School of Management, Economics,
Law, Social Sciences, International Affairs and Computer Science

Event-driven and Process-oriented Architectures

Group 5, Raffael Rot & Michał Cisło

raffael.rot@ student.unisg.ch | michal.cislo@student.unisg.ch

21th April 2024

Content

1. Project Description

The project encapsulates the supply chain of the company CiRa. Built with Spring Boot, Kafka, and Camunda, our system comprises four key parts: Machine, Factory, Warehouse, and Logistics microservices, each playing a vital role in making manufacturing smarter and more efficient.

- **Machine microservice:**

The Machine microservice lets us manage wood-shaving machine easily. We can turn it on/off and control the start of the production line.

- **Factory microservice:**

The Factory microservice coordinates various tasks including initiating the production line, monitoring machine fill levels, publishing real-time inventory data, and orchestrating complex process, of transporting the goods, using Camunda BPMN workflows. Notably, it employs Camunda to automate the start of production lines and to schedule transfer commands based on factory inventory levels, ensuring optimal resource utilization and production efficiency.

- **Warehouse microservice:**

The Warehouse microservice is responsible for managing the logistics and inventory within the manufacturing facility. It facilitates the confirmation of transport orders, quality checks, stock level updates, and the storage of goods. Additionally, it interfaces with the Factory microservice to synchronize inventory levels and ensure seamless material flow throughout the production process.

- **Logistics microservice:**

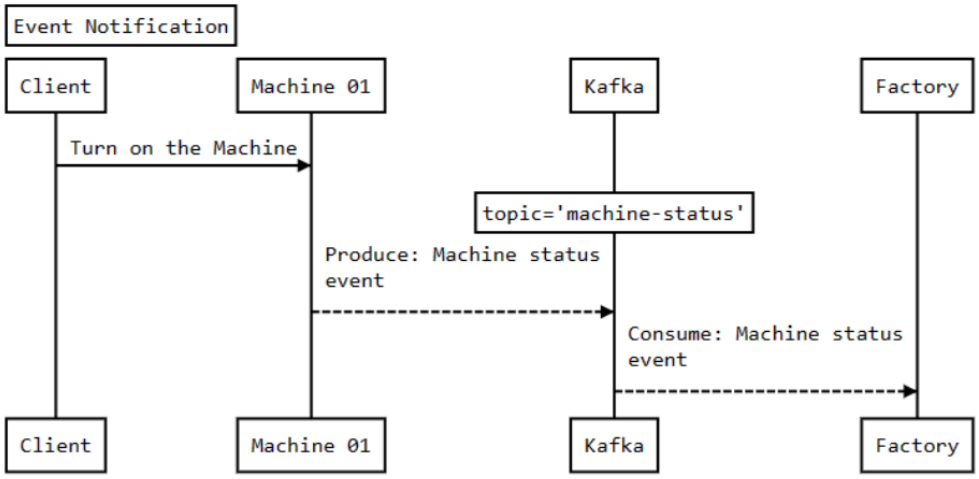
The Logistics part deals with transporting goods between different stages of production. It schedules transfers, picks up goods from the factory, and delivers them to the warehouse. This microservice plays a role in ensuring delivery of goods, optimizing supply chain efficiency.

2. Employed Concepts with diagrams

2.1. Event Notification

Event Notification is a critical feature within the system architecture, facilitating seamless communication and coordination between components. In this scenario, when an employee initiates a machine by issuing a POST request, the machine service promptly emits an event signalling its availability. This event is broadcasted to the designated 'machine-status' topic.

This event-driven approach enhances system responsiveness and agility, enabling real-time updates without necessitating continuous polling or manual intervention.



