

Monitoring an event-driven microservices ecosystem

Pedro Santos^{1,2}, Isabel Sampaio^{1,3} and Paulo Proença^{1,4}

¹ Instituto Superior de Engenharia do Porto, Portugal

² 1140545@isep.ipp.pt

³ ais@isep.ipp.pt

⁴ prp@isep.ipp.pt

Abstract. Throughout the years, software architectures have evolved deeply to attempt to address the main issues that have been emerging, mainly due to the ever-changing market needs. The need to provide a way for organizations and teams to build applications independently and with greater agility and speed led to the adoption of microservices, particularly endorsing an asynchronous methodology of communication between them via events. Moreover, the ever-growing demands for high-quality resilient and highly available systems helped pave the path towards a greater focus on strict quality measures, particularly monitoring and other means of assuring the well-functioning of components in production in real-time. Although techniques like logging, monitoring, and alerting are essential to be employed for each microservice, it may not be enough considering an event-driven architecture. Studies have shown that although organizations have been adopting this type of software architecture, they still struggle with the lack of visibility into end-to-end business processes that span multiple microservices. This paper explores how to guarantee observability over such architecture, thus keeping track of the business processes. The outcome of the paper shall do so by providing a tool that facilitates the analysis of the current situation of the ecosystem, as well as allow to view and possibly act upon the data. Two solutions have been explored and are therefore presented thoroughly, alongside a comparison to draw conclusions and provide some guidance to the readers.

Keywords: Software architecture, microservice, choreography, monitoring, observability

1 Context

1.1 Microservices

Nowadays, organizations have been adopting microservices to streamline their software delivery process. Microservices have emerged from the need to provide a way for organizations and teams to build applications independently and with more agility and speed. This architecture style effectively allows moving towards more reliable, performant, resilient, maintainable and scalable systems. It intends to do so by developing a single application as a suite of small well-bounded independent units with limited responsibility, each running in its own process and communicating through lightweight

mechanisms. These services are built around business capabilities and must be developable, testable and deployable in a fully automated manner, independently of other components that may compose the system [1]. Isolating the services might add some overhead but it also makes the whole distributed system much easier to understand and modify. Fig. 1 illustrates a possible representation of a microservices architecture.

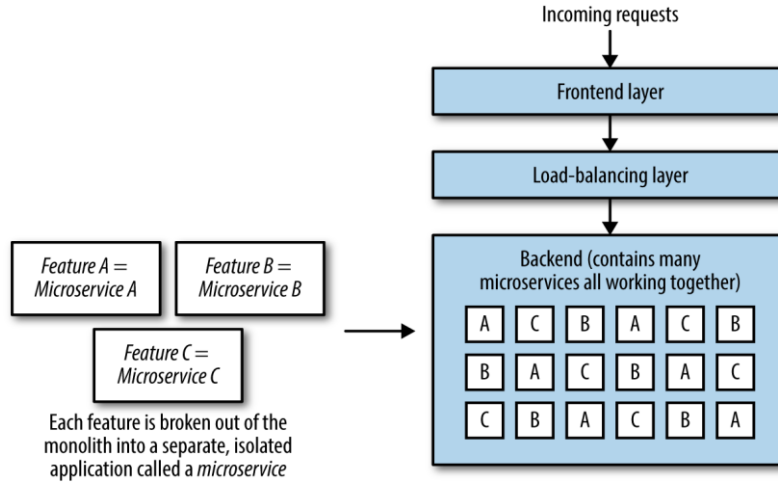


Fig. 1. The microservices architecture style [2]

Each microservice is generally characterized by three components: a frontend/client-side piece, some backend code, and a way to store and/or retrieve any relevant data. The frontend is an Application Programming Interface (API) with static endpoints, which allows microservices to interact easily and effectively by sending requests to those endpoints. The backend is responsible for processing the incoming request, apply any relevant logic, communicate with a database and finally return an appropriate answer to the client [2]. It shall be noted that every microservice should have its own independent database, which can either be an in-memory cache or an external database.

