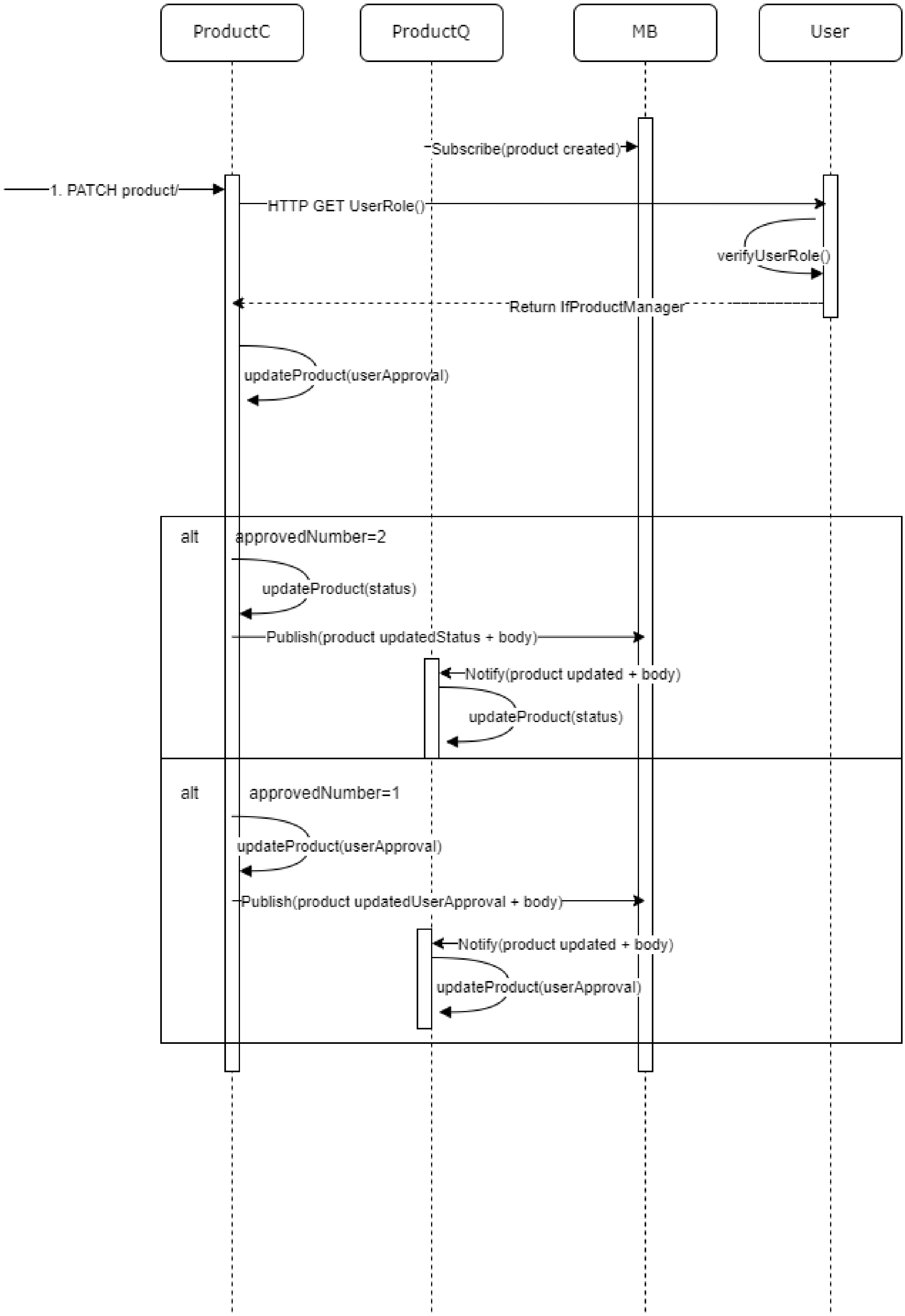


Not Addressed	Partially Addressed	Addressed
↑ UC-1	QA-3	↑ Structure the system
↑ UC-2	↑ QA-4	↑ QA-1
↑ UC-3		QA-2
		QA-5