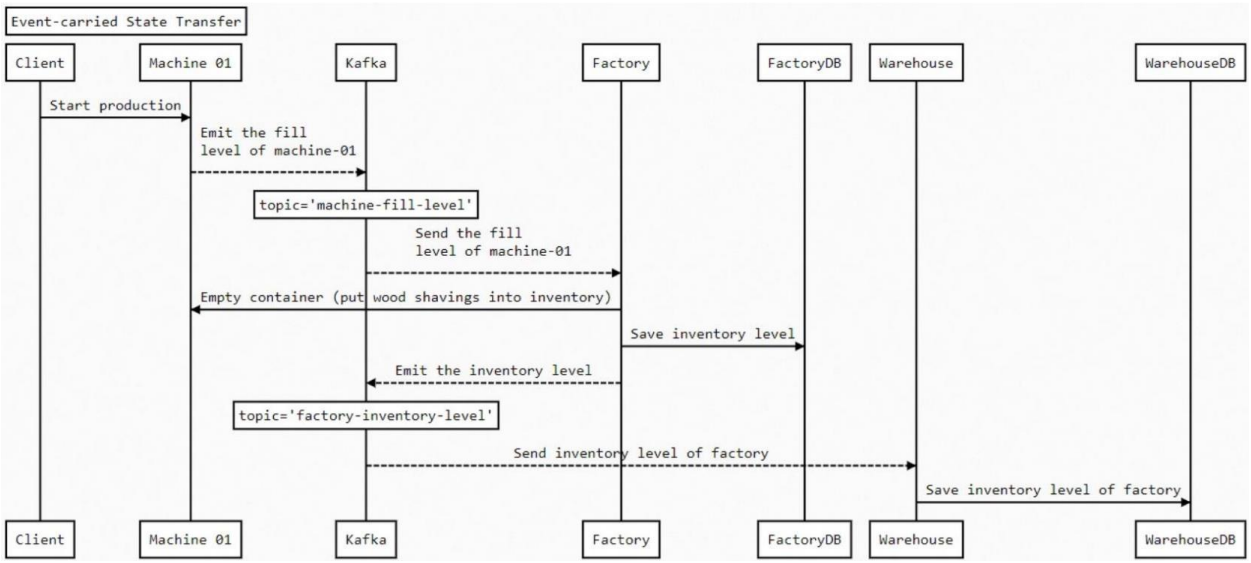
2.2. Event-carried state transfer

Event-carried state transfer is a paradigm in distributed systems architecture where changes in state are communicated through events rather than direct data transfers. In this context, the Factory microservice is responsible for maintaining the real-time inventory levels of produced goods. Once a predefined inventory threshold is met, the Factory emits an event to the 'stock-update' topic. The Warehouse microservice, subscribed to this topic, captures and retains a synchronized copy of the inventory data.

This approach offers significant advantages, including enhanced decoupling between components and reduced computational load on the Factory microservice. However, it necessitates the management of replicated data and introduces eventual consistency considerations, ensuring that all replicas converge to the same state over time. Thus, while promoting scalability and resilience, event-carried state transfer requires careful consideration of trade-offs to ensure system reliability and performance.

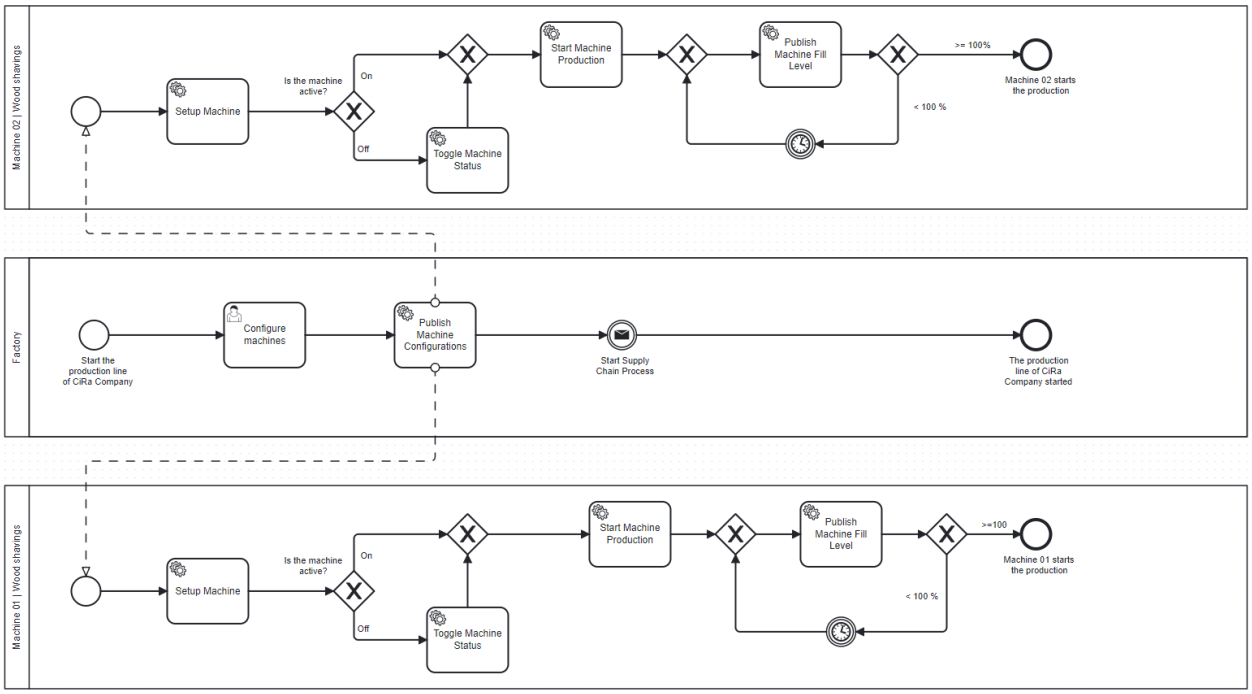
Trade-offs:

- Decoupling and reduced load on Factory
- Replicated data and eventual consistency



