SA90 - Essay
Thuisbezorgd

Ana Băltărețu, Sebastian Deaconu, Radu Gaghi, Octav Pocola

April 11, 2023

Word count: 3600 (without appendices)

1 Introduction

Are you a foodie who loves the convenience of ordering your favorite dishes right to your doorstep? Then you're probably no stranger to the world of food delivery apps! Over the past few years, these apps have taken the world by storm, and their popularity has only skyrocketed since the pandemic hit [1]. However, with this massive growth has come a whole new set of challenges for companies operating in this space. Scaling up their platforms to handle the increased demand requires a robust architecture, which is no small feat [2].

Enter Thuisbezorgd - the focus of our latest project for the IN4315 Software Architecture course. We've chosen this app because it's recently undergone a major redesign from a model-view-controller architecture to a microservice architecture. This presents us with the perfect opportunity to learn more about the real-world design choices that go into creating a robust and scalable system. Our goals for this project are ambitious, but we're up for the challenge. We're excited to explore topics like asynchronous communication using an event bus, testing the performance of complex systems, and documenting and tracking our team's progress.

So, without further ado, let's dive into the state of our architecture! In this essay, we'll be discussing the stakeholders and their priorities, the context of the problem, and the challenges that arise when developing a platform like this. We'll also be presenting our design choices, as well as alternative solutions that we considered. Finally, we'll touch on the ethical aspects of the project that we've identified.

2 Problem analysis

Food delivery apps, like any other large-scale software application, can encounter various technical issues related to performance, scalability, security, and maintainability. To address these issues, architectural choices play a crucial role in the design and implementation of such apps. Developing a food delivery app, such as Thuisbezorgd, Uber Eats or DoorDash requires careful planning, testing, and optimization at every stage of the development process. In this Section, we reason about the technical issues that might appear when developing such software. Starting from our stakeholders and their specific needs and requirements, we then analyzed the common issues related to food delivery software, especially in the Thuisbezorgd architecture. After the problem analysis, the design process of Thuisbezorgd created more specific domain challenges related to the order and restaurant logic, which we solved by further decoupling our system and using a Kafka Event Bus [3].

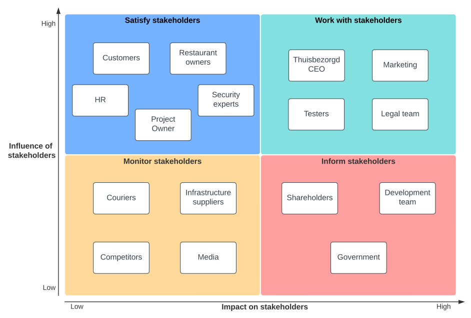


Figure 1: Stakeholders.

2.1 Who are we designing for?

To create our application's requirements, we must first determine our target audience. We conducted stakeholder analysis (Figure 1) and identified **Customers**, **Restaurant owners**, and **Couriers** as the primary users, since they are the ones mainly using the application. We also considered the input of other stakeholders, and implemented different measures for their concerns. The MoSCoW method [4] was used to prioritize requirements for the Development team (listed in Appendix A). We also recognized the importance of other stakeholders, and ensured that all of their interests and concerns were taken into account throughout the development process.

We considered the quality attributes that were important to each stakeholder in our project, including:

- **Customers:** we prioritized usability, performance, dependability, availability, and ease of support to ensure that our customers have a positive experience with the application.
- **Restaurant owners:** require us to focus on configurability, stability, and the cost of usage to meet the needs of restaurant owners and help them run their businesses more efficiently.
- **Couriers:** have a need for performance, usability, ease of support, and equality in the algorithms used to assign orders to ensure that couriers can deliver orders quickly and easily.
- **Project owner:** prioritizes scalability, reliability, recoverability, and elasticity to ensure that the project can grow and adapt to changing needs and requirements.
- **Security experts:** we took into consideration security, safety, confidentiality, integrity, authentication, authorization, and non-repudiation to ensure that the application is secure and meets regulatory and compliance requirements.
- **Development team:** we emphasized ease of integration and time-to-market with a reasonable plan that can be achieved within the required time frame, to help the development team work efficiently and deliver the product on time.

2.2 Problem context

Thuisbezorgd, one of the largest food delivery companies, had to face large-scale software issues related to scalability, security, and maintainability, firsthand. As they were gaining popularity and transitioning from a small start-up to a multinational service, the engineers of Thuisbezorgd discovered that their monolithic architecture was not able to keep up with the sudden increase in requests and users[1]. Besides scalability, the issue of maintainability arose as well: as the app evolved, it needed to be updated with new features and bug fixes. A monolithic architecture can make it challenging to maintain and update since changes to one part of the codebase can impact the entire application.

To address these issues, the architecture of Thuisbezorgd was changed to microservices. This allowed developers to handle high traffic by scaling only required services, instead of scaling the entire monolithic application. Additionally, message queues were used to handle high throughput between microservices, resulting in better performance and reliability. In terms of maintainability, in contrast to a monolithic architecture, microservices helped developers to make changes to one service without impacting the rest of the app. This approach made it easier to maintain and update the app, reducing downtime and improving the user experience. Moreover, the modularity of microservices allowed for easier integration of third-party software such as real-time tracking, and payment, which further improved the quality of the app.

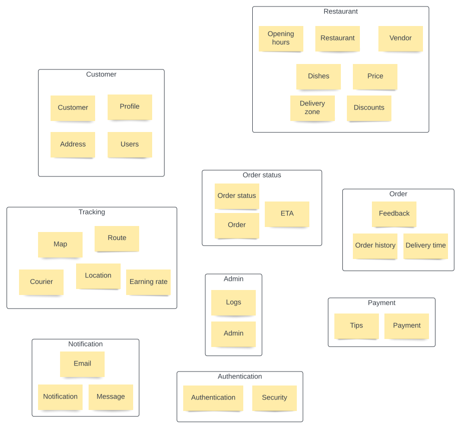


Figure 2: Coming up with the division between microservices through Keyword sorting.

[1]https://www.tarekbenguiza.nl/

2.3 Domain challenges

Architectural choices play a crucial role in addressing technical issues in food delivery apps. Because of the large scale of Thuisbezorgd, we decided to use the microservices architecture. This enabled us to mitigate the initial technical issues mentioned in the previous section which were related to performance, scalability, security, and maintainability.

In order to apply the microservices design paradigm we started by organizing services around domain-driven design (DDD) subdomains [5]. To do this we extracted domain-related keywords which we then sorted into 9 bounded contexts (Figure 2). Moreover, each context was mapped to a service, which has its own domain model. However, the use of microservices created two context challenges related to the high throughput of the system and its modularity.

The first challenge we had to address was in the order domain, more specifically, the order status. Because of the nature of our system, orders had to go through different microservices in order to validate and update their status (the order flow is shown in more detail in Figure 9). This enabled the order service to become a highly coupled "god service", by linking all synchronous communication and traffic to it and making the other services wait for its responses. Moreover, because of the high throughput of customer orders, latency became an issue as a huge amount of requests needed to be processed by the order service. The solution to this was to centralize communications related to orders on an asynchronous event bus, which we implemented using Kafka[3]. This design choice highly increased the decentralization and scalability potential of our system by letting the order-related services communicate asynchronously with each other. A further description of the Kafka event bus is found in Section 3.

The second challenge we encountered was present in the restaurant domain. Initially, we modeled all information flow related to restaurants into one microservice, separating its logic from the rest of the application. This proved to be a poor design choice as restaurants have two functionalities in food delivery apps. The first one is to provide a list of food items to customers and the second is to handle some of the ordering processes. Because of the high flow of requests and orders, the restaurant service became cluttered and slowed down on both of its functionalities. Moreover, customers need to see their food options as soon as possible, and having a slow-responding service was not an option. The solution was to split the restaurant service into two, based on its bounded contexts: a restaurant finder service, that that displays food items, and a restaurant manager service, that updates order events on the event bus. The two would then work independently on their specific domain which further uncluttered the application and improved cohesion alongside throughput.

Other issues that may arise, which we did not address directly can be related to security and integration. Food delivery apps need to handle sensitive user data, including payment information and delivery addresses. A microservices architecture can help with security by isolating sensitive data and applying different security protocols to different services. Additionally, a well-designed API gateway can provide a layer of security between the app and external services. Moreover, in terms of integration, external APIs will be used in the development of our system (e.g. payment, google maps). When working with such APIs, it's essential to avoid tight coupling with the rest of our code. If there is a direct dependence, any changes made to the API could break the code. This can be avoided by using the Dependency Inversion Principle (DIP)[6] which creates an abstraction layer that hides the details of the API and provides a common interface for the application to interact with. By applying this principle, the code becomes more flexible, extensible, and testable. It becomes easier to switch to a different API or implement new functionality without affecting the application logic.

3 Design

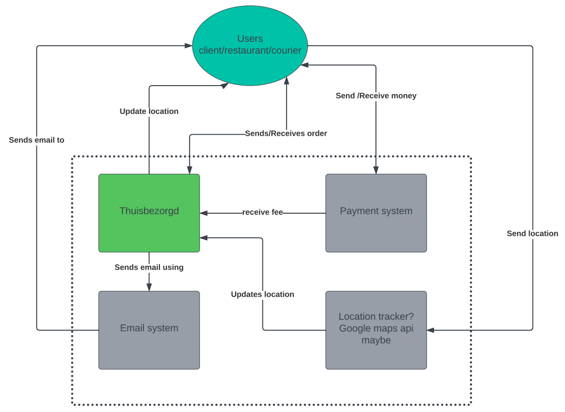


Figure 3: System context diagram.

Our solution is inspired by popular food delivery applications such as UberEats, DoorDash, and Thuisbezorgd, and it adopts the microservice paradigm. This architecture breaks down the application into small, autonomous services that are responsible for specific tasks. Each service is designed to work independently and communicates with other services through endpoints or the event bus. This approach offers several quality attributes, including scalability, performance, availability, and security [7], attributes that are relevant to our stakeholder (Section 2)

To provide a detailed view of the architecture, we present Figure 4. The Gateway design pattern [8] is used to route all incoming requests to the relevant microservice. This encapsulates the microservices from the user's perspective, allowing them to grow and adapt without affecting the client. Additionally, it reduces the attack surface, although it may become a bottleneck.

Adopting the microservice paradigm allows us to build a scalable and robust system that can handle the ever-increasing demands of our users. Each microservice can be developed, deployed, and scaled independently, enabling faster development cycles and reducing the risk of downtime. Furthermore, by isolating each service, we can reduce the impact of a failure in one component, ensuring that the entire system remains operational. Overall, the microservice architecture enables us to deliver a reliable and high-performance food delivery platform that meets the needs of our users.

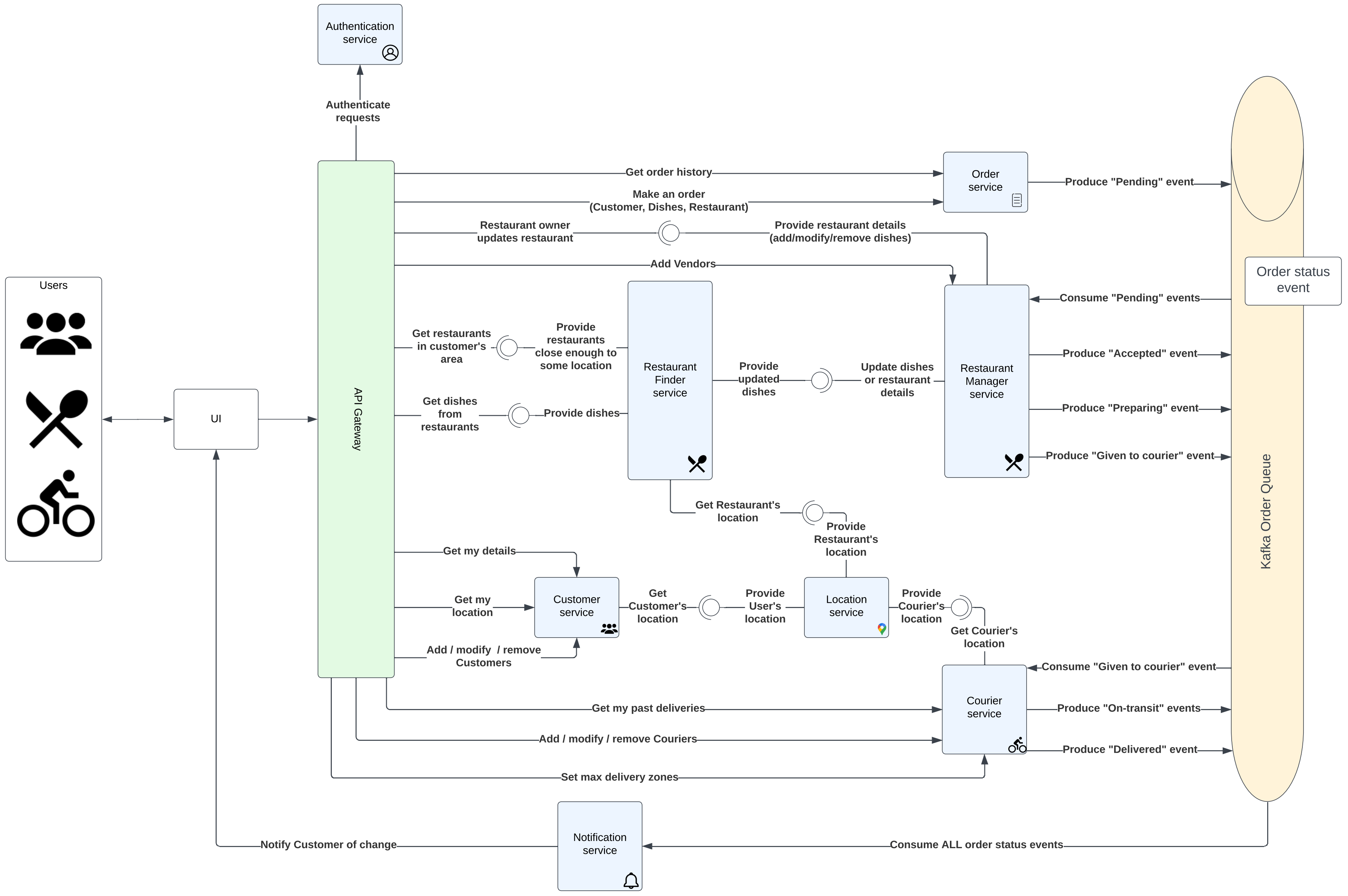

Authentication service
Authenticate requests
Users
UI
API Gateway
Get order history
Make an order (Customer, Dishes, Restaurant)
Order service
Produce "Pending" event
Restaurant owner updates restaurant
Provide restaurant details (add/modify/remove dishes)
Add Vendors
Order status event
Consume "Pending" events
Get restaurants in customer's area
Provide restaurants close enough to some location
Restaurant Finder service
Provide updated dishes
Update dishes or restaurant details
Restaurant Manager service
Produce "Accepted" event
Produce "Preparing" event
Produce "Given to courier" event
Get dishes from restaurants
Provide dishes
Get Restaurant's location
Provide Restaurant's location
Kafka Order Queue
Get my details
Get my location
Customer service
Get Customer's location
Provide User's location
Location service
Provide Courier's location
Get Courier's location
Add / modify / remove Customers
Consume "Given to courier" event
Courier service
Get my past deliveries
Produce "On-transit" events
Add / modify / remove Couriers
Produce "Delivered" event
Set max delivery zones
Notification service
Notify Customer of change
Consume ALL order status events

