SOA - Final Project Report
TrainCatcher - Train Booking Service

Victor Alecu (s2675889)
Vlad Gosa (s2755572)

1 Case Description

1.1 Problem Analysis

Planning a train journey across multiple carriers presents significant challenges for modern travelers. Without a centralized platform (for the purposes of this project we assume the NS app does not exist), users often face the daunting task of navigating various websites and applications to gather accurate information about schedules, ticket prices, and connection times. For instance, consider the journey from Enschede Kennispark to Schiphol Airport—a route that typically involves two to three transfers among different carriers such as Blauwnet, Nederlandse Spoorwegen (NS), and occasionally Arriva. In such cases, passengers must purchase separate tickets from each provider and continuously monitor multiple sources for updates on delays, cancellations, or changes in platform assignments. This fragmented process not only increases the likelihood of miscommunication and missed connections but also creates a stressful environment where even minor schedule shifts can lead to significant disruptions. Additionally, when unforeseen incidents such as technical faults or adverse weather conditions occur, the lack of a unified source for real-time information exacerbates the difficulties of managing the journey. Thus, the absence of an integrated system that consolidates data from multiple operators results in inefficiencies, diminished customer satisfaction, and a higher probability of travel-related issues that can significantly affect the overall travel experience.

1.2 Solution Proposal

TrainCatcher addresses these challenges by providing a centralized application that can aggregate real-time data from multiple train operators and related service providers. By leveraging a Service-Oriented Architecture, the system is designed as a collection of loosely coupled services, each responsible for a specific functionality such as searching for train routes, managing bookings, processing payments, issuing tickets, and delivering notifications. This modular approach not only enhances scalability and maintainability but also allows for easier integration with external systems, ensuring that users receive the most current information on train schedules, delays, and cancellations. Additionally, the platform supports a variety of user requirements, from searching for available train trips to booking, paying, and distributing the tickets via email, thus streamlining the travel process and reducing the need for multiple interactions across disparate platforms. Through dynamic service orchestration, TrainCatcher is capable of adapting to changes in the operating environment, such as the addition of new train operators. In this way, the application ensures a consistently high level of service reliability, ultimately transforming the way users experience train travel by providing an all-encompassing solution for journey management.

1.3 Objectives

To ensure that the TrainCatcher application fulfills both the business goals outlined in the project proposal and the technical requirements set by the SOA course assignment, we have identified a set of objectives that range from high-level business-oriented goals to more detailed technical implementation targets:

- **Business-Oriented Objectives:**
  - TrainCatcher should allow users to search for a train journey, select one they wish to book and, if they have the necessary funds, reserve and pay for it with the relevant train operators, as well as print the ticket and send it via email.
  - TrainCatcher should allow users to create or delete accounts, as well as add funds to their in-app balance.
  - TrainCatcher should allow train operators to register themselves in the app and have the journeys they provide shown to the users.
- **Technical Objectives:**
  - TrainCatcher should follow the Microservices architecture style.
  - TrainCatcher should be deployed on a Kubernetes cluster.
  - TrainCatcher should showcase at least one instance of synchronous communication and one of asynchronous communication among its services.
  - TrainCatcher should ensure strong consistency through distributed transactions for atomic processes such as payments or seat bookings.
  - TrainCatcher should offer secure access to the microservice cluster by securing its edge services with JWT-based access.

2 Supported Business Processes

TrainCatcher supports a range of business processes designed to fulfill both user-facing and internal operational needs. The solution primarily adopts the **Orchestrated Sagas** pattern to manage its core business logic and ensure data consistency across distributed services. However, several complementary processes, due to their low complexity and minimal coordination requirements, are implemented without the use of sagas.

2.1 Non-Saga Business Processes

- **Search for a Train Trip:** Users can search for train journeys by specifying the departure and arrival stations, a desired departure time (e.g., all trains after 07:00 AM), and the maximum number of train changes they are willing to accept. The search service collects journey data from all registered train operator services, aggregates the results, and filters them according to the user's criteria to construct suitable trip options.
- **User Authentication and Account Management:** TrainCatcher allows users to log in to their existing accounts or register new ones. During registration, basic user information, including a securely hashed and salted password, is collected and stored. Upon successful login, a JSON Web Token (JWT) is issued to the user. This token is then used to authenticate and authorize the user across the application, enabling secure access to protected services without requiring repeated logins.
- **In-App Balance Top-Up:** Users can top up their in-app balance via the payment interface, which updates their available credit, enabling them to make bookings through the system.

2.2 Saga-Based Business Processes

The core functionality of TrainCatcher revolves around orchestrated sagas that coordinate multiple services involved in a single, distributed, logical transaction. These sagas ensure reliability and consistency despite the distributed nature of the application.

- **Train Trip Booking (Main Saga):** When a user books a train trip using the search service, the ordering process is initiated. A booking record is created, and the system coordinates several sub-processes:
  - Initiate seat reservations with all relevant train operators (see Seat Booking Mini-Saga).
  - Print the ticket upon successful reservation.
  - Trigger the payment process to forward funds to each involved train operator (see Payment Processing Mini-Saga).
  - Mark all previously reserved seats as occupied upon payment confirmation.
  - Send the ticket to the user via email.
- **Seat Booking (Mini-Saga 1):** As part of the trip booking, the booking service contacts each train operator responsible for segments of the selected trip to reserve seats. Once payment is completed, the service contacts these operators again to confirm and finalize the seat allocations as occupied.
- **Payment Processing (Mini-Saga 2):** The payment service validates whether the user has enough balance to cover the entire journey cost. If so, it distributes the appropriate amounts to each involved train operator based on their respective segment prices. This saga is used as a pivot in the main saga. If it fails, the whole transaction is compensated to ensure consistency.