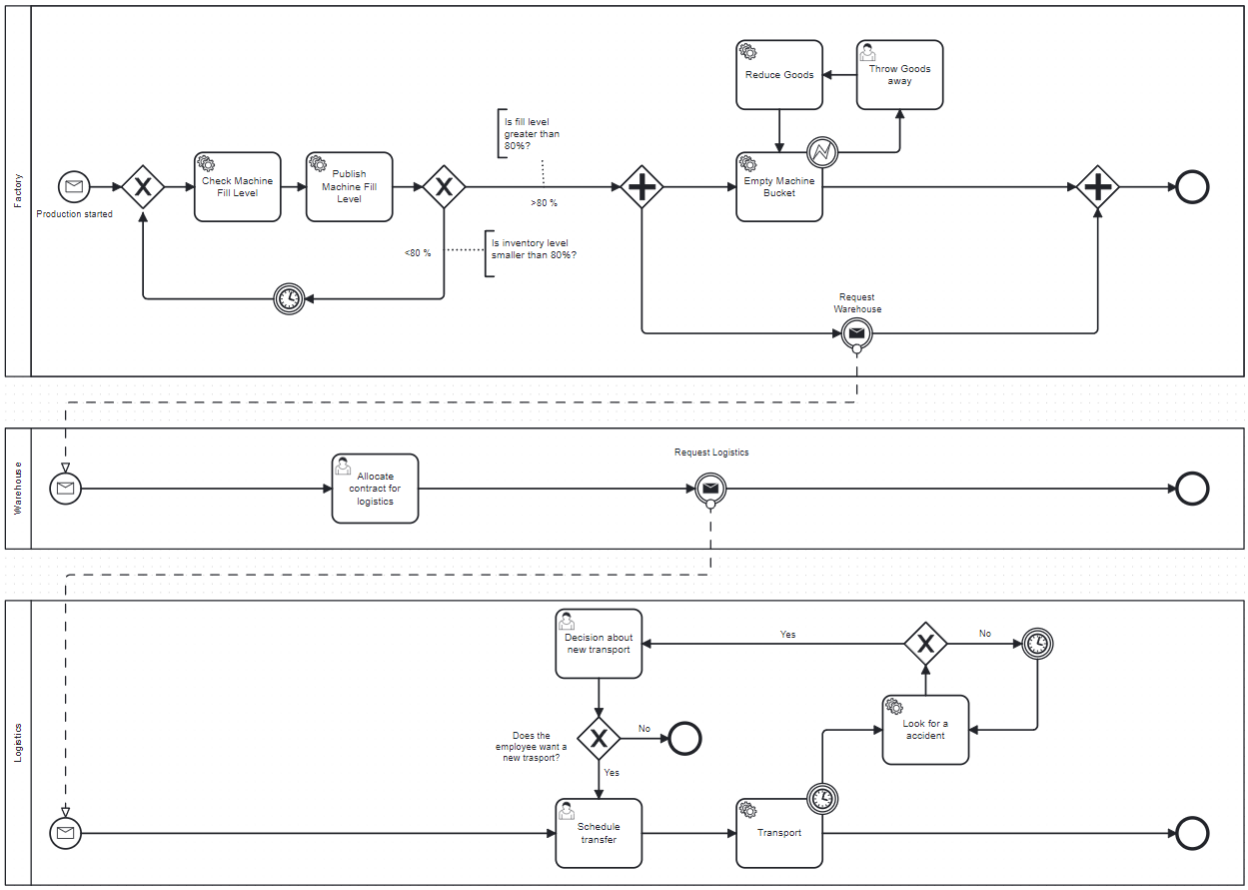
2.3. Parallel Saga

The utilization of a parallel saga pattern within our event-driven and process-oriented architecture is instrumental in ensuring robustness, scalability, and fault tolerance. By employing parallel sagas, we can concurrently execute multiple interdependent calls to our machines. This pattern facilitates asynchronous communication between services, enabling them to operate independently while maintaining consistency across the system.



