Software Architecture: Thuisbezorgd

Team 90

Ana, Sebastian, Radu, Octav

TU Delft

Introduction

- Food delivery apps have grown in popularity
- Companies had to scale their platform accordingly
- A robust architecture is required
- Propose a viable architecture for Thuisbezorgd

TU Delft

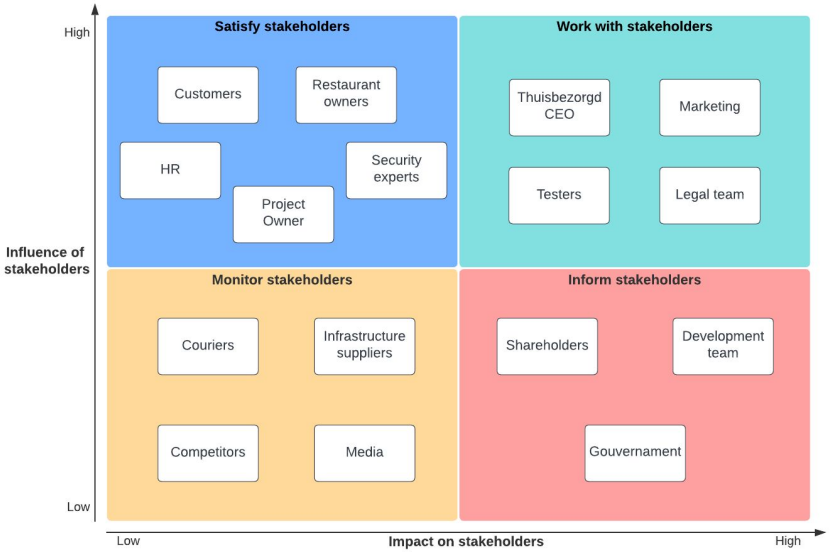
Goals

- Dive deeper into microservices
- Learn about asynchronous communication
- Explain design choices through diagramming
- Focus on fault tolerance, scalability and cohesiveness

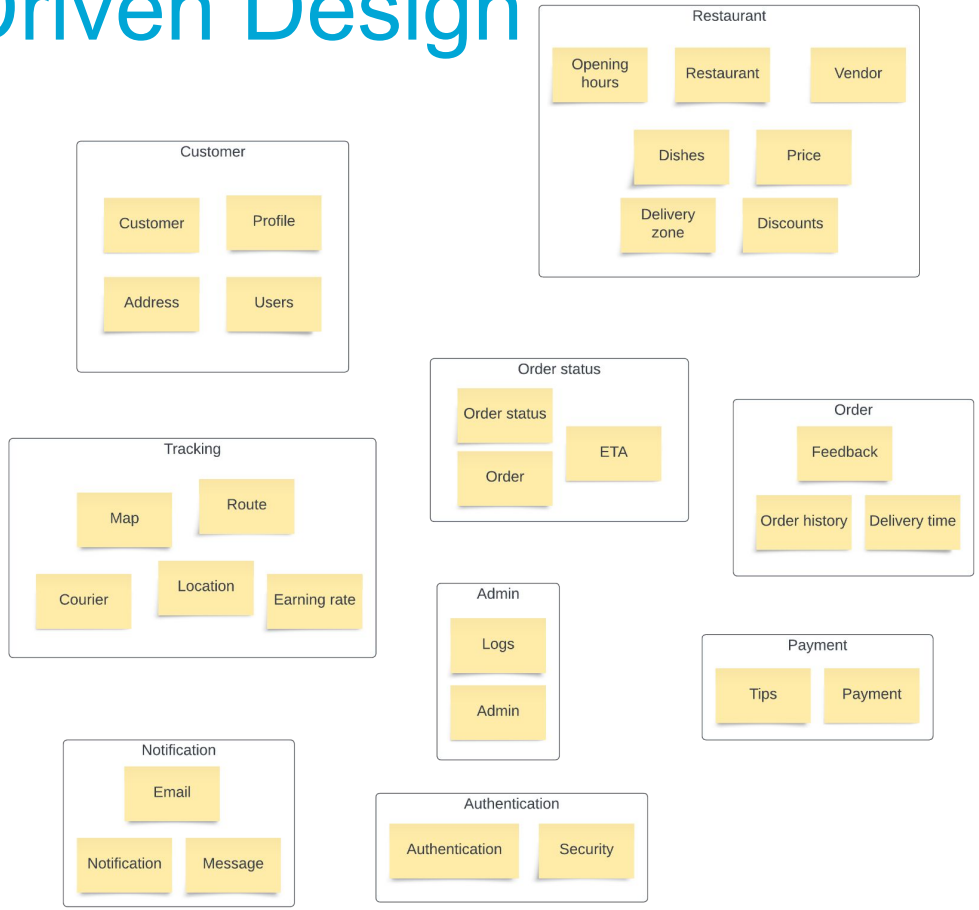
TU Delft

Stakeholders

- Customers
- Restaurant owners
- Couriers
- Project owner
- Security experts
- Development team