3 System overview

3.1 Services

3.1.1 Train Operator Service

Description One goal of our application is to allow train operators to dynamically link to our system. This implies that, in order to mimic such functionality, we need to define the interface between our system and an external train company system. The train operator service therefore represents a microservice which stores all the data on behalf of a train company, essentially acting as the company's API endpoint. For the purpose of this project, the data stored by each company has been intentionally simplified to focus on core components: trains and the journeys they operate. Each train record includes basic details such as the model name and number of seats. A journey is defined solely by a departure and an arrival station—similar to a direct plane flight—along with the scheduled times, the specific train executing the journey (including the count of occupied seats), and the ticket price. This deliberate simplification, which omits intermediate stops, enables the search service to easily aggregate journeys from multiple operators to construct more complex routes (i.e., journeys are only direct train trips while routes may have stops). Technically, our system does support multi-stop journeys; however, they are represented as distinct journey entities. For example, a journey from Enschede to Amsterdam on train X can be decomposed into three separate journeys: Enschede to Hengelo, Hengelo to Zwolle, and Zwolle to Amsterdam. In a real-life scenario, journeys would be more complex and require advanced routing algorithms with various heuristics, but for our project, this streamlined model meets our service-oriented architecture goals. Additionally, each train operator can be dynamically added to the train search service, with dedicated endpoints for booking and payment. This design allows the booking service to query available journeys, reserve a seat on each segment of a desired route, and subsequently process payments for each operator accordingly.

Communication with other services All interactions with the train operator service are performed synchronously via RESTful calls. This synchronous model is favored because booking and payment operations require immediate, verifiable responses to ensure transactional consistency and prompt error handling. RESTful APIs offer a well-defined, standardized interface that simplifies integration, making it easier to manage direct request-response cycles. Additionally, since train operators also need to provide journey data to the search service, RESTful communication ensures that the search service receives up-to-date and directly available information without the delays or complexities associated with ensuring consistency in an asynchronous setting, like message queues. Given that the booking process involves sequential steps—where the booking service must confirm seat reservations and process payments in real time—the immediacy and simplicity of synchronous RESTful communication are better suited to our operational requirements.

3.1.2 Train Search Service

Description The train search service serves as the gateway into the system and initiates the main journey booking saga. It allows train operators to register themselves via their URLs, thereby validating them as approved sources of train journey data. On the user side, passengers can search for journeys by specifying a departure station, an arrival station, a departure date and time, and a maximum number of allowed train changes. The service queries all registered operators to retrieve available journeys and aggregates these into potential routes that match the user's criteria. When a user selects a route, the complete route details—comprising all individual journeys—are forwarded to the booking service to start the main trip booking saga. Additionally, as the primary service serving the frontend, the train search service provides RESTful proxies for login, sign-in, and logout operations by forwarding authentication requests to the user service.

Communication with other services The train search service relies on synchronous RESTful communication for its core endpoints, including operator registration, journey search, and the authentication proxies (login, sign-in, and logout). RESTful APIs offer a direct, standardized interface that returns immediate responses, which is essential for real-time operations. This approach ensures that operator registrations are confirmed instantly and that search queries yield up-to-date journey data without the complexity and latency associated with asynchronous systems. Additionally, using synchronous calls for authentication processes guarantees that users receive prompt validation, and it seems to be common practice for such use-cases. Overall, the simplicity and low-latency nature of RESTful communication make it the most suitable choice for these critical, time-sensitive features.

Still, given the distributed nature of the system, certain operations—especially those involved in the trip booking saga—can introduce some delay before their final status is known. To address this, the train search service also maintains a WebSocket connection with the client application. This persistent, bidirectional communication channel enables the service to push real-time updates to the user interface regarding the status of a booking. Notifications such as successful bookings, failed payments due to insufficient funds, or failures caused by downstream service outages (e.g., ticketing or notification services) are received from the central orchestrator and immediately forwarded to the client. This design ensures that users remain informed about their booking's progress without needing to manually refresh or poll for updates, improving responsiveness and overall user experience.

3.1.3 Booking Service

Description Once a user selects a journey via the train search service, the train booking service takes over to secure the booking. It notifies each relevant train operator about the specific journey component that requires a seat to be reserved. The operators then mark the seat as booked until payment for the journey is processed; afterward, the seat status is updated to occupied. Once all operators confirm that the journey components can be reserved, the booking service forwards a ticketing request to the ticket service through a dedicated API endpoint.

Communication with other services All interactions within the booking process are mainly conducted synchronously. The service receives booking requests from the search service, then queries each train operator service to reserve seats, and finally issues a ticketing request transmitted by the

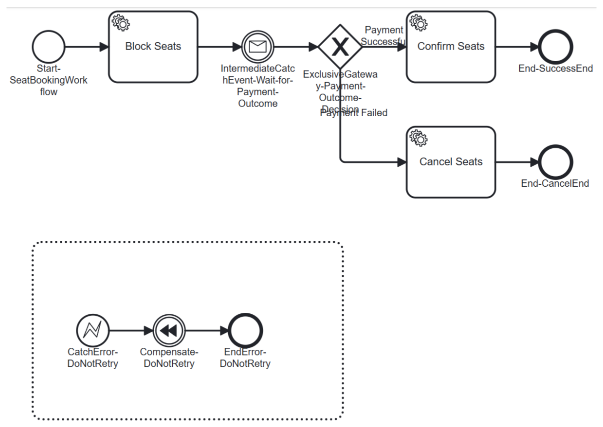


Figure 1: Camunda workflow of the seat reservation process

orchestrator. Synchronous communication is preferable in this context because it guarantees a controlled, step-by-step execution, crucial for coordinating multi-component bookings. This deterministic sequence simplifies error detection and handling during the booking process, ensuring that any failure in reserving a seat can be immediately addressed with compensation actions. Such tight coordination would be more challenging with asynchronous messaging, which could introduce delays and require additional mechanisms to maintain transactional consistency.

