Department of Mathematics and Computer Science

Interconnected Resource-aware Intelligent Systems Research Group

Real-Time Conformance Checking for Microservice Applications

Master's Thesis

Ricardo Andrade

Supervision:

Prof. Dr. Johan Lukkien

Prof. Dr. Benny Akesson (TNO-ESI)

Dr. Jacob Krüger

8th March 2024

Abstract

Traditional software systems were developed as a single unit responsible for multiple operations and containing all the business logic. However, as the system grows and is incremented with new features and functionalities, the dependencies between its parts become unmanageable and hard to understand. To overcome some shortcomings of the monolithic architecture, organizations moved away from monoliths and migrated their systems to microservice architectures. On the other hand, due to their volatile operating environment, microservices must be continuously and rigorously monitored to understand what is happening inside the system. One area that is particularly unexplored for microservice applications is the employment conformance checking techniques to verify if the execution of microservices conforms to its initial design. Existing solutions usually need the full context of execution and provide results long after the execution happens in the application, which is not suitable for the microservice environment.

In this master thesis, we create an online conformance checker tailored for microservice applications. We start by creating an offline conformance checker that effectively evaluates conformance based on MDA execution traces and sequence diagrams, operating independently of the MDA's execution for efficiency. After, we transition to an online paradigm improving performance significantly, achieving conformance results in approximately 30 seconds per trace, meeting our requirement for rapid reaction to nonconforming sequences.

Acknowledgments

I seize this moment to extend my heartfelt gratitude to all those who have contributed to this extraordinary journey of my life. Words fail to capture the immense joy I feel as I write these lines, marking the culmination of a year-long endeavor to bring my master's thesis to fruition. This milestone is the product of relentless dedication, accompanied by sweat and, perhaps, a few tears. Now that we stand at this juncture, let us celebrate the present and prepare for the future!

First and foremost, I am profoundly thankful to TNO-ESI for affording me the opportunity and support throughout my master's thesis. I extend my deepest appreciation to my supervisors, Benny and Johan, whose guidance has been invaluable. I am equally grateful to all my colleagues at TNO-ESI, especially Ben, Rajat, and Sven, whose contributions enriched my thesis.

I am indebted to my parents and grandparents for their unwavering support. To my father, Luiz, for cultivating in me the courage to pursue my dreams, and to my mother, Alina, for being my rock through it all. My heartfelt gratitude extends to my grandfather, António, whose passion for exploration ignited my own, and to my grandmother, Luiza, for the cherished memories we share. None of this would have been possible without your love and support. We did it together!

I am also grateful to my extended family for their support on this journey.

Last but certainly not least, I extend my gratitude to all my friends who have been a part of my journey thus far. Some have come and gone, but the memories we've shared will last a lifetime. To Beatriz, whose unwavering support and encouragement have been my daily motivation. To João, for the countless conversations, FIFA battles, and memorable Monsanto afternoons. To Ricardo for making me laugh with silly comments. To Ahmet and Maxim for the amazing last two years, parties, and pool games in Altstadt. To my JUNITEC friends, Afonso, Catarina, Inês, José e Lourenço for all the failed startup ideas and unforgettable parties. To my Técnico friends, Catarina, Daniel, Filipa, and Maria for being the best study partners I could've asked for and all the code nights we spent together. To the 'tugas' in The Netherlands, Catarina, Cristiana, David, Filipe, Iara, Lourenço, Madalena, Marta, Pedro, Salvador, and Vasco for making The Netherlands feel like home.

To each and every one of you, thank you for being part of this incredible journey. I hope it was as enjoyable for you, as it was for me.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CPU Central Processing Unit.

ID Identifier.

IP Internet Protocol.

JSON JavaScript Object Notation.

MDA Meal Delivering Application.

OS Operating System.

RAM Random Access Memory.

SOA Service-Oriented Architecture.

VM Virtual Machine.

Chapter 1

Introduction

1.1 Overview

Traditional software systems were developed as a single unit responsible for multiple operations and containing all the business logic, meaning that all parts of the system needed to be present for the successful execution of a program. Due to all the components being tightly coupled and interconnected, this architecture poses an easy way for small to medium-scale systems and teams to manage the code base, deployment process, and communication within an organization [20, 38]. Systems developed with this architecture are called monolithic systems.

However, as the system grows and is incremented with new features and functionalities, the dependencies between its parts become unmanageable and hard to understand, even for the team that developed the system in the first place. When the monolith gets too complex, several problems will arise, reducing the quality of the software itself. Additionally, scaling becomes more difficult, since the monolith can only scale in one dimension, by adding several replicas of the monolith on a load balancer. Finally, maintaining a big complex monolith renders it impossible to practice DevOps guidelines, mainly continuous integration and continuous deployment, since each change requires the redeployment of the complete system [38]. To overcome some shortcomings of the monolithic architecture, organizations moved away from monoliths and migrated their systems to microservice architectures.

The push and fast adoption of microservices by the industry brought new research challenges to academia. Microservices have proven to be the way to go for organizations that want to cope with the demands of a world that is moving faster than ever. Quicker development and deployment of new features are not compatible with traditional software development methodologies or architectures. Teams need to be scaled down, more independent and have more ownership of the specific part of the system they are responsible for. Thus, adopting a microservice architecture is a synonym for big changes, not only at a technical level but also at the organization as a whole [33].

Arguably, one of the most significant changes when adopting a microservice architecture is the increased attention and investment in observability and monitoring efforts. Due to their volatile operating environment, microservices must be continuously and rigorously monitored to understand what is happening inside the system. This allows software engineers to optimize and diagnose the system's behavior [31, 40].

One area that is particularly unexplored for microservice applications is the employment conformance checking techniques to verify if the execution of microservices conforms to its initial design. While formal verification has been extensively researched for monolithic systems, its adaptation and application to microservices are relatively underexplored areas. Moreover, there is a lack of appropriate tooling for the verification of microservice applications [2]. Finally, due to their inherent distributed nature and dynamic deployment environment, developers and maintainers must be alerted fast to erroneous or unexpected scenarios of execution. Existing solutions usually need the full context of execution and provide results long after the execution happens in the application, which is not suitable for the microservice environment. Thus, in this master thesis, we want to research how such a tool can be implemented, what the challenges are for its implementation, and if it can provide correct and fast results for a microservice application.

1.2 Research Questions

Building on the previous section, we aim to answer the following research question in this master thesis:

RQ *How can correct, efficient, fast, scalable, and extensible conformance checking be implemented for microservice applications?*

1.3 Outline

In Chapter 2, we formalize the problem we are trying to solve and introduce the general concepts concerning our case study. This is followed by a review of core concepts for this topic in Chapter 3. After, in Chapter 4, we detail our approach to solving this problem, along with the case study of this thesis. In Chapter 5 we survey the literature for similar works in this area, followed by the implementation of our tool in Chapter 6 and Chapter 7. Finally, we discuss our results and challenges to this study and conclude this thesis in Chapter 8.

Chapter 2

Problem Statement

2.1 Motivation

Let us consider the following scenario: we are developing a new software application from scratch. To ensure that our application is developed with rigorous standards we will follow the guidelines for good software development by going through each phase of the software development lifecycle of choice, such as the V-model. A simplified illustrative example of the most common phases of a software development lifecycle is depicted in Figure 2.1.

2.1.1 Developing a Software Application

Design

We start by generating requirements and models for our new application in order to detail how the application should behave, how it interacts with other applications, the constraints that it should adhere to, and the overall architecture of the application. For instance, one deliverable of this first phase could be a collection of sequence diagrams, like the one in Figure 2.2. The goal of a sequence diagram is to illustrate the flow of messages between different parts of an application and how they interact with each other and with external systems for a particular use case.

In Figure 2.2, we can see a simple sequence diagram composed of three actors, Service A, Service B, and Service C that exchange messages with each other. Furthermore, we observe that these messages have an explicit ordering: Service A starts the use case by sending a message to Service B, after which Service B sends a message to Service C. Finally, to complete the use case, Service C sends a message to Service A. Moreover, from our requirements, we add a timing constraint to the operation. This exchange of messages should not be executed in over 200 milliseconds.

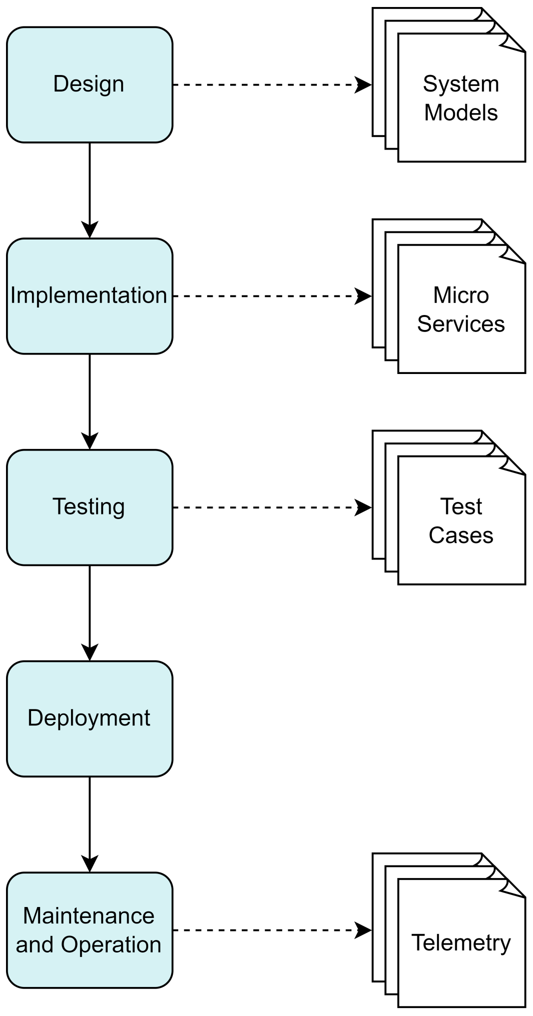

Figure 2.1: Schematic representation of a simple software development life-cycle and the deliverables of each stage of the process.

Implementation and Testing

After thoroughly defining and documenting the application we move on to its implementation. We take the system models and turn them into executable code. Test cases are defined to ensure the application behaves according to the initial specification and to avoid predictable errors and failures.

Deployment

The application is now ready to be executed. After setting up the needed infrastructure, we deploy the application into the production environment and expose the application to the outside world. We also deploy monitoring applications to keep track of the behavior of our system.

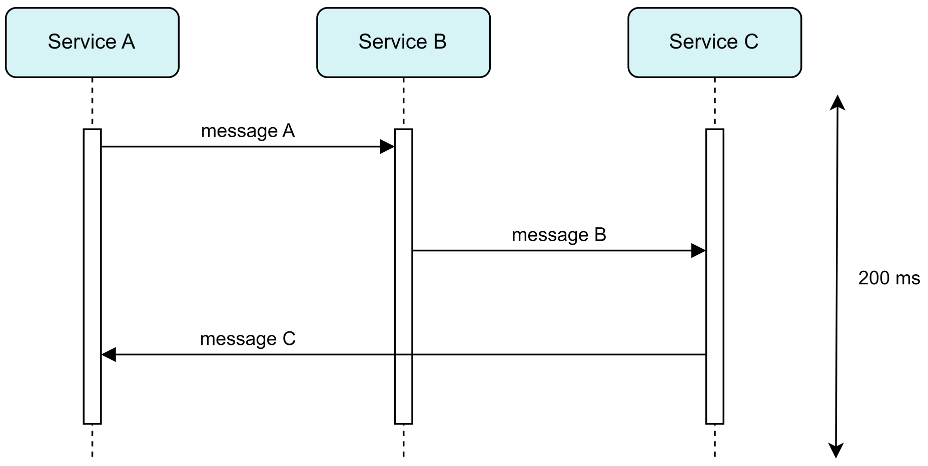

Figure 2.2: An example of a sequence diagram.

Maintenance and Operation

Maintenance plays a key role in ensuring the correct functioning of any application. Even though testing attempts to identify possible erroneous actions, no system is ever completely correct and not all faults and error scenarios can be predicted. Moreover, since we are dealing with distributed systems, in particular a microservice application that relies heavily on network communication and where microservices may evolve independently over time, it is impossible to envision all possible problems that the application can encounter during its execution in a real-world setting. Thus, we instrument the microservices to produce telemetry, allowing software developers and maintainers to monitor its execution.

This telemetry allows for different types of analysis. Some common monitoring cases include observing application and hardware metrics to monitor resource consumption and request latency, or analyzing logs and traces to check if there are any outstanding errors during the execution of the application and where they are located. A graphical representation of a trace is depicted in Figure 2.3

2.1.2 Goal

For this master thesis, we are particularly interested in evaluating if an application is actually behaving according to specification. This is called conformance checking. Despite meticulously following the software development cycle, it is hard to guarantee that the developed application follows the initial models and that during its execution, it respects previously established constraints.

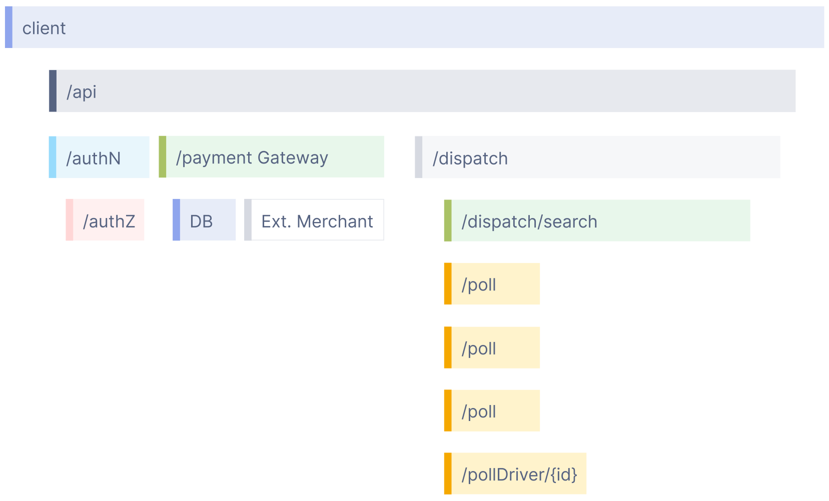

Figure 2.3: Graphical representation of a trace (from [44]). Each box is a span and boxes that are contained by other boxes represent a parent-child relationship.

To achieve this goal, we can think of a way to match the original sequence diagrams to the traces that are collected during the execution of the system. Since a trace represents (part of) an execution path of an operation as it propagates through the several services that compose the application, we can gather the relevant traces for a specific sequence diagram and analyze the services that interacted during the operation, the order in which they interacted, and even the time elapsed in each interaction.

Given a sequence diagram and a collection of traces, how can we check if the traces match the sequence diagram efficiently and effectively for a running microservice application?

We start by formally defining the conformance checking problem that was generally described above, after which we derive the desirable requirements that our conformance checking framework should adhere to.

2.2 Formal Definition

Graphical representations are useful to quickly grasp and visualize any given concept. However, they provide little information about the structure that is represented. In our case, we want to be able to describe traces and sequence diagrams in a more abstract way to make it easier to reason about possible ways to develop a conformance checking framework.

2.2.1 Sequence Diagrams

Sequence diagrams follow the UML specification and can represent complex use cases between interacting actors. In this master thesis, we are not interested in the full expressiveness of sequence diagrams. For instance, we do not intend to support sequence fragments like ALT, LOOP, or PAR that were introduced in UML 2.0. Moreover, the notation for representing messages exchanged between two actors will always use full arrows, since it is not important to distinguish between return messages, which should be represented by dashed arrows, or synchronous and asynchronous forms of communication.

Let us consider again the sequence diagram in Figure 2.2. This diagram is composed of several architectural elements. The blue boxes correspond to the actors that participate in the modeled interactions. In our case, the actors are the services that compose an application. Below each actor, we can identify white boxes that have a dashed line underneath. This is called a lifeline and represents the execution of a service during the execution of the use case. As we will see in Chapter 4, we will use a microservice application for our experiments that has all services running simultaneously, meaning that for the sequence diagrams of interest, all actors will be deployed and running during the execution of a use case. Additionally, the diagram depicts interactions between the modeled services. These interactions can be exchanging messages or remote procedure calls for instance. Finally, we see a vertical arrow on the right side of the diagram representing a time constraint for the operation. Although this is not part of the UML notation for a sequence diagram, it is sufficient for the conformance checking that we aim to implement.

We will adopt the following notation: sequence diagram s

$$s = \{A, I, t\}$$

is composed of the set of actors A that participate in s

$$A = \{a \,|\, a \text{ is a service in our application}\},$$

the tuple of ordered interactions I represented in s

$$I = (i_1, i_2, ..., i_{n-1}, i_n),$$

and the timing constrain t in milliseconds. Interaction $i \in I$ is a 4-tuple

$$\begin{aligned} i &= (type, sender, receiver, content), \\ type &\in \{message, function\}, \\ sender, receiver &\in A, \\ content &= \text{ message exchanged or function called.} \end{aligned}$$

2.2.2 Traces

Contrasting with sequence diagrams, traces don't have a rigorous standard that they adhere to. This makes it difficult to provide a general notation that would fit some of the most recent open-source tools or vendors that provide tracing libraries and process telemetry. OpenTelemetry [44] was developed as a vendor-agnostic observability framework to overcome these complications and proposed a standard protocol for many telemetry operations. Traces in OpenTelemetry support a wide range of operations and can contain lots of information, fields, and attributes that are user-customizable to aid software developers and maintainers in observing how an application is behaving. Similarly to sequence diagrams, we are not interested in describing in our notation the full range of features provided by OpenTelemetry, but rather only include the fields of interest that will be useful for our conformance checking framework.

Let us analyze the trace in Figure 2.3. Although this is a very simple representation of a trace, it is possible to grasp the basic structure of a trace and its components. The first thing we observe is the several rectangular colored boxes. These boxes are called spans and the color indicates which service emitted the span. Spans represent a unit of work or an operation [44] and can be uniquely identified by a span ID. Furthermore, spans contain several attributes that provide information about the context or operation that the span captures. One of those attributes is the span name, which depending on the developer's choice can represent an operation name or a routing path as shown in the spans in Figure 2.3. Other attributes of interest include start and end times, the name of the service that emitted the span, events, and custom user-defined fields. We will take advantage of these custom fields to include relevant information that will help us to do the desired conformance checking. For now, we will assume that our spans have a field called content that carries our custom information.

Traces are essentially a collection of spans that are aggregated together because they have the same trace ID. In our notation, we will consider the trace ID as an attribute of a trace rather than an attribute of a span, as OpenTelemetry represents it. Besides being related by a shared trace ID, spans also relate with each other in different ways. For instance, spans can be related to spans from different traces by a special field called *link* in OpenTelemetry. However, we are interested in the more simpler and common relationship: the child-parent relationship. Traces can be visualized as a tree-like structure like the one in Figure 2.4 that depicts the trace in Figure 2.3 as a tree. To be able to preserve these relations, spans include a field called parent span ID. Finally, the top span of each trace is called a root span. Root spans are different from regular spans because they have no parent span ID.

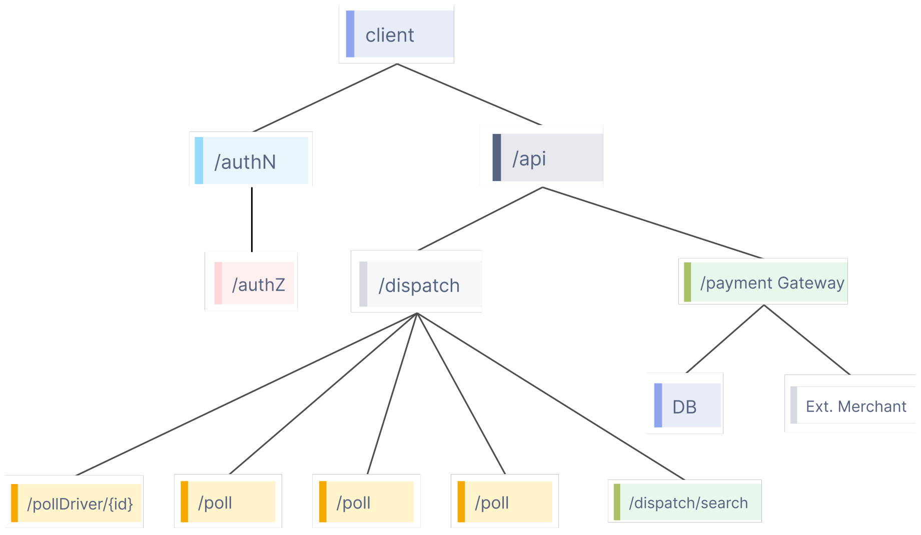

Figure 2.4: Trace represented as a tree. The horizontal axis represents the parent-child relationship.

We will adopt the following notation: the set of traces TR is generated and collected during the continuous execution of the application

$$TR = \{tr_1, tr_2, ..., tr_{n-1}, tr_n\},$$

where trace tr is a set

$$tr = \{id, SP\},$$

composed by an id, a unique character string, and the set of spans SP

$$SP = \{sp_1, sp_2, ..., sp_{n-1}, sp_n\},$$

where span sp is a set

$$\begin{aligned}
sp &= \{id, service, operation, parent, start, end, content\},\\
service &\text{ is the service that emitted the span,}\\
operation &\text{ is the name of the operation executed,}\\
parent &\text{ is the id of the parent span,}\\
start, end &\text{ are timestamps representing the start}\\
&\text{ and end times of the span, respectively, and}\\
content &\text{ is the message sent or received and/or}\\
&\text{ function name invoked during the span and/or}
\end{aligned}$$

2.2.3 Conformance Checking

We now have an accurate way of describing both traces and sequence diagrams. To be able to perform the desired conformance checking we need to find a mapping between traces and sequence diagrams.