1.2 Asynchronous Choreographed Communication

Since microservices are independent units, they need to interact and share data to fulfil business processes crucial for the whole system to function properly. The communication amongst them can occur synchronously or asynchronously. With synchronous communication, a call is made to a remote server which blocks until the operation completes. On the other hand, asynchronous communication does not imply that a caller waits for the operation to complete before returning. In fact, it does not even need to know whether the full operation eventually succeeded or not [3]. Asynchronous communication is more adequate for long-running jobs where it is impractical to maintain a connection between the client and the server for an extended period of time. Moreover, it is generally employed via event-based collaboration. With event-based collaboration, a client informs that something has changed in its domain, by publishing an

event to an event bus, and expects other parties to be aware of what they need to do. Furthermore, it does not even need to know who or what is going to react to it [3]. The event bus handles the events published and forwards them to the other components that want to be aware of such changes. This type of collaboration possesses characteristics that allow paving towards a highly decoupled system.

Following the topic of communication between microservices, there are two architecture styles usually used, either individually or as a complement to each other: orchestration and choreography. Orchestration implies having one central component to guide and drive the process, which typically follows the synchronous request/response model. This architecture style has the business processes centralized, therefore, maintaining and managing a system becomes much easier. On the contrary, choreography is characterized by the interaction between services, asynchronously, via publishing and subscribing events. Each microservice knows what to do and performs its actions independently and publishes events for other services to consume and react upon them as needed. This way dependencies between services can be avoided, meaning each service should be able to stand on its own. Therefore, this approach is significantly more decoupled and scalable, thus emphasizing the main benefits that can be taken from a microservices architecture. Although it has many benefits, it also has a significant drawback: the business processes are spread across different services, making it challenging to be aware and in control of the overall processes. This means additional work may be required to ensure we can monitor and track workflows and know if everything completed as expected or not. This paper focuses on addressing this matter.

1.3 Monitoring an event-driven microservices architecture

Nowadays, organizations should aim for tools to assure the well-functioning of their components in production in real-time. Whether it is a monolith or a complex distributed system with multiple components, logging and monitoring should be key concerns. A common solution is the usage of the ELK stack. The ELK stack is a collection of three open-source products – Elasticsearch, Logstash and Kibana. Together, these three products are widely used for monitoring and troubleshooting environments. Elasticsearch is an open-source, full-text search and analysis engine. Logstash is a log aggregator. It can collect data from multiple sources, transform it and enhance it as desired, and ship it to supported output destinations. Kibana is the visualization layer. It works on top of Elasticsearch with the intent of providing users with the ability to analyze and visualize data in a simple and user-friendly manner. By using the ELK stack, development teams become aware of what is happening with a given component of its ecosystem, by outputting that information in the form of logging and delivering it to customizable and user-friendly dashboards. However, monitoring does not consist only of gathering data from applications. Another well-known technique to achieve control over a system is the implementation of health checks. Health checks allow perceiving whether or not a specific service and its dependencies are up and running.

As we have covered in the previous section, when we are referring to complex distributed systems that communicate asynchronously via events, it becomes even harder to

track the whole ecosystem and connect all the dots. A distributed system contains multiple independent services that require data to be correlated between all of them as they collaborate to carry out business logic that crosses their boundaries [3]. And here is when the real challenge is faced. In 2018, a company named Camunda conducted a survey that showcased precisely this issue. Many people still feel like they lack visibility over the end-to-end business processes that span multiple microservices. A graphic representation of the results of the survey can be seen in Fig. 2.