Figure 4: Component diagram

Communication between microservices is realized either through API endpoints or asynchronously through the Kafka Event Bus. The use of asynchronous communication decouples services and provides a scalable way of communication between microservices. Instead of waiting for responses from other services, a service can now subscribe to a certain event and listen specifically for it. Conversely, services can send out information by publishing an event to the bus. Events on the bus are not removed when consumed and can be seen by multiple microservices, which is much more efficient than sending out multiple messages for example.

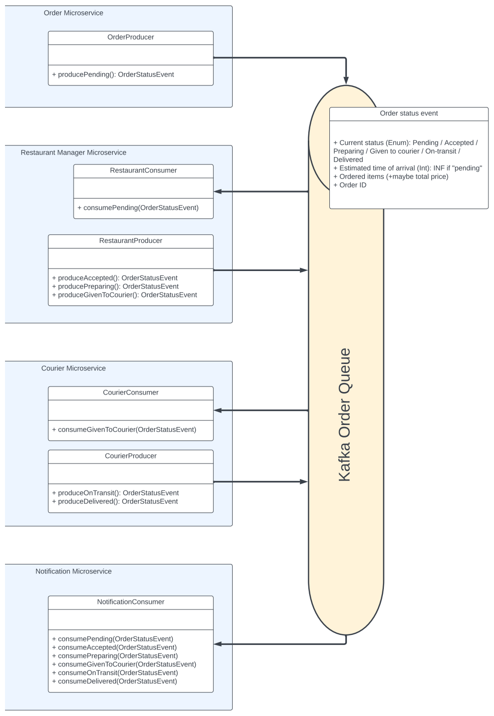

Order Microservice
OrderProducer
+ producePending(): OrderStatusEvent
Order status event
+ Current status (Enum): Pending / Accepted / Preparing / Given to courier / On-transit / Delivered
+ Estimated time of arrival (Int): INF if "pending"
+ Ordered items (+maybe total price)
+ Order ID
Restaurant Manager Microservice
RestaurantConsumer
+ consumePending(OrderStatusEvent)
RestaurantProducer
+ produceAccepted(): OrderStatusEvent
+ producePreparing(): OrderStatusEvent
+ produceGivenToCourier(): OrderStatusEvent
Courier Microservice
CourierConsumer
+ consumeGivenToCourier(OrderStatusEvent)
CourierProducer
+ produceOnTransit(): OrderStatusEvent
+ produceDelivered(): OrderStatusEvent
Kafka Order Queue
Notification Microservice
NotificationConsumer
+ consumePending(OrderStatusEvent)
+ consumeAccepted(OrderStatusEvent)
+ consumePreparing(OrderStatusEvent)
+ consumeGivenToCourier(OrderStatusEvent)
+ consumeOnTransit(OrderStatusEvent)
+ consumeDelivered(OrderStatusEvent)

Figure 5: UML of the Producers & Consumers of the Kafka Queue.

The architecture of the system can be best understood by following the process of ordering food from start to finish. Initially, the Restaurant Finder service provides customers with a list of restaurants and their dishes. After selecting an item, the Order service is notified, and an order is created. This generates an "Order Pending" event that is published on the event bus. The Restaurant Manager service then subscribes to this event, and based on its decision, either accepts or rejects the order, and posts the relevant event on the bus.

Next, the Courier service consumes these events and assigns a courier to pick up the food. Their location is updated through API requests made to the Location service. Once the courier delivers the food, and the customer pays, the "Delivered" event is published by the Courier service. All these events are subscribed to by the Notification service, which sends real-time notifications to the clients.

In addition to the above, the Customer, Restaurant Manager, and Courier services are responsible for handling CRUD operations when creating, modifying, or deleting account details. This covers the essential functionalities of the system. More information about the system's components can be found in Figure 4

3.1 Alternative solutions

Order Status microservice VS Kafka Queue

Currently, our architecture uses an event bus to handle order update events. However, communication between microservices could also be realized by sending messages asynchronously directly to each microservice. The order status microservice acts like a central hub to receive all the information regarding an order and update the status of an order, and then notify other services of this. This service seemed too coupled and too important, as our application would not be useful if the service would crash. The Order Status Service can be seen in Figure 6. A Kafka event bus (Figure 5) solves the need for this extra microservice while also improving scalability and reducing the need for direct communication between microservices.

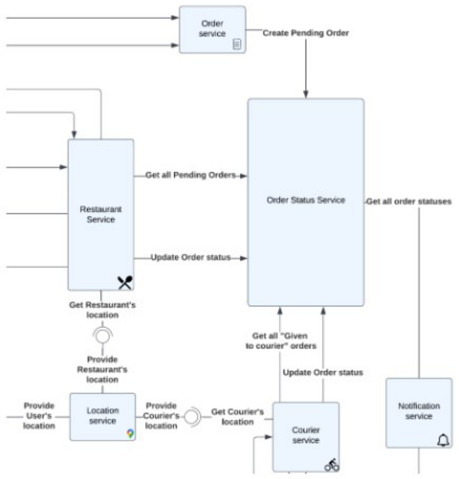


Figure 6: Problem with having a single restaurant service.

Restaurant service: one or more?

When analyzing the domain of the problem, it seemed natural to group all restaurant-related functionality in one microservice that would handle CRUD operations on restaurants and dishes, but also update the order status for every order at every restaurant, as seen in Figure 7. Our initial architecture relied heavily on this service and in order to prevent too much load on this critical component we decided to split it in two.

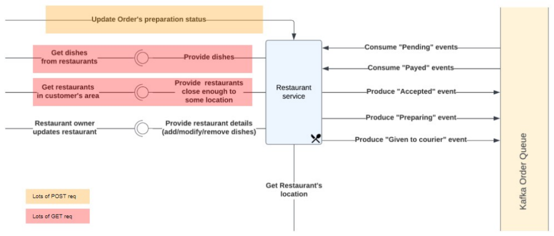

Update Order's preparation status
Get dishes from restaurants
Provide dishes
Get restaurants in customer's area
Provide restaurants close enough to some location
Restaurant owner updates restaurant
Provide restaurant details (add/modify/remove dishes)
Restaurant service
Consume "Pending" events
Consume "Payed" events
Produce "Accepted" event
Produce "Preparing" event
Produce "Given to courier" event
Get Restaurant's location
Lots of POST req
Lots of GET req
Kafka Order Queue

Figure 7: Problem with having a single restaurant service.

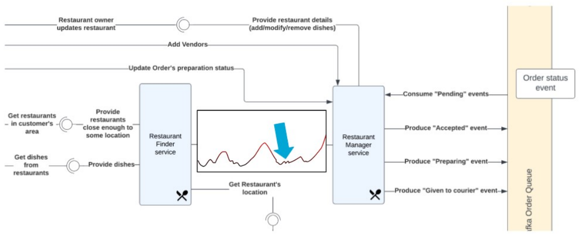

Restaurant owner updates restaurant
Provide restaurant details (add/modify/remove dishes)
Add Vendors
Update Order's preparation status
Order status event
Get restaurants in customer's area
Provide restaurants close enough to some location
Get dishes from restaurants
Provide dishes
Restaurant Finder service
Restaurant Manager service
Get Restaurant's location
Consume "Pending" events
Produce "Accepted" event
Produce "Preparing" event
Produce "Given to courier" event
ıfka Order Queue

Figure 8: Restaurant service split in 2 microservices.

The new component (Figure 8), the restaurant finder service handles all the read queries regarding restaurants and dishes, while the restaurant manager service can post and consume events from the bus. Restaurant owners can also update their information, and the restaurant finder will be notified by the restaurant manager service. While this method has the advantage of decoupling the restaurant manager service and increasing availability and fault tolerance, it adds another layer of complexity and reduces maintainability.

4 Proof of concept

The purpose of a proof of concept (POC) is to test the feasibility and viability of a new idea, technology, or product before investing significant time and resources into its development. A POC is typically a small-scale, experimental version of the final product or system, designed to demonstrate that the core features and functionality of the concept are achievable and effective. Overall, a successful POC can provide confidence in the viability of a concept and help to guide further development and investment decisions.

When deciding what to implement for our POC, this is exactly what we had in mind: to test the viability of our design with a small-scale version of our application, that could still fulfill its most important functions. To this end, we implemented the following microservices:

1. Order Service
2. Restaurant Manager Service
3. Courier Service
4. Notification Service

These services have API endpoints implemented and/or Kafka consumers or producers, as indicated in Figure 10. It is possible to simulate the entire ordering process, from creation to delivery. These services can be interacted with through the aforementioned API endpoints and logging statements expose the current status of an order or other events. Orders can be created and restaurants can accept them and update their status. Couriers can then be assigned and again the status of the order can be updated until it is delivered. An overview of the communication taking place on the message queue can be seen in Figure 9 Additionally, the notifications that would be seen by the users are logged by the notification service. The services not mentioned in the list above, such as the Location Service or the Customer Service are implemented as compilable interfaces and classes, but with no meaningful functionality. The 3 most meaningful microservices are also documented and tested, with an average line coverage of 95%. The final release of our POC can be found here