In our notation for sequence diagrams, we defined that interactions occur between two services and contain information that was exchanged. We can derive a similar concept with spans belonging to the same trace. After deriving these interactions, matching against a set of interactions of a sequence diagram is (almost) trivial. In the simplest case that traces and spans are perfectly ordered, we extract the interactions from traces and compare them one by one with the interactions in a sequence diagram to check if they appear in the same order, and concern the same services and content exchanged. Let us consider the following example:

Sequence Diagram s

$$\begin{aligned}
s &= \{A, I, t\},\\
t &= 200ms,\\
A &= \{ServiceA, ServiceB, ServiceC\},\\
I &= (i_1, i_2),\\
i_1 &= (message, ServiceA, ServiceB, messageA),\\
i_2 &= (message, ServiceB, ServiceC, messageB).
\end{aligned}$$

Trace Set TR

$$\begin{aligned}
TR &= \{tr_1, tr_2\},\\
tr_1 &= \{tr1, SP_1\},\\
SP_1 &= \{sp_{11}, sp_{12}\}\\
sp_{11} &= \{sp11, serviceA, send, none, t_0, t_2, messageA\}\\
sp_{12} &= \{sp12, serviceB, receive, sp11, t_1, t_2, messageA\}\\
tr_2 &= \{tr2, SP_2\},\\
SP_2 &= \{sp_{21}, sp_{22}\}\\
sp_{21} &= \{sp21, serviceB, send, none, t_1, t_3, messageB\}\\
sp_{22} &= \{sp22, serviceC, receive, sp21, t_2, t_3, messageB\}
\end{aligned}$$

Both sp_{11} and sp_{12} record activity with $messageA$. Moreover, sp_{11} *sends* $messageA$ and sp_{12} *receives* $messageA$. Finally, sp_{12} records the sending action at t_0 and sp_{11} records the receiving action at t_1, meaning that the sending action from $serviceA$ preceded the receiving action from $serviceB$. Hence, we derive the following interactions from TR:

$$\begin{aligned}
i_{tr_1} &= (message, ServiceA, ServiceB, messageA)\\
i_{tr_2} &= (message, ServiceB, ServiceC, messageB)\\
t_{elapsed} &= t_3 - t_0
\end{aligned}$$

We can now say that TR *conforms* to s because:

$$i_1 = i_{tr_1} \wedge i_2 = i_{tr_2} \wedge t_{elapsed} < t$$

2.3 Requirements

We want to propose a conformance checking framework that is suitable for a microservice application. As we will see in Section 3.1, microservices rely on network communication and cooperation between services to execute user requests. Thus, tracing becomes essential to understand and debug this type of application. Moreover, depending on how the microservices are instrumented, high volumes of data may be generated during their execution. Furthermore, given that microservices are distributed over the network, it is challenging to guarantee a correct time ordering of the generated traces. Given these constraints, we detail the desired requirements for our checker architecture as follows:

R1 *The checker shall provide correct results.*
In distributed systems, it is not trivial to have a correctly time-ordered sequence of events. Thus, the checker needs to implement a strategy that deals with out-of-order events to produce correct results for checking traces.

R2 *The checker shall be efficient.*
The execution of the checker shall have minimal impact on the execution of the microservice application.

R3 *The checker shall be fast.*
One important aspect of modern microservice applications is to be able to quickly detect outages and erroneous scenarios. Thus, the checker should provide fast results for nonconforming traces.

R4 *The checker shall be scalable.*
Given that we expect high volumes of data to be generated and processed, the developed architecture shall handle increased loads of data without hindering the performance of the checker or the microservice application.

R5 *The checker shall be extensible.*
The checker architecture shall have a strategy that allows for easy increments of additional sequence diagrams to be used when processing generated traces.

Chapter 3

Background

3.1 Microservices

Microservices have been around for a few years now, emerging as a trend in the industry to overcome the inherent disadvantages of service-oriented architecture (SOA) and monolithic architectures. Additionally, microservices emerge at a time when organizations prioritize faster and customer-centric development, making microservices alongside Agile and DevOps methodologies the way to go for most organizations.

Considered by some authors a subset of SOA [33], microservice architecture offers a more fine-grain development and control over each service, while maximizing their independence by breaking away from a centralized point of control or orchestrator that ultimately controls the flow of the system in SOAs. In addition to constructing each service as an independent unit, microservice architecture splits business rules and logic across services and relies on lighter technologies to support the communication paradigm between each unit [16]. This philosophy offers great independence to each team since there are no constraints on technological development stack choices and little to no coordination is needed at deployment time. Additionally, microservices can easily be scaled both horizontally and vertically, while monoliths can only be scaled horizontally if they are loosely coupled. For instance, we could scale the system horizontally by adding more instances of a deployed microservice or vertically by adding resources to the node where that microservice is hosted. Moreover, the system can be expanded with new features by adding new units to the overall system, without increasing the complexity of already developed units [27].

As an example, we can look at the Social Network application [28], which is commonly used in research papers as a proof of concept. In Figure 3.1 we can depict a possible version of the Social Network application that has a monolithic architecture, categorized by a single unit that executes all re-

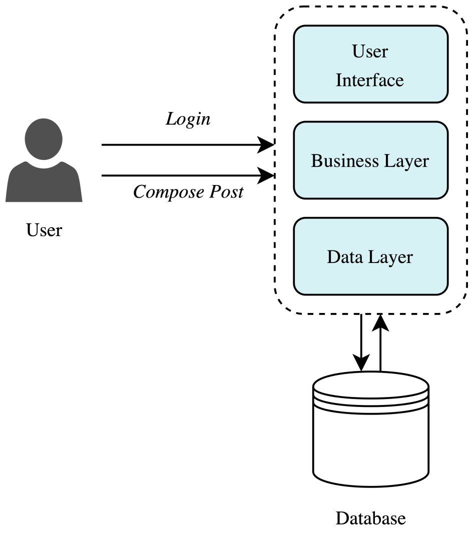

Figure 3.1: Monolithic example of the Social Network application

quests received and contains all business logic it needs to respond to users. For instance, composing a post or logging in to an account are actions performed inside this unit without communication with other instances than the database. In Figure 3.2 we can see a simplified version of the Social Network application with a microservice architecture. We can clearly see the separation of concerns and independence between the Compose Post service and the User Service that communicate with each other to execute requests when necessary. For example, when a user writes a post, the Compose Post service requests user information from the User Service asynchronously, allowing both systems to carry out work without waiting for each other. Furthermore, the Compose Post service is not connected to a database. Instead, it offloads that work to the Post Storage service to handle the persistence of data. In terms of scaling, it is possible to scale each individual service. If the Compose Post service has a lot of traffic, we can scale vertically by adding more computing power to the node where a service instance is deployed, or we can scale horizontally by adding more instances of the Compose Post service supported by a load balancer.

However, some new challenges arise with the adoption of a microservice architecture. Firstly, microservices move away from shared databases to eliminate a single point of failure and a bottleneck in the overall system. Moreover, business rules are split over microservice units. Consequently, microservice systems will have to deal with data duplication and an effective data management policy to reduce the cost of querying databases [37].

Secondly, getting a complete overview of the developed system or its working

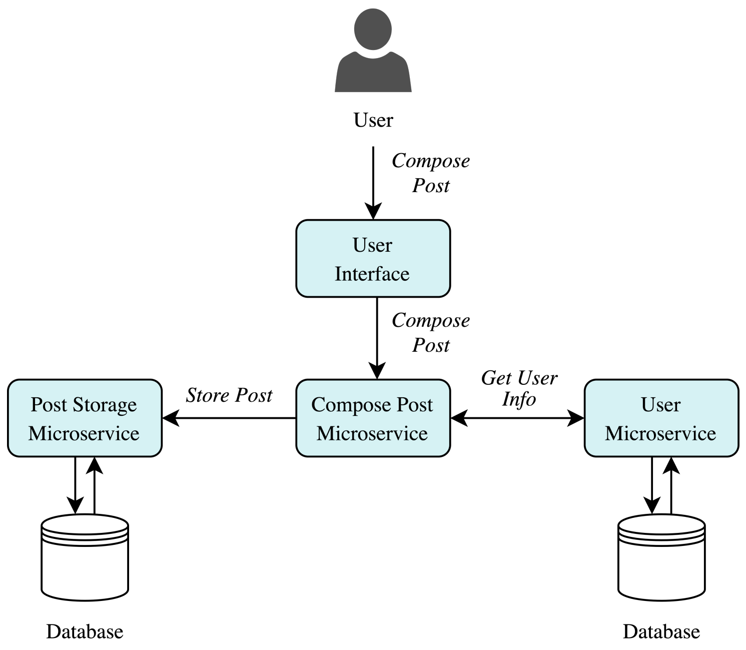

Figure 3.2: Simplified microservice architecture of the Social Network application

state is hard due to the independence between microservice units and constant scaling or descaling of the overall system. Thus, there is an increase in monitoring efforts and infrastructure costs. Even though maintenance costs are lower for microservice architecture, services must be instrumented to provide visibility into the execution of the system. Instrumentation consists essentially of producing and gathering data that gives insights into a system's execution. However, instrumenting a microservice architecture is fundamentally different from traditionally used techniques and best practices due to their inherent distributed nature. Additionally, due to frequent changes in microservice applications, such as scaling actions or updates of the microservices themselves, it is increasingly difficult to ensure and verify the correct behavior of this type of system [31].

3.2 Containerization

Before the popularization of microservices, there was a significant evolution in how applications are deployed due to the emergence of cloud-native architectures. Traditional deployment of applications was done in bare metal without any kind of virtualization. This allowed applications to run natively in the host and utilize fully the underlying hardware. However, deploying applications this way did not allow for any portability due to the application being closely tied to the hardware it was running on, which makes it difficult to redeploy the application in a server with a different configuration. Furthermore, if the application is not using fully the hardware resources at its disposal, it is not possible to allocate or share them with another application [35].

For these reasons, there was a need to change the deployment paradigm to allow for more flexibility and better use of server resources. The first abstraction of hardware resources to virtual ones was developed by IBM using hypervisor-based deployment that manages the virtual machines (VM) running on a host [9]. VMs work by providing a full guest operating system (OS) image along with the necessary binaries and libraries for the applications that the VM is running. Furthermore, the VM is fully isolated from the host's OS [45]. However, VMs come with a performance cost. There is a small amount of overhead when translating instructions from the guest OS to the host OS, the startup time of a VM can be up to a few minutes and VMs require much more storage due to packing the whole guest OS kernel [21].

Containers emerge as a lightweight alternative that allows for better use of resources of cloud infrastructure. Containers take a different virtualization approach than VMs. Instead of virtualizing the underlying hardware, it focuses on abstracting the OS level. Containers are more portable, efficient, and independent, allowing for better resource utilization of the host where they are deployed [6]. Due to containers having low startup times and being portable and fast to deploy, containerization tools like Docker [19] became a popular choice to deploy microservice applications. A schematic representation of the aforementioned deployment types is depicted in Figure 3.3.

3.2.1 Container Orchestration

Microservice applications can easily scale up to hundreds or thousands of instances deployed on a cluster. As with any traditional distributed software system, microservice applications also have to be fault tolerant, available, reliable, and dispersed geographically. Managing clusters at this scale and guaranteeing the aforementioned requirements is not trivial. Thus, container orchestration platforms such as Kubernetes [36] became an essential component of large microservice systems. These platforms provide important functionalities to manage clusters at scale such as container scheduling, security, networking, service discovery, and monitoring [34].

3.3 Observability

The term observability is often mentioned in the context of microservices and monitoring. It originates from control theory and is defined as the ability

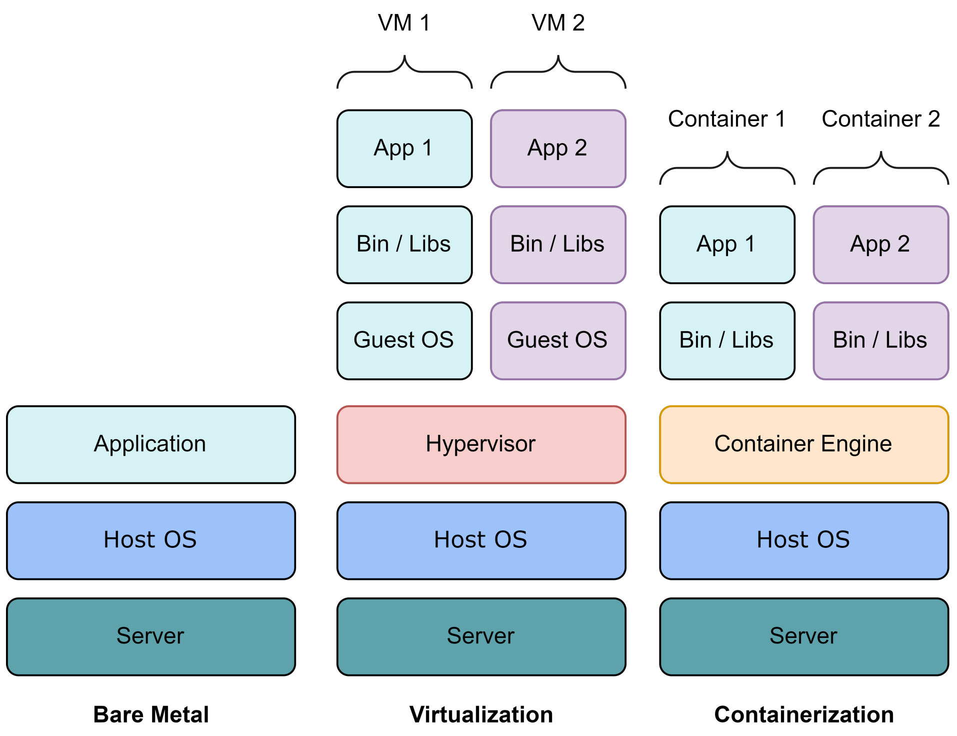

Figure 3.3: Visual representation of bare metal, virtualized, and containerized deployments.

to determine a system's internal state given its external outputs. However, there is a lack of agreement in the software engineering community regarding the definition of observability. We will use an adaptation of the definition proposed by Volpert et al. [55]:

> *Observability is a property of a system that allows an observer to gain an understanding of the system's internal state by looking at its external behavior.*

Observability complements monitoring efforts by providing ways for software developers to discover unknown and unexpected errors or failures [53]. Observability goes beyond monitoring by allowing software engineers to understand how a system is behaving, rather than just detecting issues. For the purpose of this thesis, we will consider monitoring as a subset of observability and a requirement to make a system observable.

To provide visibility into the internal state of a system, the system needs to produce the right outputs. These outputs can be created from, for example, interacting with the system and looking at the files produced during the execution of the system. Nonetheless, this is not a very efficient or structured

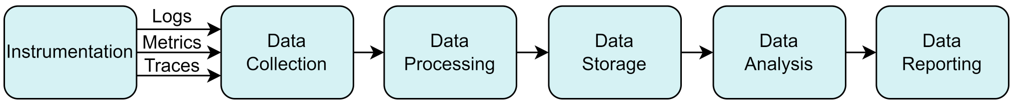

Figure 3.4: Observability Workflow (slightly modified from [55])

approach to observability. Thus, the most common way to generate the required outputs in a standardized manner is to generate telemetry data. More concretely, we want to generate logs, metrics, and traces to enable all observability activities. For instance, we can observe the services involved in a particular request by using traces and monitor its resource usage by analyzing the produced timestamped metrics during the request's execution. Furthermore, we can use logs to identify a specific problem that may have occurred during its execution.

3.3.1 Observability Workflow

The lack of consensus on a definition for observability is also extended to the phases that are related to observability itself. However, we see an overlap in the phases that make up observability across different publications [40, 55], with the main difference between the aforementioned papers being a slightly different terminology, an extra data reporting phase, and testing throughout the process proposed by Volpert et al. [55]. A simplified and adapted overview of the observability phases is depicted in Figure 3.4.

Phases of the Observability Workflow

Instrumentation is the first requirement to make a system observable. Software engineers instrument applications and platforms to generate different types of telemetry data to be analyzed and monitor the performance of the application and the platform itself.

Data Collection (or data acquisition) is the task of collecting all the instrumentation data generated by service instances, usually by a centralized collector. This collection can be performed by an agent that pulls data from the service instances or by providing an endpoint for the service to push instrumentation data.

Data Processing formats, aggregates, filters, and enriches collected telemetry data to relate the different types of telemetry data and achieve a consistent format between telemetry data collected from different service instances.

Data Storage is the task of persisting the collected and processed telemetry data. Depending on the data's intended use, it can be stored for short or long term or, archived.

Data Analysis is the phase where engineers get actionable insights from the telemetry data. Different analysis techniques can be applied to assess the current state of a running system. For instance, Li et al. [40] identifies timeline analysis, service dependency analysis, aggregation analysis, root cause analysis, and anomaly detection as some of the methods currently used in the industry for trace analysis.

Data Reporting consists mainly of data visualization. Human operators can visualize the state of a running system and react to potential issues by visualizing dashboards and responding to pre-configured alerts, like CPU usage above a certain threshold.

3.3.2 Telemetry Data

Observability is essentially a data-centric workflow, from its inception in the instrumentation phase to producing views in the data reporting phase that allow data-driven decision-making. Although observability is a relatively new concept in software engineering, telemetry data and monitoring have been the concern of software engineers for several years. Nowadays, logs, metrics, and traces have become the fundamental types of telemetry data to enable observability, denominated as the three pillars of observability [52, 53, 55].

The Three Pillars of Observability

Logs are timestamped records of events emitted during the runtime of a service. Logs can be unstructured plaintext lines, structured (usually in JSON format), or binary, usually in a domain-specific format. Logs are easy and fast to generate. Moreover, logs can be configured to allow for different types of verbosity. Logs are important for debugging efforts for a specific service instance due to their fine-grained view of service events and their real-time availability. Yet, logging presents some challenges. Firstly, processing unstructured logs can have a big impact on performance. Furthermore, using an incorrect sampling strategy can cause excessive logging and rising data storage costs.

Metrics are timestamped numeric representations of system data measured over time. In contrast to logs, metrics are easy to store and do not scale proportionally to the number of system requests. Moreover, metrics can be aggregated in time intervals to enable an analysis of daily or weekly trends in a system's performance. Metrics are also easier to analyze since they can be effortlessly inputted into mathematical and statistical models and are the type of telemetry data suited to define and trigger alerts.

Traces represent the entire lifecycle of a request as it travels throughout the entire distributed system. Traces can be seen as a type of log that contains information on how different events relate to each other. Moreover, traces are useful to perform several types of analysis to pinpoint possible bottlenecks in the system and parallel flows within the execution of a request. However, tracing can be quite complex to implement, especially because it is hard to retrofit to legacy software.

Other Telemetry Data Types

Besides the above-mentioned telemetry data types, depending on the vendor or tool chosen for instrumentation, some different data types may be included and considered important, such as events, deployment logs, and application dependencies [53]. Nevertheless, these data types can be generally derived from the three pillars of observability. For instance, deployment logs can be generated following a similar strategy for application logging and application dependencies can be obtained from traces. Events have different meanings for various authors: an event can be a log [52] or a collection of different telemetry data for a specific request of an application [41].

3.4 Conformance Checking

Many techniques and methods have been proposed to verify if a system is running according to specification. One of them is called conformance checking, a form of formal verification derived from the field of process mining [12]. The goal of conformance checking is to supply instruments to establish a link between expected modeled behavior and behavior observed during the execution of a process. This is done by recording event logs during the execution of the process and, after the process has completed its execution, matching it with a process model. This matching can be done using several different dimensions and techniques, depending on what we want to check, and yield different conclusions about the execution of the process [15].

In general, conformance checking has three different independent quality dimensions: fitness, precision, and generalization. Each dimension has a different goal and allows for different analyses. Fitness quantifies how a model and an event log fit each other, while precision measures the ability of the model to capture the observed behavior without allowing for unseen behavior in the event logs. Finally, generalization aims at quantifying the ability of the model to account for behavior that was not seen before, since event logs do not usually include all possible scenarios that a process model should represent [43].

To measure and quantify quality dimensions, three distinct techniques can be utilized. Token-replay-based approaches replay the event log over the process model, while alignment approaches attempt to describe the observed

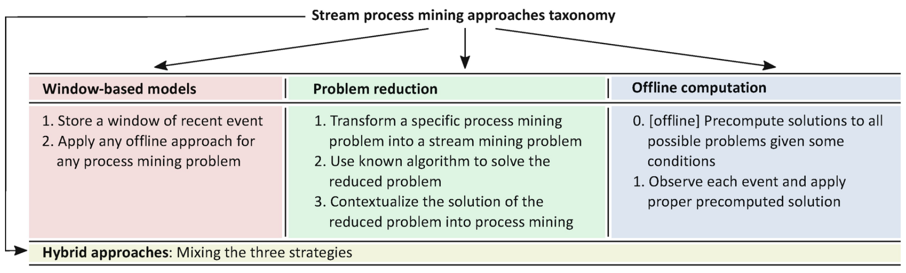
Stream process mining approaches taxonomy
Window-based models
1. Store a window of recent event
2. Apply any offline approach for any process mining problem
Problem reduction
1. Transform a specific process mining problem into a stream mining problem
2. Use known algorithm to solve the reduced problem
3. Contextualize the solution of the reduced problem into process mining
Offline computation
0. [offline] Precompute solutions to all possible problems given some conditions
1. Observe each event and apply proper precomputed solution
Hybrid approaches: Mixing the three strategies

Figure 3.5: Approaches to process event streams [11].

behavior by looking at the process model and aligning event logs to it. Finally, comparison approaches transform both the process model and event log into a common representation that makes both artifacts comparable [43].