3.1.4 Ticketing Service

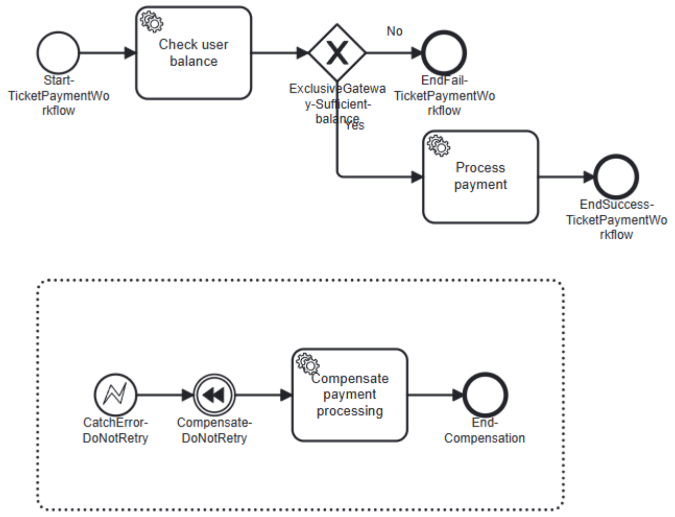
Description Once a route's journeys have been confirmed as bookable, the ticketing service aggregates all relevant journey information and passenger details into a single ticket. The service then notifies the payment service that the ticket is ready to be issued. Upon successful payment—where funds are transferred for all relevant journey legs—the ticket is sent to the user. To deliver the ticket, the ticketing service publishes it onto a message queue for the notification service, which subsequently sends it via email to the address specified on the ticket.

Communication with other services The ticketing service employs a mix of synchronous and asynchronous communication. Synchronous RESTful API calls are used when interacting with the booking and payment services (via the orchestrator), ensuring immediate confirmation and consistent transactional state during the critical steps of ticket issuance and payment processing. This direct, real-time interaction is essential for maintaining control over the sequential operations. In contrast, asynchronous communication is utilized with the notification service through a message queue, allowing the email dispatch process to be handled independently. This separation ensures that the delivery of the ticket does not impede the overall workflow, while still providing reliable and scalable notification processing.

3.1.5 Payment Service

Description After confirming that all journeys can be booked and the ticket can be printed, the main saga prompts the payment service to transfer funds from the user's account balance to the corresponding train operators - as depicted in the mini-saga of Figure 2. For each operator, the service dispatches payment instructions corresponding to the ticket price of the journeys they provide. Once payments for all journey segments are successfully processed, the payment service - via the orchestrator - notifies the ticket service to publish the already created ticket onto the message queue for the notification service, which then delivers it to the user.

Communication with other services The payment service uses synchronous RESTful communication for all interactions. Each transaction is initiated directly by the orchestrator, with the payment service performing immediate, blocking calls to the train operator APIs to execute the payments. This approach ensures that every payment is fully validated and confirmed in real time, which is



Start-TicketPaymentWorkflow
Check user balance
ExclusiveGateway-Sufficient-balance
No
Yes
EndFail-TicketPaymentWorkflow
Process payment
EndSuccess-TicketPaymentWorkflow
CatchError-DoNotRetry
Compensate-DoNotRetry
Compensate payment processing
End-Compensation

Figure 2: Camunda workflow of payment process

crucial for financial operations where accuracy and consistency are paramount. Synchronous communication eliminates the potential delays and complexity of managing eventual consistency that come with asynchronous methods. It also reduces the risk of duplicate transactions or lost messages, as each call provides a clear and immediate response that can be used to enforce atomicity and trigger compensation actions if necessary.

3.1.6 User Service

Description The user service stores all data for registered users and functions as the main authentication and access control broker. It manages user registration, profile management, and authentication via email and password. Upon successful login, the service generates a JWT token that authorizes users for subsequent operations, particularly within the ticket booking saga, ensuring secure interactions throughout the system.

Communication with other services All interactions with the user service are conducted synchronously via RESTful calls. This synchronous approach is favored because authentication requires immediate validation and token issuance, which are essential for secure access control. Real-time responses ensure that only authenticated users can proceed with actions like booking and account management. In contrast, asynchronous communication would introduce delays and complicate error handling, potentially compromising security and user experience. RESTful APIs offer a standardized, easy-to-integrate interface that meets the low-latency and high-reliability demands of authentication processes.

3.1.7 Notification Service

Description The notification service is designed to transmit relevant data to passengers via email, including tickets, reminders of upcoming trips, and updates about delays or journey modifications.

Communication with other services The service utilizes an asynchronous publisher-subscriber model by subscribing to multiple message queues based on the type of notification. When a message is received, the service processes it and dispatches an email to the intended passenger. The choice of asynchronous communication over synchronous methods is motivated by the need to decouple processing tasks, enabling parallel execution and incorporating retry mechanisms without stalling the entire workflow. This architecture facilitates load balancing and fault tolerance by isolating issues within individual message handlers. In addition, using message queues supports horizontal scaling, as additional workers can be deployed to manage increased loads. This approach results in improved throughput, enhanced error isolation, and a more resilient system under high-demand conditions. Conversely, a synchronous model would not really be feasible for this use case, as each notification request would block the process until a response is received, which increases overall latency and risks system-wide delays if a downstream component—such as the mail server—experiences high response times or failures.