2.4. Anthology Saga

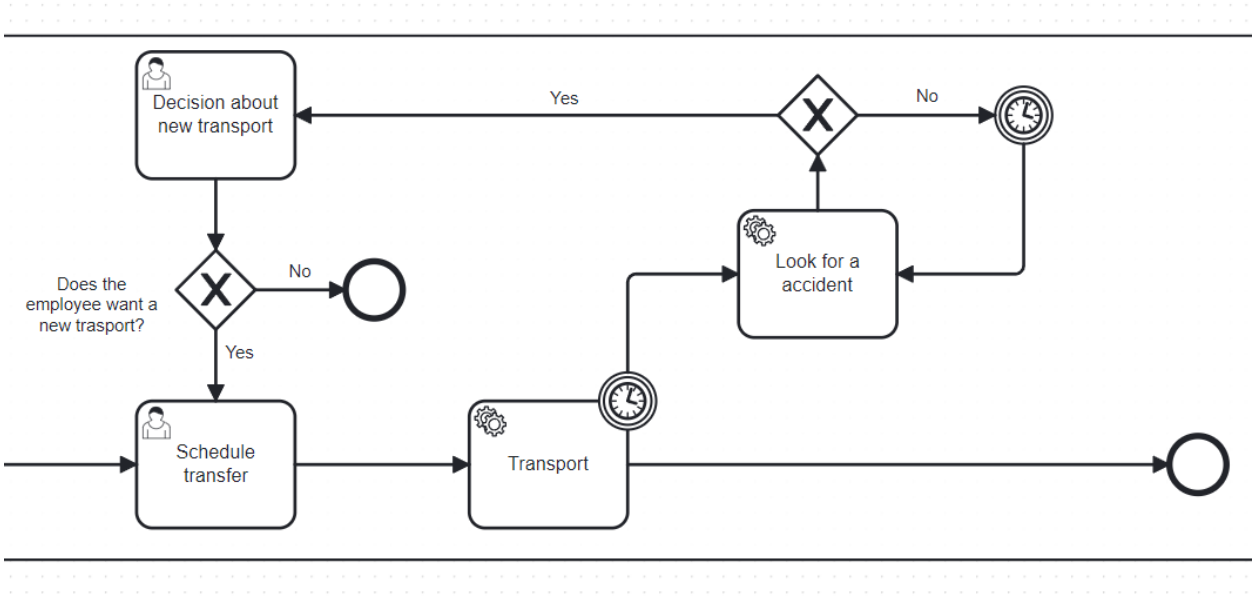
The integration of the Anthology Saga pattern brings substantial advantages in managing complex workflows with multiple related processes. By adopting the Anthology Saga, we can orchestrate cohesive sequences of events across various services, ensuring consistency and reliability throughout. This pattern allows us to group related sagas under a common overarching saga, facilitating better organization and coordination of business processes. With the Anthology Saga, we can handle intricate scenarios with ease, as it provides a structured approach to managing dependencies and interactions between different saga instances. We use this pattern for our “Supply Chain” process.



2.5. Stateful Resilience Pattern: Human Intervention

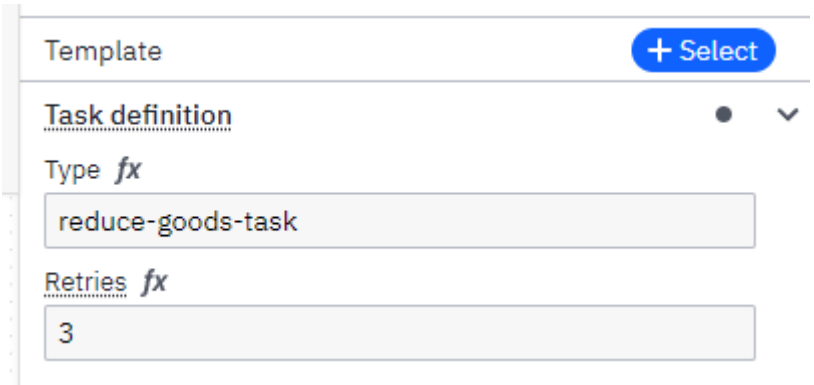
In the Human Intervention pattern the system is designed to gracefully handle exceptional situations by involving human intervention when necessary. When a critical error occurs that cannot be resolved automatically or through traditional retry mechanisms, the system leads to human operators for manual resolution. This pattern ensures that complex or ambiguous errors can be

addressed by human expertise, maintaining the integrity and reliability of the system. In our system this pattern is used to make a decision about a further transfer in case of a transfer truck accident.



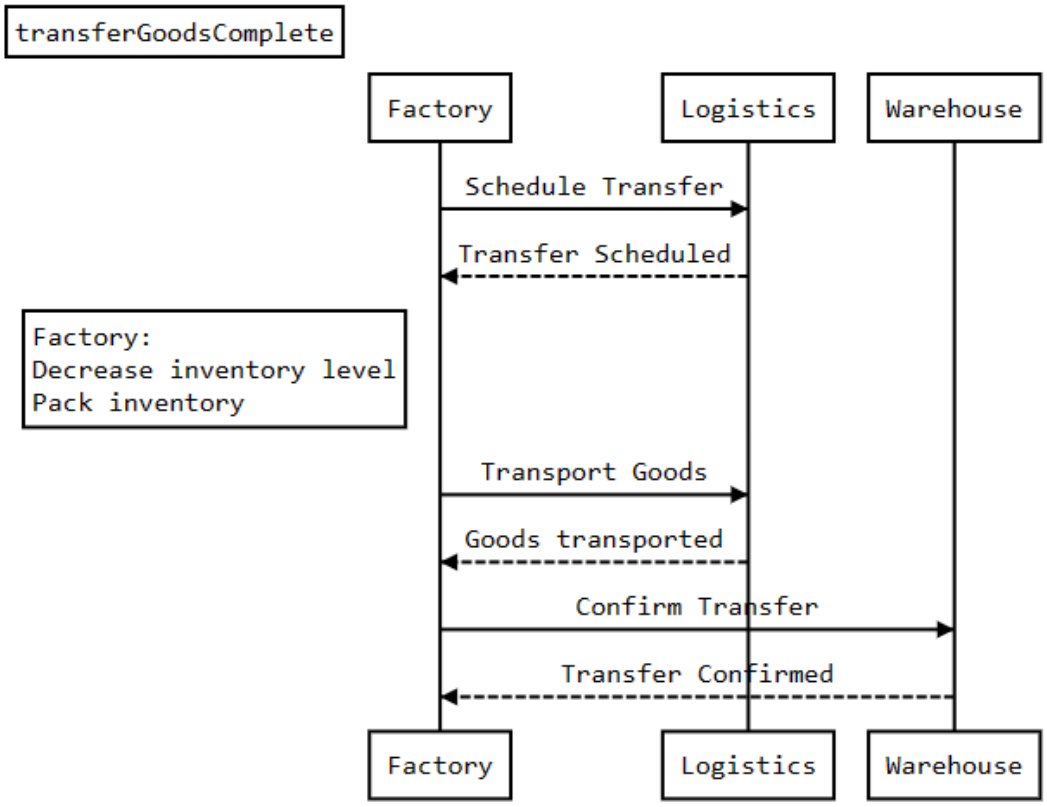
2.6. Stateful Resilience Pattern: Stateful Retry