In the case study of this thesis, we have analogous artifacts to those used in conformance checking: we can think of a set of traces as an event log and the sequence diagram as a process model. Moreover, due to how sequence diagrams and traces are represented in our application, we chose a comparison approach to perform conformance checking. However, as we will see in future chapters, some significant differences exist between our implementation and traditional conformance checking.

3.5 Streaming Conformance Checking

One open challenge in the research field of conformance checking is how to perform real-time conformance checking [15, 43]. As we saw before, most research done in the field focuses on applying conformance checking techniques after processes are concluded. There are, however, some papers that implement real-time conformance checking frameworks, often called streaming conformance checking. This discipline aims to investigate how to perform conformance checking for unbounded sequences of events in real time. Burattin [11] presents an overview of the general approaches that can be implemented when processing an event stream. The approaches are described in Figure 3.5.

The window-based models' approach is the simplest since it stores a predefined number of events before processing and analyzing them. There are different forms of storing these events. It is possible to define the window size by the number of events or by time. This choice depends on the specific needs of the algorithm implemented for performing the checking. However, this approach has the drawback of handling memory inefficiently, since it can require the storage of a large number of events, depending on the chosen window size.

Another approach is to reduce the conformance checking problem to a known stream processing one, to leverage the true and tried methods in this research area. This is only possible if the conformance problem at hand can make use of these techniques, such as frequency counting for example.

On the other hand, offline computation attempts to cache possible expensive computations before starting to handle the event stream. However, not all conformance checking problems can take advantage of this approach, and using it can make it harder to adapt the pre-computed solutions to the context that is being checked.

Finally, the author suggests considering a mix between the aforementioned approaches. Choosing an approach is mainly dependent on the problem that is being solved and the specific implementation needs and constraints.

Additionally, the author mentions some additional challenges to the successful implementation of a real-time conformance checker:

1. Arrival time of events is not equal to the time of their execution.
2. How to infer if a process instance has terminated.
3. Consider different types of streaming models.

Challenges 1 and 2 are directly related to our case study. Moreover, they are inverse qualities. For instance, we can receive events at the moment they are terminated. This would probably involve some overhead since we would not send events in batches to our checker, thus having more communication over the network. Nonetheless, spans would be delivered according to their finishing time and not starting time. Additionally, we would not be able to know when we have received all the events for a certain execution trace. In contrast, if we could wait until a trace is finished, the time between the execution of the first span of a trace and the actual processing would be greater. Furthermore, waiting for a trace to be finished means that we would have to persist data in memory for longer periods. Finally, challenge 3 calls for streaming conformance checking to allow for different types of streaming models. Streaming models can allow for events to be modified or deleted after being added to the stream, or consider all events as immutable. The latter is called an insert-only model and the former is called an insert-delete model. Streaming conformance checking research has mainly focused on insert-only models, remaining an open issue to consider insert-delete models for research in this field.

Chapter 4

Methodology

4.1 Case Study

In Chapter 2 we defined abstractly what conformance checking is and described the main components that are needed to perform the conformance checking evaluation. In Section 1.2 we outlined that we are interested in researching a way to perform fast, efficient, and correct conformance checking for a microservice application. Thus, in this section, we introduce the Meal Delivering Application (MDA) that will be the case study for this master thesis and serves as a test bed and as a data provider to allow us to develop and test our conformance checking architecture.

4.1.1 The Meal Delivering Application

MDA is a meal-delivering application composed of five unique microservices developed in C++. The application abstracts the flow of a kitchen and delivery service. The application considers that it has a certain amount of physical resources available to complete its processes. These resources are kitchens, that prepare meals, and bikes, that deliver meals. In Figure 4.1 we depict a simple diagram showcasing the microservices that comprise the MDA and the flows of communication between them. An overview of a typical workflow in the application and the role of each microservice is as follows:

Customers submit meal orders to the DELIVERY CONTROL service that are passed to the PLANNER scheduling service. After being scheduled, meal orders are dispatched by the MEAL DISPATCHING service according to the schedule defined in the previous step and the resources available at the moment dispatching occurs. Subsequently, the MEAL PREPARATION service prepares the meal in the kitchen assigned by the MEAL DISPATCHING service. Finally, the MEAL DELIVERY service dispatches the meal at the indicated time and allocates a bike to transport the meal to the receiver.

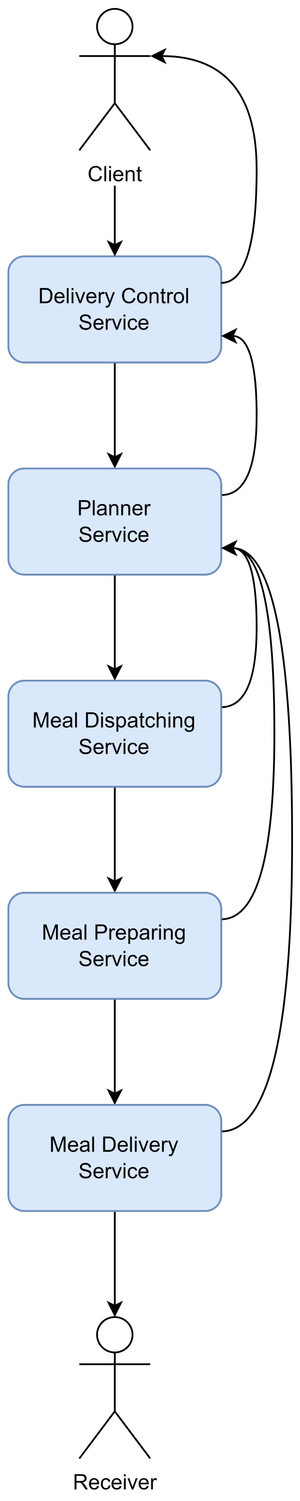

Figure 4.1: The Meal Delivering Application.

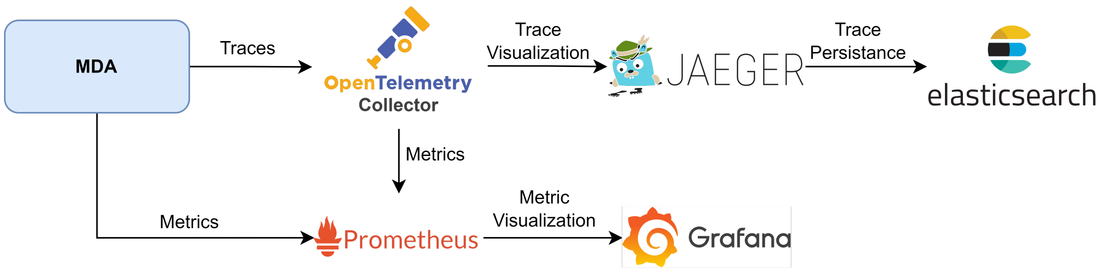

Figure 4.2: The Meal Delivering application and monitoring infrastructure.

The MDA was chosen as a case study for this master thesis because it gathers all the necessary conditions to implement our conformance checking architecture. Firstly, we have access to several sequence diagrams used to develop the application. Secondly, MDA follows a microservice architecture. Finally, the application is capable of producing traces that can then be checked for conformance against the initial sequence diagrams. All services that are part of MDA are manually instrumented with the OpenTelemetry library.

It is important to note that we are not particularly interested in the actual inner workings of the MDA. For instance, the arrows in Figure 4.1 can refer to different flows of information and operations in the application. However, we purposely leave it without a more accurate description since we only need to analyze the traces that result from the execution of the application and the initial sequence diagrams conceived at design time.

4.1.2 Monitoring Infrastructure

As we saw in Chapter 3, an important aspect of microservice applications is to be able to monitor them in order to observe their internal state and be alerted of possible erroneous or exceptional situations. As previously mentioned, all MDA services are instrumented and generate tracing data. This data needs to be collected to be suitable for analysis. Moreover, we are also interested in monitoring the resource consumption of the application. Thus, the full MDA architecture includes monitoring applications to achieve this goal. In Table 4.1 we describe the role of each application and in Figure 4.2 we depict the complete architecture and the flow of information. The box labeled "MDA" condenses all microservices that are part of the application.

4.1.3 Deployment Strategy

All services detailed in the previous section and the MDA are deployed in a Kubernetes [36] environment that runs at the in-house TNO Cluster. The

Service	**Role**
OpenTelemetry Collector [44]	Collect traces from MDA and expose its own metrics.
Jaeger [32]	Reconstruct traces received from the OpenTelemetry Collector for visualization.
ElasticSearch [24]	Persist application traces for future lookup.
Prometheus [48]	Scrape metrics from MDA and the OpenTelemetry Collector.
Grafana [29]	Query Prometheus to pull data to construct graphs and analyze metrics.

Table 4.1: Monitoring services and their role.

cluster consists of three nodes (machines), all capable of handling 110 pods (service instances). Furthermore, one node has 4 CPUs available while the other two have 16 CPU cores. Moreover, all nodes have 32GB of RAM that can be allocated. To avoid possible variability in application performance, we deploy all necessary services to the same node. This way we ensure that critical services do not end up in a less capable node and impact the execution of the whole application. It is important to note that pods within a Kubernetes cluster communicate using their internal IP addresses, over a virtual network overlay. Thus, the networking part is not lost by deploying every service in a single node.

4.1.4 Conformance Checking for MDA

Graphical representations or concrete notations are not optimized or easily used as input for a computer program. In Chapter 2, we have an accurate notation for sequence diagrams and traces that allows us to reason about the conformance checking problem in a general way. We now look into a particular instance of the problem for our case study, namely executing conformance checking for the MDA. These representations are the starting point for building our checker architecture. We will apply several transformations to both sequence diagrams and traces until we can develop a checker that can read the transformed representations and output a conforming or nonconforming result. Additionally, our approach uses common monitoring infrastructure and can be applied to other applications that are modeled with sequence diagrams.

Sequence Diagrams in PlantUML

PlantUML [47] offers a way to represent sequence diagrams as text. Moreover, this text can be used as input in the tool's website or plugins for common tools, like Eclipse [23], to generate graphic representations of the textual definition. Additionally, we include annotations to denote timing constraints as comments in PlantUML, so that the compiler can still generate the graphical representations. In Listing 4.1 we give the PlantUML specification of the From Order to Delivery sequence diagram that we will mainly use as input for our checker to run experiments. In Figure 4.3 we depict the graphical representation of the textual specification, where the timing constraint was manually added.

```
@startuml
title
Sequence diagram of "From Order to Delivery"
end title

participant DeliveryControlService
participant PlannerService
participant MealDispatchingService
participant MealPreparingService
participant MealDeliveringService

'@TimingStart 250000
PlannerService ->
    DeliveryControlService : MonitorNotification
MealDispatchingService ->
    MealPreparingService : MealPreparationRequest
MealDispatchingService ->
    PlannerService : ScheduleUpdateNotification
MealPreparingService ->
    MealDispatchingService : MealPreparationResponse
MealPreparingService ->
    MealDeliveringService : DispatchDeliveryRequest
MealDeliveringService ->
    MealPreparingService : DispatchDeliveryResponse
MealPreparingService ->
    MealDispatchingService :
       MealPreparationUpdateNotification
MealPreparingService ->
    PlannerService : MealPreparationUpdateNotification
MealDeliveringService ->
    PlannerService : DeliveryUpdateNotification
'@TimingEnd
@enduml
```

Listing 4.1: PlantUML specification of the From Order to Delivery flow.

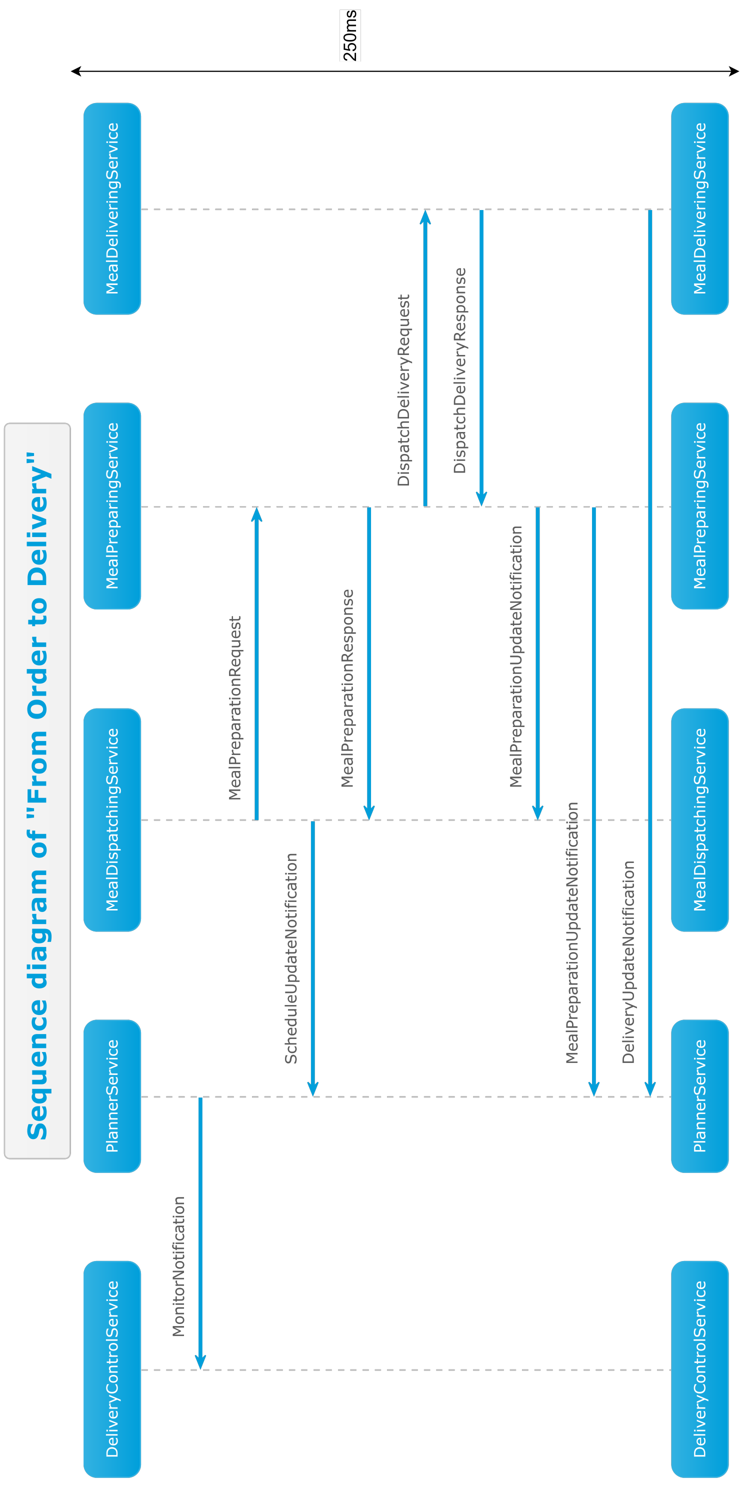

Figure 4.3: Graphical representation of Listing 4.1.

Traces in OpenTelemetry

The OpenTelemetry Collector provides a configuration that exports traces in JSON format. This is convenient because JSON is easily readable by humans and also easy to parse by computers. Listing 4.2 shows an example of a trace generated by MDA and exported by the OpenTelemetry Collector. Note that several fields were omitted since OpenTelemetry includes library-related fields that are not of interest to analyze. However, we can identify similar fields to the ones introduced in Chapter 2. We can observe a span that has a reference to the trace it belongs to, the start and end time of the span, along with its name and ID. This is a root span since its *parentId* field is empty. Moreover, we see an *orderId* field that allows for unique identification of a particular order placed to the MDA. Finally, we see that `PingMsg` was received by the *DeliveryControlService* (omitted in the listing), together with the time of reception.

```
{
  "scopeSpans": [
    {
      "spans": [
        {
          "traceId": "fc6775c3cba39099920bc0475b47627e",
            "spanId": "3f832d58d3596a00",
            "parentSpanId": "",
            "name": "postReceive",
            "startTimeUnixNano": "1699557626470949306",
            "endTimeUnixNano": "1699557626470957977",
            "attributes": [
              {
                "key": "order.id",
                "value": {
                  "stringValue": "MEAL1.0"
                }
              }
            ],
            "events": [
              {
                "timeUnixNano": "1699557626470955791",
                "name": "PingMsg"
              }
            ],
        }
      ]
    }
  ]
}
```

Listing 4.2: Example trace generated from the MDA

4.2 Approach

To tackle the research questions presented in Section 1.2, our approach involves executing a set of experiments leveraging the MDA. More specifically, we will design an offline architecture for conformance checking that will establish a benchmark for evaluating the performance of the online checker that will be developed.

4.2.1 Implementation

Given the requirements listed in Section 2.3, we can conceive a general idea of how our conformance checker will look to adhere to those requirements. For instance, R1 states that the checker shall provide correct results. To ensure that these criteria are met, we can envision an offline version of the checker that gathers all the execution traces for a predetermined amount of time and only after processing it in a big batch. This way, we can ensure that the checker will have access to all the context it needs to check if the set of traces persisted contains one or more subsets that conform to given sequence diagrams. However, this approach directly contradicts R3. In the worst case, we will get a conformance result equivalent to the amount of time that we were gathering the execution traces. One possible strategy that can be adopted is online (real-time) processing. In this paradigm, events are processed at the moment the checker receives them, therefore the checker can yield faster conforming or non-conforming results. However, the online approach presents several challenges that are not found in the offline approach. Thus, we will implement an offline version of the checker that will serve as a baseline for our online checker.

4.2.2 Experiments

Due to the inherently distributed nature of microservice applications and their dynamic deployment infrastructure, replicating experimental runs and comparing runs with each other poses some challenges. For instance, variability in execution times between runs caused by changes in networking conditions or resource availability in the cluster to handle the workload of the application can hinder direct comparisons between different runs using the same version of the MDA.

To mitigate these challenges, we propose a design for our conformance checking architecture that enables multiple instances of the conformance checker to run concurrently. This approach facilitates the comparison of different instances of the checker within the same execution of the MDA. Furthermore, we can attempt to replicate a particular run by persisting the tracing data generated during that execution of the MDA in ElasticSearch. The

Metric	**Definition**
Total Checks	Number of checks performed.
Conforming Checks	Number of checks with a conforming result.
Nonconforming Checks	Number of checks with a nonconforming result.
Check Time	Duration of a check since it is started until it finishes.
Reaction Time	Time elapsed since the first interaction is executed in the application until the checker finishes.

Table 4.2: Checker metrics.

subsequent chapter will provide a more detailed exploration of the implementation and benefits of this methodology.

The experiments will consist of executing predefined workflows in the application while executing the different instances of our checker in parallel and capturing metrics that allow us to directly compare and evaluate the performance of the instantiated checkers. Moreover, the workflows vary in size and stress the MDA in different ways, making it possible to observe how the checker handles different scenarios and loads. Additionally, these experiments intend to allow us to verify if the proposed implementations adhere to the requirements in Section 2.3.

4.2.3 Metrics for Evaluation

To compare different conformance checking architectures we need to have the right set of metrics that indicate whether a certain implementation of the checker is superior or more advantageous than another. Thus, we propose two categories of metrics to evaluate the implemented checkers: checker metrics and hardware metrics.

Checker Metrics

The objective of the checker metrics is to measure the checker's performance and compare it across different implementations. These metrics will provide an accurate way of understanding if the implemented checkers are behaving as expected. In particular *Total*, *Conforming*, and *Nonconforming Checks* allow us to assess the correctness of the online checker against the offline baseline, while *Check* and *Reaction Time* relate to how fast a checker iteration is. Table 4.2 describes the checker metrics of interest.

Metric	**Definition**
CPU Usage	CPU percentage consumed by the services related to the checker implementation. (50% would mean using 8 cores)
RAM Usage	RAM percentage consumed by the services related to the checker implementation. (50% would mean using 16GB)

Table 4.3: Resource metrics.

Hardware Metrics

Finally, the hardware metrics capture resource consumption from the checker and different services and applications that are part of the checker architecture. Table 4.3 describes the hardware metrics of interest.

Chapter 5

Related Work

5.1 Conformance Checking

In Chapter 3, we introduced the main concepts related to conformance checking as well as finding which of these concepts are also present in our case study. However, our approach and implementation differ in some parts from traditional research of conformance checking.

Firstly, we do not intend to implement formal mechanisms or calculations usually seen in conformance checking studies to derive conclusions from the execution of our application. Thus, we are not interested in calculating the dimensions described in Section 3.4. It suffices that we compare the recorded traces from the execution of the application to the sequence diagram of choice and output a conforming or non-conforming result indicating where the set of traces deviates from the expected behavior modeled in the sequence diagram.

Secondly, it is common for conformance checking papers to use Petri nets to represent process models [22]. In contrast, we choose to use sequence diagrams as our starting point to apply conformance checking. There are tools and approaches that map sequence diagrams to Petri nets [3, 25, 26, 42, 56] that would allow us to take advantage of process mining techniques to perform conformance checking. Nonetheless, this would require the execution of an extra step, either using a third-party tool or implementing one ourselves, and even then it would not be clear how to use it to our advantage, since it would require a deeper knowledge of process mining concepts and terminology.