3.2 Architecture

At a high level, the TrainCatcher system is built using a microservices architecture, in which each service is responsible for a specific business capability and communicates with others through well-defined APIs. The system integrates both synchronous and asynchronous communication mechanisms, balancing the need for real-time responsiveness with the benefits of decoupling and fault tolerance. The architecture (depicted in Figure 3) is orchestrated centrally and deployed within a containerized cloud-native environment, leveraging several core technologies to ensure scalability, discoverability, observability, and resilience.

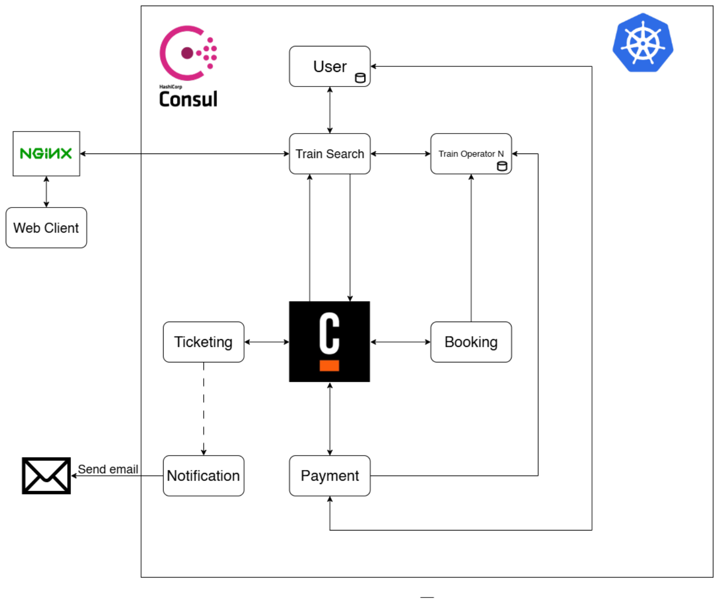


Figure 3: Architecture of the TrainCatcher system

Camunda: Orchestration and Saga Management Camunda plays a central role in orchestrating complex workflows that span multiple services. It coordinates the main booking saga, including sub-sagas such as seat reservation and payment. Each step in the saga is defined as a service task in a BPMN model, which ensures that compensation mechanisms can be triggered in case of failure. Camunda provides transparency, traceability, and reliability for long-running transactions across distributed services—making it an ideal choice for implementing the Orchestrated Saga pattern.

Consul: Service Discovery Consul is used for service discovery, allowing microservices to dynamically register themselves and discover other services without hardcoded configurations, as well as for health checking services using Spring Actuator. This is particularly beneficial in our microservice architecture where services may scale horizontally or change their cluster IPs frequently. By leveraging Consul, the system avoids the rigidity of static configuration and promotes flexibility, resilience, and automation in service interactions.

Kubernetes: Container Orchestration All services are containerized and deployed on a Kubernetes cluster. Kubernetes manages the deployment, scaling, and health monitoring of services. It ensures high availability, load balancing, and fault tolerance across service instances. The choice of

Kubernetes supports the project's goal of building a cloud-native, production-grade architecture with minimal manual intervention.

NGINX: API Gateway and Reverse Proxy NGINX acts as a reverse proxy and API gateway in the architecture. It routes incoming client requests to the appropriate services, enforces security policies, and manages load balancing. By placing NGINX at the edge of the system, we centralize request handling and simplify the integration of cross-cutting concerns such as rate limiting, logging, and authentication forwarding.

Mailhog: Mock Email Server Mailhog acts as an email interceptor, allowing us to mock an SMTP server. It is used to display the tickets send via email to the users that initiate a booking request.

Communication Model The communication architecture consists of a hybrid model:

- **Synchronous (RESTful):** Used for authentication, journey search, booking coordination, and financial transactions—where synchronous communication protocols are required to ensure atomicity.
- **Asynchronous (Message Queues):** Used for ticket delivery, where decoupling, scalability, and resilience are prioritized. Asynchronous communication is used only for features that support eventual consistency.
- **WebSockets:** Maintained between the Train Search Service and client to provide feedback on the ticket ordering workflow.

The selected technologies and architectural choices reflect a design that prioritizes modularity, resilience, and responsiveness. The microservices model aligns with the functional decomposition of the business domain, while tools like Camunda, Consul, Kubernetes, and NGINX provide the necessary infrastructure for managing complexity in a distributed system. By combining synchronous communication for transactional integrity with asynchronous patterns for scalability and decoupling, the architecture balances reliability with performance and user experience.

4 Design Decisions

4.1 Orchestration

Orchestration vs. Choreography

When designing the coordination strategy for the TrainCatcher system, we considered both orchestration and choreography as potential paradigms for managing business processes. Ultimately, we chose to adopt an orchestration-based approach, as it better aligns with the complexity, reliability requirements, and centralized control needed in the context of train trip planning across multiple operators.

Choreography—where each service autonomously reacts to events and publishes its own—is often praised for its decentralized nature, scalability, and flexibility. However, it comes with significant drawbacks when applied to processes that are highly dependent on transactional updates such as database-spanning writes. In the case of TrainCatcher, a typical trip booking spans several critical operations: route selection, seat reservation with multiple train operators, ticket generation, payment verification, and email dispatch. These operations must occur in a specific order, often with compensation mechanisms required to roll back partial progress in case of failure. Implementing such fine-grained coordination using choreography would demand extensive error handling logic within each microservice, making the system more fragile, harder to monitor, and difficult to evolve. Moreover, using asynchronous methods of communication would imply eventual consistency that is not suitable for atomic processes such as payments. There are tools such as Eventuate that abstract away some distributed transaction behaviors, but we found it to be more of a workaround rather than a solution, especially for our needs.