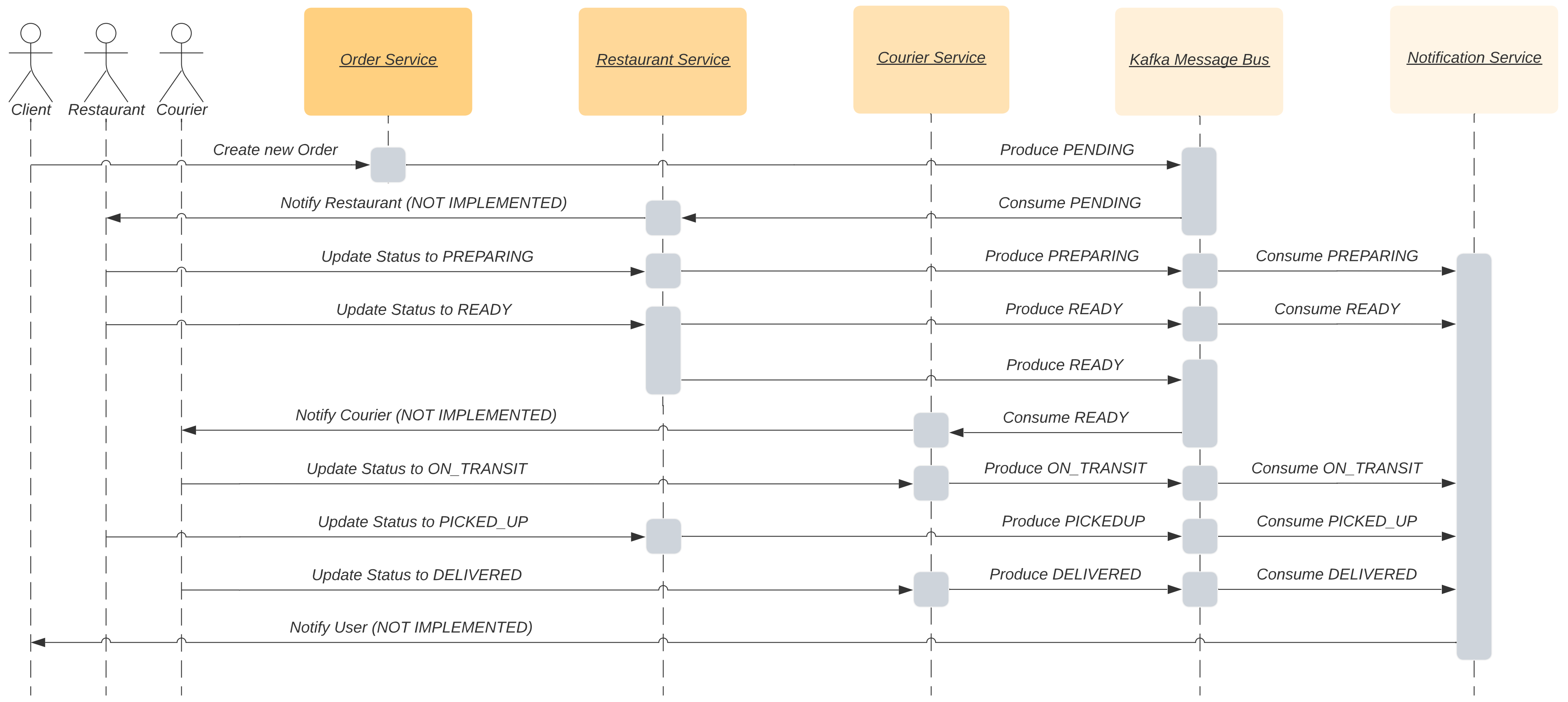


Figure 9: Sequence Diagram

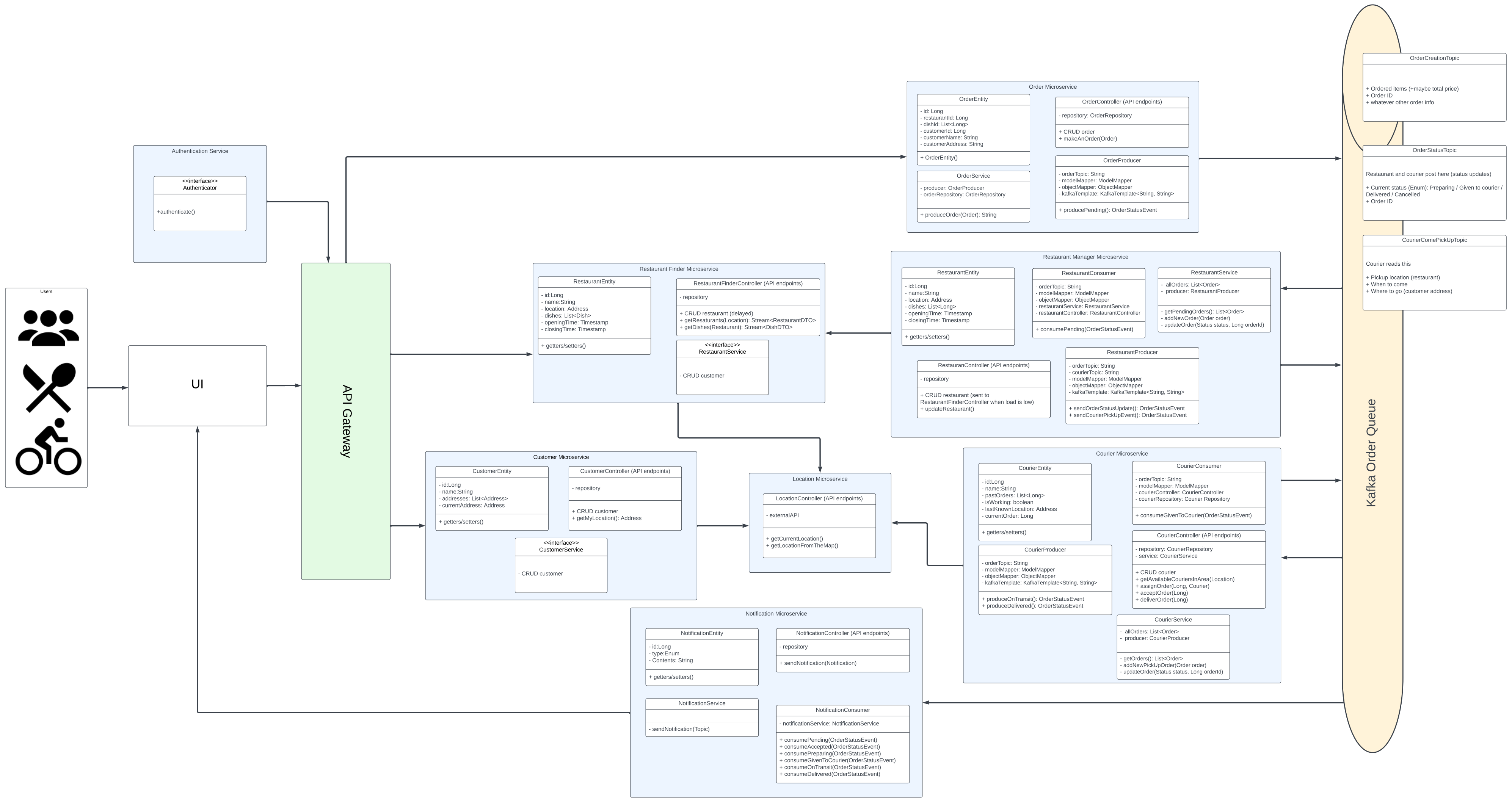

Users
UI
API Gateway
Authentication Service
<<interface>>
Authenticator
+authenticate()
Order Microservice
OrderEntity
- id: Long
- restaurantId: Long
- dishId: List<Long>
- customerId: Long
- customerName: String
- customerAddress: String
+ OrderEntity()
OrderController (API endpoints)
- repository: OrderRepository
+ CRUD order
+ makeAnOrder(Order)
OrderService
- producer: OrderProducer
- orderRepository: OrderRepository
+ produceOrder(Order): String
OrderProducer
- orderTopic: String
- modelMapper: ModelMapper
- objectMapper: ObjectMapper
- kafkaTemplate: KafkaTemplate<String, String>
+ producePending(): OrderStatusEvent
Restaurant Finder Microservice
RestaurantEntity
- id:Long
- name:String
- location: Address
- dishes: List<Dish>
- openingTime: Timestamp
- closingTime: Timestamp
+ getters/setters()
RestaurantFinderController (API endpoints)
- repository
+ CRUD restaurant (delayed)
+ getResaturants(Location): Stream<RestaurantDTO>
+ getDishes(Restaurant): Stream<DishDTO>
<<interface>>
RestaurantService
- CRUD customer
Restaurant Manager Microservice
RestaurantEntity
- id:Long
- name:String
- location: Address
- dishes: List<Long>
- openingTime: Timestamp
- closingTime: Timestamp
+ getters/setters()
RestaurantConsumer
- orderTopic: String
- modelMapper: ModelMapper
- objectMapper: ObjectMapper
- restaurantService: RestaurantService
- restaurantController: RestaurantController
+ consumePending(OrderStatusEvent)
RestaurantService
- allOrders: List<Order>
- producer: RestaurantProducer
- getPendingOrders(): List<Order>
- addNewOrder(Order order)
- updateOrder(Status status, Long orderId)
RestauranController (API endpoints)
- repository
+ CRUD restaurant (sent to RestaurantFinderController when load is low)
+ updateRestaurant()
RestaurantProducer
- orderTopic: String
- courierTopic: String
- modelMapper: ModelMapper
- objectMapper: ObjectMapper
- kafkaTemplate: KafkaTemplate<String, String>
+ sendOrderStatusUpdate(): OrderStatusEvent
+ sendCourierPickUpEvent(): OrderStatusEvent
Customer Microservice
CustomerEntity
- id:Long
- name:String
- addresses: List<Address>
- currentAddress: Address
+ getters/setters()
CustomerController (API endpoints)
- repository
+ CRUD customer
+ getMyLocation(): Address
<<interface>>
CustomerService
- CRUD customer
Location Microservice
LocationController (API endpoints)
- externalAPI
+ getCurrentLocation()
+ getLocationFromTheMap()
Courier Microservice
CourierEntity
- id:Long
- name:String
- pastOrders: List<Long>
- isWorking: boolean
- lastKnownLocation: Address
- currentOrder: Long
+ getters/setters()
CourierConsumer
- orderTopic: String
- modelMapper: ModelMapper
- courierController: CourierController
- courierRepository: Courier Repository
+ consumeGivenToCourier(OrderStatusEvent)
CourierProducer
- orderTopic: String
- modelMapper: ModelMapper
- objectMapper: ObjectMapper
- kafkaTemplate: KafkaTemplate<String, String>
+ produceOnTransit(): OrderStatusEvent
+ produceDelivered(): OrderStatusEvent
CourierController (API endpoints)
- repository: CourierRepository
- service: CourierService
+ CRUD courier
+ getAvailableCouriersInArea(Location)
+ assignOrder(Long, Courier)
+ acceptOrder(Long)
+ deliverOrder(Long)
CourierService
- allOrders: List<Order>
- producer: CourierProducer
- getOrders(): List<Order>
- addNewPickUpOrder(Order order)
- updateOrder(Status status, Long orderId)
Notification Microservice
NotificationEntity
- id:Long
- type:Enum
- Contents: String
+ getters/setters()
NotificationController (API endpoints)
- repository
+ sendNotification(Notification)
NotificationService
- sendNotification(Topic)
NotificationConsumer
- notificationService: NotificationService
+ consumePending(OrderStatusEvent)
+ consumeAccepted(OrderStatusEvent)
+ consumePreparing(OrderStatusEvent)
+ consumeGivenToCourier(OrderStatusEvent)
+ consumeOnTransit(OrderStatusEvent)
+ consumeDelivered(OrderStatusEvent)
Kafka Order Queue
OrderCreationTopic
+ Ordered items (+maybe total price)
+ Order ID
+ whatever other order info
OrderStatusTopic
Restaurant and courier post here (status updates)
+ Current status (Enum): Preparing / Given to courier / Delivered / Cancelled
+ Order ID
CourierComePickUpTopic
Courier reads this
+ Pickup location (restaurant)
+ When to come
+ Where to go (customer address)

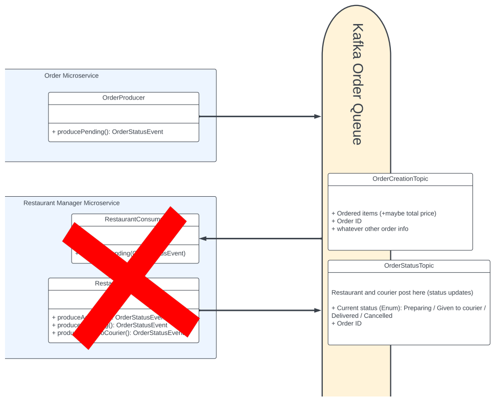
Figure 10: UML Diagram

5 Experiments

5.1 Stop-Restart

The stop-restart experiment aimed to test the system's ability to recover from a service outage and to continue processing orders without losing any data, and it involved two microservices: the restaurant service and the order service.

To perform the experiment, an order event was sent from the Order service, and then the restaurant service was intentionally stopped, but the order still remained on the Kafka Queue. This allowed the Order service to continue processing orders even when the restaurant service was stopped, unlike in a synchronous system. Once the restaurant service was restarted, because the order was still on the queue, the Restaurant service continued processing orders without loss of data.

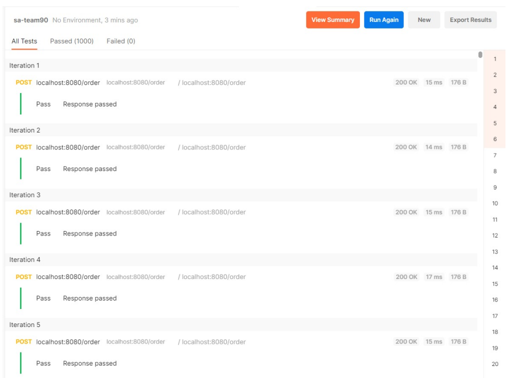


Order Microservice
OrderProducer
+ producePending(): OrderStatusEvent
Kafka Order Queue
Restaurant Manager Microservice
RestaurantConsum
nding(O usEvent)
Resta
+ produceA : OrderStatusEve
+ produce g(): OrderStatusEvent
+ produ oCourier(): OrderStatusEve
OrderCreationTopic
+ Ordered items (+maybe total price)
+ Order ID
+ whatever other order info
OrderStatusTopic
Restaurant and courier post here (status updates)
+ Current status (Enum): Preparing / Given to courier / Delivered / Cancelled
+ Order ID

Figure 11: Stopping the Restaurant microservice.

The experiment also showed the advantages of using asynchronous communication with Kafka over synchronous communication between the microservices. Asynchronous communication with Kafka decouples the microservices and provides a more resilient and scalable architecture for the system.

5.2 Performance



```
sa-team90  No Environment, 3 mins ago

All Tests   Passed (1000)   Failed (0)

Iteration 1
POST localhost:8080/order  localhost:8080/order  / localhost:8080/order   200 OK  15 ms  176 B
    Pass   Response passed
Iteration 2
POST localhost:8080/order  localhost:8080/order  / localhost:8080/order   200 OK  14 ms  176 B
    Pass   Response passed
Iteration 3
POST localhost:8080/order  localhost:8080/order  / localhost:8080/order   200 OK  15 ms  176 B
    Pass   Response passed
Iteration 4
POST localhost:8080/order  localhost:8080/order  / localhost:8080/order   200 OK  17 ms  176 B
    Pass   Response passed
Iteration 5
POST localhost:8080/order  localhost:8080/order  / localhost:8080/order   200 OK  15 ms  176 B
    Pass   Response passed
```

Figure 12: Results for the Performance experiment.

The performance experiment was conducted to determine the system's ability to handle a large volume of orders without crashing. The performance of the system was an important consideration when designing the architecture for the system.

To simulate restaurant owners and couriers, the RestaurantService and CourierService were modified to update the order status automatically every five seconds without any user requests. The Postman test feature was used to generate one thousand orders. The system's average response time was 6.9ms. The first five requests can be seen in Figure 12.

The results of the experiment suggest that the system has the potential to handle a high volume of orders. A synchronous system would have struggled with this amount of requests. However, it's important to note that the experiment was conducted in a controlled environment and with a limited amount of testable code. Therefore, the performance metrics should be taken as an initial indication of the system's capability and not necessarily reflective of real-world scenarios.