Thirdly, process models in conformance checking usually refer to business processes [15]. It is of little importance for our case study to relate the application's execution to business processes to reach actionable conclusions. In contrast, we aim to identify flaws and commonalities in the application's execution without worrying about its business relation.

Finally, a major part of the research output on conformance checking techniques focuses on situations where a complete event log is available [43]. An event log is considered to be complete when the process finishes its execution and can then be processed. As we saw before, this is called offline processing. However, in our case study, we are dealing with microservice applications that are always executing and have an unbounded sequence of traces. Hence, it is difficult to determine when a set of traces is complete, even if we only check for conformance in long fixed time intervals (eg. each day at night).

In summary, traditional conformance checking is conceptually related to the problem we want to solve, but its usual formulations and techniques do not apply to our case study. Besides, to the best of our knowledge, there are none or limited studies that research conformance checking for microservice applications. Nonetheless, we apply analogous concepts and techniques to our implementation in the following chapters.

5.2 Streaming Conformance Checking

We now review existing literature and methodologies related to streaming conformance checking. We start by noting that a substantial amount of research papers follow an alignment-based technique [12, 49–51, 57, 59], behavioral patterns [13], or a mixed approach between alignments and probability [1, 39, 54, 58] to perform streaming conformance checking. However, these are outside of the scope of this thesis since we opted to use a comparison approach to perform conformance checking and leverage some similar concepts to the ones found in token-replay-based techniques. We leave for future work to investigate the suitability of alignment techniques for our case, and if they yield better results.

Broucke et al. [10] propose a four-step process based on the token-replay technique to perform online conformance checking: decomposition, event dispatching, replay, and reporting and visualization. The initial stage of the proposed methodology involves the decomposition of the overall system Petri net. This process aims to enhance the efficiency of subsequent analyses and decision-making tasks by breaking down the complex system into a collection of subnets. This strategy allows for a faster processing of each event. Following the decomposition phase, the collection of subnets is handed over to an event dispatcher. This component is designed to listen for incoming events and assess the presence of transitions within each submodel corresponding to a given event. The prototype implementation deploys multiple worker threads, each responsible for a specific number of submodels, allowing for concurrent checking over multiple subnets. The third step involves the actual replay of events on the determined submodels from the previous phase. This token-based replay approach is preferred over alignment and behavioral techniques, as it allows for event-level analysis rather than trace-level analysis. The choice of replay is driven by scalability considerations, as alternative techniques face challenges in real-time contexts. A potential drawback of the replay technique is the generation of superfluous tokens, which may introduce undesired behavioral complexities. Finally, the system logs various statistics using worker threads and replayers. Decoupled components, such as dashboards or persistent data stores, regularly poll and collect these statistics. Additionally, the methodology incorporates triggers to detect specific situations, such as error rates surpassing predefined thresholds.

Berti and Aalst [8] present an innovative approach to token-based replay techniques aimed at addressing challenges such as the slow traversal of invisible transitions and the token flooding problem, which involves placing missing tokens to facilitate transitions, potentially leading to a state-space explosion. Instead of exhaustively exploring all possible states, the authors focus on minimal markings where a target transition is enabled. They achieve this through the implementation of a backward replay technique, departing from more conventional methods. Additionally, the paper introduces strategies for identifying frozen tokens that will never be consumed, leading to their removal from the marking. Moreover, the authors adopt techniques used in alignment-based conformance checkers to improve the performance of their innovative approach, such as post-fix caching and activity caching.

In our implementation, we will employ several strategies and techniques used in the above-mentioned papers. The process detailed by Broucke et al. [10] gives some light on how we may develop our conformance checker. In particular, the clever use of worker threads and mapping the right event logs to worker instances will be similarly implemented for our case study.

Moreover, while our implementation may not explicitly involve tokens, we anticipate encountering a similar challenge to the frozen tokens problem. Some instances of our conformance checker may become indefinitely stuck, awaiting events that signal the termination of a sequence. We can draw inspiration from Berti and Aalst [8]'s approach, to incorporate strategies into our conformance checker to address this issue. One possible strategy is to define a maximum checking time, after which the checker is automatically concluded. Additionally, the unconventional use of token-based replay with backward replay also offers valuable insights for optimizing our checker. We can adopt a less traditional sequence matching approach, where we collect all events related to a specific sequence and only after determining the sequence as complete or reaching a predefined timeout, execute the conformance checking algorithm, as opposed to processing the sequence in a strictly sequential manner.

5.3 Runtime Verification of Microservices

While related work in conformance checking helps us grasp important concepts and strategies on how to implement our conformance checker, it leaves out important details in the context of our case study: microservice applications. That is where runtime verification comes in. Runtime verification can be considered to be the boundary between forms of formal checking and testing, having a more practical focus than formal verification and a better view of the execution of the application than testing approaches. Hence, we will look at common challenges and implementation approaches that are available in the literature for runtime verification of microservice applications.

As pointed out in the rapid review from Abdelfattah and Cerny [2], there is a lack of adequate tools for verification in microservice applications. This is mainly because, in an industrial setting, several obstacles hinder the usage of proposed tools in the literature. Such obstacles are, for instance, lack of or incorrect specifications of the executing applications, which most tools depend on to detect violations in the execution. As we will see, the specification of the applications itself can be done in several different ways which makes it difficult to come up with a general solution for most cases. Let us look at some proposed tools and methodologies for the verification of microservice applications.

ucheck [46] is a tool that checks and enforces invariants, such as RPCs sequences, in real-time for microservices. Moreover, upon detecting invariant communications, *ucheck* prevents the execution of such RPC sequence. The effectiveness of the tool is dependent on user-provided models where behavior semantics are defined to allow the tool to check for invariants. *ucheck* is deployed at the virtual network layer as a module on the packet processing pipeline. This allows for isolation from the microservice itself while capturing network traffic sent and received by all microservices. This network traffic is converted into meaningful messages that can be checked for invariants against the supplied models. However, there are some meaningful limitations to this approach. Firstly, the performance of *ucheck* worsens as the size of the supplied models grows, potentially hindering application performance given the placing of the tool in the architecture of the systems. Moreover, it is not possible to detect all invariants detailed in the models. *ucheck* does not access the microservice state and some invariants might require coordination across microservices to be detected, which in the worst case would require communication between all microservices in the application, adding high performance overhead to the execution. Finally, the tool assumes that it has access to the messages exchanged between the microservices. If an encryption method is used, *ucheck* would not be able to analyze the message. To overcome this challenge, it would be necessary to rethink the tool's placement in the architecture, potentially moving it to a higher level than the network layer.

We can envision some similar problems to the ones related to the *ucheck* tool. The most related obstacle is regarding the size of the models themselves. In our case study, if a sequence diagram is very big, for instance containing dozens of interactions, the time to achieve a conforming or non-conforming result will be naturally big, with an upper bound being the difference between receiving the first and last trace necessary to complete the check. However, it will not have an impact on the application's performance since our checker will not be placed in between microservices, but rather alongside it. Furthermore, as we won't be capturing messages directly from the microservices themselves but instead instrumenting the microservices to produce tracing data, we have control over which data is placed on the traces, thus allowing for encrypted communication during the normal execution of the application.

Camilli [14] propose a different approach based on Netflix's Conductor [18] workflow orchestration tool. For applications that were built using Conductor, this approach eliminates the need to specify models for applications just for the purpose of verification. Additionally, Conductor blueprints are transformed into time-basic Petri nets, making it possible to take advantage of conformance checking techniques. This results in a complete abstraction from the microservice implementation and the verification is done by comparing execution traces with execution paths in the Petri net. This, in turn, allows for an online conformance checking paradigm but, in contrast with *ucheck*, it won't stop nonconforming operations from being executed. However, the framework is easy to integrate into the architecture of a microservice application, given that the application takes advantage of the Conductor tool for modeling. The Conductor tool is indeed the biggest disadvantage of this approach since it is not as commonly used for modeling as UML for example. Nonetheless, the general idea of transforming the initial modeling representation to a suitable one for comparison with execution traces can adapted for our implementation.

Finally, we review the lessons learned by Colombo and Pace [17] when implementing runtime verification in industry. Their work results in a list of recommendations, gathered from the experience of implementing runtime verification in two case studies, that can be followed by developers to implement runtime verification in other settings. These recommendations are divided into four different areas, which we detail in the next paragraphs.

Engineering the properties. Before developing a tool or framework for runtime verification, it is important to define what properties should be monitored that are not captured by other methods, like testing, along with who should define those properties and how they should be expressed.

Engineering the verification code. The authors recommend either finding a tool that is capable of generating verification code automatically or developing one from scratch.

Architecture design. Another choice that has to be made is what paradigm will be followed by the verification tool and where it will be placed in the overall application architecture. The authors identify three different possibilities: following an offline approach, completely decoupling the verification tool from the main application and executing it when all the needed outputs are available; following an online asynchronous approach, where the verification tool stays decoupled from the main application but delivers results in real-time; or adoption an online synchronous approach, like *ucheck*, and be able to stop wrongful messages from propagating throughout the main application, and stopping its execution.

Event extraction design. The final decision is related to how the tool will capture the events it needs to verify the executing application. The authors propose three possible implementations: a method-call-based approach that stops execution to capture the event, communication-based events that are caught at the network layer, and an event-by-design approach that involves the additional step of instrumenting the application to generate events of interest.

Chapter 6

Offline Conformance Checking

6.1 Approach

As previously detailed in Section 4.2.1, we will develop a first version of our checker under some assumptions that simplify the conformance checking problem at hand. We aim at first developing a checker application that correctly outputs conforming or non-conforming results (R1), given the execution of our MDA compared to the initial sequence diagrams, has a minor impact on the MDA's execution (R2), and can support multiple sequence diagrams checking concurrently (R5). Moreover, since the offline paradigm offers less challenge than an online approach, the offline iteration will be used as a baseline for comparison against the online iteration in Chapter 7. For this iteration of our checker, we assume we have access to all the traces generated for a given workload on the MDA. We only execute the checker application after the workload is executed in the MDA. In our approach to developing our conformance checking application, we follow the recommendations from Colombo and Pace [17], described in Chapter 5.

Engineering the Properties

As introduced before in Chapter 2, we want to monitor properties of a microservice application. More specifically we want to monitor if the communication between the microservices that compose the MDA execute according to specification and within a specific time frame. Moreover, these constraints are defined by the application developers and expressed in UML sequence diagrams. Furthermore, we verify that our current infrastructure in Figure 4.2 does not monitor these properties. While using Jaeger in combination with Grafana can help us analyze and draw some conclusions regarding the latency and the order of the execution traces from the MDA, it is very hard or nearly impossible to accurately monitor the properties we defined. It is possible to query Jaeger to display the latency of an individual or an aggregation of traces in Grafana. Moreover, this is consistent with findings of Bento et al. [7] that report the need for automated analysis besides Jaeger's ability to visualize tracing data. Still, since the traces do not *directly* correspond to the initial sequence diagrams, this approach would not suit our use case.

Engineering the Verification Code

From Chapter 5, we know that there are no tools that directly tailor to our conformance checking needs to monitor the aforementioned properties. Additionally, trying to leverage existing monitoring infrastructure is also not helpful in fulfilling our case study. Thus, we decide to develop our conformance checker from scratch.

Architecture Design

The next choice to be made is to choose what paradigm our checker should adhere to. For now, we will follow an offline approach to develop the first iteration of our conformance checker. The goal is to later improve this offline version to an online asynchronous approach to make sure our implementation adheres to our requirements in Section 2.3. Moreover, this choice has a direct influence on the modifications that will be made to the already deployed infrastructure for the MDA. As we will see in Section 6.2.1, these initial modifications are made to later allow an easier transition to the online iteration of the checker application.

Event Extraction Design

Finally, we detail how the checker will capture the execution traces from the MDA. We follow an event-by-design approach, leveraging the fact that the MDA is already instrumented with OpenTelemetry and, thus is capable of emitting traces that are captured by the OpenTelemetry Collector. Moreover, the OpenTelemetry Collector ships with multiple plugins and is highly configurable, giving us the flexibility to choose which tools to use to achieve our goals.

Checking Scope

Although this is not a part of Colombo and Pace's recommendation, we find the need to detail some particularities of the MDA application and how this influences the development of our checker applications.

Due to the way that the MDA is developed and instrumented, we have to specify which types of checks and sequence instances our checker will support. To check the conformance of execution traces against the sequence diagram in Figure 4.3, we will assume that each sequence can be uniquely identified. This is done through a user-defined field in the instrumentation of the MDA called *orderId*, a string that corresponds to the name of the order placed by the user. Looking back at our definitions in Chapter 2, the interactions derived from a trace set include an *orderId*, plus all the original fields.

Moreover, the *orderId* field can carry several values. As mentioned before, this value can be the name of the order itself, but it can also be `noOrder`. The MDA supports and applies batch processing in some of its workflows. Let us consider the following example: a user places three orders to the MDA named `Meal1`, `Meal2`, and `Meal3`. The interaction generated from the execution traces can contain any of the three order names plus `noOrder` as value for the *orderId* field. If the value of that field is one of the specific order names, we know that it relates unequivocally to that order. However, if the field assumes the `noOrder` value, it can mean that it refers to a batch operation related to all submitted orders. We will disregard this case and assume that the interaction does not fulfill the sequence we want to check since this would require a more complex logic to decide when to include this interaction as part of a valid sequence or not.

Finally, we will reject derived interactions that are not useful for the sequence diagram in use. Similarly to the `noOrder` case explained above, we will disregard interactions whose message field does not match any of the interactions in the sequence diagram, even if they possess matching *orderId* fields. This decision implies that we might give a conforming result to a particular set of traces that has an unexpected interaction that is not part of the sequence diagram but took part during the execution of the application. Furthermore, interactions that have the same attributes apart from the time they occurred will also be discarded. This is again due to the specific behavior of the MDA. Thus, we will only consider the first occurrence of an interaction for each order when performing the conformance checking. This will also be the case for the online iteration.

Let us consider again our notation from Chapter 2, where we add the *orderId* field introduced in this section as the last value of an interaction, and the sequence diagram in Figure 2.2. As an example, consider that we derive the following interactions from a set of execution traces:

$$\begin{aligned}
I &= (i_1, i_2, i_3, i_4, i_5, i_6, i_7),\\
i_1 &= (message, ServiceA, ServiceB, messageA, Meal1),\\
i_2 &= (message, ServiceA, ServiceB, messageA, noOrder)\\
i_3 &= (message, ServiceB, ServiceC, messageB, Meal1),\\
i_4 &= (message, ServiceA, ServiceB, messageA, Meal1)\\
i_5 &= (message, ServiceB, ServiceC, messageD, Meal1)\\
i_6 &= (message, ServiceA, ServiceB, messageA, Meal2),\\
i_7 &= (message, ServiceC, ServiceA, messageC, Meal1).
\end{aligned}$$

We want to check if the `Meal1` order conforms to the sequence diagram in Figure 2.2. Following the conditions outlined in this section, we would only accept i_1, i_3 and i_7. These directly match the interaction that can be derived from the sequence diagram since they follow the order depicted in the diagram, have the same message contents, and have the same actors sending and receiving the messages. i_2 would be rejected since it has `noOrder` as a value in the *orderId* field. Similarly, i_6 would be rejected because it concerns the `Meal2` order and not the intended `Meal1`. In the case of i_4, we would reject the interaction given that it is a repetition of i_1 at a different time. Since we consider *orderId* values to be unique we will not consider the cases where interactions are repeated. Finally, i_5 is also rejected since its *content* field contains value `messageD`, that does not participate in this sequence diagram. We will not give a nonconforming result since this interaction can potentially be part of another sequence diagram.

6.2 The Offline Checker

We will now present the specifics of the first iteration of our conformance checking application. We start by refining our initial architecture by including a service that collects the execution traces from the MDA, followed by the preprocessing of these traces so that they are in a suitable format to be used by the checker. Finally, we introduce the algorithm that matches a sequence diagram to the processed execution traces.

6.2.1 Architecture

To accommodate our checker we have to make some changes to the overall architecture presented in Figure 4.2. As previously stated, the OpenTelemetry Collector gathers the execution traces from the MDA. We could directly export the traces from the Collector to our checker application. However, this would couple both applications together, which could lead

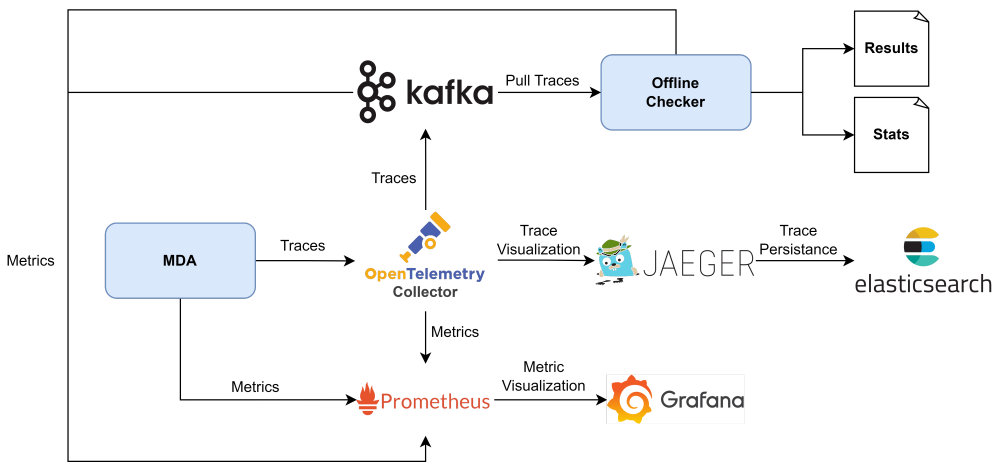

Figure 6.1: Architecture for MDA and Offline Checker.

to some traces being lost if the checker cannot process the traces as fast as the Collector can export them. Instead, we deploy Apache Kafka [5], an open-source distributed event streaming platform that sits between the Collector and the checker, uncoupling both services, while providing highly available and resilient storage for traces as a message queue. The Collector publishes execution traces to Kafka and the checker consumes these traces at different rates. Moreover, Kafka will be particularly useful when we move to the online paradigm for our checker. This is detailed in Chapter 7. The new architecture is depicted in Figure 6.1.

Furthermore, this architecture follows good practices of implementation for observability workflows [55]. The MDA is instrumented with OpenTelemetry and emits telemetry data. This data is then collected by the OpenTelemetry Collector and stored in a Kafka Queue. The checker processes and analyzes telemetry data and finally reports the results through output files.

6.2.2 Processing Execution Traces

Before delving into the checking phase, a crucial preprocessing step needs to be performed to refine and structure raw sequence diagrams and execution traces. We opted to apply a comparison-based approach to perform conformance checking. Thus, it is essential to have both the sequence diagram and execution traces in a comparable format for our checker to produce results. In Chapter 2, our formal definition gives a generic format that can be derived from both artifacts making them easily comparable. Moreover, at the beginning of this chapter, we introduced the *orderId* field, which makes a series of execution traces uniquely identifiable for checking.

On one hand, to achieve this format for sequence diagrams, we process

PlantUML definitions to generate the set of interactions that characterize a sequence diagram. The resulting format for the PlantUML specification in Figure 4.3 is given in Listing 6.1. This format is a very close match to our previous formal definition of sequence diagrams. The first and last lines allow us to identify the timing constraint that the sequence diagram imposes. All the other lines are an exact match to our interaction definition. This format is saved as a file that can be loaded by the checker when performing conformance checking.

```
["TimingStart", "250000", "", ""]
["Message", "PlannerService", "DeliveryControlService",
   "MonitorNotification"]
["Message", "MealDispatchingService",
   "MealPreparingService", "MealPreparationRequest"]
["Message", "MealDispatchingService", "PlannerService",
   "ScheduleUpdateNotification"]
["Message", "MealPreparingService",
   "MealDispatchingService", "MealPreparationResponse"]
["Message", "MealPreparingService",
   "MealDeliveringService", "DispatchDeliveryRequest"]
["Message", "MealDeliveringService",
   "MealPreparingService", "DispatchDeliveryResponse"]
["Message", "MealPreparingService",
   "MealDispatchingService",
 "MealPreparationUpdateNotification"]
["Message", "MealPreparingService", "PlannerService",
   "MealPreparationUpdateNotification"]
["Message", "MealDeliveringService", "PlannerService",
   "DeliveryUpdateNotification"]
["TimingEnd", "", "", ""]
```

Listing 6.1: Example transformation of a PlantUML specification.