Iteration 2

Goal – Address the system non-functional requirements and quality attributes


ReviewC
ReviewQ
ProductC
ProductQ
MB
User
VoteC
VoteQ
Subscribe(review created)
Subscribe(vote created)
Subscribe(product created)
Subscribe(vote created)
Subscribe(product created)
1. POST product/
Publish(product created + body)
Notify(product created + body)
createProduct(productcreated)
Notify(product created + body)
createProduct(productcreate
2. POST vote/
HTTP (checkIfUserExist)
Return User Object
createUser(userObject)
Publish(review created + body)
201 Created
createVote(votecreated)
Notify(votecreated + body)
Notify(votecreated + body)
createVote(votecreated)
3. POST reviews/
HTTP Request (checkIfUserExist)
Return User Object
createUser(userObject)
Publish(review created + body)
201 Created
createReview(reviewcreated)
Notify(review created + body)
createReview(reviewcreated)


ProductC
ProductQ
MB
User
Subscribe(product created)
1. PATCH product/
HTTP GET UserRole()
verifyUserRole()
Return IfProductManager
updateProduct(userApproval)
alt
approvedNumber=2
updateProduct(status)
Publish(product updatedStatus + body)
Notify(product updated + body)
updateProduct(status)
alt
approvedNumber=1
updateProduct(userApproval)
Publish(product updatedUserApproval + body)
Notify(product updated + body)
updateProduct(userApproval)

Messaging and Domain Events patterns

The "Messaging" and "Domain Events" patterns are fundamental components in microservices architectures that facilitate communication and coordination between independent services. In the context of an ACME project that transitioned from a monolith to microservices, these patterns were crucial for ensuring data consistency and seamless interaction between microservices. The project utilized RabbitMQ as a message broker to implement these patterns effectively.

Messaging Pattern:

The "Messaging" pattern involves the use of a message broker to enable communication between microservices in a loosely coupled manner. RabbitMQ, a widely used message broker, was employed to publish and consume messages between different microservices.

Domain Events Pattern:

The "Domain Events" pattern is based on the concept of events that represent meaningful occurrences within the domain of an application. These events capture changes or state transitions and are used to communicate these occurrences to other parts of the system.

Implementation in Microservices:

ProductCommand Microservice:

After the creation of a product, a "Product created" domain event was published to a RabbitMQ message queue. The message format included the serialized product object.

The ProductQuery Microservice subscribed to this queue, ensuring that the databases of both microservices remained synchronized. This allowed the read model in the ProductQuery microservice to be updated in real-time whenever a product was created or modified.

The ReviewCommand Microservice also subscribed to the "Product created" queue. This subscription ensured that the ReviewCommand microservice had access to the product information whenever it needed to associate a review with a product.

ReviewCommand Microservice:

After the creation of a review, a "Review created" domain event was published to another RabbitMQ message queue. The message format included the serialized review object.

The ReviewQuery Microservice subscribed to the "Review created" queue. This subscription enabled the ReviewQuery microservice to update its read model in response to new reviews being created.

Benefits and Considerations:

Loose Coupling:

The use of RabbitMQ and the "Messaging" pattern allowed for loose coupling between microservices. Microservices could publish and consume events without direct dependencies on one another.

Real-time Synchronization:

The "Domain Events" pattern ensured real-time synchronization of data between microservices. Events triggered by one microservice's actions were immediately communicated to other interested microservices.

Scalability:

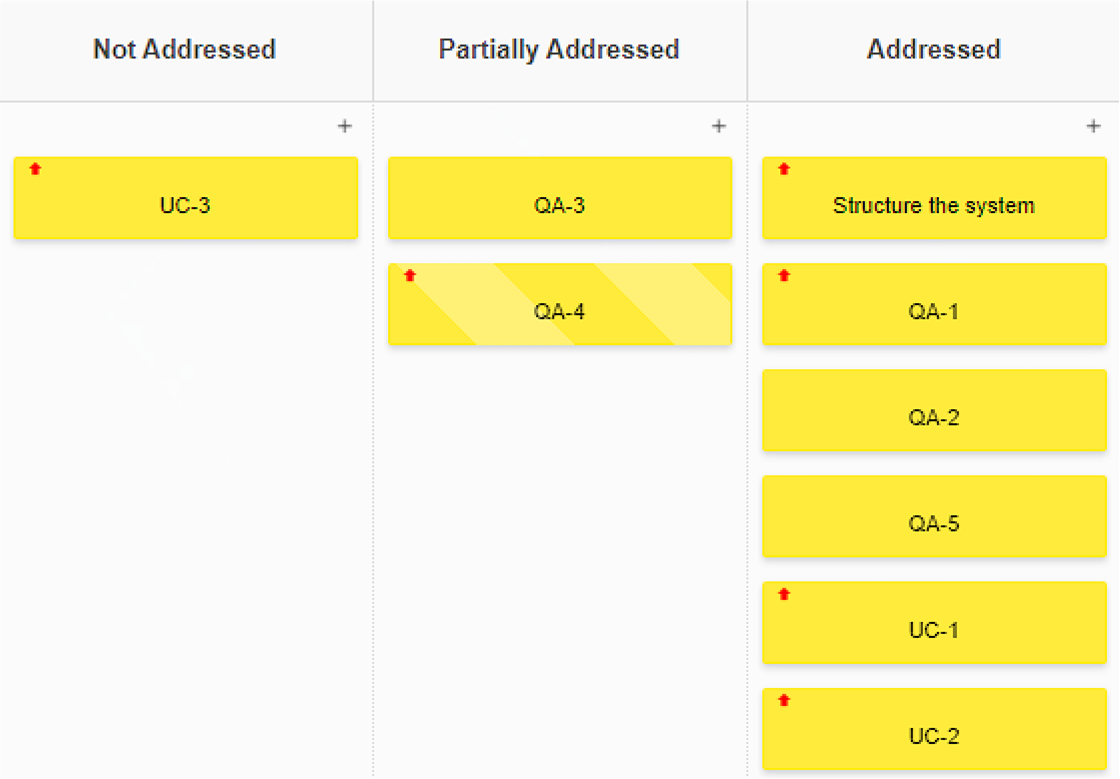
The asynchronous nature of messaging allowed for scalable and responsive communication. Microservices could continue processing messages independently, even during high loads.

Data Consistency:

By leveraging the "Domain Events" pattern, the project ensured that data consistency was maintained across microservices. Changes in one microservice triggered events that propagated to others, facilitating an eventual consistent state.

Selective Subscription:

Microservices could selectively subscribe to events based on their specific needs. This allowed for a tailored and efficient communication model.



Not Addressed	Partially Addressed	Addressed
UC-3	QA-3	Structure the system
	QA-4	QA-1
		QA-2
		QA-5
		UC-1
		UC-2

Iteration 3

Goal – Address the system functional requirements

Saga

In our project, the Saga Pattern played a pivotal role in ensuring the consistency of the voting process within the Vote microservice. The specific use case involved the creation of a vote, which, according to business requirements, should not be considered complete until a corresponding review was successfully created. This scenario introduced dependencies on other microservices, including the generation of messages and HTTP requests, making the entire process susceptible to potential failures.

Here's how the Saga Pattern was designed to address this complex workflow:

Vote Creation Step:

The saga initiates with the creation of a vote in the Vote microservice. However, this step alone does not mark the process as complete. Instead, it serves as the starting point of a series of coordinated actions.

Review Creation Step:

The subsequent step in the saga involves the creation of a review. This step is not performed directly within the Vote microservice but requires interactions with other microservices, such as sending messages or making HTTP requests.

Coordination and Compensation:

A coordinator, responsible for managing the saga, trackes the progress of these steps. If, at any point, the creation of the review fails or encounters an issue, the coordinator initiates a compensation mechanism.

Compensation Steps:

Compensation steps are designed to undo the effects of the previous steps in reverse order. For example, if the creation of a review fails after a vote is created, the compensation process should involve undoing the vote creation to maintain data consistency.

Handling Failures:

The Saga Pattern accommodates potential failures during the entire process. If any step fails, the coordinator triggers compensating transactions to roll back the changes and maintain a consistent state across microservices.