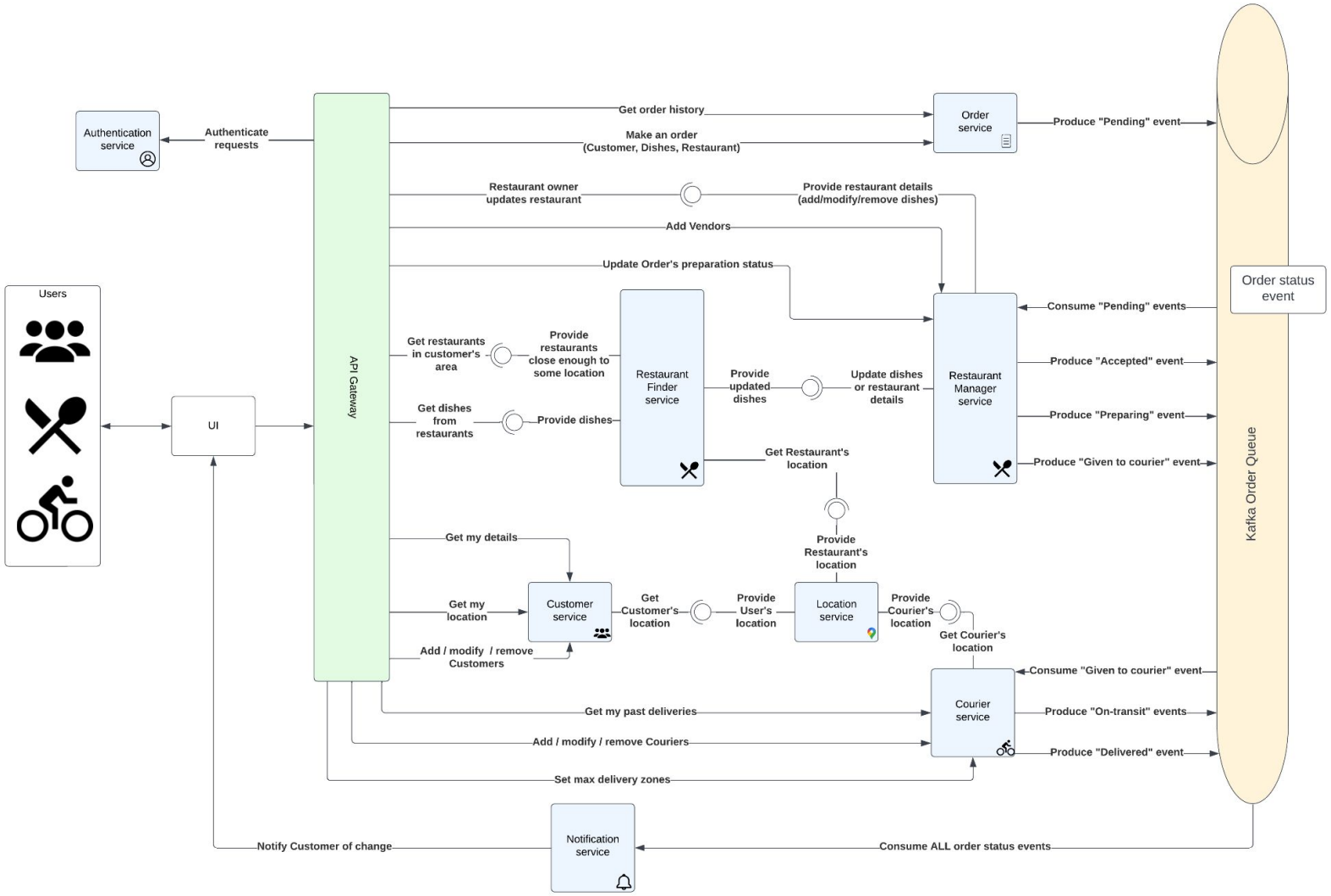
Domain Driven Design



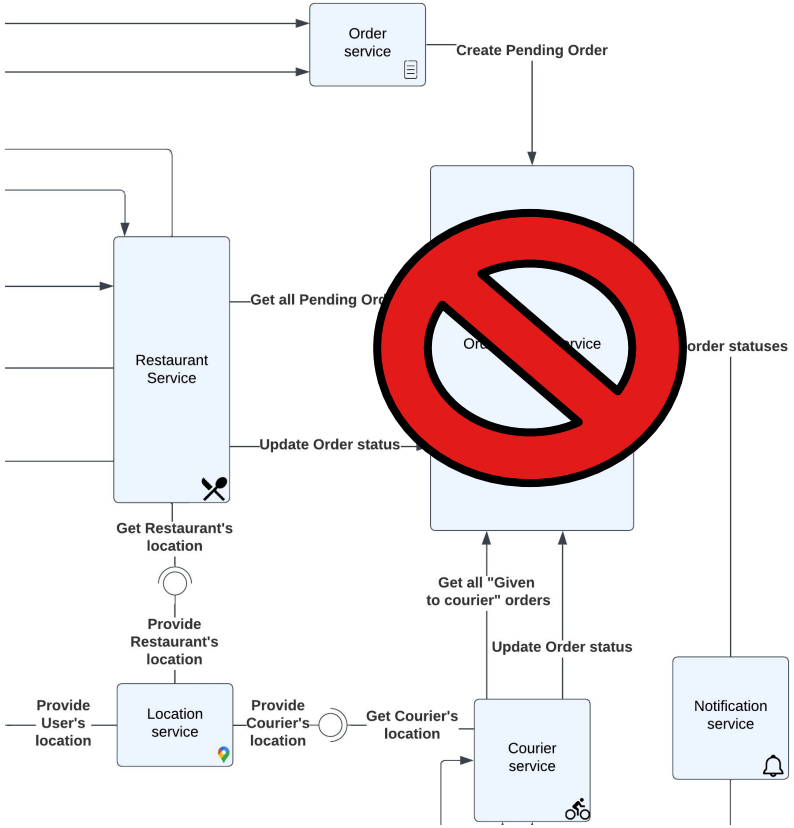
Architecture

- Microservice architecture
- Apache Kafka Queue for asynchronous communication
- Gateway for routing incoming requests
- API endpoints also available for information that does need to be on the queue

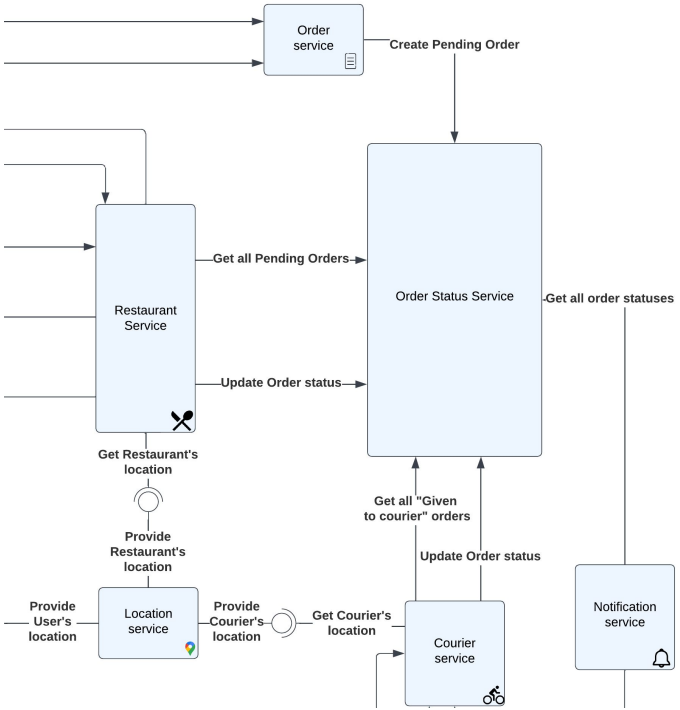
TU Delft

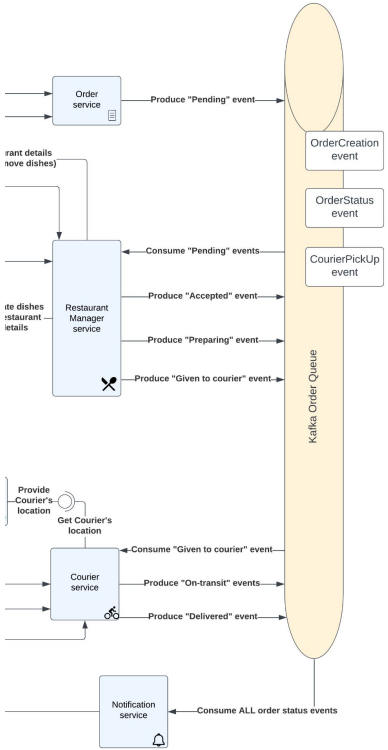

Authentication service
Authenticate requests
API Gateway
Get order history
Make an order
(Customer, Dishes, Restaurant)
Order service
Produce "Pending" event
Restaurant owner
updates restaurant
Provide restaurant details
(add/modify/remove dishes)
Add Vendors
Update Order's preparation status
Order status event
Users
UI
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
Consume "Pending" events
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