From the execution traces, there is more work that needs to be done. The traces are published to Kafka in the format shown in Listing 4.2. Even though this is a structured format, it is not very flexible and workable to perform conformance checking. Thus, we parse traces into custom Python objects that can be manipulated according to our specific needs. This is followed by the execution of the `findInteractions` procedure. This procedure follows the logic presented in Section 2.2.3 to derive interactions from the spans that are part of a specific trace. In contrast with our definition, the output of `findInteractions` has some differences. These differences simplify the processing of the traces themselves and the `findInteractions` procedure detailed below. Firstly, the interactions do not include the type field, since we assume that all interactions gathered by the procedure are of type `message`. Moreover, we include a `timestamp` field that represents the time that the interaction was started (when the message was sent). This field will be used to calculate the time elapsed between the first and last interaction in a set of interactions being checked. Additionally, as introduced earlier in this chapter we have an `orderId` field to uniquely identify each set of interactions. Also, in our object representation for the spans in each trace, there is a reference to the direct children of that span. This is done by following the `parent` field. Each trace object has a reference to its root span. Finally, we note that some extra fields are generated by the procedure that are not important for the conformance checking. Hence, we will omit them. The `findInteractions` procedure is listed in Listing 6.2 and an example of a possible output is shown in Listing 6.3.

```
function findInteractions(span, interactionList)
    if span.operation
        if span.operation = "preSend"
            interactionFound <- FALSE
            for child ∈ sp.children
                if child.operation = "preReceive"
                    interactionFound <- TRUE
                    interactionList.add([span.service,
                       child.service, child.content,
                       span.start, span.orderId])
                endif
            endfor
            if !interactionFound
                for child ∈ span.children
                    findInteractions(child,
                       interactionList)
                endfor
            endif
        endif
    endif
    else
        for child ∈ sp.children
            findInteractions(child, interactionList)
        endfor
    endelse
endfunction
```

Listing 6.2: `findInteractions` procedure.

```
["PlannerService", "DeliveryControlService",
   "MonitorNotification", t, "MEAL10.9"]
["MealDispatchingService", "MealPreparingService",
   "MealPreparationRequest", t, "MEAL10.9"]
["MealDispatchingService", "PlannerService",
   "ScheduleUpdateNotification", t, "MEAL10.9"]
["MealPreparingService", "MealDispatchingService",
   "MealPreparationResponse", t, "MEAL10.9"]
```

```
["MealPreparingService", "MealDeliveringService",
    "DispatchDeliveryRequest", t, "MEAL10.9"]
["MealDeliveringService", "MealPreparingService",
    "DispatchDeliveryResponse", t, "MEAL10.9"]
["MealPreparingService", "PlannerService",
    "MealPreparationUpdateNotification", t, "MEAL10.9"]
["MealPreparingService", "MealDispatchingService",
    "MealPreparationUpdateNotification", t, "MEAL10.9"]
["MealDeliveringService", "PlannerService",
    "DeliveryUpdateNotification", t, "MEAL10.9"]
```

Listing 6.3: Example output of the `findInteractions` procedure. The timestamp field is replaced by `t` to avoid long lines. All `t`s represent distinct timestamps in nanoseconds.

As we will see in the next section, the `findInteractions` procedure is executed after each trace is processed and a trace object is created. Each trace has a variable pointing to the root span of that trace. The procedure is first called with the root span and a list object passed as a reference. This list will be modified with the interactions found by the procedure. The procedure starts to evaluate if the operation of the span is `preSend` in Line 3. If this condition is true, the algorithm loops through the children of that span to find if any have an operation name with the value `preReceive` between Line 5 and Line 10. For each child span that contains a `preReceive` operation name, an interaction is added to the `interactionList` variable in Line 8. Alternatively, if no interaction is found, `findInteractions` is called for each child of that span between Line 11 and Line 15. Similarly, if the condition in Line 2 is false, `findInteractions` is again called for each child of the original span between Line 19 and Line 21.

6.2.3 The Checker Algorithm

Finally, we put everything together and develop the logic behind the offline checker. To perform offline conformance checking, we execute the following steps:

1. Consume and buffer all messages in Kafka.
2. Stop the MDA execution to stop the production of more execution traces.
3. Preprocess all traces to generate the set of interactions to be checked.
4. Find all unique orders present in the interactions.
5. For each order, find the relevant interactions that are part of the sequence diagram. This is where the actual checking is performed.

6. Output results and stats.

A more detailed description of this process is shown in Listing 6.4.

```
sd <- ordered list of processed sequence diagram
   interactions
t <- timing constraint in sd

kafka <- queue of execution traces

traceObjs <- []
for event ∈ kafka
    traceObj <- readTrace(event)
    tracesObjs.add(traceObj)
endfor

interactionList <- []
for trace ∈ traceObjs:
    findInteractions(trace.rootSpan, interactionList)
endfor

interactionList.sort(key <- interaction.timestamp)

orderSet <- {}
for interaction ∈ interactionList
    orderSet.add(interaction.orderId)
endfor
orderSet.remove('noOrder')

for order ∈ orderSet
    sdCopy <- sd
    selectedInteractionsSet <- {}
    // finding interactions that are part of sd
    for interaction ∈ interactionList
        if interaction.orderId != order
            continue
        endif
        if interaction ∈ sdCopy ∧ interaction ∉
           selectedInteractionsSet
            selectedInteractionsSet.add(interaction)
            sdCopy.remove(interaction)
        endif
        if sdCopy = []
            break
        endif
    endfor
    // verification
    selectedInteractionsSet.sort(key <- interaction.
       timestamp)
    duration <- selectedInteractionsSet[-1].end -
```

```
            selectedInteractionsSet[0].start
        if sdCopy != []
            writeResults("NONCONFORMING_MISSING",
                selectedInteractionsSet, duration)
        endif
        for j ∈ [0..len(selectedInteractionsSet)]
            if selectedInteractionsSet[j] != sdCopy[j]
                writeResults("NONCONFORMING_OUT_OF_ORDER",
                    selectedInteractionsSet, duration)
            endif
        if duration > t
            writeResults("NONCONFORMING_TIME",
                selectedInteractionsSet, duration)
        endif
        writeResults("CONFORMING", selectedInteractionsSet,
            duration)
endfor
```

Listing 6.4: Algorithm for the offline conformance checker.

The `readTrace` function receives a Kafka event in JSON format, parses it, and returns a custom Python trace object containing relevant information for each trace. Each trace object is a collection of span objects, similar to our formal definition in Chapter 2.

The `writeResults` procedure outputs the conformance checking results to a file so that after the execution of the checker we can visualize the results. Moreover, in this listing, we omitted code related to capturing and calculating statistics for simplicity. These statistics are also an output of our conformance checker.

Additionally, we adopt a strategy that minimizes the impact of out-of-order or incomplete traces. This is mitigated by configuring the OpenTelemetry Collector with the `Group by Trace Processor` [30] that is configured to hold individual spans for 10 seconds and aggregate them by `traceId`. This ensures that if all spans that compose an execution trace arrive at the Collector within a 10-second time window, the complete trace will be forwarded to our Kafka queue. Most execution traces of the MDA have a duration that is lower than 10 seconds, with some occasionally taking 4 minutes or longer. This is due to the way the MDA is instrumented. However, this detail does not impact the execution of the checker or its results, since these execution traces are not part of the sequence diagram in use. It is important to note that this is a configurable parameter. It is possible to configure the OpenTelemetry Collector to hold traces for longer periods, depending on the memory available for the Collector. Additionally, given that the checker collects all the execution traces available in Kafka and only after starts processing them allows for a trivial solution to out-of-order events: it is sufficient to sort interactions by their timestamp as indicated in Line 11 of Listing 6.4.

The algorithm is divided into three different parts: initialization, finding interactions, and verification.

Initialization. We start by loading the desired sequence diagram, already processed like in Listing 6.1, along with the timing constraint associated with the diagram in Lines 1 and 2. This is followed by the checker initiating a connection to the Kafka queue where the OpenTelemetry Collector is publishing raw traces in Line 4. After we stop the MDA's execution, we read all traces in Kafka, creating a trace object for each event present in the queue and adding the trace objects into the `traceObjs` list between Line 6 and Line 10. After the creation of all the trace objects, we proceed to execute the `findInteractions` procedure for all traces and sort the resulting `interactionsList` in Lines 12 to 17. To finalize the initialization part, we again loop through all interactions to find all the unique order ids present in the list between Line 10 and Line 22. Additionally, we remove the `noOrder` value from the `orderSet`, per the scope we defined at the beginning of this chapter.

Finding Interactions. In this part, we start the actual checking by finding all the interactions that relate to the orders present at the `orderSet`, one by one. First, we create a copy of our sequence diagram in Line 26, to allow for the reuse of the sequence diagram without loading it again. This is followed by collecting all the interactions that have the *orderId* field equal to the `order` variable in the for loop that starts on Line 25, and that are part of `sdCopy`. In Line 30 we reject interactions that don't have a matching *orderId* and in Line 33 we accept interactions that are part of `sdCopy` and are not already present in the `selectedInteractionSet` variable, to avoid duplicates. Additionally, we remove the equivalent interaction from `sdCopy` in Line 35, given that when we have an empty sequence diagram, we know to have found all the needed interactions that belong to the sequence diagram for the initial order id. This is evaluated in Line 37.

Verification. Upon finding a complete set of interactions, we check if the interaction present in `selectedInteractionsSet` adhere to the constraints imposed by the sequence diagram. We start by sorting the interactions in the set in Line 42 by their timestamp. After, we calculate the total duration of the sequence by subtracting the timestamp of the first event in the sorted set from the last event in the set in Line 43. We start by checking in Line 44 if the `sdCopy` variable is empty. If not, it means we haven't found a complete set of interactions that satisfy the sequence diagram in the loop of Line 29. Thus, we output a nonconforming result due to some interactions being missing. If we have a complete set of interactions, we loop over the sorted set between Lines 47 and 50 and compare it to the order in `sdCopy`. If at any point there is a mismatch between the `selectedInteractionsSet` and the sequence diagram, we output a nonconforming result due to some interactions being out of order. The last check to be made is if the set of interactions adheres to the timing constraint `t`. If the total duration is bigger than `t`, we output a nonconforming result due to time violation in Lines 51 to 53. Finally, if all conditions are satisfied, we output a conforming result in Line 54.

6.2.4 Evaluation

We now evaluate the implementation of our offline conformance checker. The goal of this section is to assess if the offline implementation meets the requirements defined in Section 2.3. We start by outlining the experiments' design, followed by an in-depth analysis of all the experimental runs and how the checker is impacted by higher-volume experiments.

Experimental Design

As we saw before, in order to generate telemetry data to be used by the offline checker, we need to submit meal orders to the MDA. These orders can be configured to be executed in a determined amount of time. Moreover, the more orders are submitted to the MDA, the more telemetry data is generated, and the longer the execution time of the MDA and the checkers are. We draw up the experiments described in Table 6.1 to evaluate the performance of the offline checker, following the metrics described in Section 4.2.3 and against the requirements in Section 2.3. Experiment 0 allows us to evaluate if the checker is correct (R1), by submitting an order that should be labeled as conforming, one that should be labeled as nonconforming due to a time constraint violation, and finally, an order that will be stopped mid-execution and should be labeled as nonconforming due to missing spans. These three orders demonstrate that the checker correctly identifies and processes all relevant cases that we want to check. The rest of the experiments consist of 20-second orders with varying order sizes. Orders are submitted to the MDA with a 10-millisecond interval in each experiment. This allows us to grasp the efficiency (R2), fastness (R3), and scalability (R4) of the offline checker. Finally, we will also discuss the checker's ability to use multiple sequence diagrams (R5) to conclude our evaluation.

R1: Correctness

We start by determining if the checker can correctly identify conforming and nonconforming sequences. As mentioned earlier, this is the main purpose of Experiment 0. In the sequence diagram used for these experiments,

Experiment No	No of Orders	Order Time	Delay b/w Orders
0	3	20s, 30s, 60s	10 ms
1	1	All 20s	10 ms
2	10	All 20s	10 ms
3	50	All 20s	10 ms
4	100	All 20s	10 ms
5	200	All 20s	10 ms
6	300	All 20s	10 ms
7	400	All 20s	10 ms
8	500	All 20s	10 ms

Table 6.1: Description of experiments executed. The delay between orders corresponds to a waiting time before submitting a new order to the MDA.

presented in visually Figure 4.3 and textually in Listing 6.1, it is defined what interactions must be identified in a set of traces for the traces to be labeled as conforming, along with the maximum duration that the sequence can take. The first order of Experiment 0, with *orderId* set as `CORRECT` is configured to take 20 seconds and should be labeled as conforming by the checker, while the second order named `TIME_VIOLATION` is configured to take 30 seconds, violating the timing constraint present on the sequence diagram of 25 seconds, and thus should be labeled as nonconforming. Finally, the final order with *orderId* value `UNFINISHED` is configured to take 60 seconds. However, we will interrupt the execution before the full extent of the order can be executed by the MDA, meaning that the checker will not see the necessary traces to label the sequence as conforming. Thus, we expect a nonconforming result due to missing interactions. In the following listings, we present the output of the offline checker for this experiment. For all listings below, the timestamp field is replaced by `t` to avoid long lines. Moreover,`t`s represent distinct timestamps in nanoseconds, and the metrics in the last line of each listing are measured in seconds.

Starting with the 20-second order of Experiment 0, we can observe in Listing 6.5 the interactions that the checker chose to verify the conformance against the supplied sequence diagram. As expected, the checker gives a conforming result, since it was able to capture all the interactions present in the sequence diagram, and the *Sequence Time* was lower than the 25 seconds allowed by the sequence diagram (represented by *Max Time* in the listing). We look at the *Check Time* and *Reaction Time* metrics later in this chapter.

```
Result: CONFORMING
["PlannerService", "DeliveryControlService", "
   MonitorNotification", t, "CORRECT"]
["MealDispatchingService", "MealPreparingService", "
   MealPreparationRequest", t, "CORRECT"]
["MealDispatchingService", "PlannerService", "
   ScheduleUpdateNotification", t, "CORRECT"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationResponse", t, "CORRECT"]
["MealPreparingService", "MealDeliveringService", "
   DispatchDeliveryRequest", t, "CORRECT"]
["MealDeliveringService", "MealPreparingService", "
   DispatchDeliveryResponse", t, "CORRECT"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationUpdateNotification", t, "CORRECT"]
["MealPreparingService", "PlannerService", "
   MealPreparationUpdateNotification", t, "CORRECT"]
["MealDeliveringService", "PlannerService", "
   DeliveryUpdateNotification", t, "CORRECT"]
Check Time: 0.000106854 Sequence Time: 20.09
   Max Time: 25.00 Reaction Time: 82
```

Listing 6.5: Output of the offline checker for the 20-second order of Experiment 0.

Moving on to the 30-second order submitted on Experiment 0, we observe in Listing 6.6 that the chosen interactions are labeled as `NONCONFORMING_TIME`, due to the *Sequence Time* of the interaction being greater than the allowed 25 seconds.

```
Result: NONCONFORMING_TIME
["PlannerService", "DeliveryControlService", "
   MonitorNotification", t, "TIME_VIOLATION"]
["MealDispatchingService", "MealPreparingService", "
   MealPreparationRequest", t, "TIME_VIOLATION"]
["MealDispatchingService", "PlannerService", "
   ScheduleUpdateNotification", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationResponse", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDeliveringService", "
   DispatchDeliveryRequest", t, "TIME_VIOLATION"]
["MealDeliveringService", "MealPreparingService", "
   DispatchDeliveryResponse", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationUpdateNotification", , "TIME_VIOLATION
   "]
["MealPreparingService", "PlannerService", "
   MealPreparationUpdateNotification", t, "
   TIME_VIOLATION"]
```

```
["MealDeliveringService", "PlannerService", "
   DeliveryUpdateNotification", t, "TIME_VIOLATION"]
Check Time: 0.000151412 Sequence Time: 30.06
   Max Time: 25.00 Reaction Time: 69
```

Listing 6.6: Output of the offline checker for the 30-second order of Experiment 0.

Finally, we look into the 60-second order, the last order submitted in Experiment 0. We purposefully interrupt the execution of this order to show that the offline checker can handle cases where traces, and in this case interactions, are missing from the telemetry data captured during the MDA's execution. As expected, the checker outputs a nonconforming result due to some spans being missing. This can be seen in Listing 6.7.

```
Result: NONCONFORMING_MISSING
2024-02-14 10:14:31
["PlannerService", "DeliveryControlService", "
   MonitorNotification", t, "UNFINISHED"]
Check Time: 0.000148652 Sequence Time: 0.0 Max Time: 25
   Reaction Time: 57
```

Listing 6.7: Output of the offline checker for the 60-second order of Experiment 0.

As a final remark on the checker's correctness, we note that the implementation also outputs a correct result if the interactions were to be out-of-order. This would happen if the application executes a certain meal order in a different sequence than the one specified in the target sequence diagram. This behavior is detected in Listing 6.4 in Lines 47 to 50. However, we were not able to experimentally demonstrate this, as it would require changing the MDA application to execute operations in a different order. In this case, the checker would output a `NONCONFORMING_OUT_OF_ORDER` result.

Let us now consider Experiments 1 to 8. We will analyze the results and look at how the metrics change with the higher loads of the experiments. This will allow us to verify if the checker meets R2, R3, and R4. All experiments include 20-second orders, varying only the number of orders submitted in each experiment. Moreover, we expect that all orders will be labeled by the checker as `CONFORMING`. In Figure 6.2, we can see the total checks realized in each experiment, along with the number of checks labeled as `CONFORMING` and `NONCONFORMING`. The checker outputs the expected result for all submitted orders in each experiment.

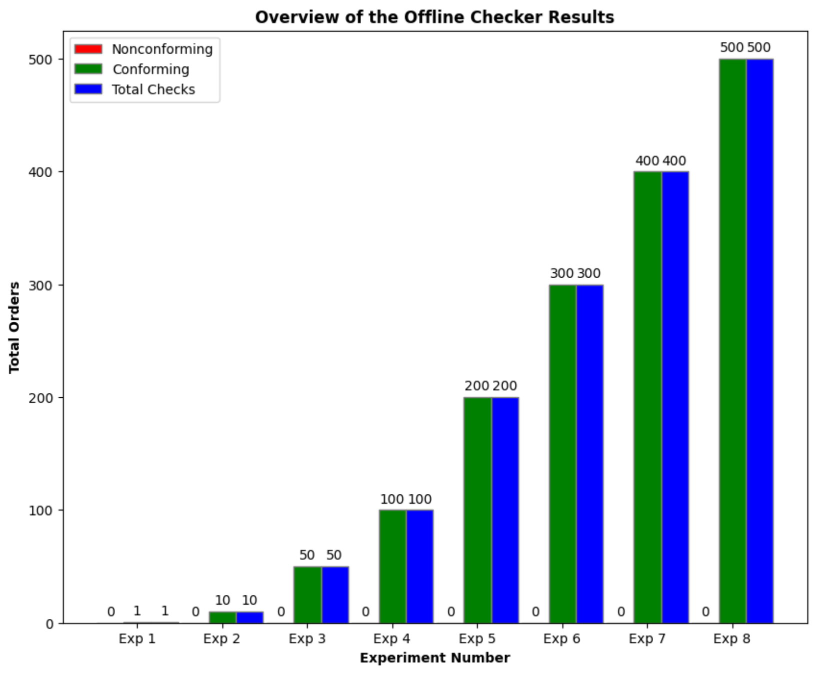

Figure 6.2: Overview of Experiments 1 to 8 and their checking results.

R2: Efficiency

To evaluate the efficiency of the offline checker we look at the CPU and memory (RAM) consumption throughout the experiments, and the weight of the needed infrastructure for Experiment 8, since it is the experiment that gives a higher load to the system. Starting with the CPU and memory consumption for all infrastructure in Experiment 8, we can observe in Figure 6.3 it is very low through most of the experimental run, almost always below 0.5%, except at the end where the offline checker peaks around 3% CPU usage. We will examine this behavior later in this chapter. On the other hand, in Figure 6.4 we can see that RAM consumption presents a different pattern. While the offline checker's RAM remains fairly low across the experiment, Kafka and the OpenTelemetry Collector have a rising RAM consumption throughout the experiment and with higher values than the checker. This can be attributed to the fact that the Collector and Kafka handle and store dense JSON files, while the checker handles slimmer versions of these JSON files as Python objects. Thus, it is expected that with larger data volumes, the RAM consumption will keep increasing. This can be mitigated in Kafka by setting smaller data expiration dates instead of the default 7 days. This expiration date has to be sufficient for the checker to consume the execution traces from Kafka. Regarding the Collector, it is harder to optimize since it handles telemetry data emitted by the target application.

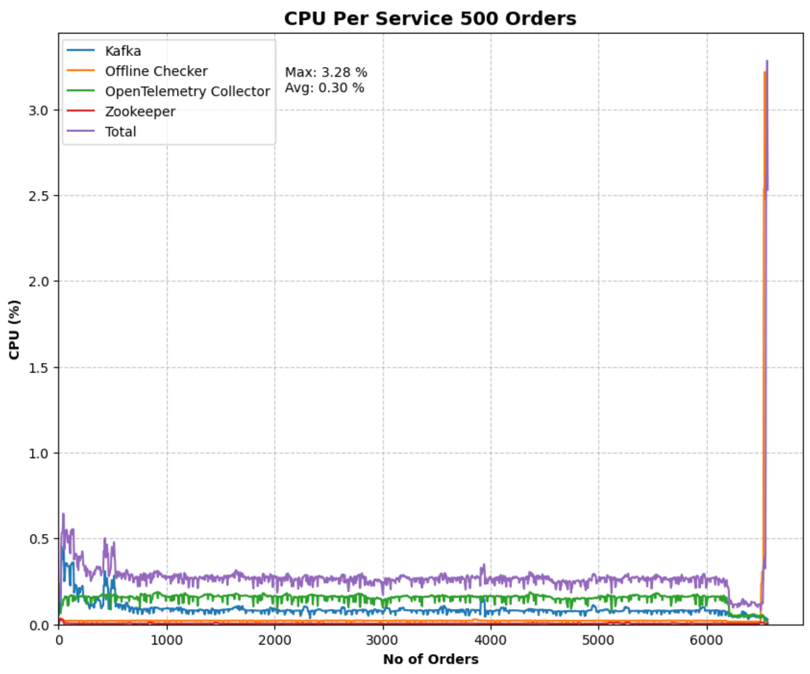

Figure 6.3: CPU consumption for all infrastructure necessary for the checker in Experiment 8.