It's worth noting that Kafka provides several tools to monitor and test system performance. Unfortunately, the Kafka image used for this project did not include these tools, and porting the system to a production image proved more challenging than anticipated.

6 Ethics

Before releasing the application, it is important to take a look at some ethical consideration. This section is divided into subsections based on our stakeholders' different needs.

6.1 Customers

One ethical consideration related to our customers is ensuring that the price charged for delivery is reasonable. To address this, administrators of the application could adjust the pricing based on the income levels in different countries. Other concerns may include data privacy and security of our customers, as most food delivery applications collect a significant amount of data from their users, including location, payment information, and ordering history. It is important to ensure that this data is stored securely, that users' privacy is respected, and that GDPR [9] [10] is respected.

6.2 Couriers

An ethical consideration related to couriers is distributing the incoming orders to available drivers to provide them with an equal opportunity to earn money. To address this concern, an algorithm should be implemented that ensures fair distribution of orders among drivers. Another concern is about the fair treatment of couriers, i.e. making sure that they are not subjected to exploitative practices such as overly long working hours or inadequate compensation. A way to enforce this would be to limit a driver's allowed working hours each day, and implement a fixed percentage cut out of what the customer pays to use our service.

6.3 Restaurants

Ensuring fair competition is an important ethical consideration when it comes to the perspective of restaurants. The order in which restaurants are displayed to a customer can affect their decision on which restaurant to order from. To address this concern, we will implement a fair sorting mechanism for restaurants. For example, we could use customers' ratings to sort the restaurants. This would not only provide customers with more information to make their decisions, but it also incentivizes restaurants to perform better.

6.4 Environment

Food delivery can have a notable environmental impact, mainly because of the carbon emissions from delivery vehicles. To minimize this impact, the application could consider promoting the use of more eco-friendly delivery options, such as electric vehicles or bicycles. Additionally, an algorithm for "batch deliveries" could be implemented, where a driver's route is optimized to pick up multiple deliveries and distribute them to different customers efficiently, reducing the overall number of trips required.

7 Conclusion

In conclusion, our microservice-based Thuisbezorgd design was a success. Throughout the project, the team developed an in-depth understanding of asynchronous communication, which enabled us to design scalable, fault-tolerant microservices. We tested our design through experiments and analysis of the diagram, which demonstrated low coupling within microservices, and the ability to recover in the event of a failure. Although we did not test our performance in a production environment, we are confident in its potential based on our findings.

Our final goal was to improve team dynamics and decision-making, which we achieved through collaboration and effective communication. All design decisions were made together, after intense debates and meetings, with documentation on Gitlab, including retrospective meetings and notes.

The team is proud of our work on this project. We are confident in the scalability and fault-tolerance of our microservices design, which has great potential for future development. We have learned the value of effective communication and collaboration in achieving success and have made great strides in improving our team dynamics. The documentation and retrospective meetings were invaluable in identifying areas for improvement, and we are eager to apply what we've learned in future projects.

References

[1] K. Ahuja, V. Chandra, V. Lord, and C. Peens, "Ordering in: The rapid evolution of food delivery," *McKinsey & Company*, Sept. 2021.

[2] A. A. Ltd, "Software Architecture Robustness Analysis: A Case Study Approach," *UKDiss*, July 2022.

[3] N. Garg, *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[4] D. Tudor and G. A. Walter, "Using an agile approach in a large, traditional organization," in *AGILE 2006 (AGILE'06)*, pp. 7–pp, IEEE, 2006. MoSCoW method blog post: https://blog.ganttpro.com/en/prioritization-techniques-and-methods-for-projects-with-advantages-of-moscow-model/.

[5] E. Evans and E. J. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[6] M. Noback and M. Noback, "The dependency inversion principle," *Principles of Package Design: Creating Reusable Software Components*, pp. 67–104, 2018.

[7] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and Software Technology*, vol. 131, p. 106449, 2021.

[8] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," in *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*, SciTePress, 2018.

[9] "GDPR checklist for data controllers." [Online]. Available: `https://gdpr.eu/checklist/`.

[10] ICO, "GDPR." [Online]. Available: `https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/principles/storage-limitation/`.

A Requirements

A.1 Functional requirements

The functional requirements are organized based on the user roles that they relate to, those are:

1. Customer = person that wants to order food from a restaurant
2. Vendor = owner of the restaurant (+ for the device receiving orders in the restaurant)
3. Courier = person delivering the food
4. Admin = person taking care of the system (+ customer support)

A.1.1 Must haves

1. A User can log in / log out.
2. A Customer can set their (current) delivery address.
3. A Customer can see a list nearby restaurants.
4. A Customer can see the dishes offered by each restaurant (price, description).
5. A Customer can select items from a single restaurant to add to their order.
6. A Customer can see the total price of their order.
7. A Customer can place their order.
8. A Customer can track their order's status (accepted / preparing / on-transit / delivered).
9. A Customer can see an ETA (estimated time of arrival) for their order.
10. A Vendor can add/modify/remove dishes to/from their restaurant.
11. A Vendor can accept incoming orders.
12. A Vendor see what items they need to prepare for an order.
13. A Vendor can see how much they earn from an order.
14. A Vendor can indicate when they start different parts of the order fulfilment process (start preparation, order given to courier)
15. A Vendor can give a time estimation for when the order will be ready.
16. A Vendor can set the delivery zone for their restaurant.
17. A Courier can be assigned an order.
18. A Courier can see their route (from their location to pick up point to delivery location).
19. A Courier can see their location being updated while they are following the route.
20. A Courier indicate when they picked up the order.
21. A Courier can indicate when they have delivered an order.
22. A Courier can see how much they have earned from past deliveries.
23. An Admin can add new users to the system.
24. An Admin can change the type of the user (Customer/Courier/Vendor/Admin)
25. An Admin can set the (max) delivery zones.

A.1.2 Should haves

1. A User can sign up (create an account).
2. A User can get an email notification.
3. A Customer can specify when an order should be delivered (ASAP or at a later time - within restaurant working hours)
4. A Customer can select their payment method (Paypal / iDeal / pay at delivery/ etc.)
5. A Customer can track the location of their order on a map.
6. A Customer can see their past orders.
7. A Customer can rate past orders (and/or restaurants).
8. A Customer can give tips.
9. A Customer can search / filter the list of restaurants.
10. A Vendor can set opening hours for their restaurant.
11. A Vendor can see a history of orders that they have fulfilled.
12. A Courier can pick an order from the list of unassigned orders that they want to fulfill.
13. An Admin can view logs (payments / orders / system crashing).
14. An Admin can adjust the earning rate of Couriers (+ how much the Customer would need to pay for delivery).
15. An Admin can handle refunds.

A.1.3 Could haves

1. A User can see (and change) their own profile and preferences.
2. A Customer can add a home address (or saved addresses).
3. A Customer can view restaurants on a map.
4. A Customer can see images for the dishes offered by each restaurant.
5. A Customer can apply discounts to their order.
6. A Customer can give written feedback on their past orders.
7. A Customer can modify their selected dish (make a custom order).
8. A Customer can choose to pickup an order instead of having it delivered.
9. A Vendor can set-up the restaurant to only have their own couriers.
10. A Vendor can set their maximum delivery zone if they have their own couriers.
11. A Courier can see (a range of) how much they would earn from an order.
12. An admin can view the feedback left by the customer.

A.1.4 Won't haves

1. A User won't be able to have 2 different roles for 1 account (e.g: if a Customer is also working as a Courier they would need a separate account).
2. A Customer won't be able to add dishes from 2 different restaurants in the same order.
3. A Customer won't be able to cancel their order after it has been accepted by the restaurant.
4. A Customer won't be able to change their order (dishes / custom requests) after it has been accepted by the restaurant.
5. A Courier won't be able to handle more than 1 order at a time.
6. A Vendor won't be able to have multiple restaurants in the same account.

A.2 Non-Functional requirements

1. The system's backend programming language will be Java.
2. The system will be designed using a microservices architecture to promote modularity.
3. The system will provide security features, including authentication, authorization, and accountability (AAA).
4. The restaurant owner app will synchronize with the POS terminal in real-time.
5. The system will comply with GDPR regulations.
6. The system should be able to handle an average of 100 requests per second.
7. The system should be designed with consideration for color-blind and visually impaired users, and will undergo expert evaluation to ensure usability.
8. The system should be designed with consideration for older and technologically inexperienced users to ensure usability.
9. The system should have an uptime of at least 99% after launch.
10. The system should integrate with the Google Maps API.
11. The system's API endpoints should be clearly documented to ensure ease of use and further development.

B Accountability

B.1 Goals/learning objective

In this appendix we mention the team's original goals, an updated rubric, explaining how to assess the extent to which the team has reached its goals, as well as an actual assessment of the team's results in terms of the updated rubrics.

B.1.1 Initial goals

The original goals of the team were as follows:

1. How to design scalable systems with microservices and refine our ability to isolate each microservice as much as possible.
2. Research and get familiar with various concepts: methods of communication (asynchronous communication/message queues), elasticity in load balancing, working with serverless applications, sharding databases, and having multiple workers that handle requests.
3. To research concrete technologies such as Kafka or Eureka, relevant AWS / Google services.
4. How to measure the scalability of our application through various metrics, including failure rate, throughput, etc. Additionally, we strive to understand the techniques used for testing the scalability of our application.
5. To be more organized, to distribute our work evenly, and to make sure we take decisions together based on concrete arguments.

B.1.2 Updated goals

However, the initial goals shifted throughout the course of the project into the following:

1. How to design scalable systems with microservices and refine our ability to isolate each microservice as much as possible.
2. Research and get familiar with methods of communication such as asynchronous communication, and message queues
3. Research and use concrete technologies such as Kafka, that implement asynchronous communication.
4. How to measure the scalability of our application through metrics, including failure rate and, throughput. Additionally, we strive to understand these techniques used for testing the scalability of our application.
5. To be more organized, to distribute our work evenly and to make sure we take decisions together based on concrete arguments.

B.1.3 Goal assessment

To assess these goals we have devised the following rubric:

1. Assessment of the first goal is done through the quality of our architecture. This can be measured by analyzing our UML Diagrams regarding Coupling and Cohesion between microservices. The team has fully achieved this goal if its architecture has a low Coupling Between Microservices (CBM) value (lower than 2) and a low Lack of Cohesion in Events (LCE), which can be seen through the clear separation of services through bounded contexts, thus creating an event-driven architecture.

2. The quality of research done on asynchronous communication, message queues and more specifically Kafka can be assessed based on two criteria: firstly, on the extent of knowledge gathered, which we measured using research sheets(these should explain the key points and takeaway of the research topic in question) and secondly, through the feasibility of our implementation in regards to using message queues and Kafka. The team has achieved its goals if extensive and well-documented research sheets have been made and if the use of Kafka is feasible in regard to the POC.

3. Assessment of the scalability measurement goals is done through the quality of the experiments and the team's analysis regarding throughput and failure rate. The team has achieved this goal fully if an extensive scalability analysis, including multiple experiments and methods that were tested, was done.

4. The last goal can be measured by the team's GitLab wiki. This goal was achieved if the team devised an extensive wiki including the following:

   - notes during team meetings
   - weekly agendas
   - sprint retrospectives
   - other organizational documents such as a team plan, milestones etc.

B.1.4 Team's own assessment

Based on the rubric above, the team's assessment is the following:

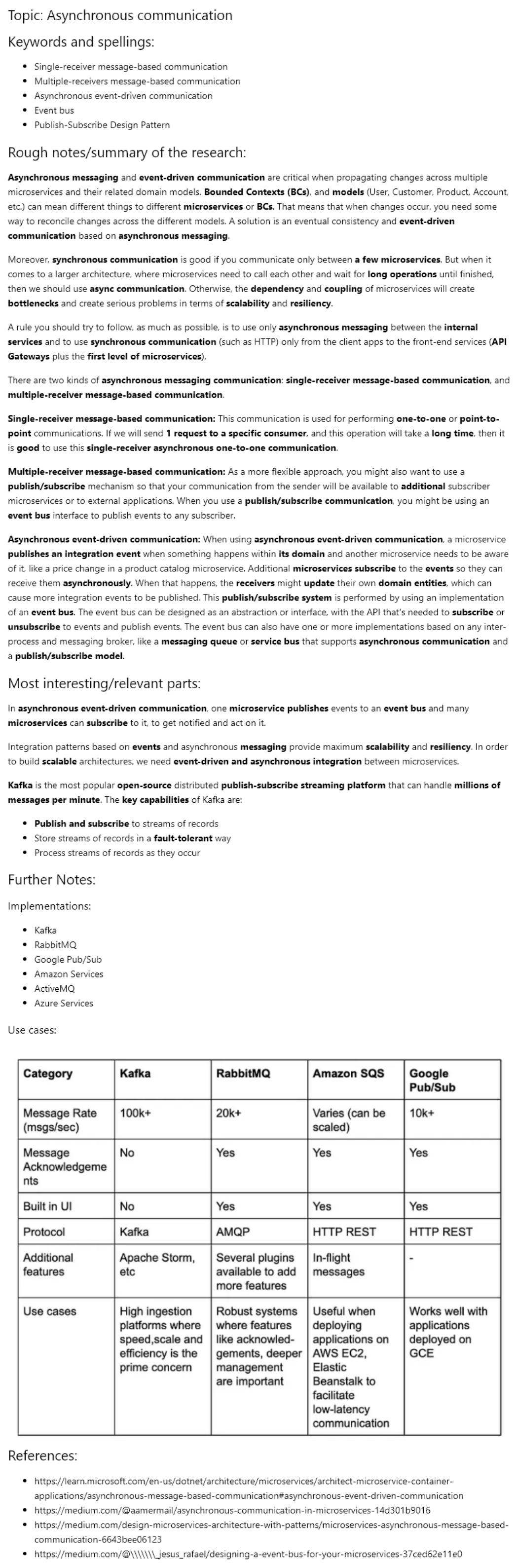
1. Regarding the goal of designing scalable systems with microservices, the team has achieved this fully by using an architecture with a low coupling factor(1.2) and a domain-driven design, which ensured high cohesion between events within a microservice and overall event-based communication.