Design choice 1a: OrderStatus Microservice

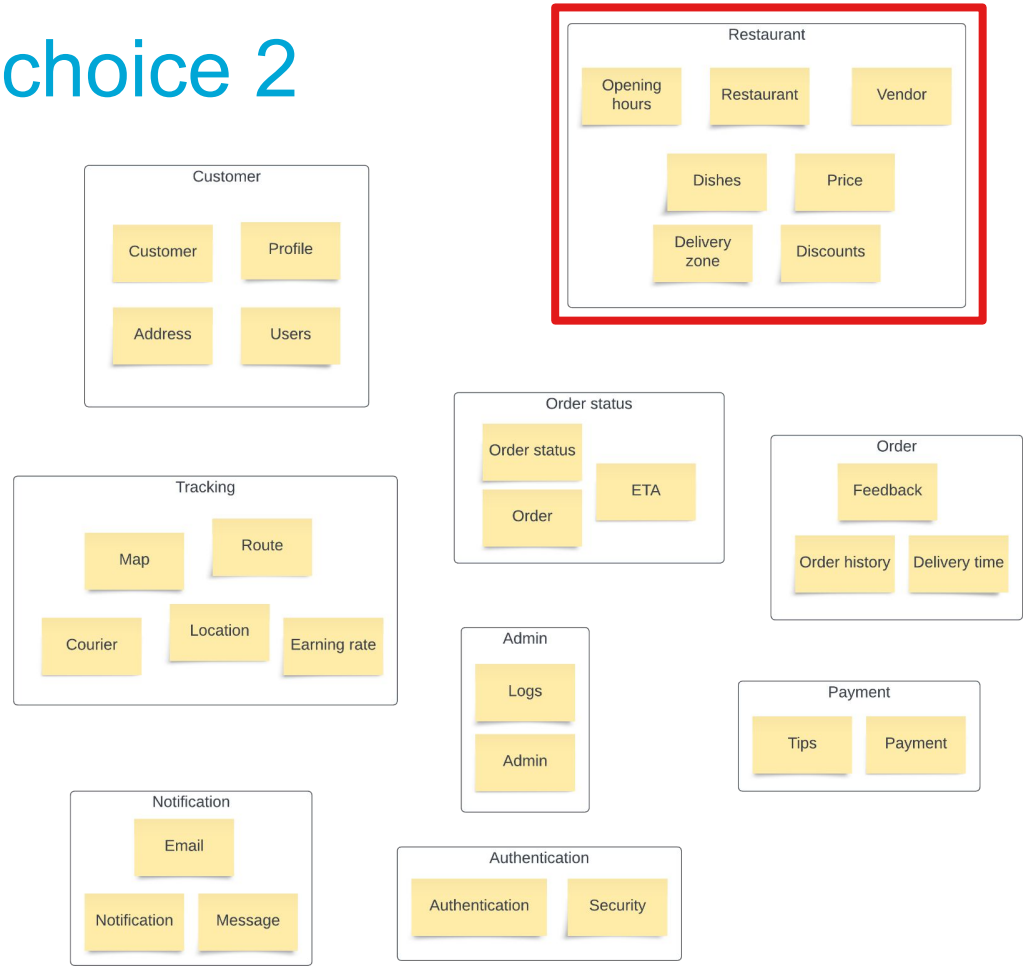


Design choice 1b: OrderStatus Microservice VS Kafka Event Bus

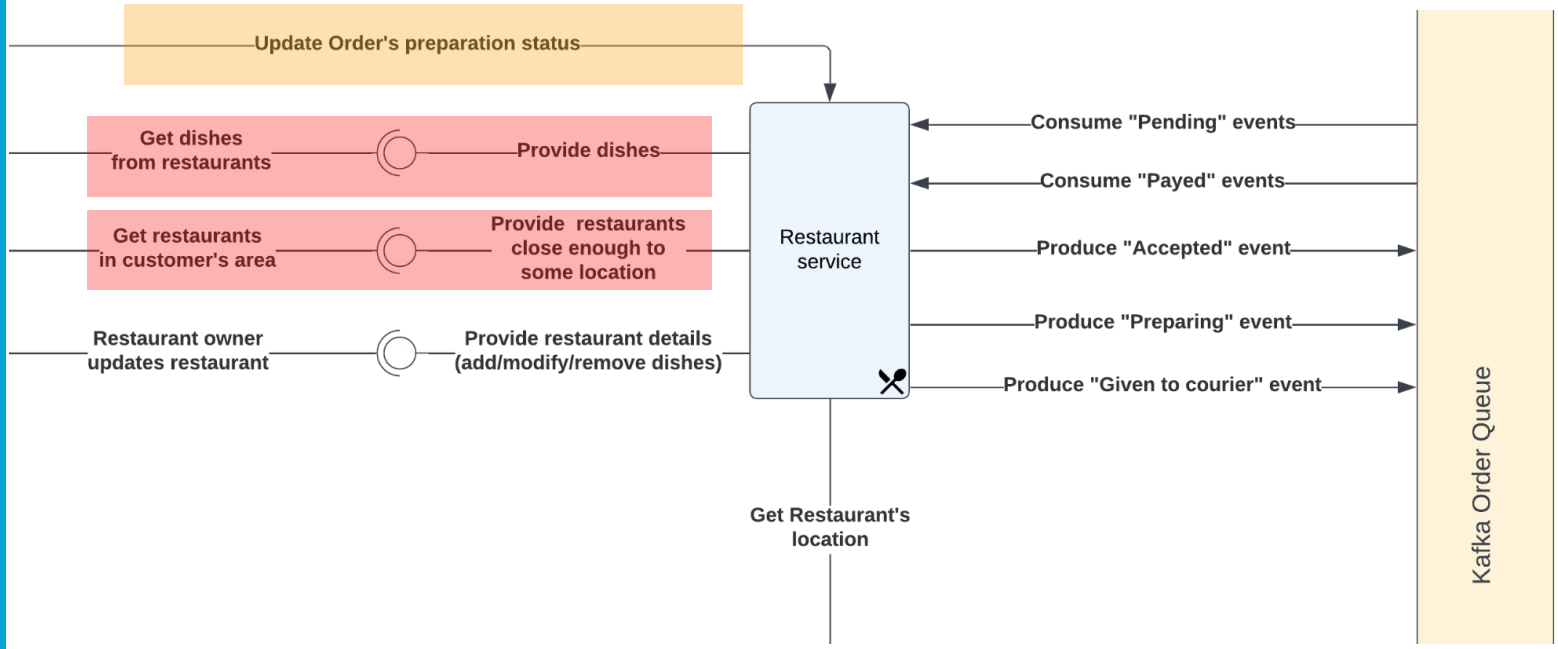




Design choice 2



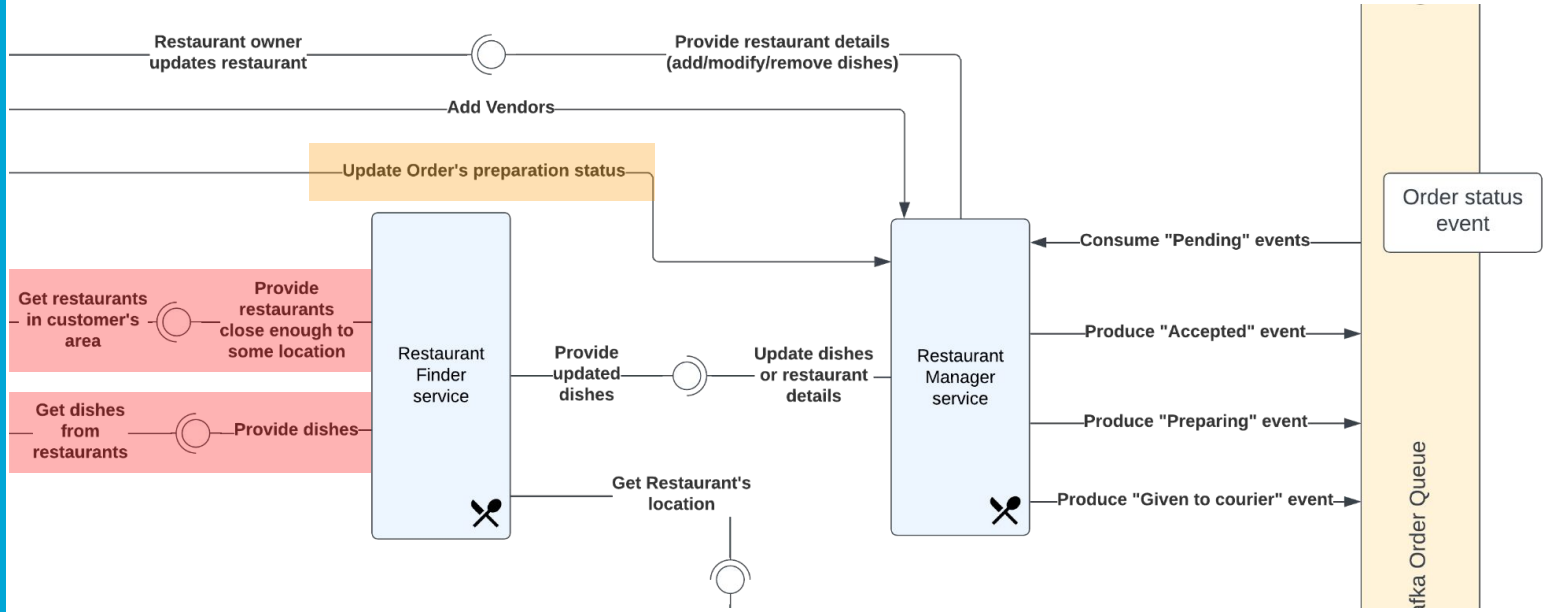
Design choice 2a: Single Restaurant Service



Lots of POST req

Lots of GET req

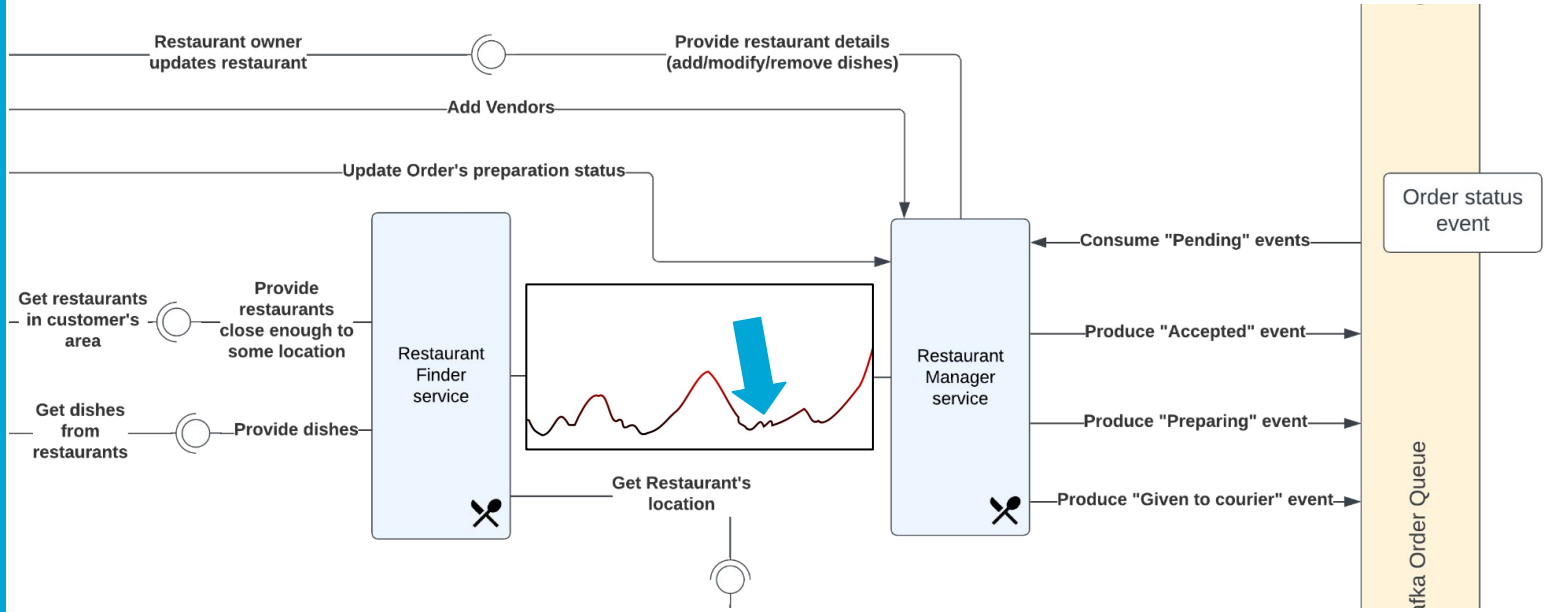
Design choice 2b: Split Restaurant Service