Instead, TrainCatcher adopts a centralized orchestration model using Camunda to manage the main booking saga and related sub-sagas (e.g., seat reservation and payment). With orchestration, the overall flow is explicitly defined in a single BPMN process model, and Camunda coordinates the execution of tasks by invoking the appropriate services in sequence. This design allows for:

- **Centralized Error Handling and Compensation:** Failures such as insufficient funds or unavailable seats can be handled through well-defined compensation steps directly within the orchestrator.
- **Improved Observability and Monitoring:** Through Camunda Cockpit, we can visually trace the state of any ongoing process, facilitating easier debugging and system understanding.
- **Clear Process Ownership and Maintainability:** The business logic is consolidated into a single orchestrator component rather than being scattered across multiple services, improving maintainability and consistency.
- **Better Fit for Sequential, Transactional Workflows:** The linear and conditional nature of booking workflows benefits from a deterministic execution order, which orchestration enforces by design.

In our specific context, where the application must coordinate external services (e.g., operator APIs), process financial transactions, and ensure the delivery of final outcomes (i.e., tickets), orchestration provides the guarantees and control mechanisms necessary to maintain system integrity and user trust. While choreography might scale better in loosely coupled systems with high autonomy and predominant read-only operations, the interdependence of actions and heavy reliance on atomic updates in TrainCatcher's workflows makes orchestration the more suitable and pragmatic choice, in our opinion.

Business Flow Orchestration

Our application leverages Camunda to orchestrate one main business flow, along with two complementary flows (Payment process, see Figure 2 and Booking process, see Figure TODO). In the ticket booking flow (depicted in Figure 4), a logged-in user searches for a route and selects one to book. The booking service then sends booking requests to each relevant train operator, providing details on the journey components that require seat reservations. Upon confirmation that all segments are available (i.e. seats are available), the operators mark the journeys as pending an additional booking (seat reservation), and the booking service issues a ticket creation request to the ticketing service. The ticketing service aggregates all journey and passenger information into a single ticket and then calls on the payment service—our pivot process—to process the payments for each journey leg, as depicted in Figure 2, under the Process payment service task. Once all payments are confirmed, the payment service notifies the main booking saga through a Receive Task (asynchronous message that can be sent between processes) about its return state. Once the validity of the payment is confirmed, the reserved train seats are finally confirmed for each journey leg. The last step is then to publish the ticket on a message queue for the notification service to deliver via email.

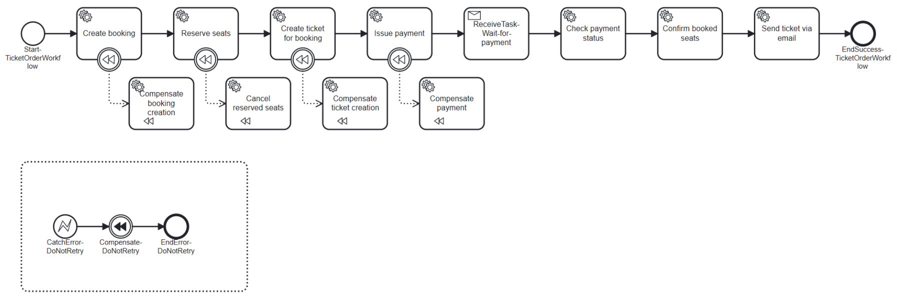


Figure 4: Camunda workflow of main process. Triggered when a user orders a ticket from the Train Search dashboard.

Rationale for Using Camunda

We selected Camunda for orchestration instead of creating an orchestrating process entirely in Java due to its robust workflow management and monitoring capabilities. Camunda's BPMN engine provides a clear, visual representation of the sequential and conditional steps involved in the ticket booking saga, facilitating easier adjustments and error handling. While a pure Java implementation might offer tighter control over each component, it would require extensive custom development to manage compensating transactions, retries, and state monitoring, which would also have to be tested to ensure functional correctness. Camunda simplifies these aspects through tried-and-tested built-in features, reducing development overhead. For the scope of this project—where the focus lies in demonstrating a service-oriented architecture—Camunda offers an optimal balance between flexibility and operational transparency.

Instead of defining BPMN models using Camunda Modeler, we have built a Java-based model builder class which allows us to programmatically define BPMN workflows and deploy them to the embedded Camunda engine. The main advantage of this approach is that we can now more easily debug workflows both visually through Camunda Cockpit, as well as programmatically through actually debugging live code at runtime. Moreover, each deployed workflow is versioned in our git repository, making this approach highly flexible compared to modifying XML models or defining them using a GUI editor like Camunda Modeler.

Pivot Process Design

In designing the orchestration for ticket issuance, we evaluated two primary approaches. In Option 1, the flow follows: booking, ticketing, then payment (serving as the pivot), and finally sending the ticket. In this sequence, the ticket is prepared after booking, but before payment is processed. Here, payment acts as the final go/no-go point, ensuring that funds are only withdrawn if every preceding operation is successful. This approach follows the natural business flow, ensuring that the complete ticket information is confirmed before any financial transactions occur, which minimizes the risk of having to refund payments if later steps fail. Conversely, Option 2 follows: booking, payment, then ticketing (as the pivot) before sending the ticket. This alternative, which aligns more with the process/logical flow, offers the advantage of reducing wait times by processing payment earlier, allowing the ticket to be created and dispatched immediately upon confirmation. However, processing payment too early may lead to situations where, if subsequent operations fail, funds must be returned—adding complexity and potential user dissatisfaction. Ultimately, we chose Option 1 because it best meets our hypothetical business requirements by ensuring transactional consistency and a more robust, user-friendly experience.