2. The team has achieved the goal of researching asynchronous communication with message queues and Kafka as the devised research sheets(13 and 14) have extensive information, backed up by references, as well as key strengths and weaknesses for using this type of communication in comparison to a synchronous paradigm. Moreover, the use of Kafka aids the high load of food delivery apps as it decreases coupling, and ensures high loads with fault-tolerant message flows.

3. The team has partially achieved its goal in regard to the scalability measurements. This can be seen through the two experiments that have been done: stop-restart and throughput. The stop-restart experiment was successful, as it demonstrated that the architecture is resilient and that every other microservice can keep processing requests, even if a microservice was shut down. The use of a Kafka queue for messaging enables the system to maintain its resilience by ensuring that messages are not lost even in the event of service failures. This, in turn, allows the system to easily scale up by creating multiple instances of the same microservice as the workload and demand increase, while still maintaining high system performance and reliability. The performance experiment demonstrated that the system could handle a large volume of orders without crashing. However, the experiment was conducted in a controlled environment and with limited testable code, so the performance metrics should be taken as an initial indication of the system's capability, and not necessarily reflective of real-world scenarios. Note that a more advanced experiment was tried, one using Kafka internal tools, however, these tools were not available for the image used in this project, and porting the system to a production image proved more difficult than anticipated.

4. In terms of organizational skills, the team has achieved this goal as an extensive wiki was created. It includes an overview of the whole work process. It also documents team meetings and TA meetings. A weekly role rotation for chairs and note-takers was used in order to improve resource management. Moreover, weekly agendas and retrospectives were made to keep track of progress. Finally, important deadlines and deliverables were included in order to keep the team on track. Alongside these, design diagrams were also added to monitor the progress of the architectural development.

B.2 Research sheets



Topic: Asynchronous communication

Keywords and spellings:

- Single-receiver message-based communication
- Multiple-receivers message-based communication
- Asynchronous event-driven communication
- Event bus
- Publish-Subscribe Design Pattern

Rough notes/summary of the research:

Asynchronous messaging and **event-driven communication** are critical when propagating changes across multiple microservices and their related domain models. **Bounded Contexts (BCs)**, and **models** (User, Customer, Product, Account, etc.) can mean different things to different **microservices** or **BCs.** That means that when changes occur, you need some way to reconcile changes across the different models. A solution is an eventual consistency and **event-driven communication** based on **asynchronous messaging**.

Moreover, **synchronous communication** is good if you communicate only between **a few microservices**. But when it comes to a larger architecture, where microservices need to call each other and wait for **long operations** until finished, then we should use **async communication**. Otherwise, the **dependency** and **coupling** of microservices will create **bottlenecks** and create serious problems in terms of **scalability** and **resiliency**.

A rule you should try to follow, as much as possible, is to use only **asynchronous messaging** between the **internal services** and to use **synchronous communication** (such as HTTP) only from the client apps to the front-end services (**API Gateways** plus the **first level of microservices**).

There are two kinds of **asynchronous messaging communication**: **single-receiver message-based communication**, and **multiple-receiver message-based communication**.

Single-receiver message-based communication: This communication is used for performing **one-to-one** or **point-to-point** communications. If we will send **1 request to a specific consumer**, and this operation will take a **long time**, then it is **good** to use this **single-receiver asynchronous one-to-one communication**.

Multiple-receiver message-based communication: As a more flexible approach, you might also want to use a **publish/subscribe** mechanism so that your communication from the sender will be available to **additional** subscriber microservices or to external applications. When you use a **publish/subscribe communication**, you might be using an **event bus** interface to publish events to any subscriber.

Asynchronous event-driven communication: When using **asynchronous event-driven communication**, a microservice **publishes an integration event** when something happens within **its domain** and another microservice needs to be aware of it, like a price change in a product catalog microservice. Additional **microservices subscribe** to the **events** so they can receive them **asynchronously**. When that happens, the **receivers** might **update** their own **domain entities**, which can cause more integration events to be published. This **publish/subscribe system** is performed by using an implementation of an **event bus**. The event bus can be designed as an abstraction or interface, with the API that's needed to **subscribe** or **unsubscribe** to events and publish events. The event bus can also have one or more implementations based on any inter-process and messaging broker, like a **messaging queue** or **service bus** that supports **asynchronous communication** and a **publish/subscribe model**.

Most interesting/relevant parts:

In **asynchronous event-driven communication**, one **microservice publishes** events to an **event bus** and many **microservices** can **subscribe** to it, to get notified and act on it.

Integration patterns based on **events** and asynchronous **messaging** provide maximum **scalability** and **resiliency**. In order to build **scalable** architectures, we need **event-driven and asynchronous integration** between microservices.

Kafka is the most popular **open-source** distributed **publish-subscribe streaming platform** that can handle **millions of messages per minute**. The **key capabilities** of Kafka are:

- **Publish and subscribe** to streams of records
- Store streams of records in a **fault-tolerant** way
- Process streams of records as they occur

Further Notes:

Implementations:

- Kafka
- RabbitMQ
- Google Pub/Sub
- Amazon Services
- ActiveMQ
- Azure Services

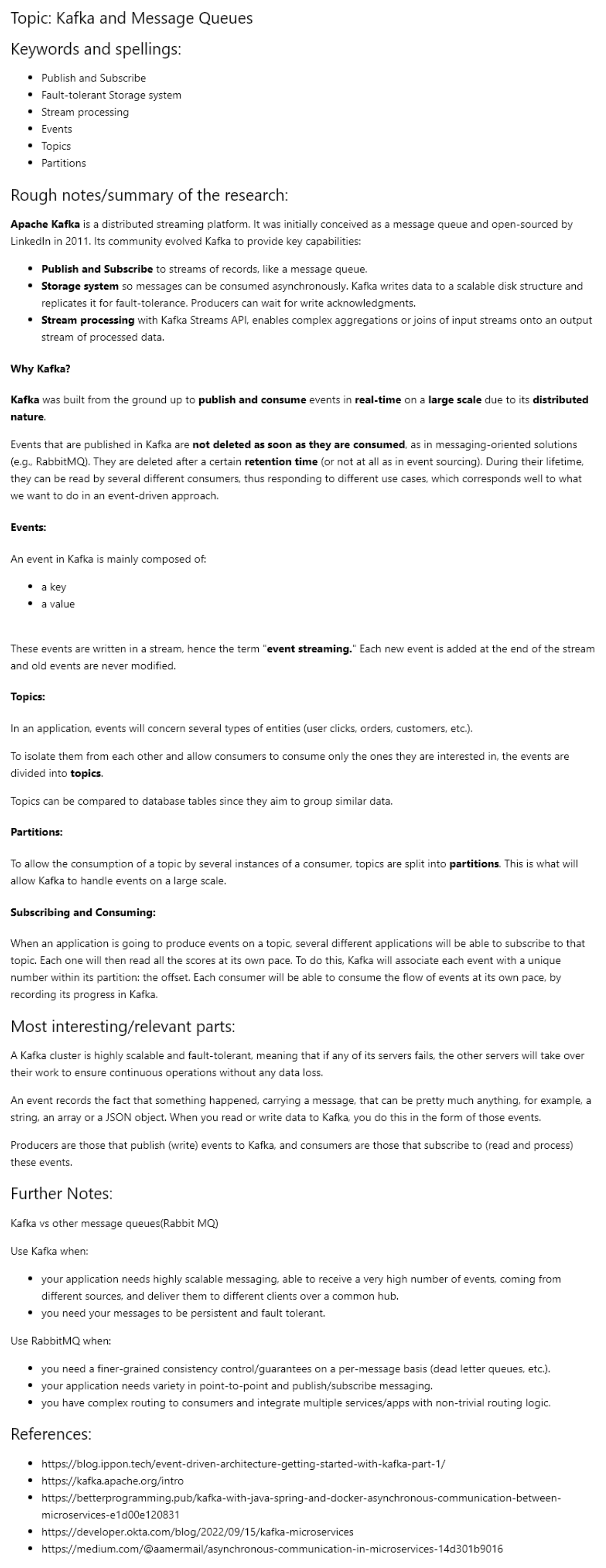
Use cases:

Category	**Kafka**	**RabbitMQ**	**Amazon SQS**	**Google Pub/Sub**
Message Rate (msgs/sec)	100k+	20k+	Varies (can be scaled)	10k+
Message Acknowledgements	No	Yes	Yes	Yes
Built in UI	No	Yes	Yes	Yes
Protocol	Kafka	AMQP	HTTP REST	HTTP REST
Additional features	Apache Storm, etc	Several plugins available to add more features	In-flight messages	-
Use cases	High ingestion platforms where speed,scale and efficiency is the prime concern	Robust systems where features like acknowled-gements, deeper management are important	Useful when deploying applications on AWS EC2, Elastic Beanstalk to facilitate low-latency communication	Works well with applications deployed on GCE

References:

- https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication#asynchronous-event-driven-communication
- https://medium.com/@aamermail/asynchronous-communication-in-microservices-14d301b9016
- https://medium.com/design-microservices-architecture-with-patterns/microservices-asynchronous-message-based-communication-6643bee06123
- https://medium.com/@\\\_jesus_rafael/designing-a-event-bus-for-your-microservices-37ced62e11e0

Figure 13: Research sheet: Asynchronous communication



Topic: Kafka and Message Queues

Keywords and spellings:

- Publish and Subscribe
- Fault-tolerant Storage system
- Stream processing
- Events
- Topics
- Partitions

Rough notes/summary of the research:

Apache Kafka is a distributed streaming platform. It was initially conceived as a message queue and open-sourced by LinkedIn in 2011. Its community evolved Kafka to provide key capabilities:

- **Publish and Subscribe** to streams of records, like a message queue.
- **Storage system** so messages can be consumed asynchronously. Kafka writes data to a scalable disk structure and replicates it for fault-tolerance. Producers can wait for write acknowledgments.
- **Stream processing** with Kafka Streams API, enables complex aggregations or joins of input streams onto an output stream of processed data.

Why Kafka?

Kafka was built from the ground up to **publish and consume** events in **real-time** on a **large scale** due to its **distributed nature**.

Events that are published in Kafka are **not deleted as soon as they are consumed**, as in messaging-oriented solutions (e.g., RabbitMQ). They are deleted after a certain **retention time** (or not at all as in event sourcing). During their lifetime, they can be read by several different consumers, thus responding to different use cases, which corresponds well to what we want to do in an event-driven approach.

Events:

An event in Kafka is mainly composed of:

- a key
- a value

These events are written in a stream, hence the term "**event streaming.**" Each new event is added at the end of the stream and old events are never modified.

Topics:

In an application, events will concern several types of entities (user clicks, orders, customers, etc.).

To isolate them from each other and allow consumers to consume only the ones they are interested in, the events are divided into **topics**.

Topics can be compared to database tables since they aim to group similar data.

Partitions:

To allow the consumption of a topic by several instances of a consumer, topics are split into **partitions**. This is what will allow Kafka to handle events on a large scale.

Subscribing and Consuming:

When an application is going to produce events on a topic, several different applications will be able to subscribe to that topic. Each one will then read all the scores at its own pace. To do this, Kafka will associate each event with a unique number within its partition: the offset. Each consumer will be able to consume the flow of events at its own pace, by recording its progress in Kafka.

Most interesting/relevant parts:

A Kafka cluster is highly scalable and fault-tolerant, meaning that if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

An event records the fact that something happened, carrying a message, that can be pretty much anything, for example, a string, an array or a JSON object. When you read or write data to Kafka, you do this in the form of those events.

Producers are those that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events.

Further Notes:

Kafka vs other message queues(Rabbit MQ)

Use Kafka when:

- your application needs highly scalable messaging, able to receive a very high number of events, coming from different sources, and deliver them to different clients over a common hub.
- you need your messages to be persistent and fault tolerant.

Use RabbitMQ when:

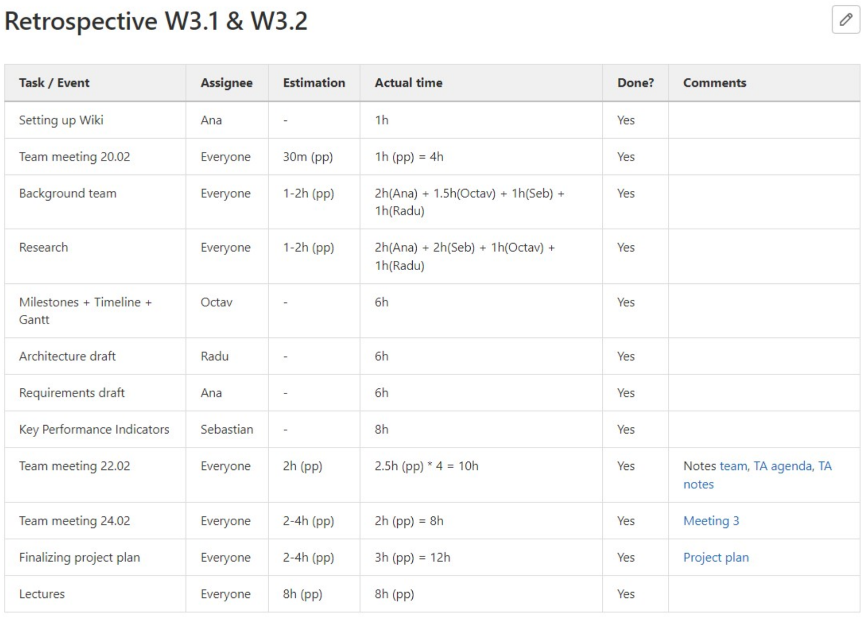
- you need a finer-grained consistency control/guarantees on a per-message basis (dead letter queues, etc.).
- your application needs variety in point-to-point and publish/subscribe messaging.
- you have complex routing to consumers and integrate multiple services/apps with non-trivial routing logic.