Lots of POST req

Lots of GET req

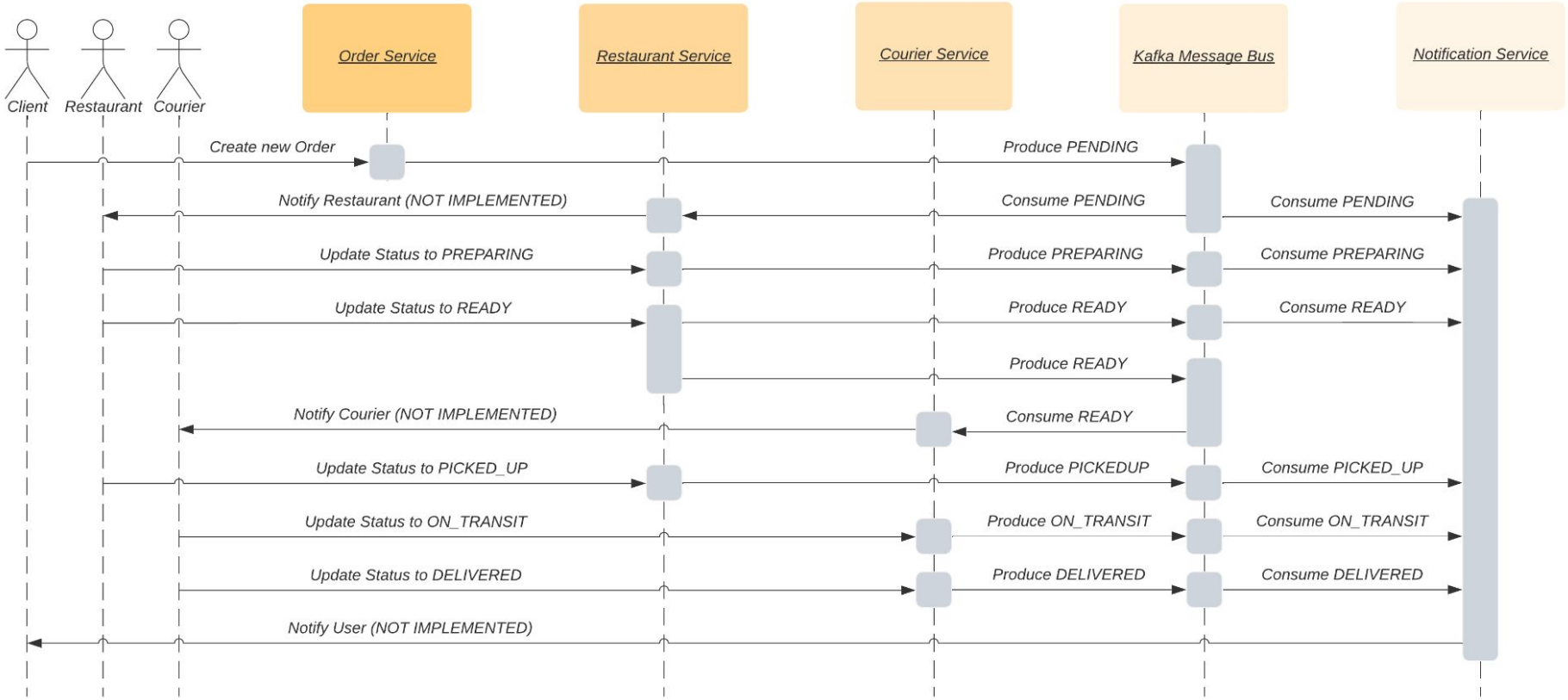
Design choice 2b: Updating the Restaurant finder



Proof of concept

- Kafka Message Bus
- Relevant Microservices:
  - Order Service
  - Restaurant Service
  - Courier Service
  - Notification Service

TU Delft


Client
Restaurant
Courier
Order Service
Restaurant Service
Courier Service
Kafka Message Bus
Notification Service
Create new Order
Produce PENDING
Notify Restaurant (NOT IMPLEMENTED)
Consume PENDING
Consume PENDING
Update Status to PREPARING
Produce PREPARING
Consume PREPARING
Update Status to READY
Produce READY
Consume READY
Produce READY
Notify Courier (NOT IMPLEMENTED)
Consume READY
Update Status to PICKED_UP
Produce PICKEDUP
Consume PICKED_UP
Update Status to ON_TRANSIT
Produce ON_TRANSIT
Consume ON_TRANSIT
Update Status to DELIVERED
Produce DELIVERED
Consume DELIVERED
Notify User (NOT IMPLEMENTED)

Demo

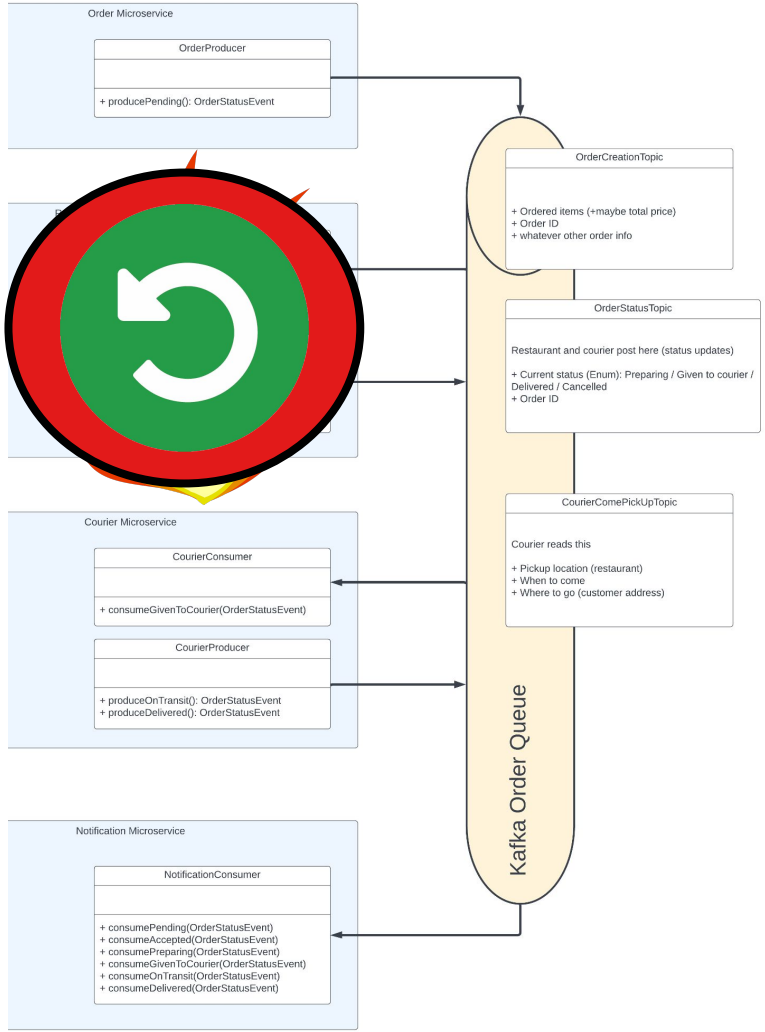
- Let’s make a few requests

TU Delft

Measurements and Experiments

- Assumption: robust architecture by using microservices and Kafka.
- Measure architecture robustness trough:
  - Fault Tolerance
  - Throughput
  - UML Analysis: Coupling and Cohesion

TU Delft


Order Microservice
OrderProducer
+ producePending(): OrderStatusEvent
Courier Microservice
CourierConsumer
+ consumeGivenToCourier(OrderStatusEvent)
CourierProducer
+ produceOnTransit(): OrderStatusEvent
+ produceDelivered(): OrderStatusEvent
Notification Microservice
NotificationConsumer
+ consumePending(OrderStatusEvent)
+ consumeAccepted(OrderStatusEvent)
+ consumePreparing(OrderStatusEvent)
+ consumeGivenToCourier(OrderStatusEvent)
+ consumeOnTransit(OrderStatusEvent)
+ consumeDelivered(OrderStatusEvent)
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)
Kafka Order Queue

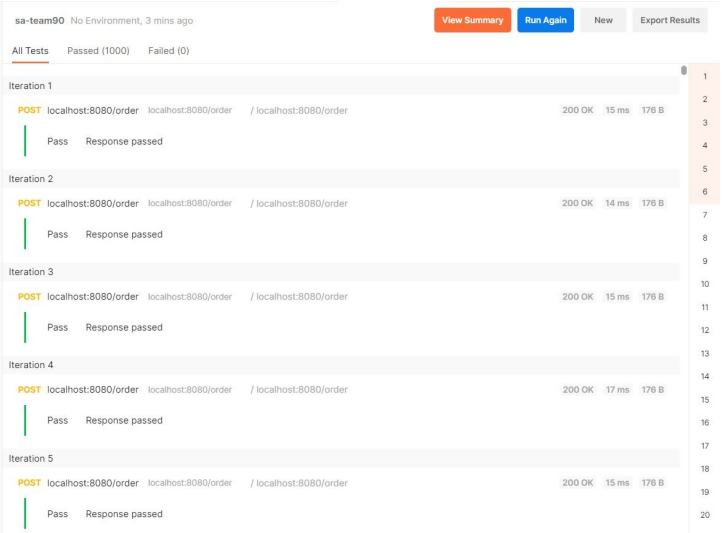
Experiment: Fault Tolerance

- Stop-restart experiment:
  - Restaurant and Order Services
- Information stored on kafka is used to restore the state.
- After restart information flow continues
- Better than synchronous communication