In the Stateful Retry pattern within event-driven architecture, resilience is achieved by implementing a stateful mechanism for retrying failed service tasks. Specifically, each service task is retried up to three times upon encountering a failure. This approach aims to improve the robustness and fault tolerance of the system by allowing failed tasks to be automatically retried.



3. Implementation diagrams

transferGoodsComplete Process



4. ADRs

4.1. Selection of Microservice Architecture

...

Decision:

For this, we use the microservice architecture. The microservice architecture offers a flexible experimentation environment.

...

Github link to full ADR

4.2. Single Table Ownership for Bounded Contexts

...

Context:

Single table ownership refers to the principle where each microservice exclusively manages and owns its database. In this assignment, we have to create an event pattern.

...

Github link to full ADR

4.3. Code sharing

...

Decision:

We decided against implementing shared libraries for Kafka configuration at this stage of the project. However, as the project is expected to grow, this decision will be revisited and evaluated again in the future.

Consequences:

Potential for increased code duplication: Without the implementation of a shared library, there is a risk of continued code duplication in Kafka configuration across different modules of the application.

Potential for inconsistent configuration: The absence of a centralized configuration approach may result in inconsistencies in Kafka configuration settings across different parts of the codebase.

Flexibility for current needs: By not implementing a shared library for Kafka configuration at this stage, we can maintain flexibility to address immediate needs and priorities without the overhead of designing, implementing, and integrating a shared library.

...

Github link to full ADR

4.4. Continue with Camunda 7 or Camunda 8

...

Decision:

After evaluation, we have decided to migrate from Camunda 7 to Camunda 8. This decision is based on several factors outlined below:

- Desire to experiment: We had been using Camunda 7 since the beginning of the semester, so the moment we had the opportunity to use Camunda 8 and learn about the changes and differences between the versions, we decided to migrate.
- New skills: Camunda 8 introduced new developments to its engine. We decided to switch to Camunda 8 in order to gain more skills in using the platform and the new developments in the code.
- Future-proof solution: After considering the scalability and architectural differences between the Zeebe engine in Camunda 8 and the Camunda 7 engine, we've decided to use to Camunda 8. This decision is based on the scalability of Zeebe, does not rely on a relational database, eliminating its major bottleneck for scaling the engine.

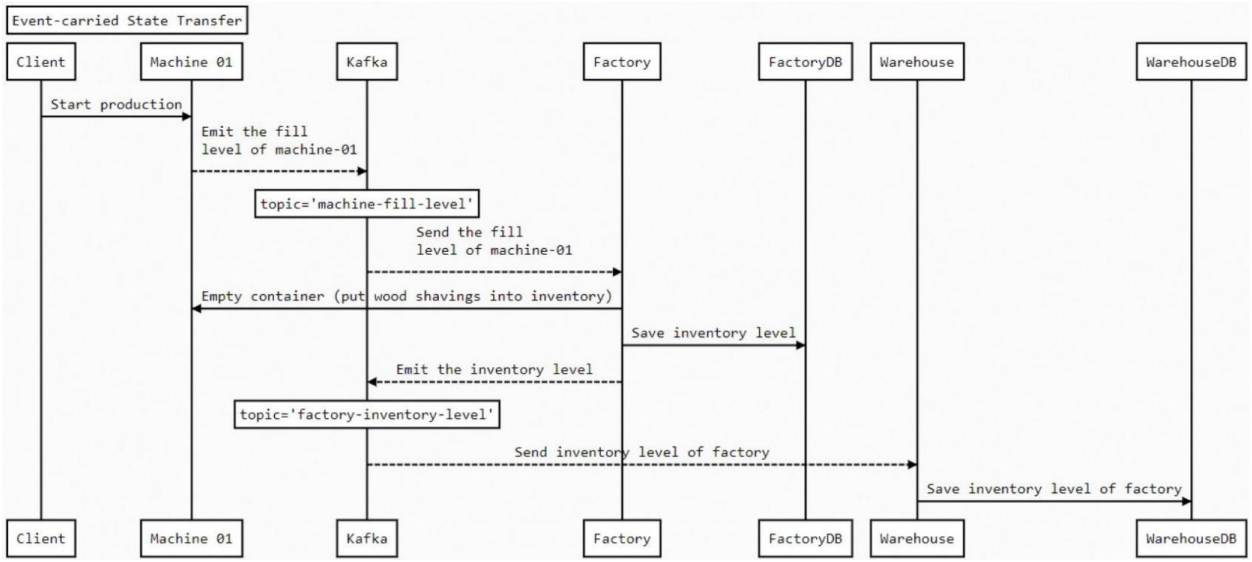
...