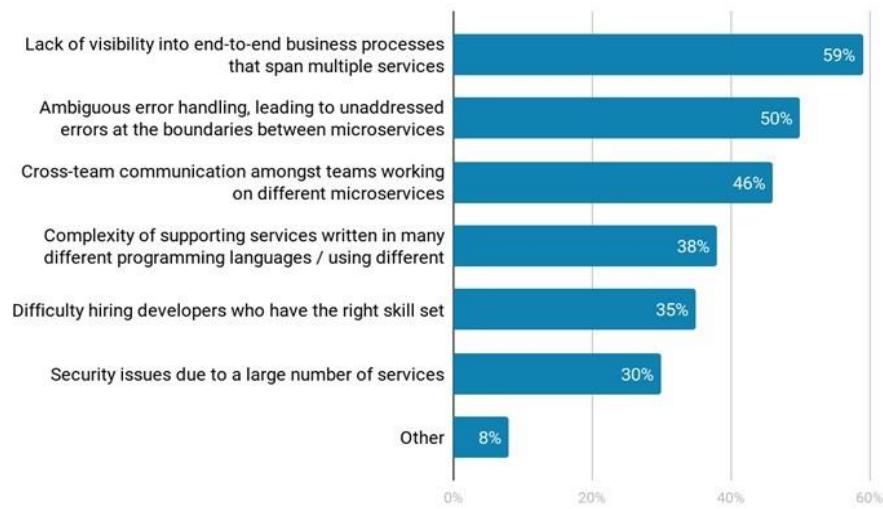


Fig. 2. Study on the challenges of microservices

This paper intends to address this issue by providing a solution for gaining sight and control of the logical flows of events throughout a system, and by also focusing on managing Service-Level Agreements (SLAs) and assuring resilience. The strategy employed relies on tracking business processes via workflow automation. The processes can be modelled using Business Process Model and Notation (BPMN) and run on top of a workflow engine responsible for managing it. The solution implemented and showcased in this paper is characterized by a tool responsible for explicitly matching the view of the business processes and showcasing them to authenticated system administrators, and possibly allow for also reacting upon the data, either manually or automatically.

2 Solution

The solution is composed of three main components:

- Target Application: An event-driven microservices ecosystem was needed to serve as the target application to apply the observability solution;

- `CamundaEventObservabilityApp`: An application implemented on top of the workflow engine named Camunda BPMN Workflow Engine [4];
- `ZeebeEventObservabilityApp`: Implemented using a workflow engine named Zeebe, developed by the same company as the Camunda BPMN Workflow Engine but with a greater emphasis on high throughput and scalability concerns [5].

2.1 Target Application

The target application was created based on an open-source project known as `eShopOnContainers` developed by Microsoft. `eShopOnContainers` is a reference application for .Net Core and microservices, designed to be deployed using Docker containers [6]. Its main purpose is to showcase how to apply many well-known architectural patterns.

The application is an online store focused on selling various products, such as t-shirts, sweatshirts and mugs. Therefore, its main use cases involve managing users, catalogue items and orders. For example, a user can view all the products being sold on the platform and choose to add or remove them to and from its basket, so that an order can be fulfilled. Those workflows are accomplished through the interaction between multiple microservices. Between microservices, the communication is performed asynchronously by publishing and subscribing to events. The main workflows carried out by the application are the ones that concern the order fulfilment process. Therefore, those are the ones being covered by the observability solutions depicted below, as they are also the ones that entail the publishing and subscribing events the most.

2.2 Event Observability Applications

Both applications were developed with the purpose of integrating the microservices architecture described previously with a workflow engine. The integration was performed by focusing on the main workflow of fulfilling an order. Many microservices are involved and communicate with each other asynchronously through the usage of events. We must ensure the full flow is not compromised, as well as the user's experience. For example, an issue can occur with any microservice causing it not to consume events or not produce the event it is expected to produce in order to proceed with the flow. Furthermore, the event bus used, in this case, RabbitMQ, can also fail at any given point in time. These solutions ensure that if any of these issues happen, the development team is notified and corrective measures are put in place, thus trying to minimize the impact on the user and even attempt to do it in a way the user won't even notice any issue occurred with its order. To do so, the solutions consume the events flowing through the ecosystem and attempt to correlate each one with the right step of a business process, providing the status of the system in real-time.

`CamundaEventObservabilityApp` is a Spring Boot application, with the Camunda BPMN Workflow Engine embedded, developed to integrate the microservices architecture described previously with the workflow engine. Once the business processes are defined, and those definitions are deployed to the workflow engine, it can manage multiple simultaneous instances of each process. Fig. 3 contains the definitions of the business processes, via BPMN, in the scope of this solution.