Considering CPU and memory consumption for the checker only, we can observe how these metrics vary with higher loads in the different experiments in Figure 6.5 and Figure 6.6, by analyzing the peaks for each metric. Starting with RAM consumption, we can see that the peak RAM consumption remains more or less constant below 20MB in each experiment, highlighting that the strategy chosen to process the incoming events from Kafka applies to our use case. On the other hand, the maximum CPU consumption shows a growing trend with a higher volume of orders submitted to the MDA. This is due to the strategy implemented to do the checking in the offline paradigm, where the checker only consumes events throughout an experiment and only in the end processes everything. This behavior is consistent with the one shown in Figure 6.3 for Experiment 8. At the end of the experiment, we see a spike in CPU consumption that corresponds to when the actual checking takes place.

Nonetheless, regarding the efficiency of consuming resources, the checker presents a low resource consumption overall, for the resources available for these experimental runs.

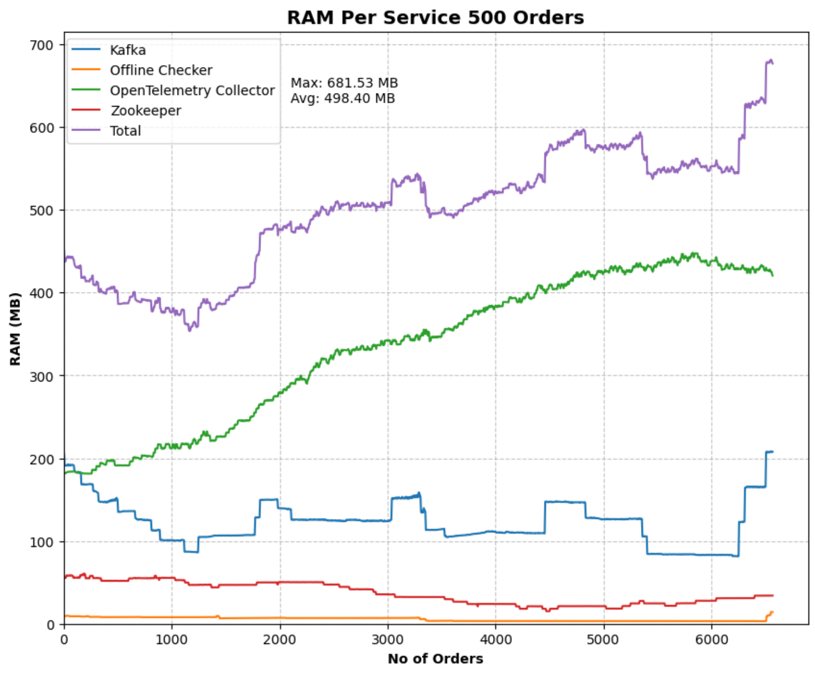

Figure 6.4: RAM consumption for all infrastructure necessary for the checker in Experiment 8.

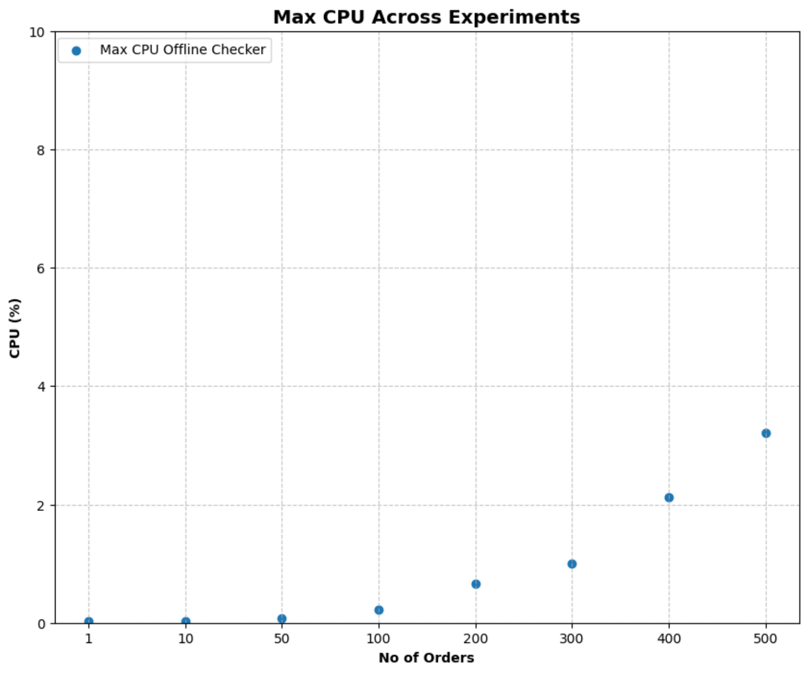

Figure 6.5: Maximum CPU consumption for the offline checker through Experiments 1 to 8.

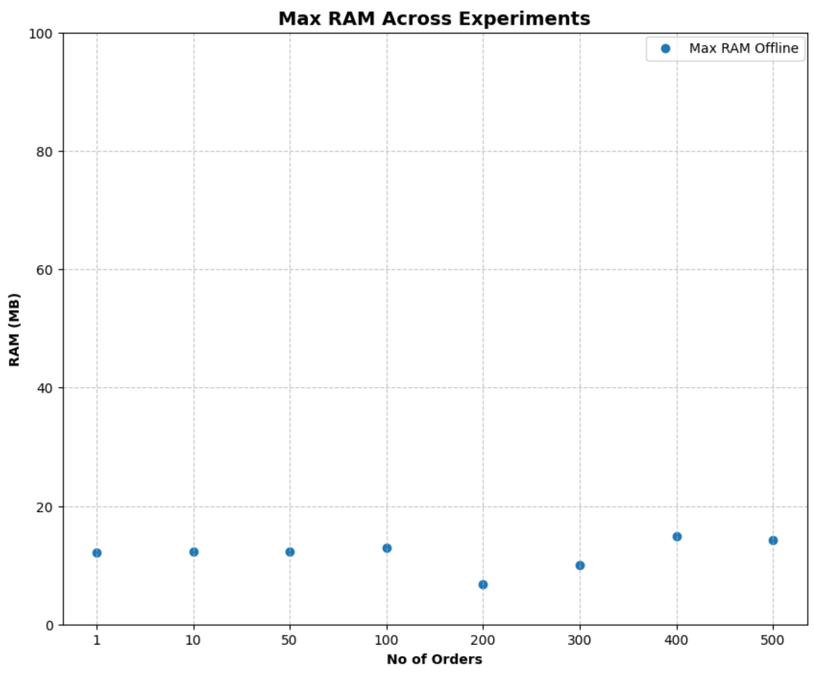

Figure 6.6: Maximum RAM consumption for the offline checker in each experiment through Experiments 1 to 8.

R3: Fastness

We now turn our attention to two other metrics that tell us how fast the checker can process traces and output conformance results. The first one we look at is the check time. This metric measures the time elapsed since the checker first started checking a certain order until it's finished. In Figure 6.7, we can see the average check time for all orders in each experiment. No matter the size of the experiment, the average time remains constant and close to 0 seconds. This means that once the checker has all the data it needs to produce a conformance result, it can do so in a very short amount of time. In contrast, we can observe the average reaction times for orders in each experiment in Figure 6.8. This metric represents the time elapsed between the timestamp in the first interaction and the time that the checker outputs a conforming or nonconforming result. There is a consistent increase in the average reaction time with experiments that are composed of more orders. Our current implementation only starts processing the execution traces when the MDA application is stopped. This means that the checker only outputs a conformance result first order submitted to the MDA after it possesses all execution traces for all orders. Thus, the time to get a result will be equal to the time that the MDA was running. For instance, Experiment 8, which consists of 500 orders, has an average reaction time

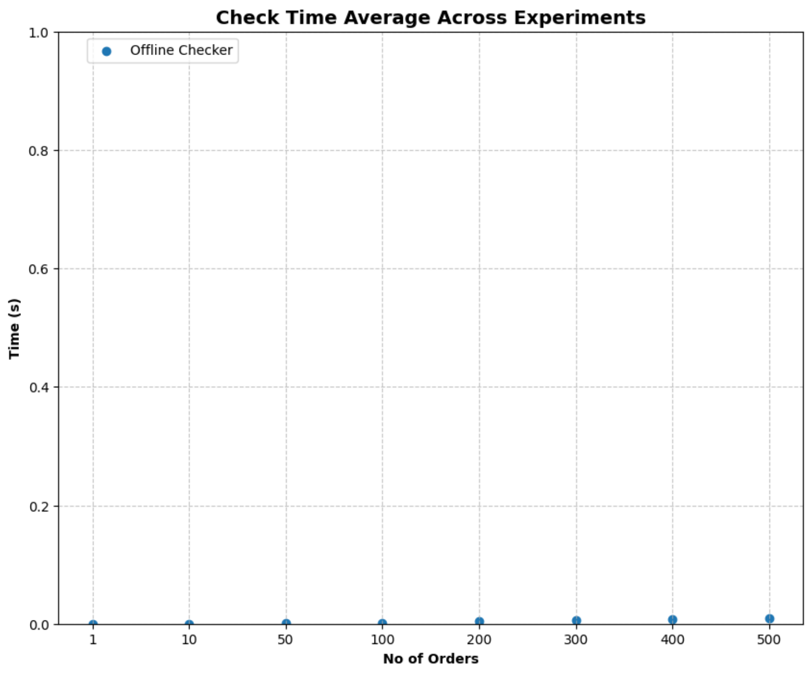

Figure 6.7: Average check times for the offline checker through Experiments 1 to 8.

close to 3500 seconds and we can see in Figure 6.3 that the experiment ran for close to 6500 seconds, which is close to double the average reaction time for the experiment. Hence, our current implementation does not meet this requirement, since we want to be able to act fast against nonconforming traces.

R4: Scalability

As previously observed, while resource consumption remains consistently low across all experimental runs, a concerning pattern emerges in CPU usage as the volume of orders processed increases. This indicates a scalability issue with our offline implementation, as peak CPU usage continues to rise with the growing number of orders. Additionally, we have noted a detrimental impact on reaction times with larger experiments, evidenced by the increasing average time to react as experiment size expands. This further underscores the inadequate scalability of our implementation, as it will increasingly delay the fulfillment of conformance results for submitted orders.

R5: Extensibility

For our final requirement, we designated that the checker should be easily extensible to allow for the use of multiple sequence diagrams. Although we

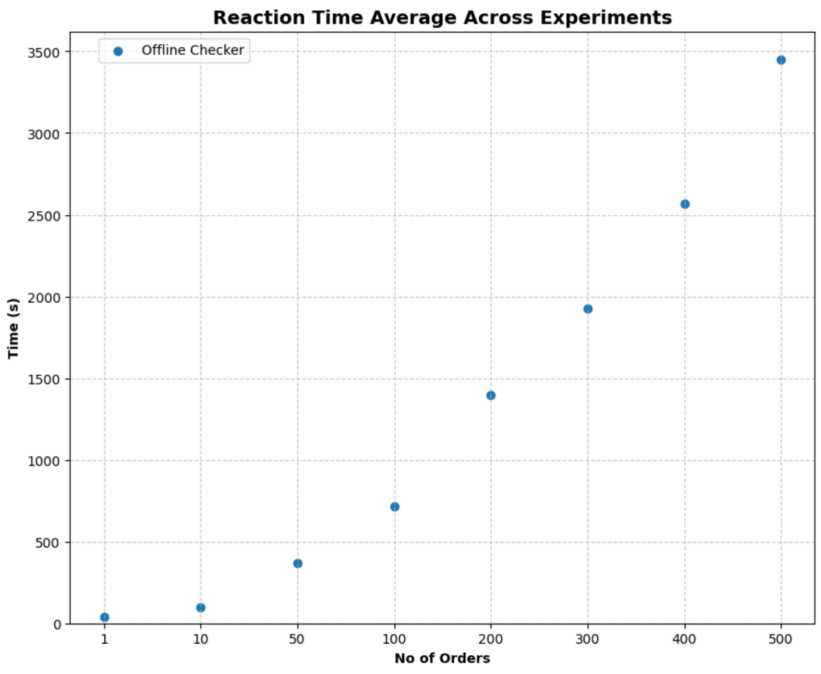

Figure 6.8: Average reaction times for the offline checker through Experiments 1 to 8.

do not present an experiment in this section that directly validates that this requirement is met, we note that the checker is agnostic with regard to the sequence diagram that is used. This means that the checker can be deployed with any sequence diagram in the PlantUML specification language, and following the initial constraints in Chapter 2, since we do not support the full syntax of the diagrams. Moreover, to check multiple sequence diagrams at the same time, it is only necessary to deploy the checker application with a different sequence diagram, as they work independently and get the same traces from Kafka.

Chapter 7

Online Conformance Checking

7.1 Motivation

In the previous chapter, we presented our first attempt at building a conformance checker for our microservice application. We used an offline processing paradigm to analyze execution traces in batches after a certain amount of time had elapsed.

This approach already fulfills most of our requirements. The checker correctly captures, processes, and matches execution traces to a provided sequence diagram, producing correct conformance results (R1). Moreover, the design decision regarding the placement of the checker in the overall architecture, with the addition of using Kafka ensures that its execution is decoupled from the MDA, thus not impacting the normal flow of the main application, while also consuming minor hardware resources (R2). Furthermore, we can argue that the checker is also extensible (R5) since checking different sequence diagrams is achieved by deploying another checker instance with a different sequence diagram as input. However, as predicted, the offline approach falls short of the other two requirements. The offline version of the checker is not fast, since the reaction time to each interaction will be, in the worst case, as slow as the time elapsed between the actual execution of the interaction and the time that we actually process the trace, that is equal to the time that the MDA is running and executing workflows (R3). Moreover, it is hard to make the case that our checker is scalable (R4) since it is currently developed as a single unit of work that needs the full execution context to be able to operate correctly.

7.2 From Offline to Online Conformance Checking

To address requirements R3 and R4 we switch our approach to online (real-time) processing of execution traces. Instead of gathering all execution traces and only doing the necessary processing for conformance checking afterward, we aim to process a trace immediately after it is available in Kafka. This would decouple the reaction time from the MDA's execution time, allowing for faster conformance results. Moreover, as this change in paradigm will need several adaptions to how we process and deliver conformance results, it is a chance to modularize our checker and more easily identify where the checker can be scaled to handle bigger loads.

More concretely, we adopt an online asynchronous approach [17] to develop the online version of our checker, in favor of the online synchronous approach, to maintain the MDA and the checker application decoupled and independent from one another, and minimize the impact the checker has on the MDA's regular execution (R2). This also directly influences our decision regarding the placement of the online checker in our architecture: we deploy the checker similarly to the offline checker, instead of following an approach like *ucheck* [46], which is deployed close to each microservice and intercepts the communications between them.

Additionally, our algorithm also needs to be adapted. In the online processing paradigm, we don't have access to the full context nor do we know when that context will be available. Thus, we need to find a strategy that allows the checker to follow an online approach for faster reaction times, without sacrificing its correctness. Drawing inspiration from Burattin's streaming conformance checking approaches [11], we use a hybrid approach between window-based and problem-reduction approaches. We start by applying a window of a single trace followed by reducing our problem from matching an entire set of traces to a sequence diagram to checking if the interactions derived from a single trace are part of the sequence diagram. Moreover, our approach correctly handles interactions that come out of order. In this approach, we capture relevant events in any order that they are received, and only after having a complete set of interactions that can be compared to the sequence diagram, do we initiate the conformance checking by sorting those interactions based on their execution time.

This strategy creates the necessity of keeping the state of the orders that are being checked since now multiple orders can be checked simultaneously, in contrast to our offline approach where we searched for all the relevant interactions for each *orderId* present, one order at a time. Thus, we implement a mechanism similar to the one proposed by Broucke et al. [10]: instead of using worker threads to concurrently check subnets, we create a mapping between an *orderId* and a checker instance to correctly route each interaction to the right checker. Finally, we have to account for the problem of not knowing if a given interaction will arrive or if it was actually executed by the MDA. We apply a similar approach to the one followed by Berti and Aalst [8], by assigning to each checker instance in use a 10-second keep-alive timeout. When the timeout expires, the checker outputs a nonconforming result, considering that there are missing interactions to validate against the sequence diagram. Moreover, this prevents checker instances from accumulating, due to never being terminated and thus overly consuming memory resources.

7.3 The Online Checker

Building on the previous sections in this chapter, we present the online version of the conformance checker introduced in Chapter 6. As before, we modify the overall architecture, followed by changing the algorithm to take into account the new checker paradigm.

7.3.1 Architecture

We are now able to fully leverage the Kafka service first deployed in the offline version of the checker. The checker polls Kafka and consumes each event present in the queue immediately, instead of buffering its full content as before. Moreover, in Figure 7.1, we can see that we maintained the offline checker deployed next to the online version. This is consistent with the approach described in Chapter 4 to allow for an accurate evaluation of the online implementation against the offline checker, due to the challenges of replicating and comparing different experimental runs. Moreover, we can delegate to Kafka the work of keeping track of which events have been consumed for each checker instance. This is done by using the consumer group's configuration in Kafka. Each checker instance is assigned a different consumer group ID, and Kafka manages which event should be consumed by each checker autonomously.

7.3.2 Checker Algorithm

The most significant changes lie in the algorithm used to perform the actual checking. Instead of collecting and buffering every execution trace available in Kafka in one go, we introduce the `poll` function that returns a single event present in the queue. That event is then processed as before resulting in a list of interactions. Each interaction is then passed to the right checker instance that possesses the same *orderId* as the interaction being used. Moreover, we keep track of the checker instances through a map indexed by *orderId* and whose value is the actual checker instance. Each checker is a class that contains an attribute `startTime`, a timestamp equivalent to the time

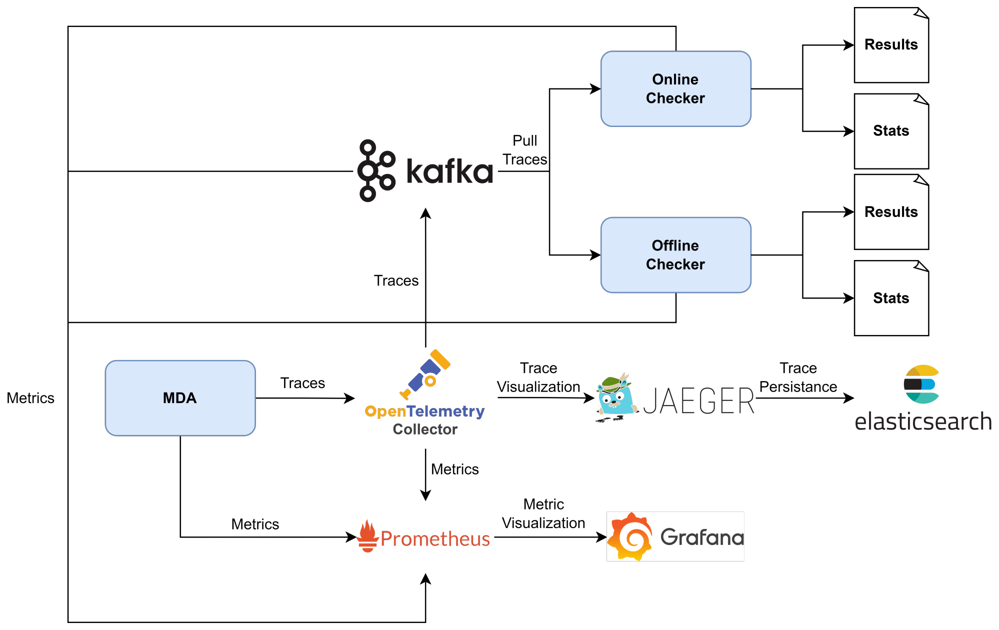

Figure 7.1: Architecture for MDA and Online Checker.

that the checker instance was created, and methods `run` and `verify`. The `run` method updates the checker state upon the arrival of an interaction, while the `verify` method is executed once a set of interactions satisfies a complete sequence diagram, or the checker is given a termination signal. If the checker instance still exists after a 10-second window, we output the conformance results up to that point and eliminate the instance. Finally, although not visible in the algorithm presented here, we separated the code that captures the checker execution stats into its separate class to allow it to be shared between instances and make sure that the stats are coherent. The new algorithm is described in Listing 7.1.

```
sd <- ordered list of processed sequence diagram
    interactions
t <- timing constraint in sd

kafka <- queue of execution traces

activeCheckersMap <- {}

while kafka != []
     trace <- poll(kafka)
     traceObj <- readTrace(trace)

     interactionList <- []
     findInteractions(trace.rootSpan, interactionList)
    interactionList.sort(key <- interaction.timestamp)

    for interaction ∈ interactionList
        if interaction.orderId ∉ activeCheckersMap
            activeCheckersMap[interaction.orderId] = new
                Checker(sd, getCurrentTime(), t)
        endif
        isCheckFinished <- activeCheckersMap[interaction.
           orderId].run(interaction)
        timeElapsed <- getCurrentTime() -
           activeCheckersMap[interaction.orderId].
           startTime
        if isCheckFinished ∨ timeElapsed > 10
            activeCheckersMap[interaction.orderId].verify
                ()
            delete activeCheckersMap[interaction.orderId]
        endif
    endfor
endwhile

class Checker(sd, timestamp, t)
    sdCopy <- sd
    startTime <- timestamp
    maxDuration <- t
    selectedInteractionsSet <- {}

    function run(interaction)
        if interaction ∈ sdCopy ∧ interaction ∉
           selectedInteractionsSet
            selectedInteractionsSet.add(interaction)
            sdCopy.remove(interaction)
        endif
        if sdCopy = []
            return TRUE
        endif

        return FALSE
    endfunction

    function verify()
        selectedInteractionsSet.sort(key <- interaction.
           timestamp)
        duration <- selectedInteractionsSet[-1].end -
           selectedInteractionsSet[0].start
        if sdCopy != []
            writeResults("NONCONFORMING_MISSING",
                selectedInteractionsSet, duration)
        endif
        for I ∈ [0..len(selectedInteractionsSet)]
```

```
54              if selectedInteractionsSet[i] != sd[i]
                    writeResults("NONCONFORMING_OUT_OF_ORDER
                        ", selectedInteractionsSet, duration)
56              endif
            if duration > maxDuration
58              writeResults("NONCONFORMING_TIME",
                    selectedInteractionsSet, duration)
            endif
60          writeResults("CONFORMING",
                selectedInteractionsSet, duration)
        endfunction
62  endclass
```

Listing 7.1: Algorithm for the online conformance checker.