Github link to full ADR

5. Trade-offs

Asynchronous Communication - Eventual Consistency: In the context of event-driven and process-oriented architecture, asynchronous communication allows systems to operate without waiting for immediate responses, leading to higher throughput and reduced latency. However, this approach requires handling eventual consistency, meaning that data might not be instantly synchronized across all parts of the system.

Example from the project:

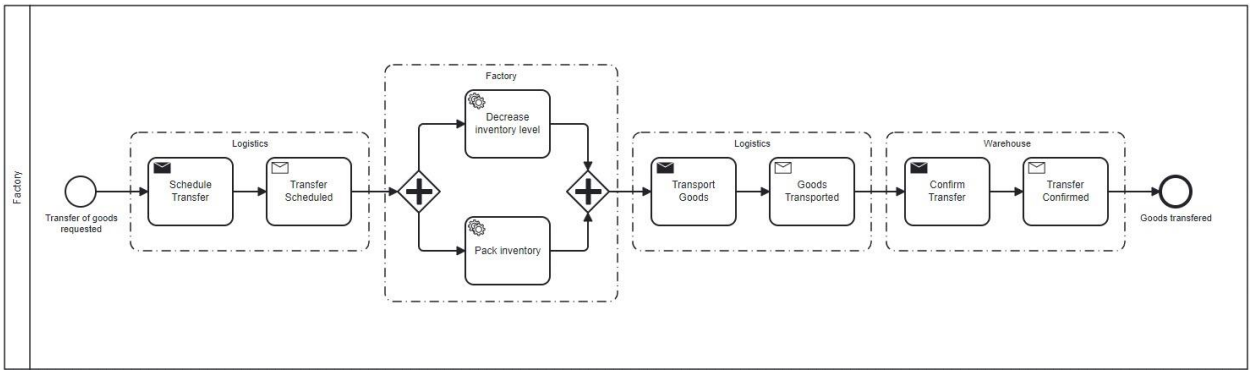


We try to leverage asynchronous communication and in the same time we are aware that we end up with eventual consistency in Warehouse microservice which might not have a synchronized 'inventory level' at every moment of a running application.

Parallel saga and anthology saga contribute to achieving high responsiveness and scalability by supporting low coupling between components. Parallel saga enables concurrent execution of tasks, allowing systems to respond to user requests or events, thus enhancing the user experience. Meanwhile, anthology saga facilitates horizontal scaling, ensuring the system can efficiently handle growing workloads or dynamic demands without compromising performance. This combination of parallel and anthology sagas optimizes responsiveness and scalability, laying the foundation for robust and adaptable event-driven architectures.

Balancing Orchestration and Choreography: In our system, we faced an important trade-off between orchestration and choreography. To address the challenges posed by a complex process, we employed a combination of commands and events. By utilizing events for communication and commands for control, we've achieved a delicate balance between centralized coordination and decentralized autonomy. Our approach is based on an understanding of responsibilities, guiding us in determining whether to use commands or events. We've assessed our process using a simple rule: if it's acceptable for a component to emit an event that might be ignored, it's likely an event; otherwise, it's probably a command.