TU Delft

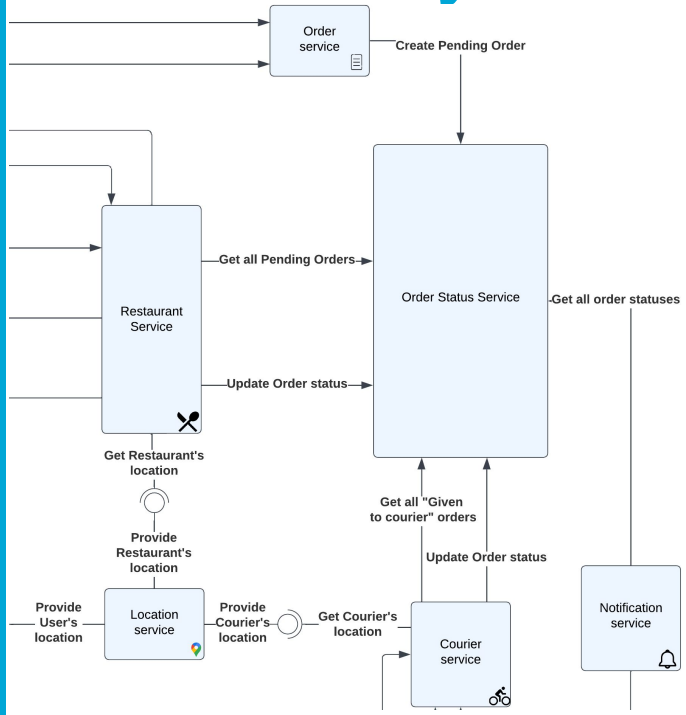
Experiment: Throughput

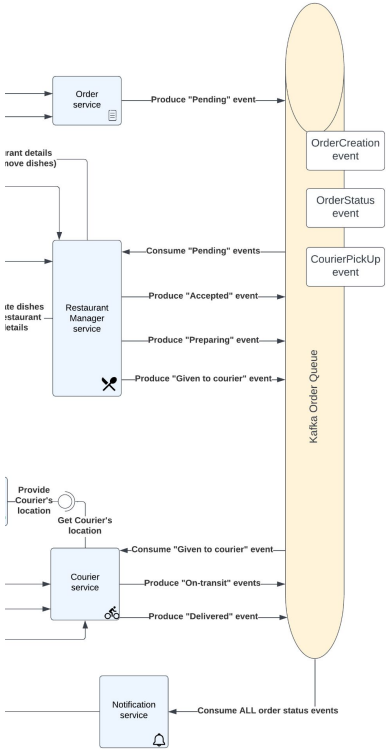
- Tests throughput of kaka queue
- Kafka queue chaining
- Order generation through postman requests(1000 req)
- Average performance 6.9ms/req



TU Delft

UML analysis: Coupling

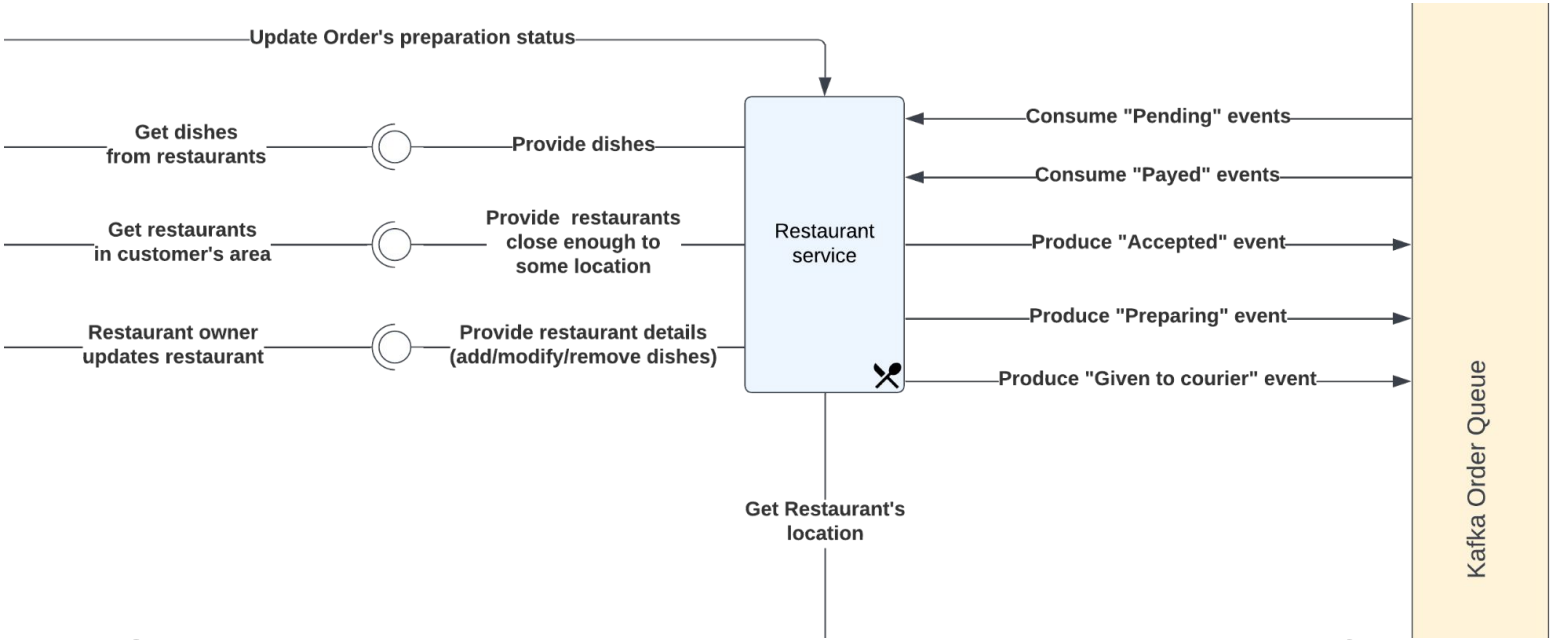




Avg coupling: 2

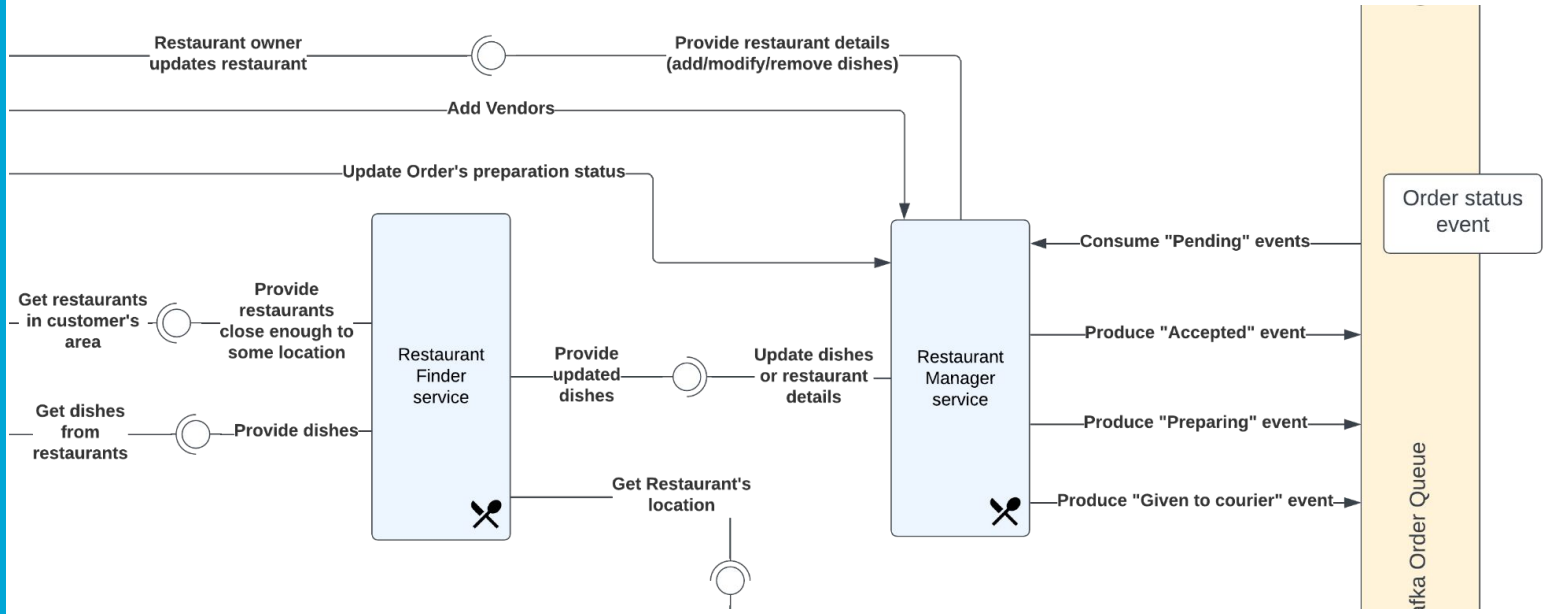
Avg coupling: 1.2

UML Analysis: cohesion



- Small coupling improvement: Average of 1
- Low cohesion: Order vs Dish domain

UML Analysis: cohesion



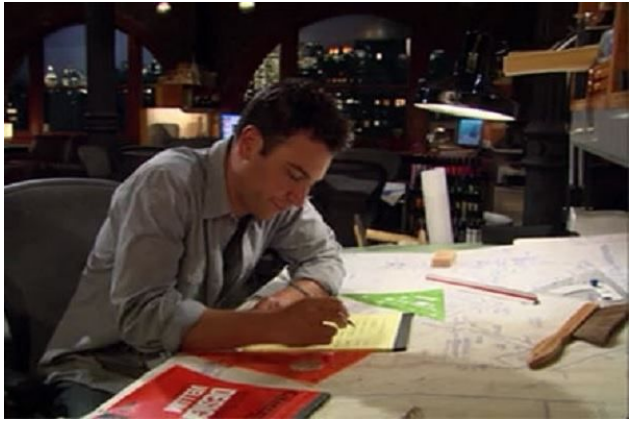
- High cohesion in both services due to separation of domains