4.2 Security

Securing the edge services of a microservice architecture is important to ensure that the integrity, confidentiality, and availability of data inside the actual service cluster are maintained. To ensure that our system is secure, we adopted the following measures:

JWT Authentication

Initially, our project proposal showed a potential implementation of OAuth2 using KeyCloak. However, because of the limited time for development and the fact that the project was developed by a small team consisting of two people, we resorted to a more approachable security implementation that does not sacrifice the security of the system, which is JWT tokens. Using JJWT and Spring Security, we created a common library that hosts all the functionality for issuing and verifying JWT tokens across all the services that need it. Moreover, we used AspectJ and Spring Boot's AOP dependency to create a custom annotation "@RequiresAuthentication" that automatically checks that the token present in the HTTP Authorization header is valid and corresponds to the given userId. The annotation was used to secure all the service operations that lead to internal service calls, all of which reside on the single entry point to our service architecture: the Train Search service.

The requirements created for this project do not specifically identify the need for role-based authorization; however, by using JWT, authorization can easily be integrated. Moreover, using the commonly defined library, all the microservices in our cluster have access to a shared JWT verification utility which simplifies the development process of adding security in a given microservice.

Lightweight Docker Images

A recommended practice in containerized environments is to create container images that are lightweight, shell-less, and root-less, to ensure that even if access is gained to an internal service, there are little to no attack vectors that an attacker could exploit. In our case, we used the Alpine-flavored docker images with Eclipse-Temurin as a base for our containers because of their reduced size and default security features. For the purpose of this project, we did not resort to removing the shell or root user from these base images; however, in a production environment, this is a highly recommended practice, which can also be accompanied by blacklisting unused ports and ensuring tight security at the API Gateway level to protect the internal cluster of services as much as possible.

4.3 Technology Stack and Communication Protocols

Table 1: Technologies often used for SoA along with design rationales of use / non-use

Service / Component	**Technology**	**Rationale for Use / Non-Use**
Most Microservices	**REST (JSON)**	Used for synchronous communication; lightweight, standard, and easy to integrate and document. JSON preferred over XML for better readability and common support.
Notification Service	**Message Queue**	Used for asynchronous communication to decouple services. Improve scalability and fault tolerance during ticket delivery.
Train Search Service	**WebSockets**	Used to provide real-time booking status updates to users without polling, enhancing user experience.
All Services	**SOAP / XML**	Not used due to complexity and overhead. REST with JSON provides simpler and more modern alternative.
All Services	**GraphQL**	Not used as our use cases are request-driven and do not require flexible querying; REST is more appropriate and widely supported.
Infrastructure	**YAML**	Used to define Kubernetes manifests, Docker Compose files for service deployment and configuration files for Spring.
Business Process Coordination	**Orchestration (Camunda)**	Used to implement orchestrated sagas, providing centralized control, visibility, and error handling for complex multi-service workflows.
Business Process Coordination	**Choreography**	Possible through frameworks like Eventuate, not needed for our use-case due to the need for synchronized request-responses.
Deployment	**Docker**	Used to containerize all microservices, ensuring consistent runtime environments.
Service Discovery	**Consul**	Used for dynamic service registration and discovery, K/V storage and health monitoring.
API Gateway	**NGINX**	Used as a reverse proxy to route client requests, enforce access rules, and balance load.
Container Orchestration	**Kubernetes**	Used to manage deployments, scaling, and service availability; required by project specification.
Container Orchestration	**Docker Swarm**	Not used because Kubernetes offers more robust features and was a mandatory project requirement.

5 Testing

System Testing

To ensure that our system is validated accordingly, we created a test suite of system tests that target the distributed functionality of our overall architecture. Each test is defined so that we can test requirements such as: functional correctness, system robustness against service failures, compensation activities that lead to a consistent state, and retry policies. To make our tests understandable, we apply the Behaviour Driven Design practice which states that tests must take the form of Given-When-Then scenarios.

Scenario 1: User books a ticket with insufficient funds.

Given the user has insufficient funds for the requested train trip

When the user requests a booking for the trip

Then the application should refuse the operation due to insufficient funds, inform the user about the issue and compensate the activities of creating a ticket and reserving a seat.

`Result` the system sends the request. The system immediately responds with the error seen in Figure 5.

Scenario 2: User books a ticket with sufficient funds.

Given the user has sufficient funds for the requested train trip

When the user requests a booking for the trip

Then the application should complete the full ticket booking workflow, inform the user that his ticket is to be sent via email and send it to Mailhog.

`Result` the system sends the request. The system immediately responds with the success message seen in Figure 9. The ticket is sent to the user's inbox which can be seen in Figure 11.

Scenario 3: User books a ticket with sufficient funds but the booking/ticket/payment services are down

Given the user has sufficient funds for the requested train trip, but one of the following services is down: booking, ticket, payment

When the user requests a booking for the trip

Then the application should complete the ticket booking workflow up until it hits the down service(s). The workflow step should be retried two more times at a 5 second interval (PT5S according to ISO-8601). If the retries still fail, the user should receive an informative error message and all the succeeded service tasks must be compensated.

`Results` the system sends the request. Depending on which service is down, the following error messages and behavior are observed:

Booking service down An informative error message is shown (see Figure 6). The compensation activity for booking is tried. Since booking is the first step in the orchestration, no other compensation is triggered. No booking is created in the system and no seats are blocked/reserved because the booking service is offline. The state is not affected because no writes are done, therefore the system state is consistent.

Ticket service down An informative error message is shown (see Figure 7). The compensation activity for booking is triggered and succeeds, therefore leaving the system in a consistent state. The workflow ends at the ticket service, and after 2 more retires, the error is sent to the user's dashboard.