References:

- https://blog.ippon.tech/event-driven-architecture-getting-started-with-kafka-part-1/
- https://kafka.apache.org/intro
- https://betterprogramming.pub/kafka-with-java-spring-and-docker-asynchronous-communication-between-microservices-e1d00e120831
- https://developer.okta.com/blog/2022/09/15/kafka-microservices
- https://medium.com/@aamermail/asynchronous-communication-in-microservices-14d301b9016

Figure 14: Resarch sheet: Kafka and message queues

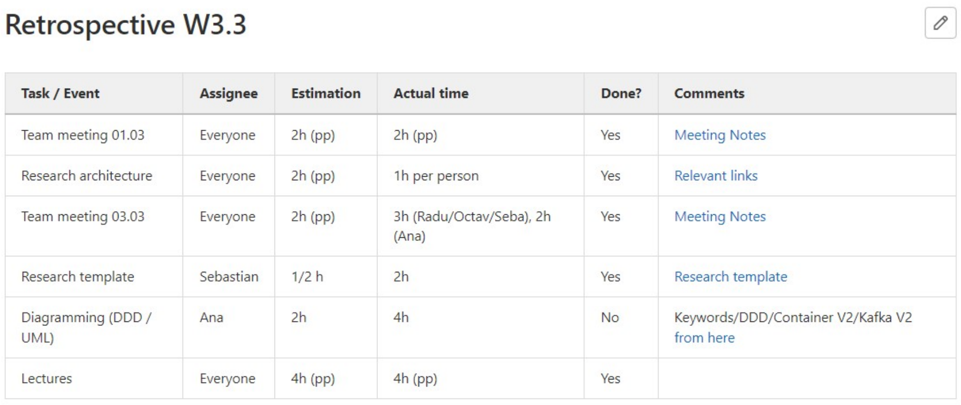
B.3 Sprint Retrospectives



Retrospective W3.1 & W3.2

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Setting up Wiki	Ana	-	1h	Yes	
Team meeting 20.02	Everyone	30m (pp)	1h (pp) = 4h	Yes	
Background team	Everyone	1-2h (pp)	2h(Ana) + 1.5h(Octav) + 1h(Seb) + 1h(Radu)	Yes	
Research	Everyone	1-2h (pp)	2h(Ana) + 2h(Seb) + 1h(Octav) + 1h(Radu)	Yes	
Milestones + Timeline + Gantt	Octav	-	6h	Yes	
Architecture draft	Radu	-	6h	Yes	
Requirements draft	Ana	-	6h	Yes	
Key Performance Indicators	Sebastian	-	8h	Yes	
Team meeting 22.02	Everyone	2h (pp)	2.5h (pp) * 4 = 10h	Yes	Notes team, TA agenda, TA notes
Team meeting 24.02	Everyone	2-4h (pp)	2h (pp) = 8h	Yes	Meeting 3
Finalizing project plan	Everyone	2-4h (pp)	3h (pp) = 12h	Yes	Project plan
Lectures	Everyone	8h (pp)	8h (pp)	Yes	

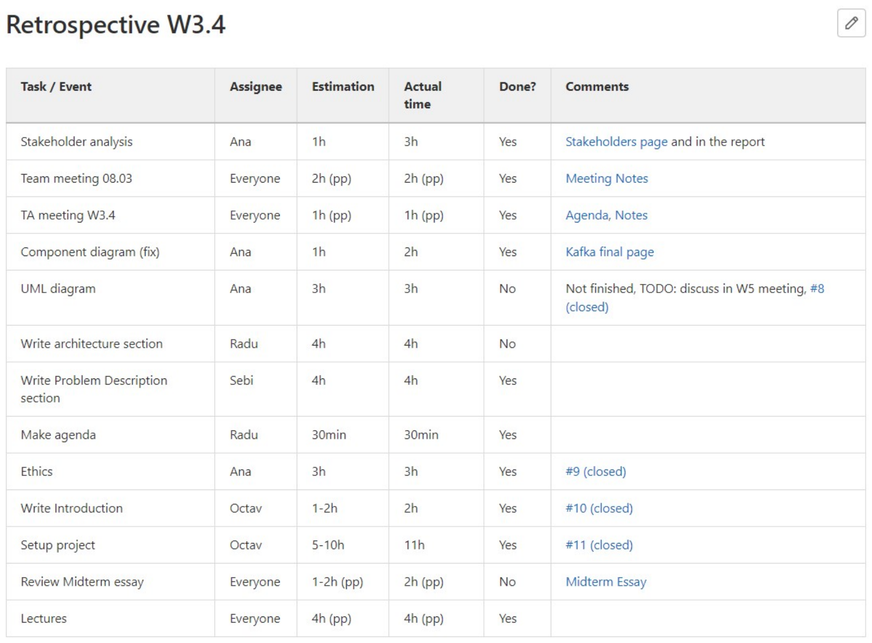
Figure 15: RetrospectiveW3.1-3.2



Retrospective W3.3

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Team meeting 01.03	Everyone	2h (pp)	2h (pp)	Yes	Meeting Notes
Research architecture	Everyone	2h (pp)	1h per person	Yes	Relevant links
Team meeting 03.03	Everyone	2h (pp)	3h (Radu/Octav/Seba), 2h (Ana)	Yes	Meeting Notes
Research template	Sebastian	1/2 h	2h	Yes	Research template
Diagramming (DDD / UML)	Ana	2h	4h	No	Keywords/DDD/Container V2/Kafka V2 from here
Lectures	Everyone	4h (pp)	4h (pp)	Yes	

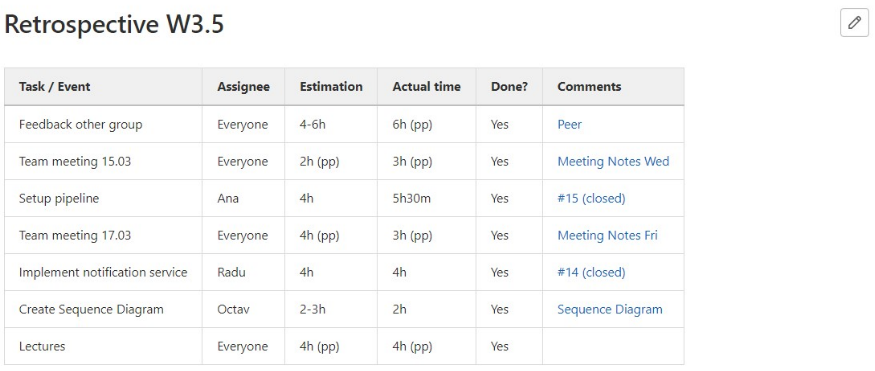
Figure 16: RetrospectiveW3.3



Retrospective W3.4

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Stakeholder analysis	Ana	1h	3h	Yes	Stakeholders page and in the report
Team meeting 08.03	Everyone	2h (pp)	2h (pp)	Yes	Meeting Notes
TA meeting W3.4	Everyone	1h (pp)	1h (pp)	Yes	Agenda, Notes
Component diagram (fix)	Ana	1h	2h	Yes	Kafka final page
UML diagram	Ana	3h	3h	No	Not finished, TODO: discuss in W5 meeting, #8 (closed)
Write architecture section	Radu	4h	4h	No	
Write Problem Description section	Sebi	4h	4h	Yes	
Make agenda	Radu	30min	30min	Yes	
Ethics	Ana	3h	3h	Yes	#9 (closed)
Write Introduction	Octav	1-2h	2h	Yes	#10 (closed)
Setup project	Octav	5-10h	11h	Yes	#11 (closed)
Review Midterm essay	Everyone	1-2h (pp)	2h (pp)	No	Midterm Essay
Lectures	Everyone	4h (pp)	4h (pp)	Yes	

Figure 17: RetrospectiveW3.4

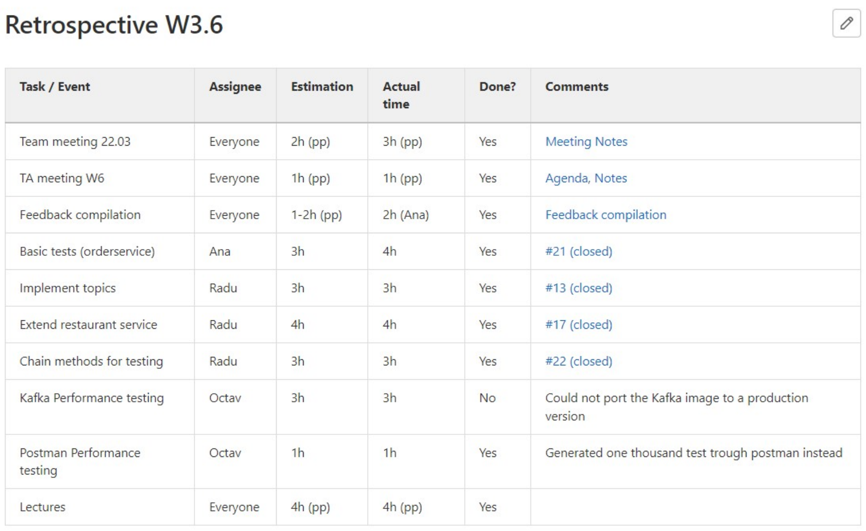


Retrospective W3.5

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Feedback other group	Everyone	4-6h	6h (pp)	Yes	Peer
Team meeting 15.03	Everyone	2h (pp)	3h (pp)	Yes	Meeting Notes Wed
Setup pipeline	Ana	4h	5h30m	Yes	#15 (closed)
Team meeting 17.03	Everyone	4h (pp)	3h (pp)	Yes	Meeting Notes Fri
Implement notification service	Radu	4h	4h	Yes	#14 (closed)
Create Sequence Diagram	Octav	2-3h	2h	Yes	Sequence Diagram
Lectures	Everyone	4h (pp)	4h (pp)	Yes	

Figure 18: RetrospectiveW3.5

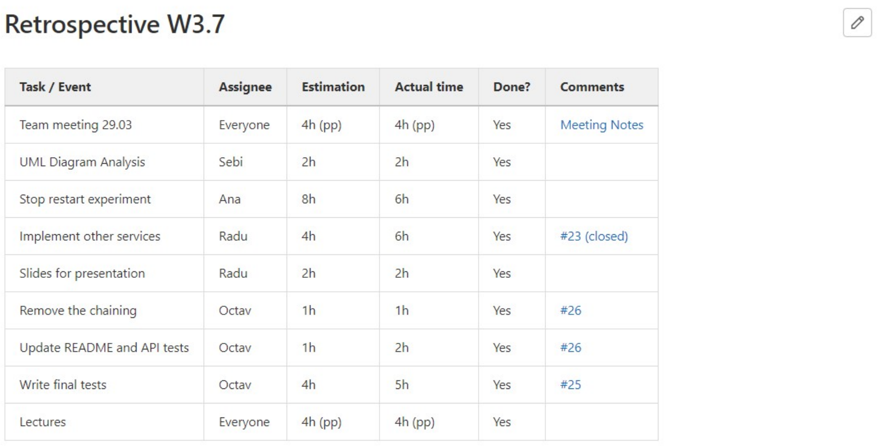
.



Retrospective W3.6

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Team meeting 22.03	Everyone	2h (pp)	3h (pp)	Yes	Meeting Notes
TA meeting W6	Everyone	1h (pp)	1h (pp)	Yes	Agenda, Notes
Feedback compilation	Everyone	1-2h (pp)	2h (Ana)	Yes	Feedback compilation
Basic tests (orderservice)	Ana	3h	4h	Yes	#21 (closed)
Implement topics	Radu	3h	3h	Yes	#13 (closed)
Extend restaurant service	Radu	4h	4h	Yes	#17 (closed)
Chain methods for testing	Radu	3h	3h	Yes	#22 (closed)
Kafka Performance testing	Octav	3h	3h	No	Could not port the Kafka image to a production version
Postman Performance testing	Octav	1h	1h	Yes	Generated one thousand test trough postman instead
Lectures	Everyone	4h (pp)	4h (pp)	Yes	

Figure 19: RetrospectiveW3.6

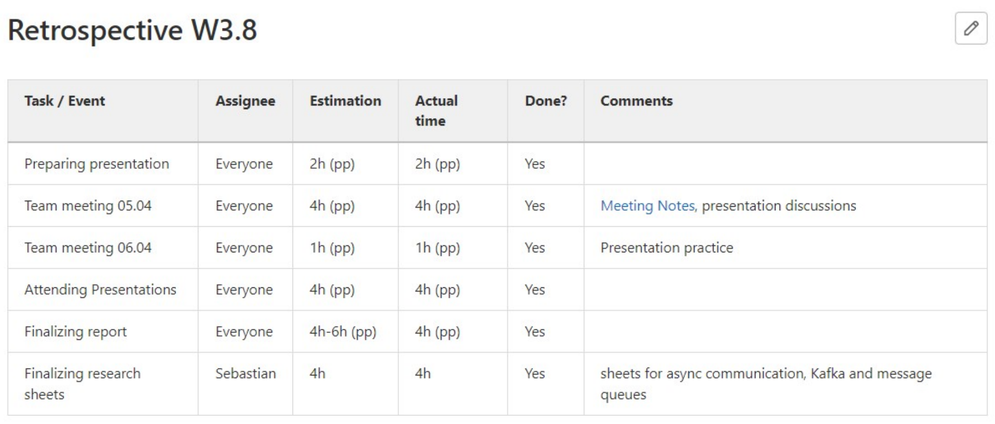


Retrospective W3.7

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Team meeting 29.03	Everyone	4h (pp)	4h (pp)	Yes	Meeting Notes
UML Diagram Analysis	Sebi	2h	2h	Yes	
Stop restart experiment	Ana	8h	6h	Yes	
Implement other services	Radu	4h	6h	Yes	#23 (closed)
Slides for presentation	Radu	2h	2h	Yes	
Remove the chaining	Octav	1h	1h	Yes	#26
Update README and API tests	Octav	1h	2h	Yes	#26
Write final tests	Octav	4h	5h	Yes	#25
Lectures	Everyone	4h (pp)	4h (pp)	Yes	

Figure 20: RetrospectiveW3.7

.