Conclusion

- Fault tolerance and high throughput by using message queues
- Improved diagramming readability and overall designing skills.
- Reiteration of architecture: explainability through diagramming.

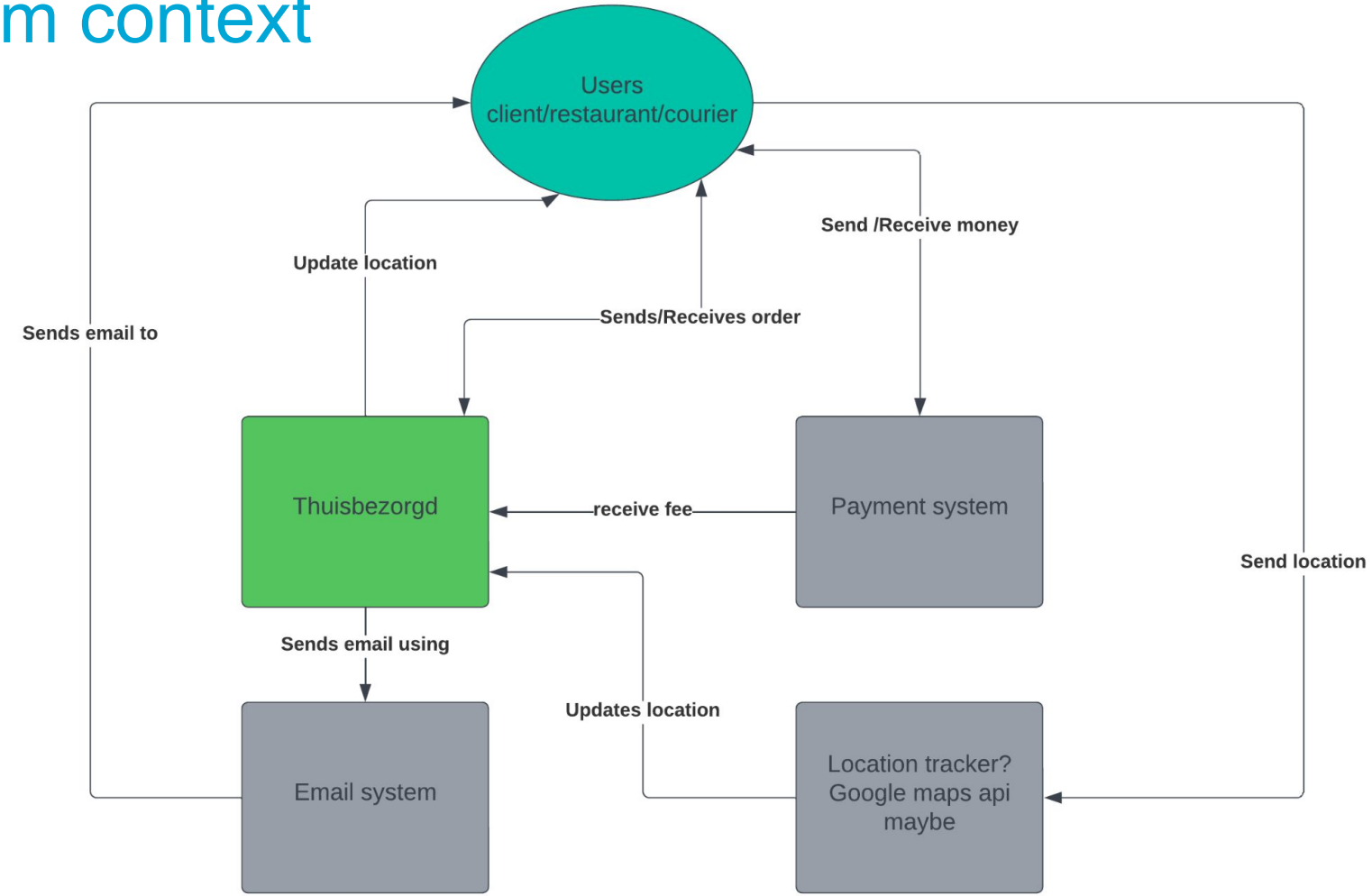
TU Delft

Thanks architects!

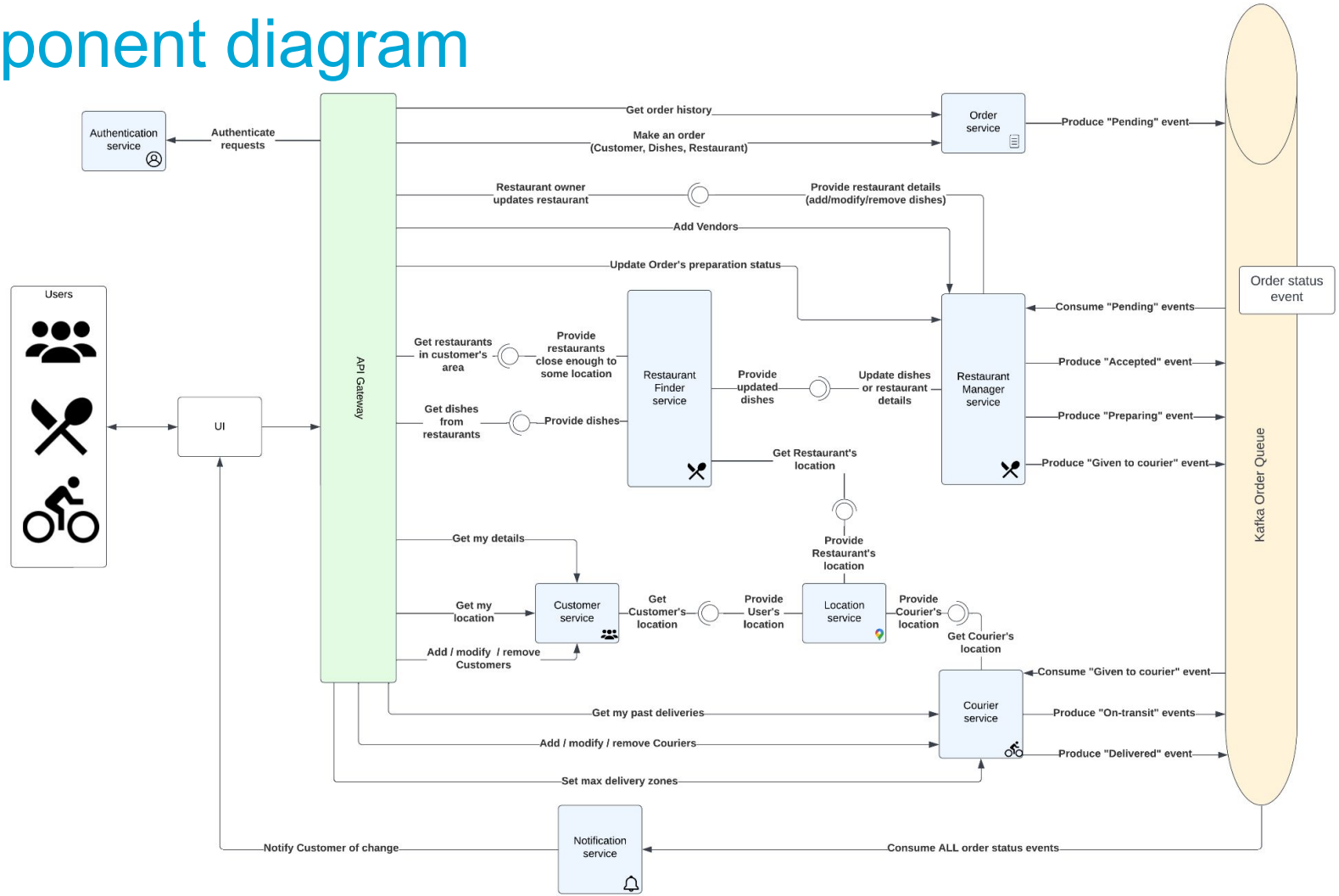


Questions?