Payment service down An informative error message is shown (see Figure 8). The compensation activities for booking and ticketing are triggered and succeed, therefore leaving the system in a consistent state. The workflow ends at the payment service, and after 2 more retires, the error is sent to the user's dashboard.

Scenario 4: User books a ticket with sufficient funds but the notification service is down

Given the user has sufficient funds for the requested train trip but one of the following services are down: notification

When the user requests a booking for the trip

Then the application should complete ticket booking workflow up until it hits the down service(s). The workflow should succeed since the notification service is not part of the atomic core of the transaction. The notification service will only consume the email from the message queue once it is back up and it will eventually send the email to the user.

`Result` the application returns a successful message (see Figure 9). The email is not sent, but once the notification service is back up, the message on the MQ is consumed and the email is sent (see Figure 11).

Scenario 5: User tries to book the same ticket 10 times in a short time-span but has money only for 8.

Given the user has sufficient funds for the 8 out of 10 requested train trips

When the user requests the bookings for the trips

Then the application should complete ticket booking workflows in a isolated fashion. Only 8 tickets must be bought and the other 2 must be refused due to insufficient funds. Only 8 emails should be sent to the user's inbox.

`Result` To test this, we assigned 344/430 euros required to buy 8/10 tickets worth 43 euros each. The result was evaluated using Mailhog and the user balance. The user balance reached 0 and only 8/10 workflows succeeded (only 8 tickets were sent to email, see Figure 10).

Scenario 6: A train operator is offline when searching for a route

Given the user is searching for train routes using the search function

When the user requests the routes based on his input parameters

Then the application must only send the routes of the online operators (hence not fail if an operator is down). The application should resume querying the down operator only when it's API is back up.

`Result` A scheduler runs on the train search service to activate/deactivate querying the operator. The operator results are excluded successfully from the search results. When the operator is back up, his journeys are included again the search algorithm.

Scenario 7: A train operator is added to the system during runtime

Given that there are only two train operators registered

When a new train operator requests to register to the Train Search platform

Then the train search service must register its service hostname and use it in its queries.

`Result` The operator is registered and it is now part of the scheduler state space. The operator results are included now in the search results. When the operator is down, scenario 6 shows the expected and actual behavior of the system.

6 Discussion

Overall, we are very satisfied with the outcome of the TrainCatcher project. We successfully implemented all the core business functionalities envisioned in our proposal, such as multi-operator journey search, ticket booking with orchestrated distributed transactions, payment coordination, and notification delivery via email. Beyond the implementation itself, the project served as a valuable learning experience in distributed system design. We deliberately chose architectural patterns and tools that were outside our comfort zone in order to broaden our understanding of service-oriented architecture. Specifically, integrating orchestrated sagas using Camunda and implementing WebSocket communication between the client and backend were completely new territory for both team members, so the practice provided by this project greatly enriched our practical skills with these concepts.

That said, the project did present several notable challenges. The integration of Camunda, while ultimately successful, came with a steep learning curve. Understanding how sagas are modeled and executed—particularly with compensating transactions after a pivot operation—took considerable time and experimentation. Previously, our experience had been limited to monolithic systems where transactional consistency was largely managed by a single database, or by a single RDBMS handling multiple databases with database links. Designing distributed compensation logic manually exposed us to the complexities of ensuring system consistency in a distributed context. Another major challenge was deployment. While we were familiar with Docker at a basic level, writing deployment configurations for a multi-service architecture and transitioning from Docker Compose to Kubernetes proved demanding. Specific difficulties included deploying multiple instances of the same service with isolated state (e.g., different operator databases), and correctly routing requests through NGINX while maintaining secure access and proper service registration in Consul.

Looking forward, we see two promising directions for a hypothetical continuation of TrainCatcher's development. First, we could implement a train monitoring service. This feature would track live train statuses (e.g., delays, cancellations) by periodically querying the operator services or listening for updates. In the event of a service disruption, affected bookings could be flagged, and the notification service could inform users proactively. Second, we would like to simulate the heterogeneous nature of train service providers by having each operator expose its data in a different format (e.g., JSON, XML, SOAP). To handle this, we would introduce an integration layer that normalizes incoming messages and translates internal requests to the proper external protocol, enhancing the realism and complexity of the system.

In conclusion, TrainCatcher not only achieved its functional goals, but also served as a platform for exploring modern distributed system design and cloud-native deployment practices. The challenges and experimentation which came with this project helped us develop a deeper understanding of the trade-offs involved in building scalable, reliable microservice-based systems; and made us gain the confidence to apply these techniques in more complex, real-world scenarios.

7 Appendix A - Links to relevant documentation

OpenAPI

A link to the OpenAPI Specification used in this project can be found at this link: https://vgosa.github.io/train-catcher.

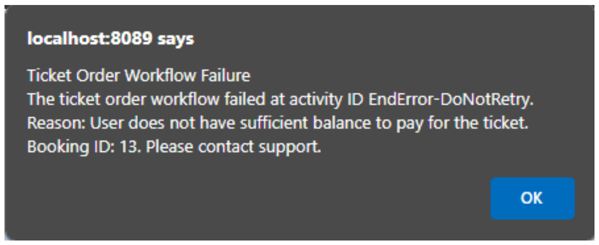
Github Repository

A link to the public Github repository can be found at this link: https://github.com/vgosa/train-catcher

Kubernetes

The Kubernetes cluster is explained in the README.md file present in the submission and Github repository. Clear steps on how to deploy the application for both development and production are explained in the readme.