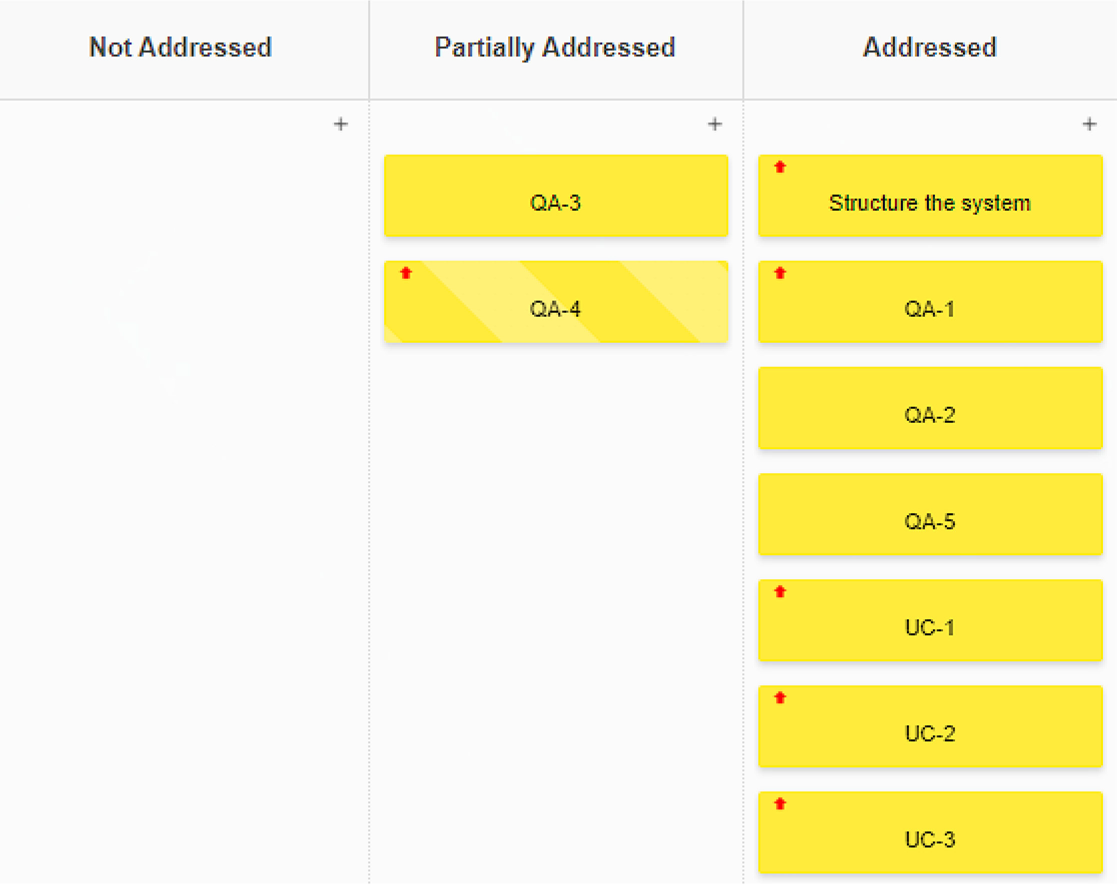
Eventual Consistency:

By using the Saga Pattern, the project should embrace the concept of eventual consistency. It acknowledges that, due to the distributed nature of microservices and the potential for failures, immediate consistency might not always be achievable. Instead, the system aims for a consistent state over time.

Isolation of Concerns:

The Saga Pattern allows for the isolation of concerns related to the voting and review creation process. Each microservice focuses on its specific responsibilities, and the coordination is managed by the saga coordinator.

By adopting the Saga Pattern for the Vote microservice, our project not only ensures data consistency but also improves the overall resilience of the system. This approach enables the handling of complex workflows involving interactions with multiple microservices and external dependencies while gracefully managing failures and maintaining a coherent state across the distributed environment.



Not Addressed	Partially Addressed	Addressed
	QA-3	Structure the system
	QA-4	QA-1
		QA-2
		QA-5
		UC-1
		UC-2
		UC-3