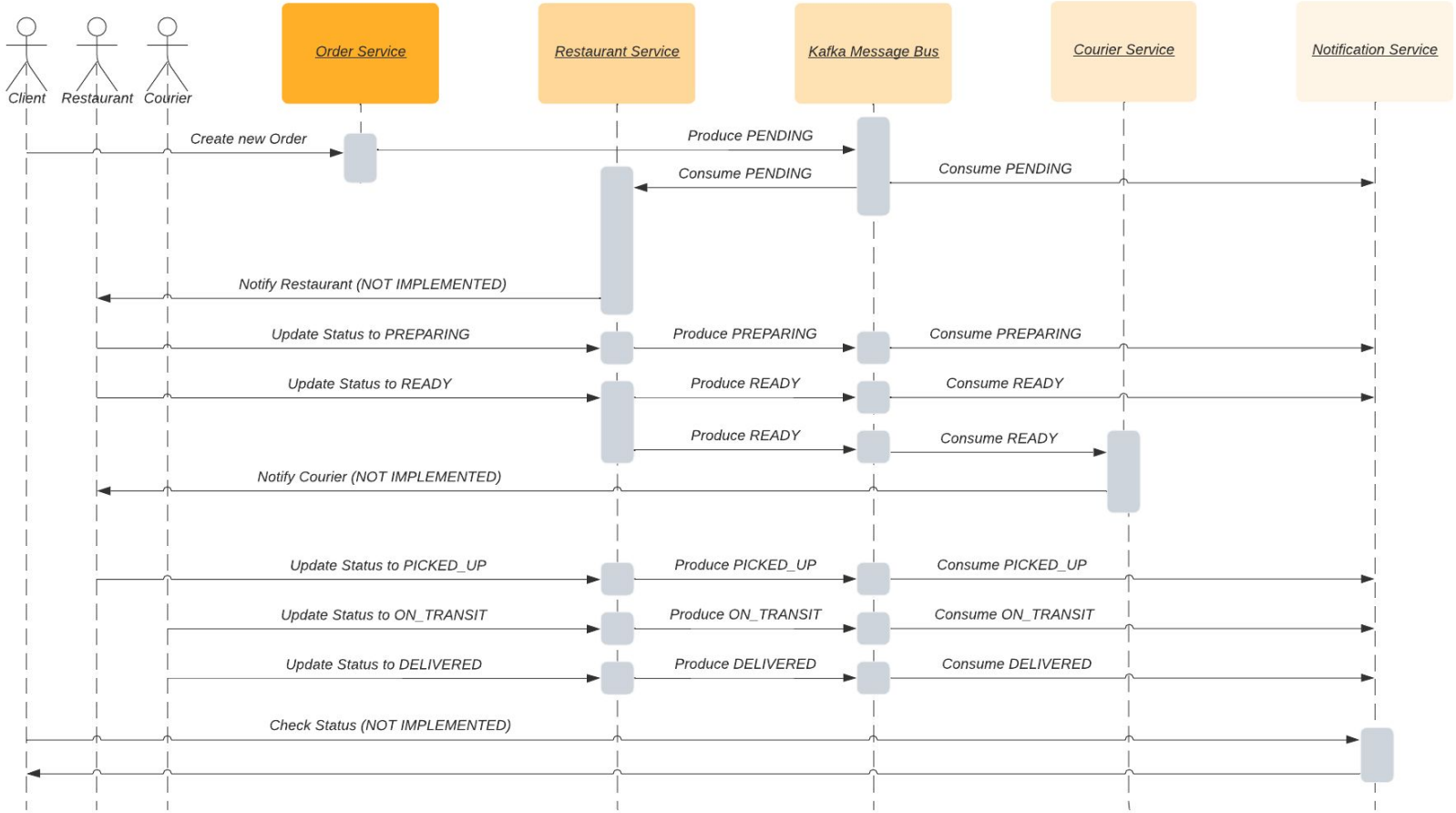
System context



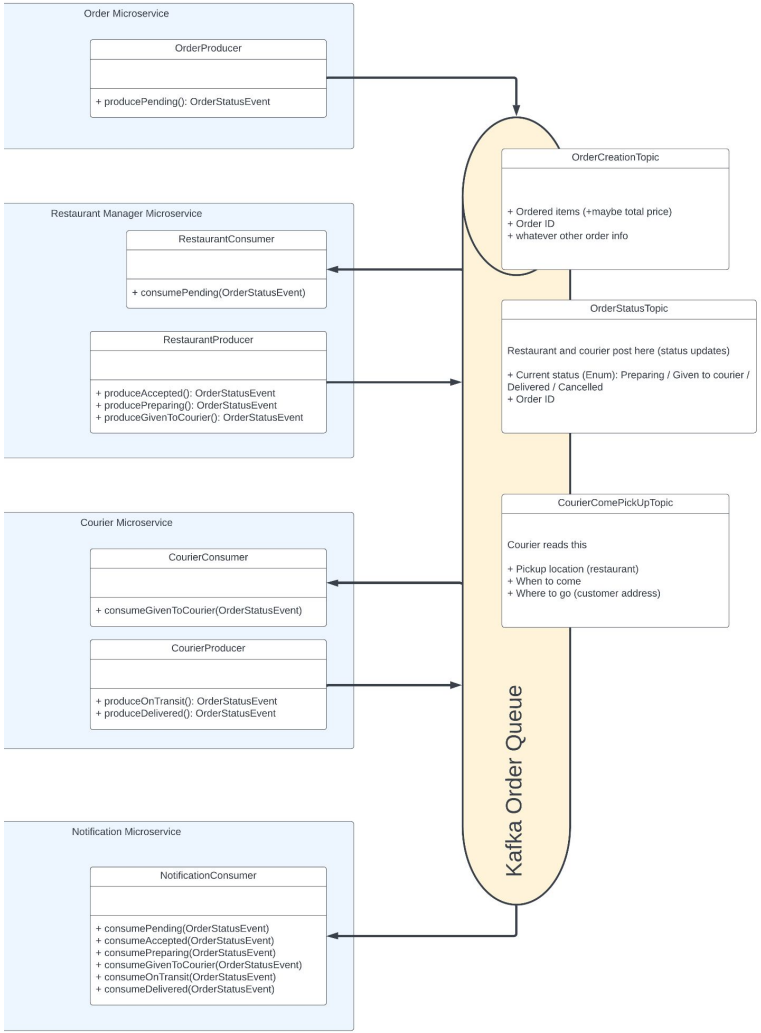
Component diagram



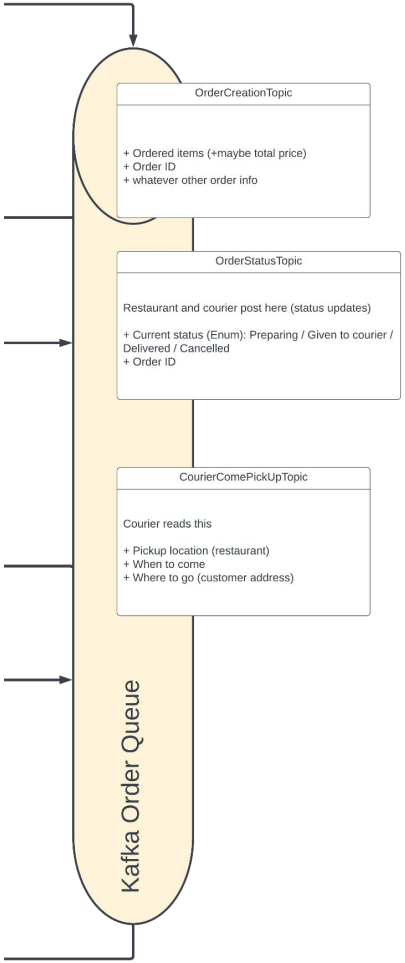
Sequence diagram



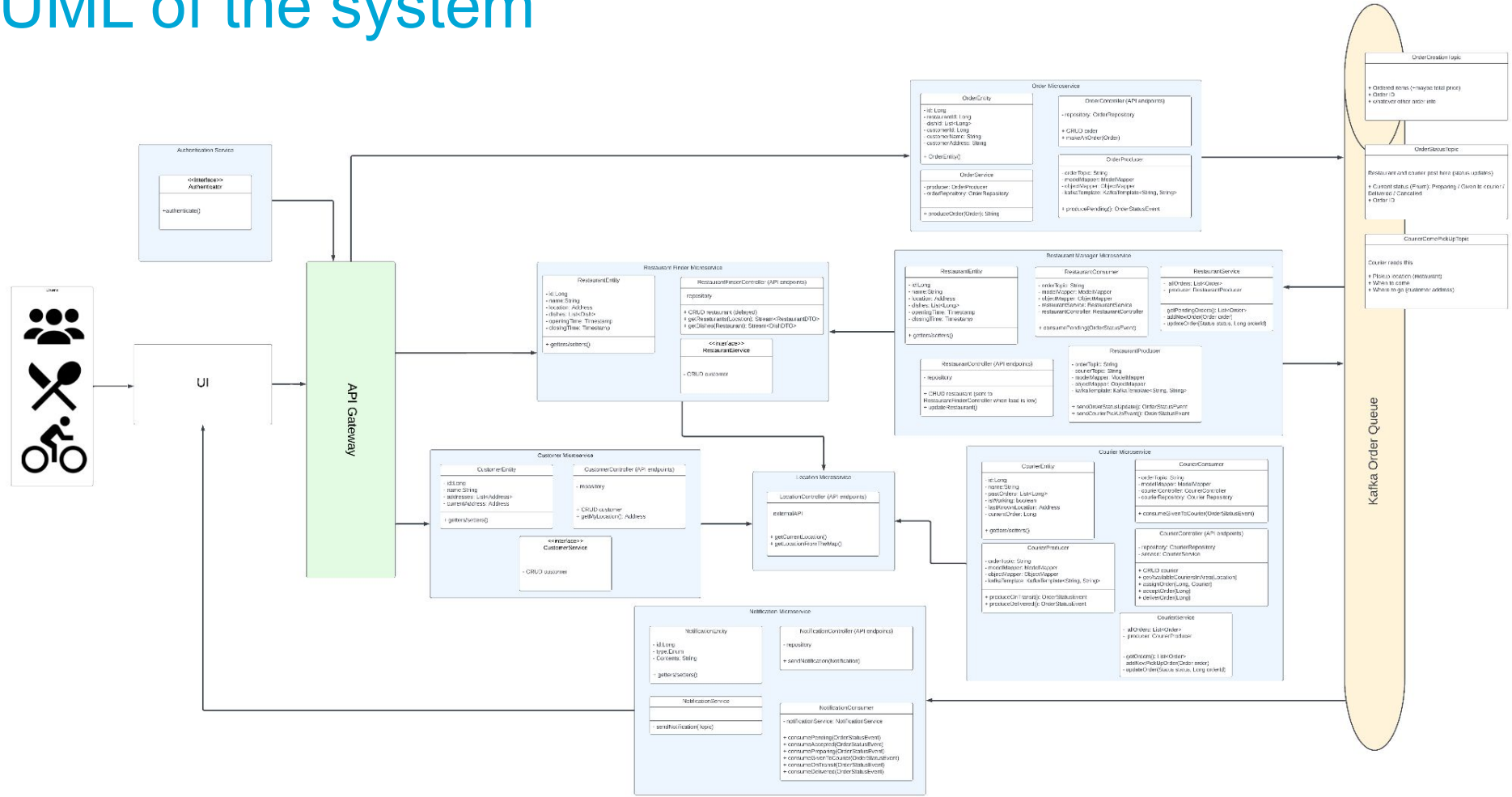
Kafka Consumers and Producers



Kafka events

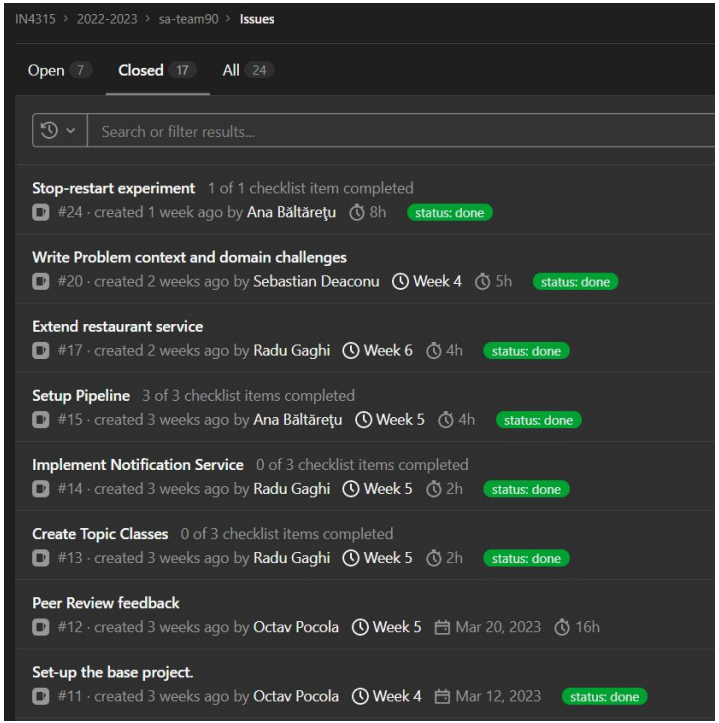


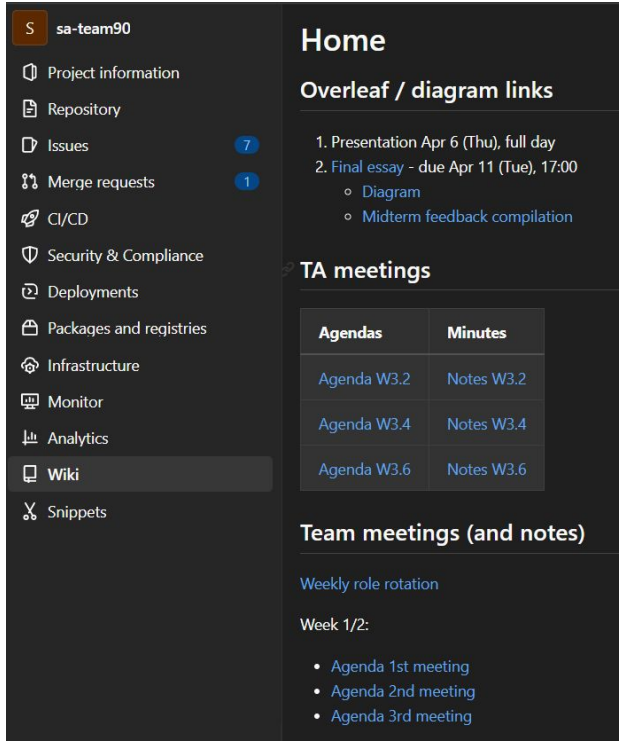
UML of the system



Organization

- Issues
- Milestones
- Github Wiki

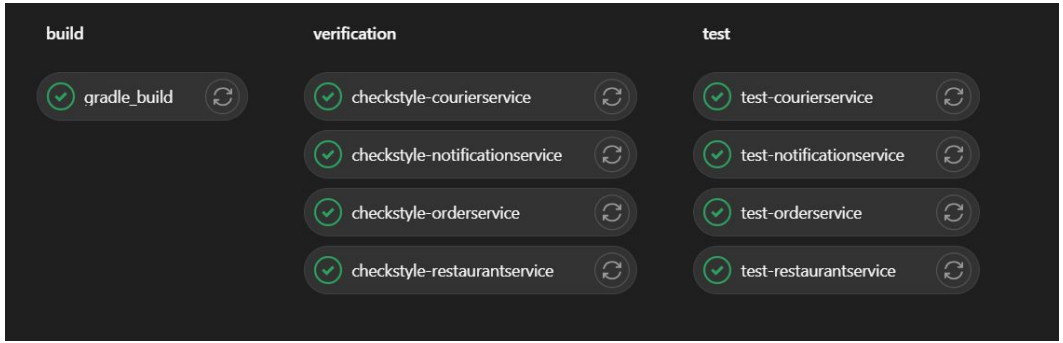




Testing & CI/CD

- Experiments
- Unit tests
- Integration tests
- Pipeline

```java
@DataJpaTest
class OrderServiceTests {

        @InjectMocks /// The one in which you put all the mocked things
        private OrderService orderService;

        @Mock /// Which things you are mocking
        private Producer producer;

        @Mock /// Needs to be everything that is @Autowired
        private OrderRepository orderRepository;

        private static Order order1;

        @BeforeEach
        public void setup() {
                order1 = new Order();
        }

        @Test
        public void processOrderTest() throws JsonProcessingException {
                when(orderRepository.save(any())).thenReturn(order1);
                when(producer.sendMessage(any())).thenReturn("message sent");
                assertEquals("message sent", orderService.processOrder(order1));
        }
}
```



Documentation

- Readme
- Javadoc
- Checkstyle

Run the aplication

1. Kafka Docker Conainter

We will run the Kafka server in a Docker container. First, start you Docker Desktop and wait for the engine to start. Then, run the following command in the root directory of the project:

```
docker-compose up
```

If all went well, you should see three running containers.

Showing 4 items

Search

NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
sa-team90 3 containers	-	Running (3/3)	-		
kafdrop 9739ff32e722	obsidiandynamics/kafdrop:late	Running	9000	2 hours ago	
kafka 25e1e95f2c1f	obsidiandynamics/kafka:latest	Running	9092	2 hours ago	
zookeeper 556aaa8db879	zookeeper:3.7.0	Running	2181	2 hours ago	

You should also be able to access the Kafkadrop web interface at localhost:9000.

```java
    /**
     * Returns all pending orders in string format.
     *
     * @return all the pending orders in string format
     */
    @GetMapping("/get-all-pending")
    public String getAllPendingOrders() {
        log.info("Returning all orders: " + restaurantService.getPendingOrders().toString());
        return restaurantService.getPendingOrders().toString();
```