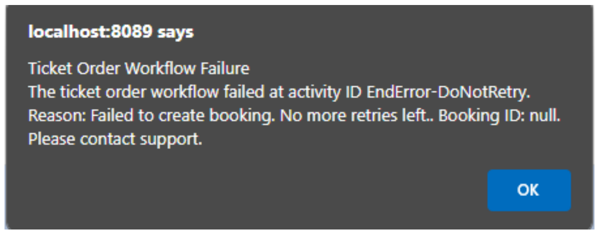
8 Appendix B - Testing results



localhost:8089 says

Ticket Order Workflow Failure
The ticket order workflow failed at activity ID EndError-DoNotRetry.
Reason: User does not have sufficient balance to pay for the ticket.
Booking ID: 13. Please contact support.

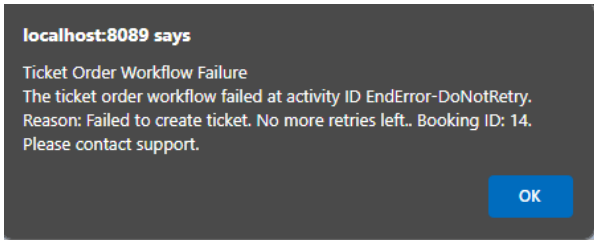
Figure 5: Insufficient funds error message shown to the client



localhost:8089 says

Ticket Order Workflow Failure
The ticket order workflow failed at activity ID EndError-DoNotRetry.
Reason: Failed to create booking. No more retries left.. Booking ID: null.
Please contact support.

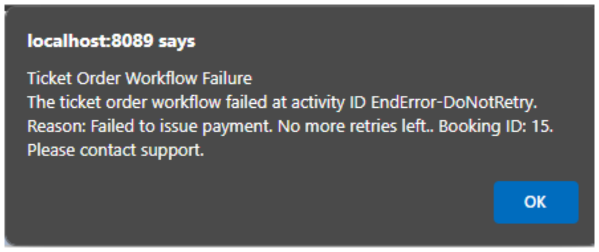
Figure 6: Error message shown when the booking service fails (it is down or an error occurs)



localhost:8089 says

Ticket Order Workflow Failure
The ticket order workflow failed at activity ID EndError-DoNotRetry.
Reason: Failed to create ticket. No more retries left.. Booking ID: 14.
Please contact support.

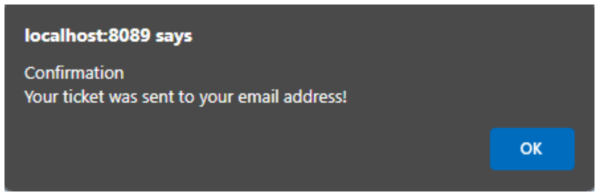
Figure 7: Error message shown when the ticketing service fails (it is down or an error occurs)



localhost:8089 says

Ticket Order Workflow Failure
The ticket order workflow failed at activity ID EndError-DoNotRetry.
Reason: Failed to issue payment. No more retries left.. Booking ID: 15.
Please contact support.

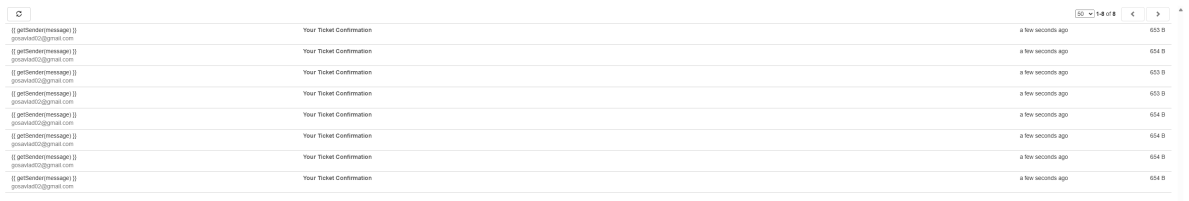
Figure 8: Error message shown when the payment service fails (it is down or an error occurs)



localhost:8089 says

Confirmation
Your ticket was sent to your email address!

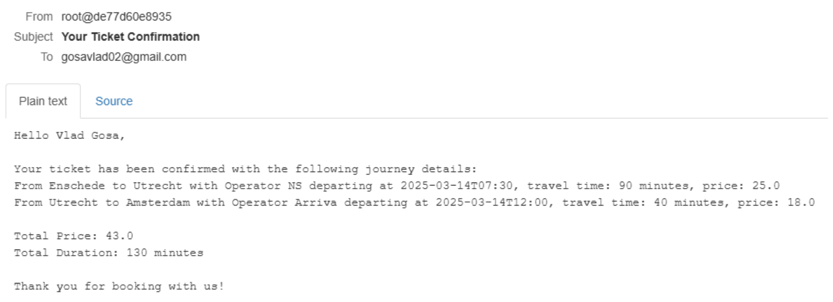
Figure 9: Ticket order workflow successful message shown when the SAGA completes successfully.



50 1-8 of 8

{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	653 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	654 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	653 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	653 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	654 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	654 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	654 B
{{ getSender(message) }} gosavlad02@gmail.com	Your Ticket Confirmation	a few seconds ago	654 B

Figure 10: Snippet for scenario 5 where only 8 emails are sent even though 10 requests are sent in a very short amount of time.



From	root@de77d60e8935
Subject	**Your Ticket Confirmation**
To	gosavlad02@gmail.com

```
Hello Vlad Gosa,

Your ticket has been confirmed with the following journey details:
From Enschede to Utrecht with Operator NS departing at 2025-03-14T07:30, travel time: 90 minutes, price: 25.0
From Utrecht to Amsterdam with Operator Arriva departing at 2025-03-14T12:00, travel time: 40 minutes, price: 18.0

Total Price: 43.0
Total Duration: 130 minutes

Thank you for booking with us!
```

Figure 11: Email received by a user containing his requested ticket