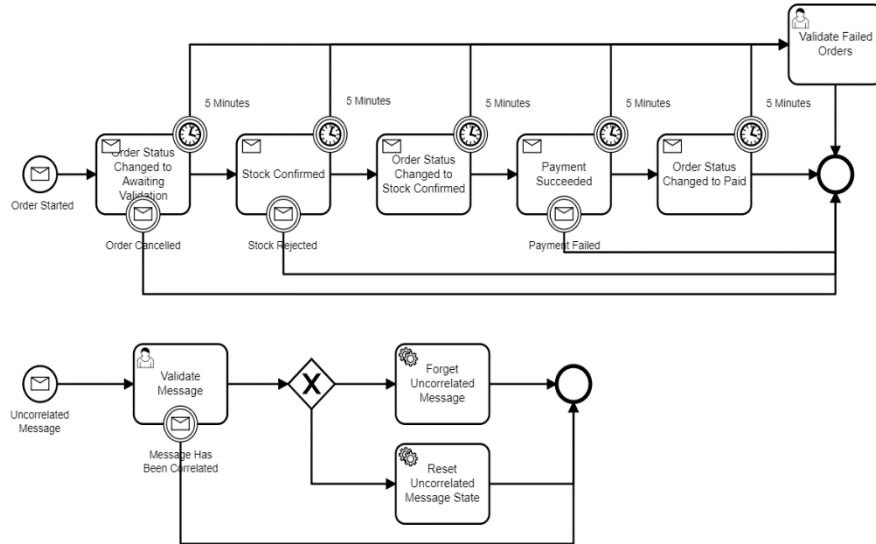


Fig. 3. Business processes defined in the scope of CamundaEventObservabilityApp

As we can see in the picture above, we have two business processes defined. The first process is the one defined for the order fulfilment scenario, whereas the second is a process triggered by a scheduler to handle uncorrelated messages. The first process expects a start event to be published to the workflow engine for it to correlate it to a new instance of the process. Each instance represents an order that is being fulfilled by the target application. Afterwards, the process expects a sequence of events, represented with the BPMN symbol Message Boundary Event, that represents the remaining order fulfilment flow. If the application for some reason cannot consume any of the events in five minutes, there may be an issue with the process. Therefore, a timer is activated and a User Task triggered. This user task provides administrators info regarding that order and users must provide acknowledgement for the process to end. There are also some Message Boundary Event symbols that indicate an event that has triggered the cancellation of the order. These events however are expected and do not indicate failure, therefore the process simply completes. If any of the messages consumed cannot be correlated to any step of a running instance of the process, it will be stored in a specific table for the uncorrelated messages. Correlation may fail due to one of the reasons already mentioned in this section, which actually constitute an issue, or because of the events being consumed misleadingly. The latest meaning that, depending on the event bus we are dealing with, the events may not be consumed by CamundaEventObservabilityApp in the same order they were produced. In this scenario, those events must wait for other events to be successfully correlated. Because of this, every Message Boundary Event contains start and end listeners. Prior to the task execution, the application will look for the event in the uncorrelated messages table. If the correlation was performed due to the data on the table, the end listener will be responsible to remove that data as it is no longer necessary. If not, the application will simply remain waiting

for the event to arrive in its queue. The application contains a scheduler, triggered every thirty minutes, that initiates the second process. This process will then present to the platform's administrators, via a User Task, every message that could not be correlated and has been awaiting correlation for more than thirty minutes. The user may then decide whether the message should be kept for future correlation or if it can be removed. This decision depends on the user's assessment of the message. The user can perceive it as being a normal event that can still be correlated in the future or consider it an issue, in which case the user may pursue that issue and end up deciding to remove it from the database since it does not have a chance of being correlated. If the message ends up being correlated while the user is expected to provide a decision, the flow captures that event which automatically leads the process to an end.

ZeebeEventObservabilityApp is also a Spring Boot application that attempts to integrate the target architecture with the workflow engine but, instead of Camunda BPMN Workflow Engine, it uses Zeebe. This application does not possess Zeebe embedded, therefore it depends on a connection to a Zeebe broker, which in turn depends on an Elasticsearch cluster. Also, Zeebe does not provide visualization features on the fly. Instead, it requires another tool named Camunda Operate for such purpose. Zeebe also has the downside of supporting less BPMN symbols as Camunda BPMN Workflow Engine. Therefore, its process definition is slightly different, as illustrated in Fig. 4.