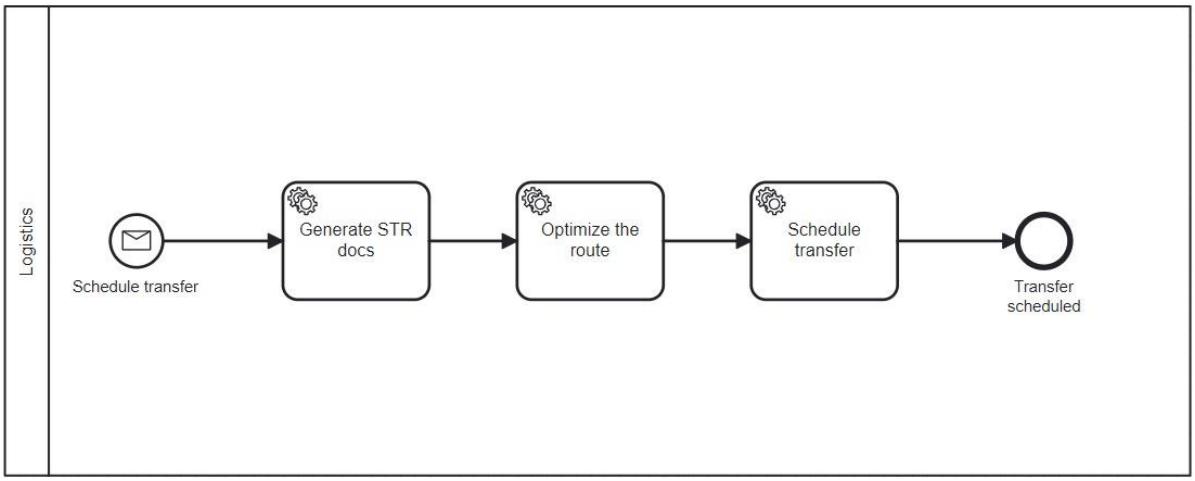
Commands vs Events: Commands represent explicit directives to perform a specific action, while events signify something that has happened in the system. Understanding the distinction between the two helps in defining clear responsibilities for components and enables efficient communication between different parts of the architecture. Commands drive the state changes in the system, while events communicate those changes to interested parties, facilitating loose coupling and flexibility. The employment of this trade-off can be seen in the *"transferGoodsComplete"* process.



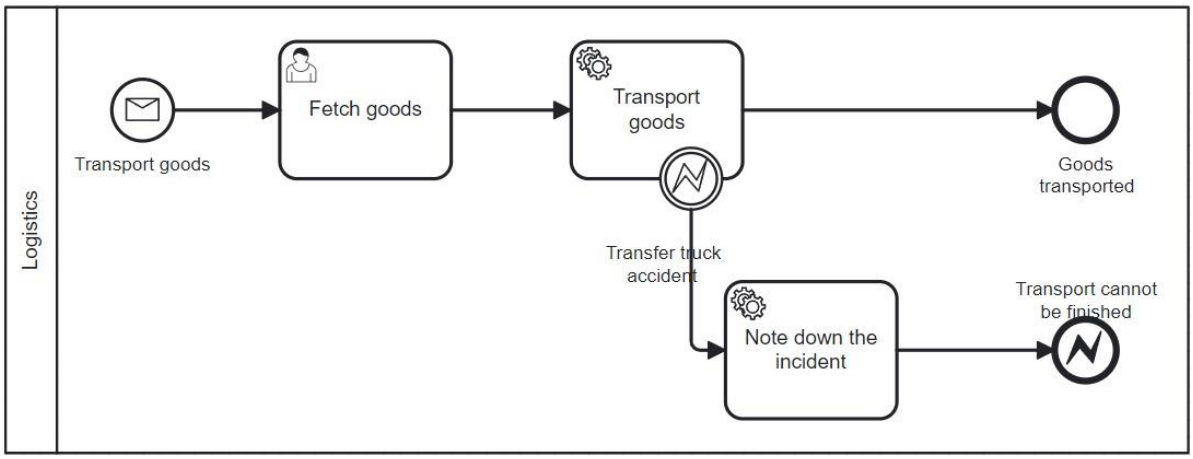
Respect Boundaries and Avoid Process Monolith: Avoiding process monoliths involves breaking down complex systems into smaller, manageable components with well-defined boundaries. This approach promotes modularity, allowing for independent development, deployment, and scaling different parts of the system. By distributing responsibilities and isolating concerns, a team can achieve better maintainability and scalability in the architecture.

Example from the project: We have divided a huge “transferGoodsComplete” process into smaller processes respecting their bounded contexts (microservices).