There are some fundamental changes to the algorithm presented in Listing 6.4. Instead of reasoning about the algorithm as being divided into three different parts we redesigned and modularized the previous logic into three components: initialization, main loop, and checker class.

Checker Class. Starting with the latter component, we encompassed the checking logic onto a class that can be instantiated on the fly. As mentioned before, we need to keep track of multiple orders at a time. Thus, in Lines 29 to 33, we initialize a checker instance object that receives a sequence diagram, a timestamp, and t as input. The timestamp denotes the time that the checker object was created and helps determine if the checker has been running for longer than 10 seconds, while the rest of the parameters are similar to the ones we had before. Between Line 35 and Line 45, we create the `run` function that contains the logic behind deciding whether to accept a given interaction or not. Finally, the verification logic in the offline algorithm is contained in the `verify` function between Line 47 and Line 61.

Initialization. Going back to the beginning of the listing, we do a similar initialization of the variables seen before. We load the processed sequence diagram and its timing constraint in Lines 1 and 2, followed by the connection to the Kafka queue in Line 6. In this algorithm, we introduce the `activeCheckersMap` explained at the start of this section to keep track of the orders that are being checked in Line 6.

Main Loop. The change of paradigm means that instead of gathering all the execution traces, we process traces one by one, as soon as they are available in Kafka. Thus, we poll Kafka until no more traces are left in the queue. Kafka will stop receiving new traces as soon as the MDA's execution is stopped. This condition is evaluated in Line 8. From here, we apply a similar logic to the one seen in the previous chapter, with a few new conditions. We start by consuming a trace from Kafka in Line 9 and creating a trace object in Line 10. After, we execute the `findInteractions` procedure as before, between Line 12 and Line 14. For each resulting interaction in *interactionList*, we go to our map to see if a key already has the same value as the *orderId* field in the interaction. If this is a new order that isn't being checked, we create a new Checker object and add it to the map in Lines 17 to 19. We advance by taking the right checker on the map and executing the `run` method and storing its return value in the *isCheckFinished* variable in Line 20. Additionally, we calculate the time that has elapsed since the checker for that order was instantiated in Line 21. The loop logic ends with the decision if the checker for that order should be terminated. We evaluate if the checking is concluded or if 10 seconds have passed since the checker was started. If any of these conditions are true, we execute the `verify` method of the checker that outputs the final result in Line 23 and delete the checker from the map in Line 24.

7.3.3 Evaluation

Just as we assessed the offline checker in Chapter 6, we now follow a similar approach to evaluate the implementation of our online checker. Here, we revisit the requirements outlined in Section 2.3, examining whether the online checker satisfies them. Additionally, we conduct a direct comparison between this new online version and the offline checker introduced in the previous chapter by examining the biggest experimental run and the overall evolution of metrics for both checkers across experiments.

Experimental Design

To be able to distinguish and decide what implementation is better, the focus of this evaluation section is to compare both the offline and the online checker against each other, to determine if the online implementation outperforms the offline one, without sacrificing correctness or efficiency. Thus, we will conduct the same set of experiments as in the previous chapter. However, instead of having only the offline checker deployed in our infrastructure, we also have the online checker. Both checkers will consume the same data from Kafka and operate independently from each other. The experiments mentioned in the following paragraphs are described in Table 6.1.

R1: Correctness

To demonstrate the correctness of our online implementation we show the output of the online checker in Listing 7.2, Listing 7.3, and Listing 7.4 for Experiment 0. The online checker gives correct results to all three orders and is consistent with what we saw on the offline checker's evaluation. Moreover, in Figure 7.2 we observe that the online checker also evaluates correctly all orders for Experiments 1 to 8 and is on par with the offline checker's behavior. For all listings below, the timestamp field is replaced by `t` to avoid long lines. Moreover,`t`s represent distinct timestamps in nanoseconds, and the metrics in the last line of each listing are measured in seconds.

```
Result: CONFORMING
["PlannerService", "DeliveryControlService", "
   MonitorNotification", t, "CORRECT"]
["MealDispatchingService", "MealPreparingService", "
   MealPreparationRequest", t, "CORRECT"]
["MealDispatchingService", "PlannerService", "
   ScheduleUpdateNotification", t, "CORRECT"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationResponse", t, "CORRECT"]
["MealPreparingService", "MealDeliveringService", "
   DispatchDeliveryRequest", t, "CORRECT"]
["MealDeliveringService", "MealPreparingService", "
   DispatchDeliveryResponse", t, "CORRECT"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationUpdateNotification", t, "CORRECT"]
["MealPreparingService", "PlannerService", "
   MealPreparationUpdateNotification", t, "CORRECT"]
["MealDeliveringService", "PlannerService", "
   DeliveryUpdateNotification", t, "CORRECT"]
Check Time: 20.092 Sequence Time: 20 Max Time: 25.00
   Reaction Time: 30
```

Listing 7.2: Output of the online checker for the 20-second order of Experiment 0.

```
Result: NONCONFORMING_TIME
["PlannerService", "DeliveryControlService", "
   MonitorNotification", t, "TIME_VIOLATION"]
["MealDispatchingService", "MealPreparingService", "
   MealPreparationRequest", t, "TIME_VIOLATION"]
["MealDispatchingService", "PlannerService", "
   ScheduleUpdateNotification", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationResponse", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDeliveringService", "
   DispatchDeliveryRequest", t, "TIME_VIOLATION"]
["MealDeliveringService", "MealPreparingService", "
   DispatchDeliveryResponse", t, "TIME_VIOLATION"]
["MealPreparingService", "MealDispatchingService", "
   MealPreparationUpdateNotification", , "TIME_VIOLATION
   "]
["MealPreparingService", "PlannerService", "
   MealPreparationUpdateNotification", t, "
```

```
    TIME_VIOLATION"]
["MealDeliveringService", "PlannerService", "
    DeliveryUpdateNotification", t, "TIME_VIOLATION"]
Check Time: 30.056 Sequence Time: 30.06 Max Time: 25.00
    Reaction Time: 40
```

Listing 7.3: Output of the online checker for the 30-second order of Experiment 0.

```
Result: NONCONFORMING_MISSING
2024-02-14 10:14:31
["PlannerService", "DeliveryControlService", "
    MonitorNotification", t, "UNFINISHED"]
Check Time: 47.347 Sequence Time: 0.0 Max Time: 25
    Reaction Time: 57
```

Listing 7.4: Output of the online checker for the 60-second order of Experiment 0.

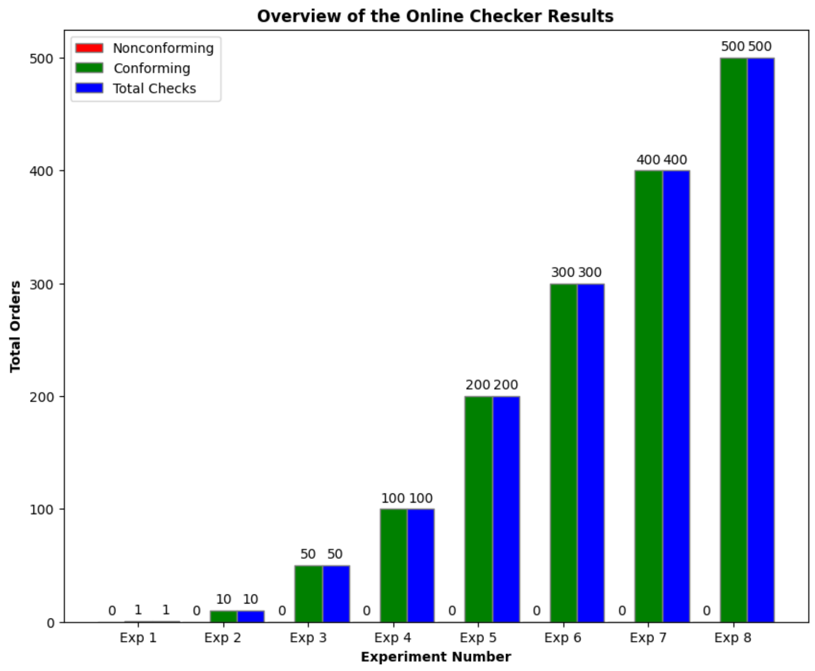

Figure 7.2: Overview of Experiments 1 to 8 and their checking results.

R2: Efficiency

Let us now revisit the CPU and RAM usage by the checker during the experiments. Starting with CPU usage for Experiment 8, we can observe in Figure 7.3 that adding the online checker to the overall infrastructure had a very low impact on CPU usage. Since the online checker is directly connected to Kafka, like the offline checker, there is no extra effort required to power the infrastructure, as CPU usage is similar to what we saw in the offline checker case. Moreover, we can see that the online implementation is more efficient with CPU consumption than the offline checker. The online checker does not have a peak at the end of the experiment, since the checking effort is done throughout the execution, in contrast with the offline approach that did it all in the end. Additionally, the RAM usage was also unaffected by the addition of the online checker. In Figure 7.4, we can see that, like the CPU, the infrastructure does not use more RAM than before. Moreover, the online checker consumes marginally more RAM than the offline version, maintaining the memory efficiency of the previous implementation.

This behavior is also consistent across experiments. In Figure 7.5 and Figure 7.6 we compare the maximum CPU and RAM usage of each checker throughout Experiments 1 to 8. The online checker's maximum CPU usage remains consistently low and close to 0 in every experiment while the offline checker's maximum CPU consumption grows with the number of orders. This shows that the online implementation made a step forward by reducing CPU usage while maintaining the correctness of the offline checker. Additionally, the maximum RAM usage for both checkers is close to 20MB. The online checker uses marginally more memory since it keeps more objects in memory given that it checks multiple orders at the same time and needs to maintain state for all of them, while the offline checker only keeps state for one order at a time when it starts doing the actual checking in final stages of an experiment.

R3: Fastness

One of the main drives to adopt an online processing approach was to achieve better reaction times to nonconforming traces than the offline approach gave us. In Figure 7.7 we can compare the average reaction times for both checkers in the different experiments. The online checker is a significant improvement in this metric. First, the reaction time is not dependent on the number of orders that are checked. Secondly, the online checker achieves a constant reaction time of 30 seconds for each order in all experiments. This is a consequence of the time that an order takes to be executed, which for Experiments 1 to 8 is 20 seconds, plus the 10 seconds that traces are retained by the OpenTelemetry Collector to aggregate all spans that make a particular trace before they are sent to Kafka and subsequently to the checker for processing. Looking again at Listing 7.2 and Listing 7.3 we can see that this is in fact the case. For the 20-second order, the reaction time is 30 seconds while for the 30-second order, the reaction time goes up 10 seconds, to a total of 40 seconds. Thus, we devise the following formula for

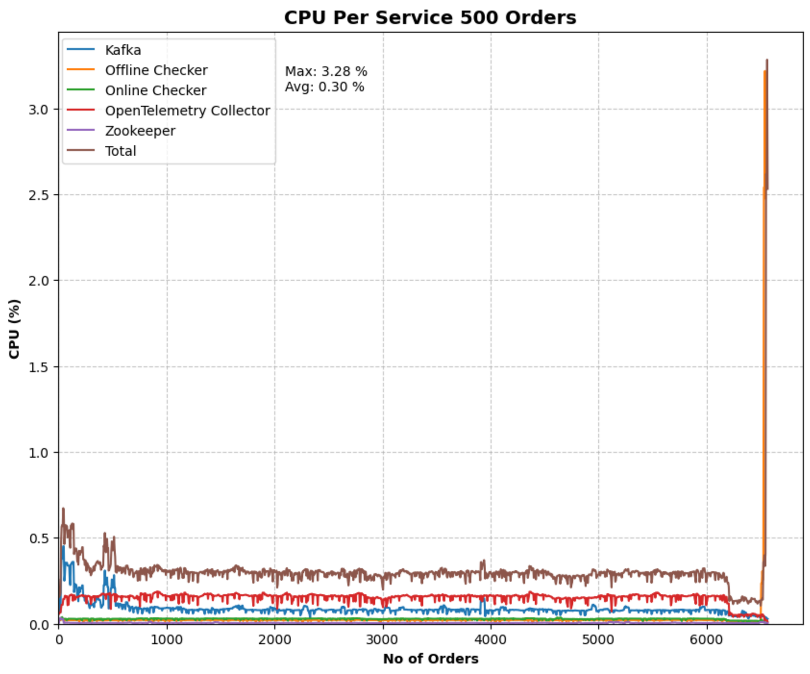

Figure 7.3: CPU consumption for all infrastructure necessary for the checker in Experiment 8.

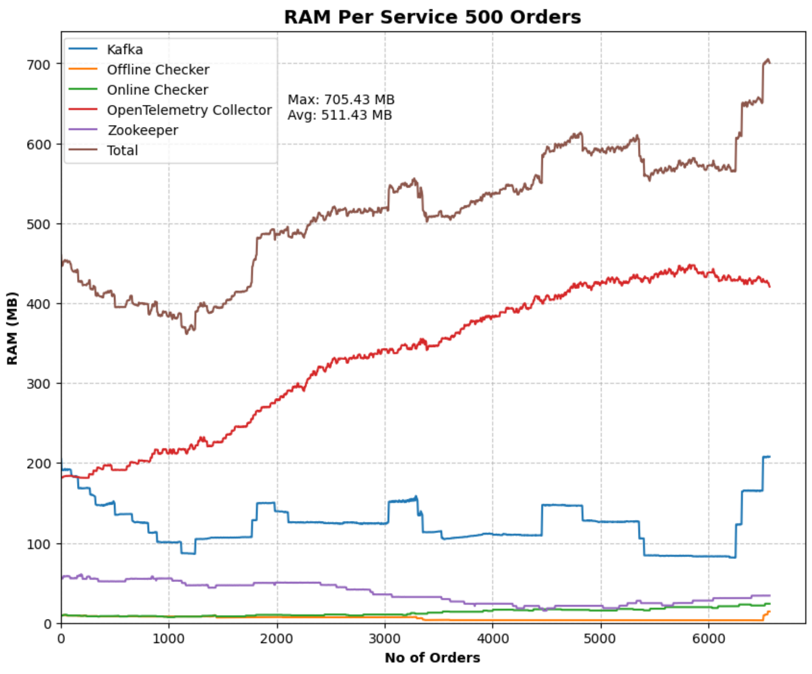

Figure 7.4: RAM consumption for all infrastructure necessary for the checker in Experiment 8.

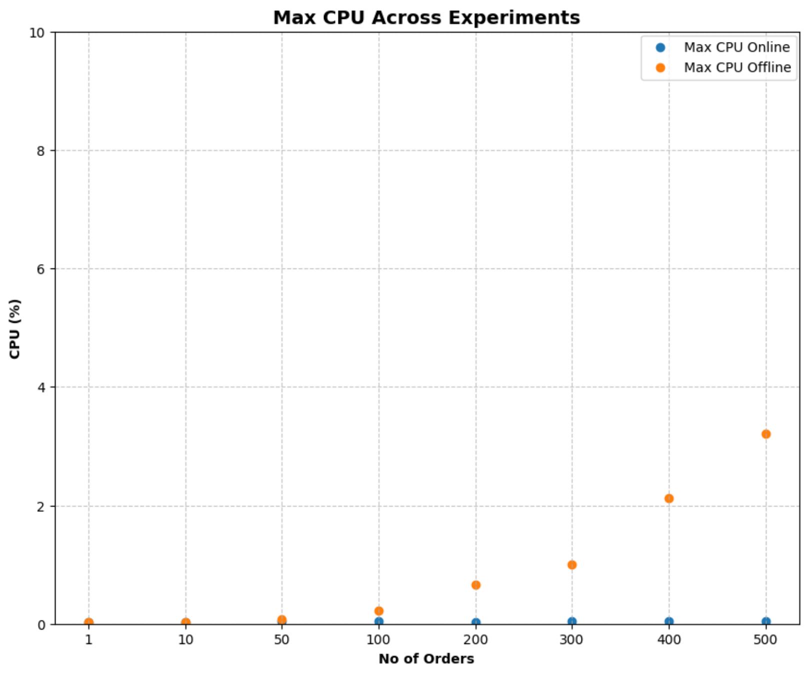

Figure 7.5: Maximum CPU consumption for the offline and online checkers through Experiments 1 to 8.

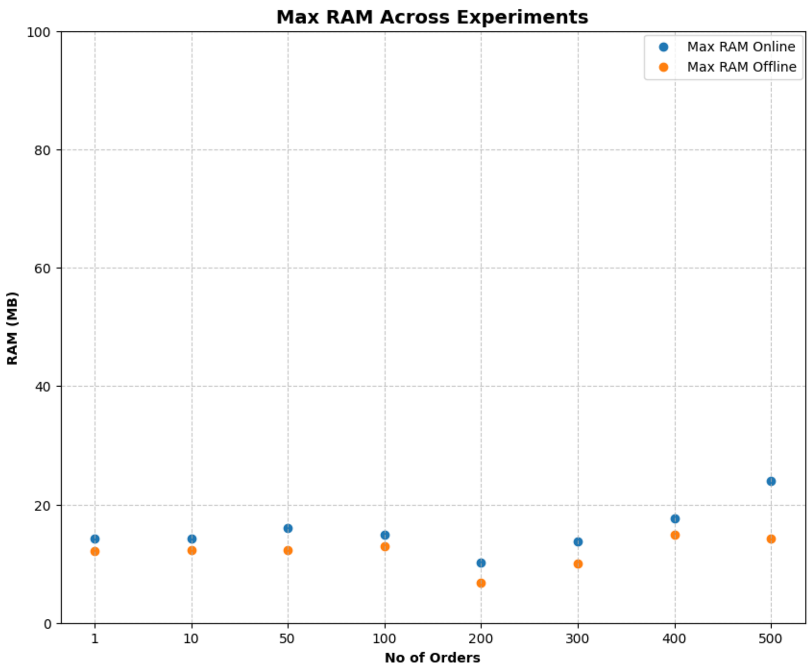

Figure 7.6: Maximum RAM consumption for the offline and online checkers in each experiment through Experiments 1 to 8.

the reaction time of our online implementation:

$$ReactionTime = SequenceTime + CollectorTime + Other$$

where *SequenceTime* is the time that takes a set of traces subject to checking takes to execute in the application (in our case, it corresponds to an order in the MDA), *CollectorTime* is the delay time before the OpenTelemetry Collector published execution traces to Kafka and *Other* corresponds to other factors that can influence the time to react like network delays, or in the case of missing interactions, the 10-second timeout defined earlier. This shows that the reaction time is independent of the checker implementation, only depending on the application itself and the predefined time to gather all the spans of a trace in the OpenTelemetry Collector.

We also observe some changes regarding the average check time for each order. The average check times for both checkers throughout Experiments 1 to 8 can be seen in Figure 7.8. At first glance, it may seem that the online checker is substantially slower than its counterpart. However, the consistent 20-second duration check time for each order can easily be explained by the way the online checker works. In contrast with the offline implementation, the online checker starts checking for an order the moment that the first trace for that order is received by the checker. Since the orders that consist of Experiments 1 to 8 are 20 seconds long, the time elapsed between receiving the first and last interactions for a particular order will naturally be the time that it takes to execute that order in the application, plus some network delays.

R4: Scalability

As we already mentioned, the online checker improves several aspects that the offline version lacked. This also makes the online implementation more scalable than its predecessor. For instance, as we saw CPU usage is no longer dependent on the number of orders, not peaking higher with higher volumes of data like the offline checker. Moreover, the reaction times are also not linked to the number of orders processed and are consistent for each order in each experiment. Thus we can consider that our online approach meets the scalability needs that we defined.