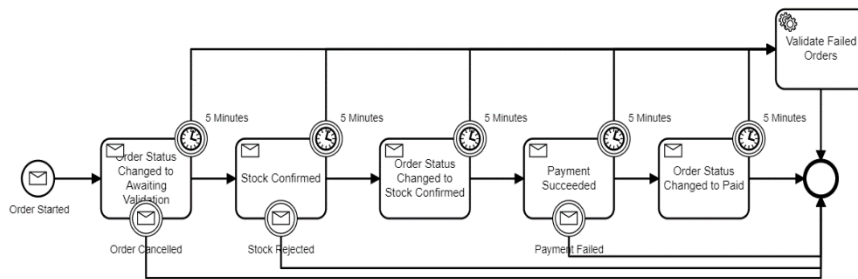


Fig. 4. Business processes defined in the scope of ZeebeEventObservabilityApp

The business process defined is the same as the previous solution but with slight changes originated from Zeebe's limitations. The first limitation is the lack of support for the User Task BPMN symbol. To address this, the solution needed to contain a database and a user interface. The user task was replaced by a service task which calls a Java function. The purpose of the service task is to mimic the behaviour of the user task, therefore, that Java function stores the data in a database. From that point onwards, an authenticated user can access the exposed interface and visualize the user tasks assigned and pending for acknowledgement. Once the acknowledgement is provided, the application issues a command for the workflow engine to correlate the step and therefore terminate the instance of the process. The other limitation relies on the fact that we cannot know if a message was not correlated because Zeebe's correlation process is asynchronous. Therefore, the Zeebe client does not report the result of the correlation it attempted to perform.

3 Conclusions and Future Work

This paper provides valuable insights towards microservices architectures and particularly in regards to architectures employing asynchronous communication through events. The problem identified is acknowledged by professionals of the area as being a significant setback and worth the research. We could also conclude both solutions present valid approaches to the problem. Camunda BPMN Workflow Engine is more mature, embeddable and more versatile as it supports a wider range of BPMN symbols. Furthermore, it is easier to set up as its architecture only depends on a database. On the contrary, Zeebe provides a narrower range of BPMN symbols and requires a more extensive set of dependencies. However, it may address performance and scalability requirements to an extent that Camunda BPMN Workflow Engine cannot. It is a cloud-native solution specifically designed with high throughput and low latency scenarios in mind. Each solution has its advantages as well as its disadvantages and may be seen as the best solution depending on the scenario to which it is expected to be applied.

For the future, there is already identified work that can be pursued. One improvement could be to enhance the interface developed for addressing the user tasks limitation on ZeebeEventObservabilityApp. This was approached as a proof of concept, so this feature was implemented with the focus of having it fully functioning, with the interface not being the main concern. Now that a fully-featured solution was achieved, the focus can shift towards these details that improve user interaction and experience.

Apart from the implementation details, the solution could be evaluated in a real-life scenario. By implementing it in a real production environment, it is possible to collect relevant metrics to effectively evaluate whether or not it improves a team's efficiency. Some metrics could be for example the time it takes for the development team to detect malfunctions in the system and the time to recover from those failures. By collecting these metrics before the implementation of the solution, and comparing to the behaviour after implementing the solution, further concrete conclusions can be taken in regards to its efficiency in real-life scenarios.

References

1. Martin Fowler and James Lewis, "Microservices", <https://martinfowler.com/articles/microservices.html>, (Last accessed in 2020)
2. Susan Fowler, "Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization", 2016
3. Sam Newman, "Building Microservices: Designing Fine-Grained Systems", 2014
4. Camunda, "BPMN Workflow Engine", <https://camunda.com/products/bpmn-engine/>, (Last accessed in 2020)
5. Camunda, "What is Zeebe?", <https://zeebe.io/what-is-zeebe/>, (Last accessed in 2020)
6. Microsoft, "Introducing eShopOnContainers reference app", <https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduce-eshoponcontainers-reference-app>, (Last accessed in 2020)