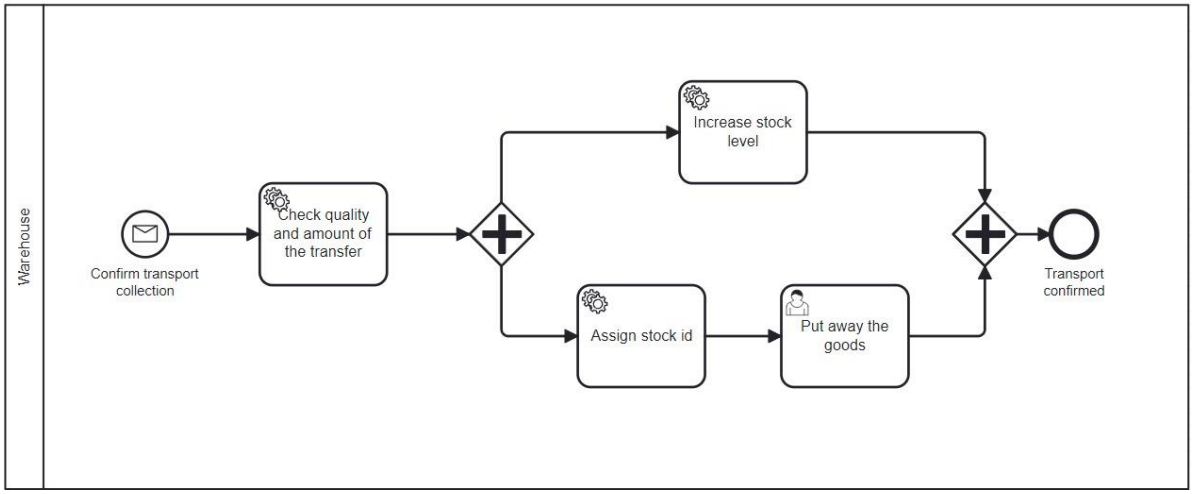
Schedule transfer:



Transport goods:



Confirm transport collection:



6. Kafka experiments

Through testing various aspects of Kafka, several additional results and insights have emerged.

The outage of Zookeper test:

ZooKeeper has a critical role in a Kafka cluster by providing distributed coordination and synchronization services. It maintains the cluster's metadata, manages leader elections, and enables consumers to track their consumption progress. Conclusion - Zookeeper enables consumers to track their consumption progress, so if the Zookeeper is not working consumers cannot continue to consume messages after they have to be for example restarted.

The risk of data loss due to offset misconfigurations test:

In that we have tested several configuration options. Some of the conclusions –

1. *auto.offset.reset* = *earliest* and 2. *auto.offset.reset* = *latest* both duplicated messages and started again after Consumer was up from the Last committed offset.

auto.offset.reset - This property controls the behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid. This is the reason why both options =true and =false started from the last committed offset (It was valid committed offset).

3. *enable.auto.commit* = *false* with *auto.offset.reset* = *earliest* this test did not commit any offset and since the offset was invalid when the Consumer was up again it started reading messages from the beginning.

Acknowledgement configuration test:

Based on the conducted Kafka experiments in Spring Boot, several conclusions can be drawn. Firstly, with a replica factor of 1 and one broker, there is a marginal difference in the time taken to send 100,000 messages regardless of the acknowledgment setting, with acks = 0 showing slightly better performance. When introducing a second broker with the same replica factor, again, there's minimal

variation in message sending time among acknowledgment settings, with acks = 0 consistently being the fastest. This is due to the fact that in both cases we only used --replication-factor 1.

However, with a replica factor of 2 and 2 brokers, there's a noticeable increase in the time taken, especially when acknowledging all replicas. This trend continues as the replica factor and number of brokers increase, indicating a trade-off between data redundancy and message throughput. Notably, as redundancy increases, so does message transmission time, particularly evident with acknowledgment of all replicas. Therefore, depending on the application's requirements, the choice of replica factor and acknowledgment setting should be carefully considered to strike a balance between fault tolerance and performance.

Link to all 7 Kafka experiments: link

7. Link to a release version

Link: release

8. Work division

Generate ideas and discuss the possible project – Together

Support project setups and resolve problems with docker – Together

Camunda Diagrams:

‘transferGoodsComplete’, ‘scheduleTransfer’, ‘transportGoods’, ‘confirmTransport’ – Michał

‘’startProductionLine’, ‘Supply Chain Process’, ‘StartProductionLineProcess’ - Raffael

Code reviews – Together

Create assignment reports – Together

Final documentation – Together

Writing ADRs – Together

Create presentation – Michał

9. Reflections

Throughout our journey of the first part of the “Event-driven and Process-oriented Architectures” course, we have encountered various challenges, gained valuable insights, and developed important skills that have not only enhanced our academic understanding but also team work. One of the most significant takeaways from that part of the course is a solid understanding of the fundamental principles underlying event-driven and process-oriented architectures and Kafka. From the basics of

Kafka configurations, event-driven programming to the intricacies of workflow modelling and application of Camunda platform. This knowledge has not only equipped us with the necessary vocabulary to discuss and analyse architectural patterns but has also provided us with a framework for approaching real-world software design challenges. One of the most rewarding aspects of the course was the opportunity to apply theoretical concepts to real-world scenarios through hands-on projects and case studies. Additionally, the lab materials prepared to test some solutions at home were incredibly helpful. Having access to resources that enabled us to experiment and apply what we have learned, further enhanced our learning experience. Whether it was setting up development environments, running demos, or troubleshooting implementation challenges. Furthermore, the visit from a Camunda employee was particularly motivating and enriching. Having the opportunity to interact with someone directly involved in the development of the platform Camunda provided valuable insights into its functionalities and capabilities straight from the creators. We are excited to continue exploring new concepts and applying them to practical scenarios in the second part of the semester, building on the foundation we have established so far.