R5: Extensibility

Finally, we analyze if the online checker can also easily be extended to check multiple sequence diagrams at the same time. Like the offline checker, we do not have an experiment that can validate our claim. However, we again note that the online checker is sequence diagram agnostic, and different checker instances work independently from each other, not requiring any coordination between them. Thus, checking multiple sequence diagrams

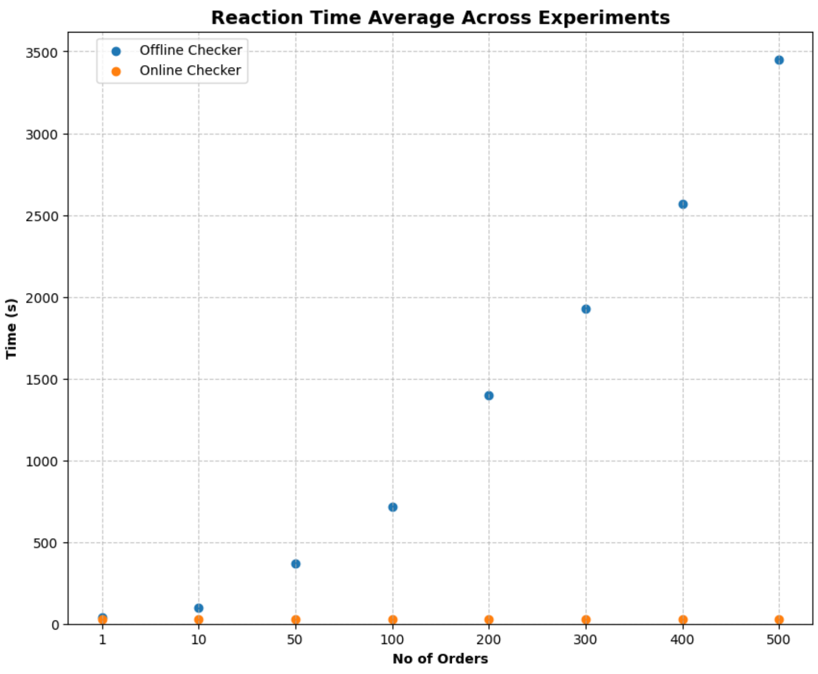

Figure 7.7: Average reaction times for the offline and online checkers through Experiments 1 to 8.

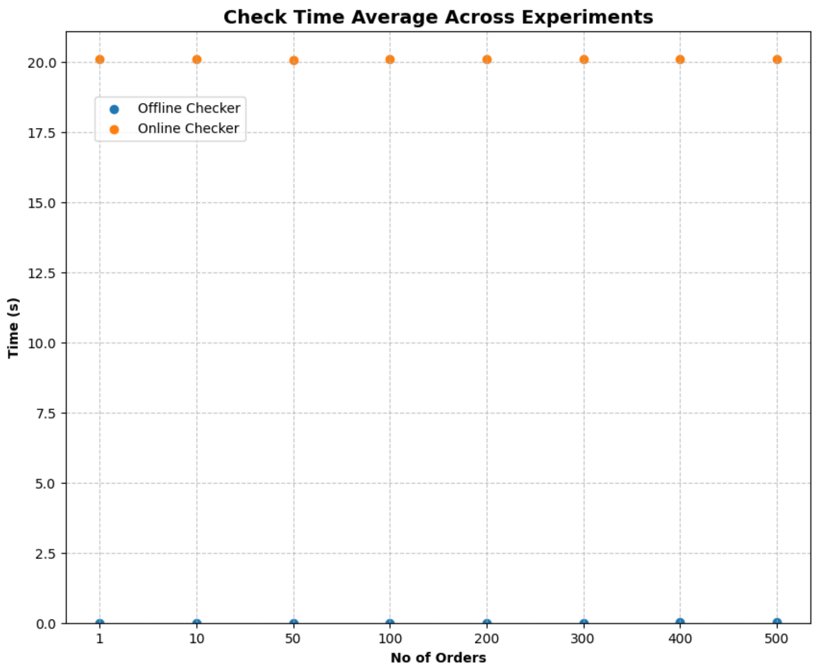

Figure 7.8: Average check times for the offline and online checkers through Experiments 1 to 8.

would only require deploying a new online checker instance with a different sequence diagram.

Chapter 8

Conclusion

In this master thesis, we set out to address the challenge of implementing fast, efficient, and correct conformance checking for microservice applications. Microservice architectures have gained popularity due to their scalability, flexibility, and maintainability benefits. However, ensuring the conformance of these distributed and independently deployable services to a predefined specification remained a challenge. Traditional conformance checking approaches were not suitable for the dynamic microservice environment and no tool had been used in the specific combination of our use case of sequence diagram and modern execution traces. Through in-depth research and experimentation, we developed a real-time conformance checker using open-source tools and provided insights and contributions to the field.

RQ *How can correct, efficient, fast, scalable, and extensible conformance checking be implemented for microservice applications?*

Our approach to the research question involved creating an offline conformance checker tailored for microservice applications. This tool effectively evaluated conformance based on MDA execution traces and sequence diagrams, operating independently of the MDA's execution for efficiency. Transitioning to an online paradigm improved performance significantly, achieving conformance results in approximately 30 seconds per trace, meeting our requirement for rapid reaction to nonconforming sequences. Ultimately, we developed a proof-of-concept online conformance checker suitable for microservice applications.

Contributions. With this thesis, we contribute with a novel approach to perform conformance checking using a comparison of transformations of sequence diagrams and execution traces. Moreover, we also contributed to the field with an architecture that follows good observability pipelines' principles and that can be adapted and used by other telemetry analysis applications or dataflows. Our approach is application-agnostic and ensures reaction times as fast as the execution of requests in the application itself, with minor delays in processing and checking for conformance. Moreover, the implemented online checker is a lightweight service with a minor impact on the overall infrastructure of a system.

8.1 Beyond Conformance Checking

The main focus of our work was conformance checking and we proposed an adequate architecture that makes it possible to evaluate conformance for microservice applications. However, we argue that this architecture can potentially serve more purposes than just conformance checking. Through a clever use of observability pipelines, we envision that our architecture in Figure 7.1 can be extended with different applications that perform other telemetry data analysis. For example, the work of processing execution traces can be separated from the checker and deployed as a separate service that publishes the transformed traces into a new Kafka topic. In turn, this new service can be used in several observability pipelines that adhere to the format posted by this service to a Kafka topic. For instance, a bottleneck analysis application can consume from the Kafka topic that contains the preprocessed traces and use them for this analysis, without needing to incorporate this preprocessing in the application itself. Moreover, these services would be able to be completely decoupled from one another thanks to Kafka which would serve as an intermediary in all communications.

8.2 Threats to Validity

In this master thesis, we developed and presented a proof-of-concept conformance checking application that delivers real-time results for the Meal Delivery microservice application. We achieved promising results and fulfilled our initial requirements concerning the design and implementation of our checker. However, it is important to note that there are some risks and threats to the validity of our study. We outline them in this section.

Limited evaluation. To evaluate the quality of our checker we leaned on tracing data produced by the MDA. While this allowed us to tailor our checker to this application and, in general, to microservice systems, the use of a single application is a limitation. More extensive testing would need to be conducted to accurately access our checker's behavior to similar systems. Moreover, the MDA is not a production or real-world system. Thus, during our experiments, we might not have been able to test important behavior that more complex real-world systems might exhibit compared to the MDA.

Request latency. Adding to the previous point, as we mentioned before, the MDA traces are mostly 2 seconds long. Moreover, the sequence diagram being checked is composed of interactions that ideally should be executed in 25 seconds or less. If another application has longer traces or longer sequences to be checked, we may need more memory resources to cope with the extra time that each interaction is kept in memory and that each checker instance is executing.

Stress testing. A more restricting aspect of using the MDA as the source of our telemetry data is that we are not able to stress test our checker during our experiments. The MDA is not capable of generating higher telemetry data loads which can be a hurdle to both focal points of our architecture: the OpenTelemetry Collector and Kafka. Kafka in particular can handle close to a million messages per second, making it very hard for the MDA to reach those levels of throughput. Thus, the checker is not impacted by possible losses of traces anywhere in the process.

Sequence diagrams. Another limitation of our checker is the support for the UML syntax of sequence diagrams. The checker currently does not support more complex statements such as loops or optional sequences. Checking these types of interactions could increase the complexity of the checker, as well as introduce new challenges to the checker's development.

Instrumentation. Finally, we assume that the MDA is instrumented in such a way that the emitted telemetry data correctly resembles the actual execution done by the application. If this would not be the case, the checker would be outputting wrong results since its initial data would be incorrect.

8.3 Future Work

During the development of the online conformance checker in this thesis, we identified possible improvements that can be implemented in the future. Throughout the thesis, we made some choices regarding the direction we wanted to head towards to implement the checker, cutting some possibilities along the way, since we did not have time to pursue all possible alternatives.

One immediate improvement that could be made is to support the full syntax of sequence diagrams. In Chapter 2, we excluded some syntax elements like loops and alternative sequences to simplify our implementation. However, supporting the complete syntax would allow our checker to be used with a wider range of diagrams and be suitable for more use cases.

Another possible direction to improve our checker is to revisit the related work on conformance checking. Although we chose to deviate from the known conformance checking techniques, due to the particularity of using sequence diagrams, we open the possibility of researching how more traditional methods can be implemented in our use case. This can be done by transforming the sequence diagrams to Petri nets and finding a way to replay the OpenTelemetry-generated traces in these Petri nets. Furthermore, with the Petri net as our model, it is possible to test the suitability of other techniques, particularly alignments, which are subject to the most research output for conformance checking.

Additionally, we could consider integrating concepts from other research fields to improve our checker's implementation. For instance, a more thorough research of real-time processing might reveal interesting techniques that improve the processing logic of our checker. Additionally, research on complex event processing has some strong similarities to the processing that we do on the execution traces side. Together with Apache Flink [4], the aggregation and filtering of execution traces can be optimized and parallelized. Moreover, Flink would help expand our architecture to different telemetry analysis pipelines since it would decouple most of the processing from the logic of the checker and other tools that might be present in the architecture.

Finally, reporting results and metrics could be enhanced. Currently, the checker writes its outputs to locally stored files. However, we can imagine building a real-time dashboard that keeps track of the metrics accompanied by a repository that flags nonconforming execution traces and produces a PlantUML specification highlighting the cause of a nonconforming result. The generated metrics could also be reformatted and scrapped by Prometheus, taking advantage of existing visualization tools like Grafana.

Bibliography

[1] Han van der Aa, Henrik Leopold and Hajo A. Reijers. ‘Efficient Process Conformance Checking on the Basis of Uncertain Event-to-Activity Mappings’. In: *IEEE Transactions on Knowledge and Data Engineering* 32.5 (2020), pp. 927–940. DOI: 10.1109/TKDE.2019.2897557.

[2] Amr S. Abdelfattah and Tomas Cerny. ‘Roadmap to Reasoning in Microservice Systems: A Rapid Review’. In: *Applied Sciences* 13.3 (2023). DOI: 10.3390/app13031838.

[3] Aysh Alhroob, Keshav Dahal and Alamgir Hossain. ‘Transforming UML sequence diagram to High Level Petri Net’. In: *2010 2nd International Conference on Software Technology and Engineering*. Vol. 1. 2010, pp. V1-260-V1–264. DOI: 10.1109/ICSTE.2010.5608842.

[4] *Apache Flink*. URL: https://flink.apache.org/.

[5] *Apache Kafka*. URL: https://kafka.apache.org/.

[6] Ouafa Bentaleb et al. ‘Containerization technologies: Taxonomies, applications and challenges’. In: *The Journal of Supercomputing* 78.1 (2022), pp. 1144–1181.

[7] Andre Bento et al. ‘Automated Analysis of Distributed Tracing: Challenges and Research Directions’. In: *Journal of Grid Computing* 19.1 (2021). DOI: 10.1007/s10723-021-09551-5.

[8] Alessandro Berti and Wil M. P. van der Aalst. ‘A Novel Token-Based Replay Technique to Speed Up Conformance Checking and Process Enhancement’. In: *Transactions on Petri Nets and Other Models of Concurrency XV*. Ed. by Maciej Koutny, Fabrice Kordon and Lucia Pomello. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 1–26. DOI: 10.1007/978-3-662-63079-2_1.

[9] Aditya Bhardwaj and C Rama Krishna. ‘Virtualization in cloud computing: Moving from hypervisor to containerization—a survey’. In: *Arabian Journal for Science and Engineering* 46.9 (2021), pp. 8585–8601.

[10] Seppe K. L. M. vanden Broucke et al. ‘Event-Based Real-Time Decomposed Conformance Analysis’. In: *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*. Ed. by Robert Meersman et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 345–363.

[11] Andrea Burattin. ‘Streaming Process Mining’. In: *Process Mining Handbook*. Ed. by Wil M. P. van der Aalst and Josep Carmona. Cham: Springer International Publishing, 2022, pp. 349–372. DOI: `10.1007/978-3-031-08848-3_11`.

[12] Andrea Burattin and Josep Carmona. ‘A Framework for Online Conformance Checking’. In: *Business Process Management Workshops*. Ed. by Ernest Teniente and Matthias Weidlich. Cham: Springer International Publishing, 2018, pp. 165–177.

[13] Andrea Burattin et al. ‘Online Conformance Checking Using Behavioural Patterns’. In: *Business Process Management*. Ed. by Mathias Weske et al. Cham: Springer International Publishing, 2018, pp. 250–267.

[14] Matteo Camilli. ‘Continuous Formal Verification of Microservice-Based Process Flows’. In: *Software Architecture*. Ed. by Henry Muccini et al. Cham: Springer International Publishing, 2020, pp. 420–435.

[15] Josep Carmona, Boudewijn van Dongen and Matthias Weidlich. ‘Conformance Checking: Foundations, Milestones and Challenges’. In: *Process Mining Handbook*. Ed. by Wil M. P. van der Aalst and Josep Carmona. Cham: Springer International Publishing, 2022, pp. 155–190. DOI: `10.1007/978-3-031-08848-3_5`.

[16] Tomas Cerny, Michael J. Donahoo and Michal Trnka. ‘Contextual Understanding of Microservice Architecture: Current and Future Directions’. In: *SIGAPP Appl. Comput. Rev.* 17.4 (Jan. 2018), pp. 29–45. DOI: `10.1145/3183628.3183631`.

[17] Christian Colombo and Gordon J. Pace. ‘Industrial Experiences with Runtime Verification of Financial Transaction Systems: Lessons Learnt and Standing Challenges’. In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Cham: Springer International Publishing, 2018, pp. 211–232. DOI: `10.1007/978-3-319-75632-5_7`.

[18] *Conductor*. URL: `https://conductor.netflix.com/`.

[19] *Docker*. URL: `https://www.docker.com/`.

[20] Nicola Dragoni et al. ‘Microservices: Yesterday, Today, and Tomorrow’. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. DOI: `10.1007/978-3-319-67425-4_12`.

[21] Rajdeep Dua, A Reddy Raja and Dharmesh Kakadia. ‘Virtualization vs Containerization to Support PaaS’. In: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 610–614. DOI: `10.1109/IC2E.2014.41`.

[22] Sebastian Dunzer et al. ‘Conformance Checking: A State-of-the-Art Literature Review’. In: *Proceedings of the 11th International Conference on Subject-Oriented Business Process Management*. S-BPM ONE

’19. Seville, Spain: Association for Computing Machinery, 2019. DOI: 10.1145/3329007.3329014.

[23] *Eclipse*. URL: https://www.eclipse.org/.

[24] *ELK Stack*. URL: https://elastic.co/.

[25] Sima Emadi and Fereidoon Shams Aliee. ‘Transformation of Usecase and Sequence Diagrams to Petri Nets’. In: vol. 4. Sept. 2009, pp. 399–403. DOI: 10.1109/CCCM.2009.5267604.

[26] Montserrat Estañol et al. ‘Conformance checking in UML artifact-centric business process models’. In: *Software & Systems Modeling* 18.4 (2018), pp. 2531–2555. DOI: 10.1007/s10270-018-0681-6.

[27] Martin Fowler and Jamal Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 17/02/2023).

[28] Yu Gan et al. ‘An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems’. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. DOI: 10.1145/3297858.3304013.

[29] *Grafana*. URL: https://grafana.com/.

[30] *Group by Trace Processor*. URL: https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor/groupbytraceprocessor.

[31] Robert Heinrich et al. ‘Performance Engineering for Microservices: Research Challenges and Directions’. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE ’17 Companion. L’Aquila, Italy: Association for Computing Machinery, 2017, pp. 223–226. DOI: 10.1145/3053600.3053653.

[32] *Jaeger*. URL: https://jaegertracing.io/.

[33] Pooyan Jamshidi et al. ‘Microservices: The Journey So Far and Challenges Ahead’. In: *IEEE Software* 35.3 (2018), pp. 24–35. DOI: 10.1109/MS.2018.2141039.

[34] Asif Khan. ‘Key Characteristics of a Container Orchestration Platform to Enable a Modern Application’. In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933.

[35] Charalampos Gavriil Kominos, Nicolas Seyvet and Konstantinos Vandikas. ‘Bare-metal, virtual machines and containers in OpenStack’. In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. 2017, pp. 36–43. DOI: 10.1109/ICIN.2017.7899247.

[36] *Kubernetes*. URL: https://www.kubernetes.io/.

[37] Rodrigo Laigner et al. ‘Data Management in Microservices: State of the Practice, Challenges, and Research Directions’. In: *Proc. VLDB Endow.* 14.13 (Oct. 2021), pp. 3348–3361. DOI: 10.14778/3484224.3484232.

[38] Nebrass Lamouchi. 'Introduction to the Monolithic Architecture'. In: *Pro Java Microservices with Quarkus and Kubernetes: A Hands-on Guide*. Berkeley, CA: Apress, 2021, pp. 35–38. DOI: 10.1007/978-1-4842-7170-4_2.

[39] Wai Lam Jonathan Lee et al. 'Orientation and conformance: A HMM-based approach to online conformance checking'. In: *Information Systems* 102 (2021), p. 101674. DOI: https://doi.org/10.1016/j.is.2020.101674.

[40] Bowen Li et al. 'Enjoy Your Observability: An Industrial Survey of Microservice Tracing and Analysis'. In: *Empirical Softw. Engg.* 27.1 (Jan. 2022). DOI: 10.1007/s10664-021-10063-9.

[41] Charity Majors, Liz Fong-Jones and George Miranda. *Observability Engineering: Achieving production excellence*. O'Reilly, 2022.

[42] Dulani Meedeniya, Indika Perera and Juliana Bowles. 'Tool support for transforming Unified Modelling Language sequence diagram to coloured Petri nets'. In: *Maejo international journal of science and technology* 10 (Nov. 2016), pp. 272–283.

[43] Jorge Munoz-Gama. 'Conformance Checking'. In: *Encyclopedia of Big Data Technologies*. Ed. by Albert Zomaya, Javid Taheri and Sherif Sakr. Cham: Springer International Publishing, 2020, pp. 1–12. DOI: 10.1007/978-3-319-63962-8_89-2.

[44] *OpenTelemetry*. URL: https://opentelemetry.io/.

[45] Claus Pahl. 'Containerization and the PaaS Cloud'. In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31. DOI: 10.1109/MCC.2015.51.

[46] Aurojit Panda, Mooly Sagiv and Scott Shenker. 'Verification in the Age of Microservices'. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 30–36. DOI: 10.1145/3102980.3102986.

[47] *plantuml*. URL: https://plantuml.com/.

[48] *Prometheus*. URL: https://prometheus.io/.

[49] Kristo Raun, Riccardo Tommasini and Ahmed Awad. 'I Will Survive: An Event-Driven Conformance Checking Approach Over Process Streams'. In: *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*. DEBS '23. Neuchatel, Switzerland: Association for Computing Machinery, 2023, pp. 49–60. DOI: 10.1145/3583678.3596887.

[50] Kristo Raun et al. 'C-3PA: Streaming Conformance, Confidence and Completeness in Prefix-Alignments'. In: *Advanced Information Systems Engineering*. Ed. by Marta Indulska et al. Cham: Springer Nature Switzerland, 2023, pp. 437–453.

[51] Daniel Schuster and Sebastiaan J. van Zelst. 'Online Process Monitoring Using Incremental State-Space Expansion: An Exact Algorithm'.

In: *Business Process Management*. Ed. by Dirk Fahland et al. Cham: Springer International Publishing, 2020, pp. 147–164.

[52] C. Sridharan. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018.

[53] Muhammad Usman et al. 'A Survey on Observability of Distributed Edge & Container-Based Microservices'. In: *IEEE Access* 10 (2022), pp. 86904–86919. DOI: `10.1109/ACCESS.2022.3193102`.

[54] Han van der Aa, Henrik Leopold and Matthias Weidlich. 'Partial order resolution of event logs for process conformance checking'. In: *Decision Support Systems* 136 (2020), p. 113347. DOI: `https://doi.org/10.1016/j.dss.2020.113347`.

[55] Simon Volpert et al. 'The view on systems monitoring and its requirements from future Cloud-to-Thing applications and infrastructures'. In: *Future Generation Computer Systems* 141 (2023), pp. 243–257. DOI: `https://doi.org/10.1016/j.future.2022.11.024`.

[56] Nianhua Yang et al. 'Modeling UML Sequence Diagrams Using Extended Petri Nets'. In: *2010 International Conference on Information Science and Applications*. 2010, pp. 1–8. DOI: `10.1109/ICISA.2010.5480384`.

[57] Rashid Zaman, Marwan Hassani and Boudewijn F. van Dongen. 'Efficient Memory Utilization in Conformance Checking of Process Event Streams'. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 437–440. DOI: `10.1145/3477314.3507217`.

[58] Rashid Zaman, Marwan Hassani and Boudewijn F. Van Dongen. 'Prefix Imputation of Orphan Events in Event Stream Processing'. In: *Frontiers in Big Data* 4 (2021). DOI: `10.3389/fdata.2021.705243`.

[59] Sebastiaan J. van Zelst et al. 'Online conformance checking: Relating event streams to process models using prefix-alignments'. In: *International Journal of Data Science and Analytics* 8.3 (2017), pp. 269–284. DOI: `10.1007/s41060-017-0078-6`.