Retrospective W3.8

Task / Event	Assignee	Estimation	Actual time	Done?	Comments
Preparing presentation	Everyone	2h (pp)	2h (pp)	Yes	
Team meeting 05.04	Everyone	4h (pp)	4h (pp)	Yes	Meeting Notes, presentation discussions
Team meeting 06.04	Everyone	1h (pp)	1h (pp)	Yes	Presentation practice
Attending Presentations	Everyone	4h (pp)	4h (pp)	Yes	
Finalizing report	Everyone	4h-6h (pp)	4h (pp)	Yes	
Finalizing research sheets	Sebastian	4h	4h	Yes	sheets for async communication, Kafka and message queues

Figure 21: RetrospectiveW3.8

B.4 Other schemas

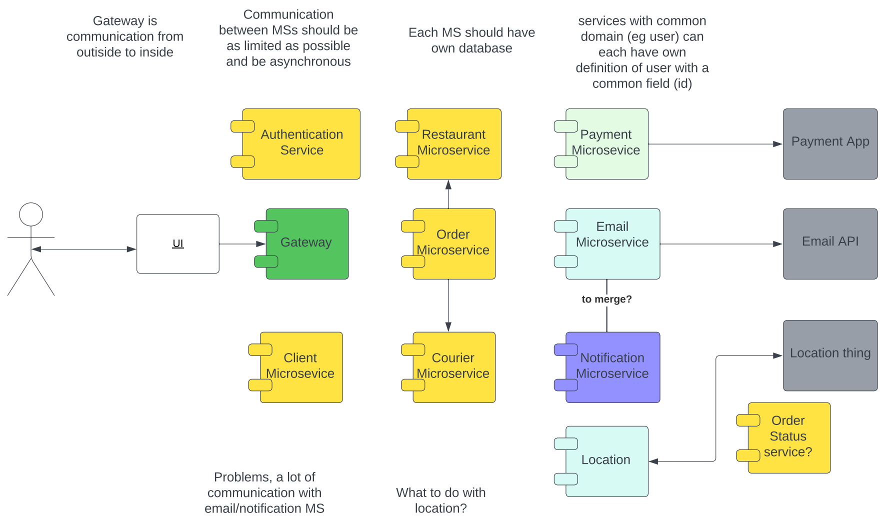

Gateway is communication from outside to inside
Communication between MSs should be as limited as possible and be asynchronous
Each MS should have own database
services with common domain (eg user) can each have own definition of user with a common field (id)
Authentication Service
Restaurant Microservice
Payment Microsevice
Payment App
UI
Gateway
Order Microservice
Email Microservice
Email API
to merge?
Client Microsevice
Courier Microservice
Notification Microservice
Location thing
Order Status service?
Location
Problems, a lot of communication with email/notification MS
What to do with location?

Figure 22: ContainerV1

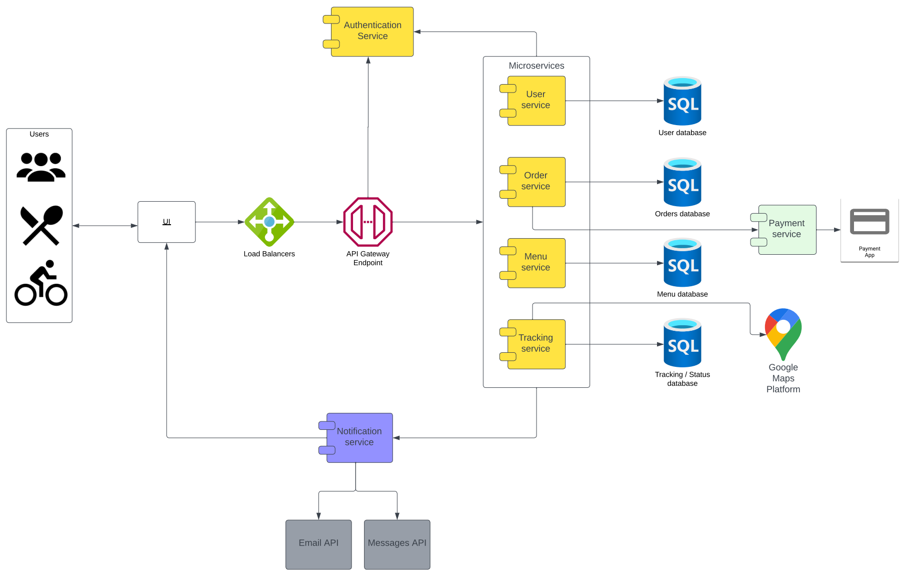


Figure 23: ContainerV2

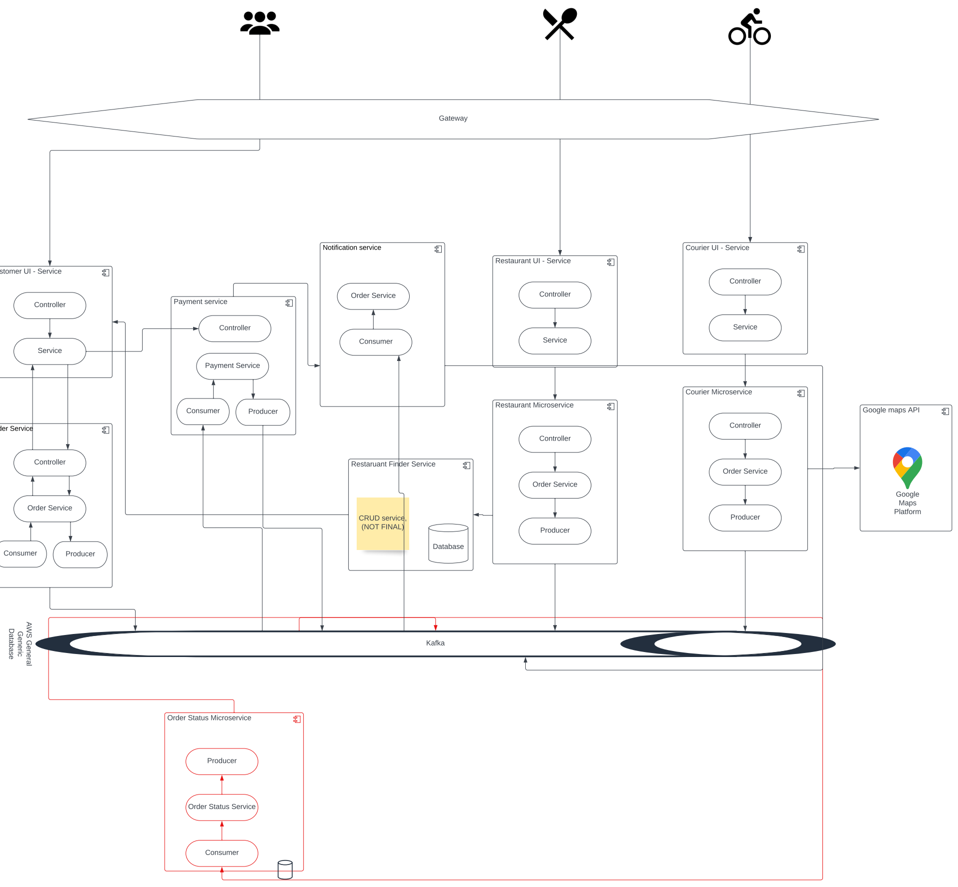
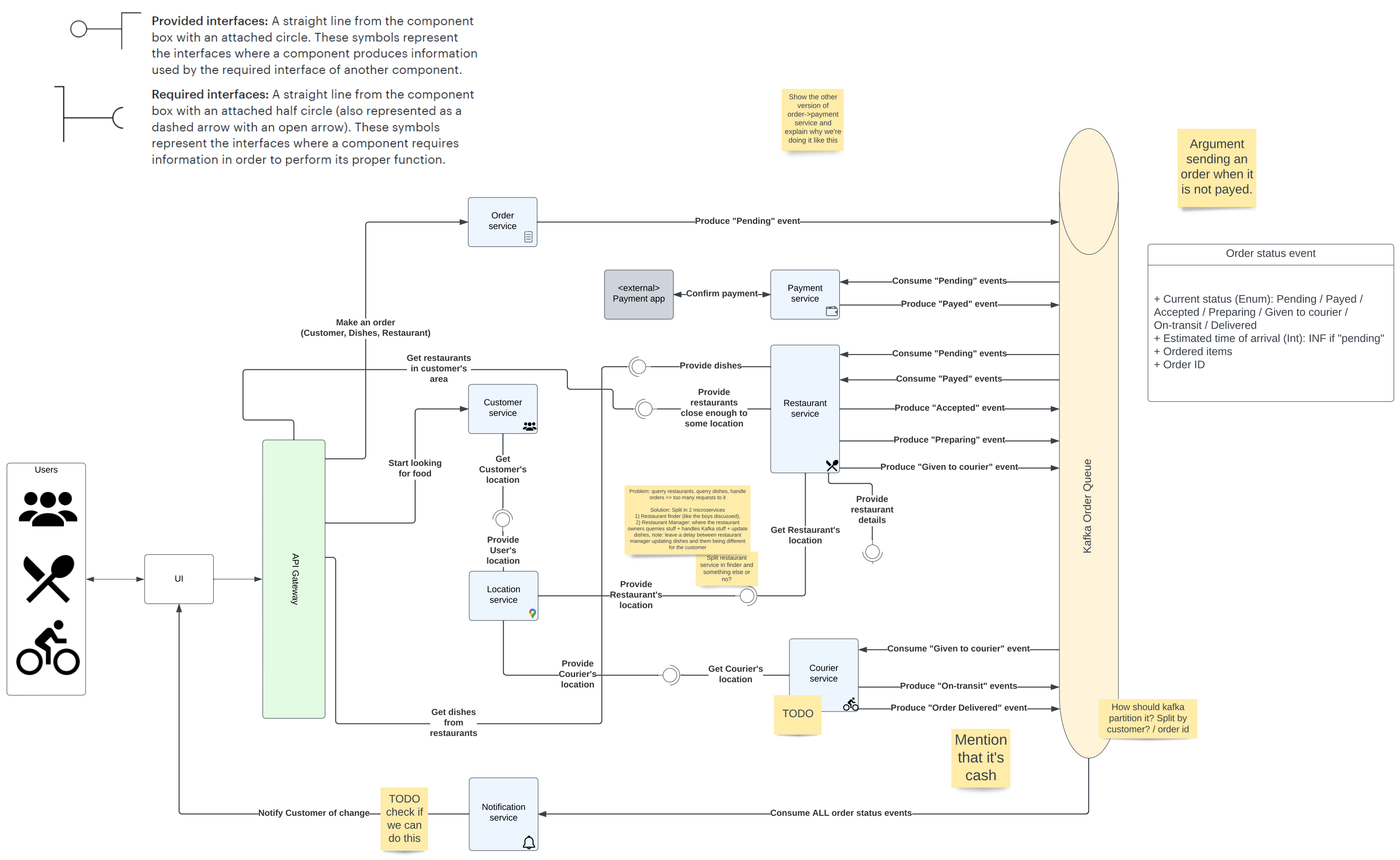


Figure 24: KafkaV1



Provided interfaces: A straight line from the component box with an attached circle. These symbols represent the interfaces where a component produces information used by the required interface of another component.

Required interfaces: A straight line from the component box with an attached half circle (also represented as a dashed arrow with an open arrow). These symbols represent the interfaces where a component requires information in order to perform its proper function.

Show the other version of order->payment service and explain why we're doing it like this

Argument sending an order when it is not payed.

Order status event

+ Current status (Enum): Pending / Payed / Accepted / Preparing / Given to courier / On-transit / Delivered
+ Estimated time of arrival (Int): INF if "pending"
+ Ordered items
+ Order ID

Problem: querry restaurants, querry dishes, handle orders => too many requests to it

Solution: Split in 2 microservices
1) Restaurant finder (like the boys discussed),
2) Restaurant Manager: where the restaurant owners querries stuff + handles Kafka stuff + update dishes, note: leave a delay between restaurant manager updating dishes and them being different for the customer

Split restaurant service in finder and something else or no?

How should kafka partition it? Split by customer? / order id

Mention that it's cash

TODO check if we can do this

Order service
Produce "Pending" event
<external> Payment app
Confirm payment
Payment service
Consume "Pending" events
Produce "Payed" event
Make an order (Customer, Dishes, Restaurant)
Get restaurants in customer's area
Provide dishes
Provide restaurants close enough to some location
Restaurant service
Consume "Pending" events
Consume "Payed" events
Produce "Accepted" event
Produce "Preparing" event
Produce "Given to courier" event
Customer service
Start looking for food
Get Customer's location
Provide User's location
Provide restaurant details
Get Restaurant's location
Users
UI
API Gateway
Location service
Provide Restaurant's location
Kafka Order Queue
Provide Courier's location
Get Courier's location
Courier service
Consume "Given to courier" event
Produce "On-transit" events
Produce "Order Delivered" event
TODO
Get dishes from restaurants
Notify Customer of change
Notification service
Consume ALL order status events

Figure 25: KafkaV2

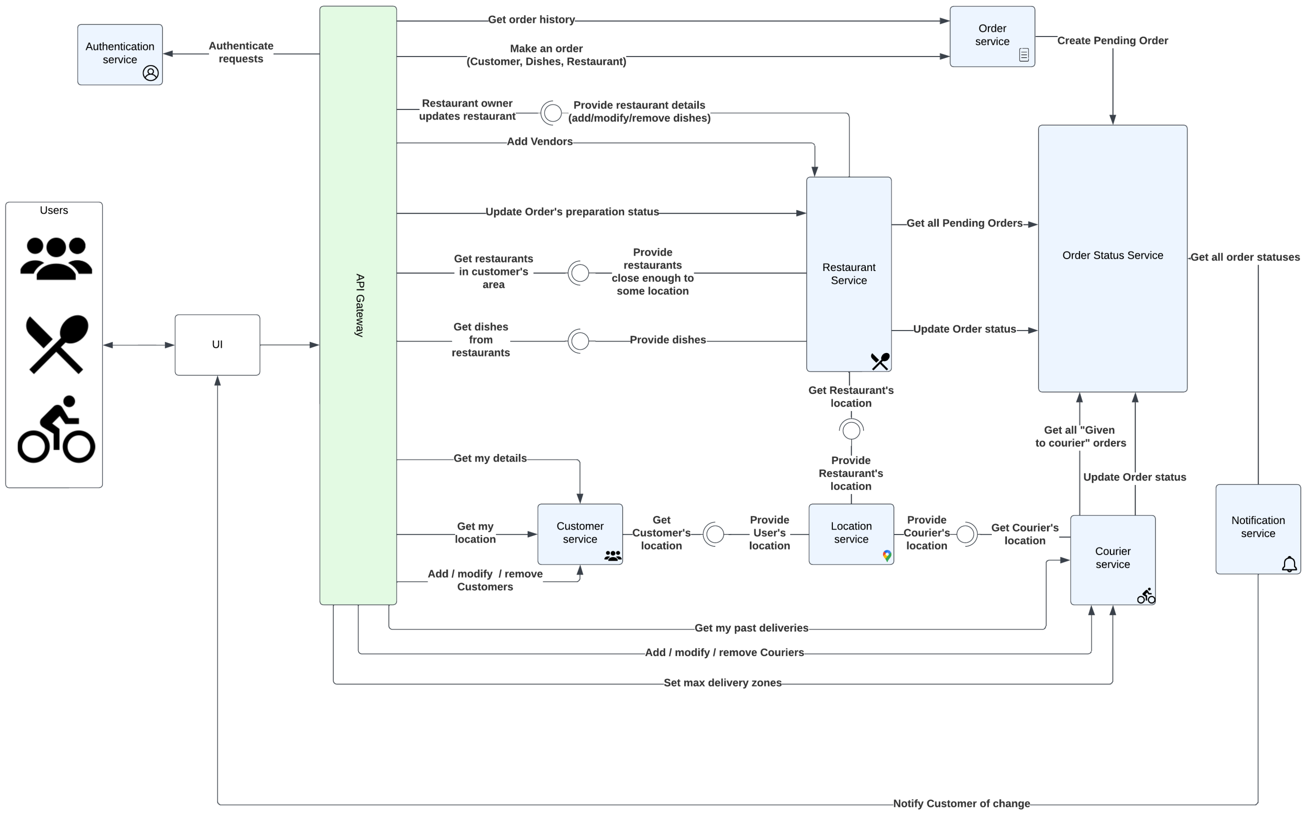

Authentication service
Authenticate requests
API Gateway
Get order history
Make an order (Customer, Dishes, Restaurant)
Order service
Create Pending Order
Restaurant owner updates restaurant
Provide restaurant details (add/modify/remove dishes)
Add Vendors
Users
Update Order's preparation status
Restaurant Service
Get all Pending Orders
Order Status Service
Get all order statuses
Get restaurants in customer's area
Provide restaurants close enough to some location
UI
Get dishes from restaurants
Provide dishes
Update Order status
Get Restaurant's location
Provide Restaurant's location
Get all "Given to courier" orders
Update Order status
Get my details
Get my location
Customer service
Get Customer's location
Provide User's location
Location service
Provide Courier's location
Get Courier's location
Courier service
Notification service
Add / modify / remove Customers
Get my past deliveries
Add / modify / remove Couriers
Set max delivery zones
Notify Customer of change

Figure 26: Alternative design, using a single restaurant service and the order status microservice.

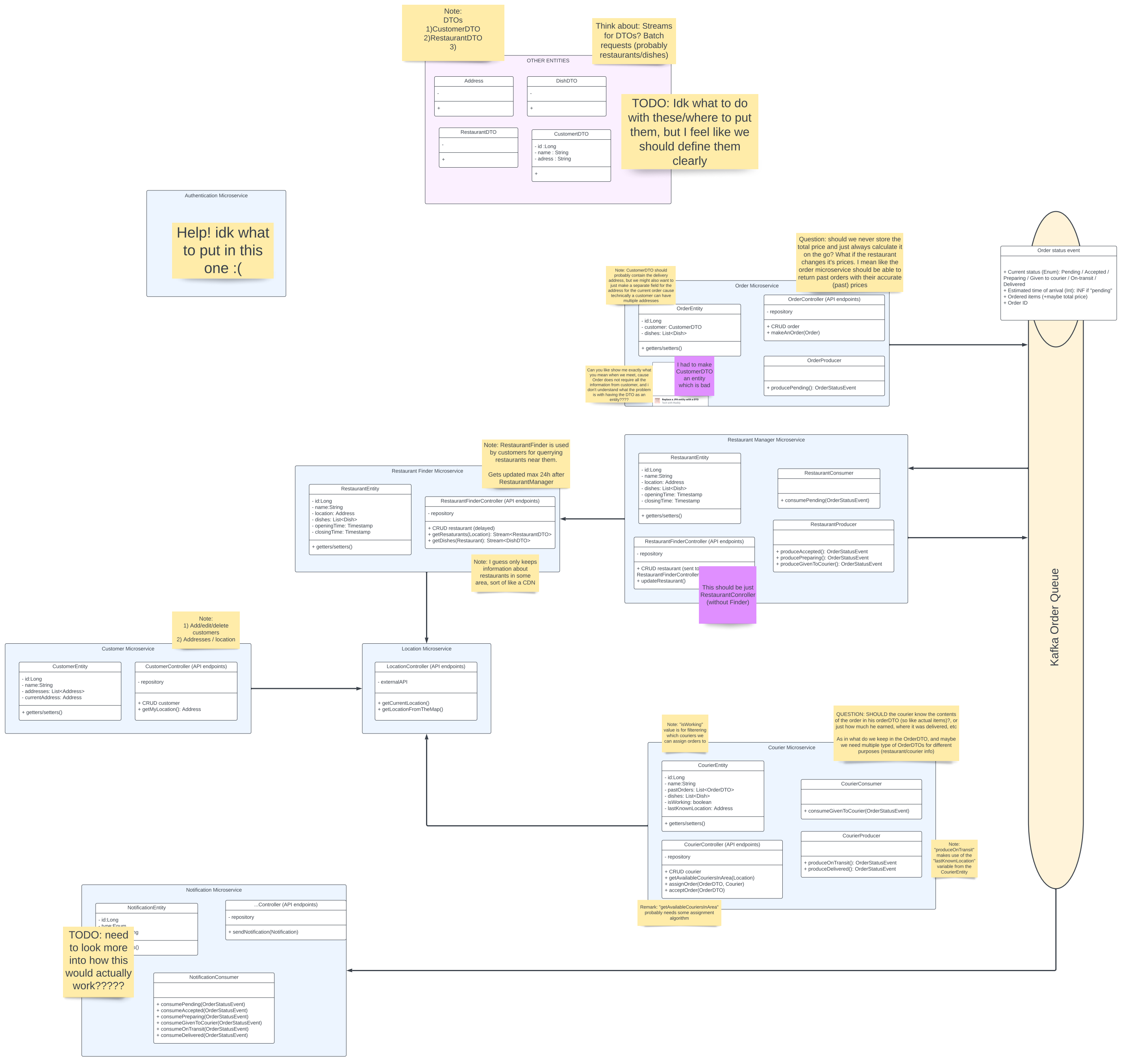

Note:
DTOs
1)CustomerDTO
2)RestaurantDTO
3)
OTHER ENTITIES
Think about: Streams for DTOs? Batch requests (probably restaurants/dishes)
Address
-
+
DishDTO
-
+
RestaurantDTO
-
+
CustomertDTO
- id :Long
- name : String
- adress : String
+
TODO: Idk what to do with these/where to put them, but I feel like we should define them clearly
Authentication Microservice
Help! idk what to put in this one :(
Note: CustomerDTO should probably contain the delivery address, but we might also want to just make a separate field for the address for the current order cause technically a customer can have multiple addresses
Question: should we never store the total price and just always calculate it on the go? What if the restaurant changes it's prices. I mean like the order microservice should be able to return past orders with their accurate (past) prices
Order Microservice
OrderEntity
- id:Long
- customer: CustomerDTO
- dishes: List<Dish>
+ getters/setters()
OrderController (API endpoints)
- repository
+ CRUD order
+ makeAnOrder(Order)
OrderProducer
+ producePending(): OrderStatusEvent
Can you like show me exactly what you mean when we meet, cause Order does not require all the information from customer, and i don't understand what the problem is with having the DTO as an entity????
I had to make CustomerDTO an entity which is bad
Replace a JPA entity with a DTO
Tech with Maddy
Order status event
+ Current status (Enum): Pending / Accepted / Preparing / Given to courier / On-transit / Delivered
+ Estimated time of arrival (Int): INF if "pending"
+ Ordered items (+maybe total price)
+ Order ID
Kafka Order Queue
Note: RestaurantFinder is used by customers for querrying restaurants near them.
Gets updated max 24h after RestaurantManager
Restaurant Finder Microservice
RestaurantEntity
- id:Long
- name:String
- location: Address
- dishes: List<Dish>
- openingTime: Timestamp
- closingTime: Timestamp
+ getters/setters()
RestaurantFinderController (API endpoints)
- repository
+ CRUD restaurant (delayed)
+ getResaturants(Location): Stream<RestaurantDTO>
+ getDishes(Restaurant): Stream<DishDTO>
Note: I guess only keeps information about restaurants in some area, sort of like a CDN
Restaurant Manager Microservice
RestaurantEntity
- id:Long
- name:String
- location: Address
- dishes: List<Dish>
- openingTime: Timestamp
- closingTime: Timestamp
+ getters/setters()
RestaurantConsumer
+ consumePending(OrderStatusEvent)
RestaurantProducer
+ produceAccepted(): OrderStatusEvent
+ producePreparing(): OrderStatusEvent
+ produceGivenToCourier(): OrderStatusEvent
RestaurantFinderController (API endpoints)
- repository
+ CRUD restaurant (sent to RestaurantFinderController
+ updateRestaurant()
This should be just RestaurantConroller (without Finder)
Note:
1) Add/edit/delete customers
2) Addresses / location
Customer Microservice
CustomerEntity
- id:Long
- name:String
- addresses: List<Address>
- currentAddress: Address
+ getters/setters()
CustomerController (API endpoints)
- repository
+ CRUD customer
+ getMyLocation(): Address
Location Microservice
LocationController (API endpoints)
- externalAPI
+ getCurrentLocation()
+ getLocationFromTheMap()
Note: "isWorking" value is for filterering which couriers we can assign orders to
QUESTION: SHOULD the courier know the contents of the order in his orderDTO (so like actual items)?, or just how much he earned, where it was delivered, etc
As in what do we keep in the OrderDTO, and maybe we need multiple type of OrderDTOs for different purposes (restaurant/courier info)
Courier Microservice
CourierEntity
- id:Long
- name:String
- pastOrders: List<OrderDTO>
- dishes: List<Dish>
- isWorking: boolean
- lastKnownLocation: Address
+ getters/setters()
CourierConsumer
+ consumeGivenToCourier(OrderStatusEvent)
CourierProducer
+ produceOnTransit(): OrderStatusEvent
+ produceDelivered(): OrderStatusEvent
CourierController (API endpoints)
- repository
+ CRUD courier
+ getAvailableCouriersInArea(Location)
+ assignOrder(OrderDTO, Courier)
+ acceptOrder(OrderDTO)
Note: "produceOnTransit" makes use of the "lastKnownLocation" variable from the CourierEntity
Remark: "getAvailableCouriersInArea" probably needs some assignment algorithm
Notification Microservice
NotificationEntity
- id:Long
- type:Enum
...Controller (API endpoints)
- repository
+ sendNotification(Notification)
TODO: need to look more into how this would actually work?????
NotificationConsumer
+ consumePending(OrderStatusEvent)
+ consumeAccepted(OrderStatusEvent)
+ consumePreparing(OrderStatusEvent)
+ consumeGivenToCourier(OrderStatusEvent)
+ consumeOnTransit(OrderStatusEvent)
+ consumeDelivered(OrderStatusEvent)

Figure 27: UMLV1