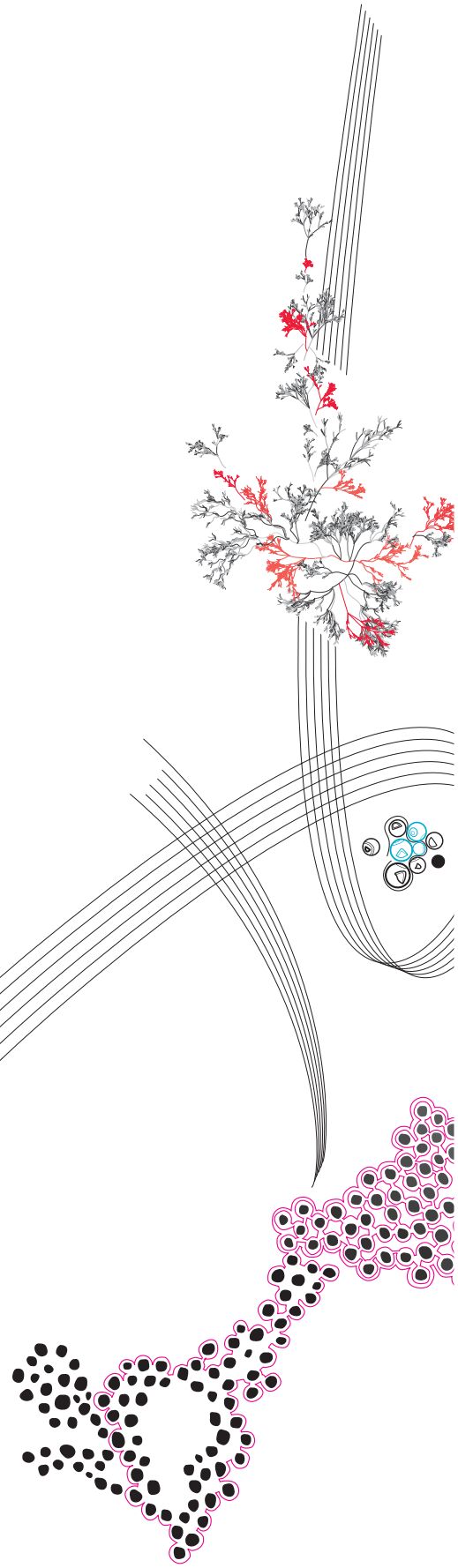

MSc Computer Science
Final Project

Analysing Service Interactions in Traced Microservice Architectures Using Process Mining

Michael Janssen

Supervisors:
dr. ir. Vadim Zaytsev
dr. Faiza Bukhsh
ir. Jan-Jelle Kester

June, 2026

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Microservice systems are hard to understand: the behaviour of a single request is spread across many independently deployed services and visible nowhere in full. Distributed tracing frameworks such as OpenTelemetry record that behaviour one request at a time, but a pile of individual traces does not reveal the general patterns of interaction that characterise a system. Process mining can turn such data into structured models automatically, but when applied naively to OpenTelemetry traces, it results in unintelligible and incorrect models.

This thesis asks how process mining can be applied to OpenTelemetry traces so that the result helps engineers and architects understand, validate, and improve their systems. Its answer is *layered process mining*: rather than one global model, mine one model per service operation from only the direct children of that operation's spans, and represent each call to a child operation as an expandable subprocess reference. This preserves the hierarchy of a trace end to end, producing models that are structurally faithful and stay navigable however large the system grows. The approach is grounded in interviews with six engineers and architects at Info Support and realised in **Ariadne**, an open-source framework that mines a hierarchical model per operation with a lifecycle-aware Inductive Miner, checks conformance against a reference architecture, and renders the models as an interactive BPMN visualisation.

Evaluated on the MediaMicroservices benchmark, the mined model of its most complex endpoint matches the application source code, capturing concurrency a completion-order miner would have serialised. Evaluated on Uber's CRISP dataset of 100,000 production traces, Ariadne mines all 18,386 operations in under six minutes on a single workstation while keeping model quality high. Once the hierarchy of traces is preserved rather than discarded, process mining becomes a practical lens on runtime microservice behaviour.

Keywords: process mining; distributed tracing; OpenTelemetry; microservice architecture; conformance checking; software observability.

Chapter 1

Introduction

Modern software systems are increasingly built as collections of small, independently deployable services. Companies such as Netflix, Amazon, Google, and Uber have successfully implemented this microservice architectural style at scale [40, 21]. It promises independent deployment, fault isolation, and development agility [20]. However, the same properties that make microservices attractive also make them difficult to understand [31]. As the number of services grows, so does the complexity of the interactions between them. A single user-facing request may traverse dozens of services, in a web of nested calls whose combined behaviour is nowhere visible in full.

Observability tools, such as OpenTelemetry, have emerged to help address this. Distributed tracing systems, and OpenTelemetry in particular, capture the journey of a request as it propagates through a system, recording each operation as a span and linking spans into hierarchical traces that reflect the full call graph. In principle, this representation captures everything needed to reason about runtime behaviour. In practice, making sense of it remains difficult. Traces describe individual requests, not the general patterns that emerge across thousands of them. Engineers inspecting traces in tools such as Jaeger [7] see instances, not processes. The question of what the system typically does, which service calls which, in what order, with what concurrency, remains difficult to answer.

Process mining offers a way to bridge this gap. Originally developed for business process management, process mining derives structured process models directly from event logs, without requiring a manually specified model beforehand [39]. Applied to distributed traces, it offers the potential to transform raw observability data into interpretable, structured representations of runtime behaviour. However, existing applications of process mining to microservice traces have treated traces as flat event logs [35], discarding the parent–child relationships between spans that are fundamental to their meaning. The resulting models are shallow and structurally incorrect, misrepresenting the systems they are supposed to describe.

This thesis addresses that problem. We propose a layered process mining approach in which each service operation is mined independently into its own process model, with child operations represented as subprocesses. This preserves the hierarchical structure of OpenTelemetry traces end-to-end, producing models that are both structurally faithful and practically navigable. To ground the work in real engineering needs, we conducted interviews with six engineers and architects at Info Support, the Dutch software company where this research was carried out. These interviews surfaced two recurring needs: structural visibility and automated verification. These needs informed the design of a framework that integrates the layered mining approach with a visualisation interface and a conformance checking module, enabling engineers to both explore mined models and verify whether

system behaviour matches a reference architecture.

The framework was evaluated on two systems of different scale: the MediaMicroservices application from the DeathStarBench benchmark suite, and a large public dataset of traces released by Uber. Together, these evaluations demonstrate that the approach is feasible across both synthetic and real-world environments, and that it surfaces behavioural patterns and deviations that would otherwise remain hidden in raw trace data.

The goal of this thesis is to answer the following question:

> How can process mining techniques be applied to OpenTelemetry traces to support engineers and architects in understanding, validating, and improving microservice systems?

Figure 1.1 gives an overview of how this project unfolded and how the chapters of this thesis relate. The remainder of this thesis is structured as follows. Chapter 2 presents the layered process mining approach. Chapter 3 describes the expert interviews and their findings. Chapter 4 details the framework design and implementation. Chapter 5 evaluates the approach on the two systems described above. Chapter 6 concludes the thesis, discusses the results, reflects on limitations, and outlines directions for future work.

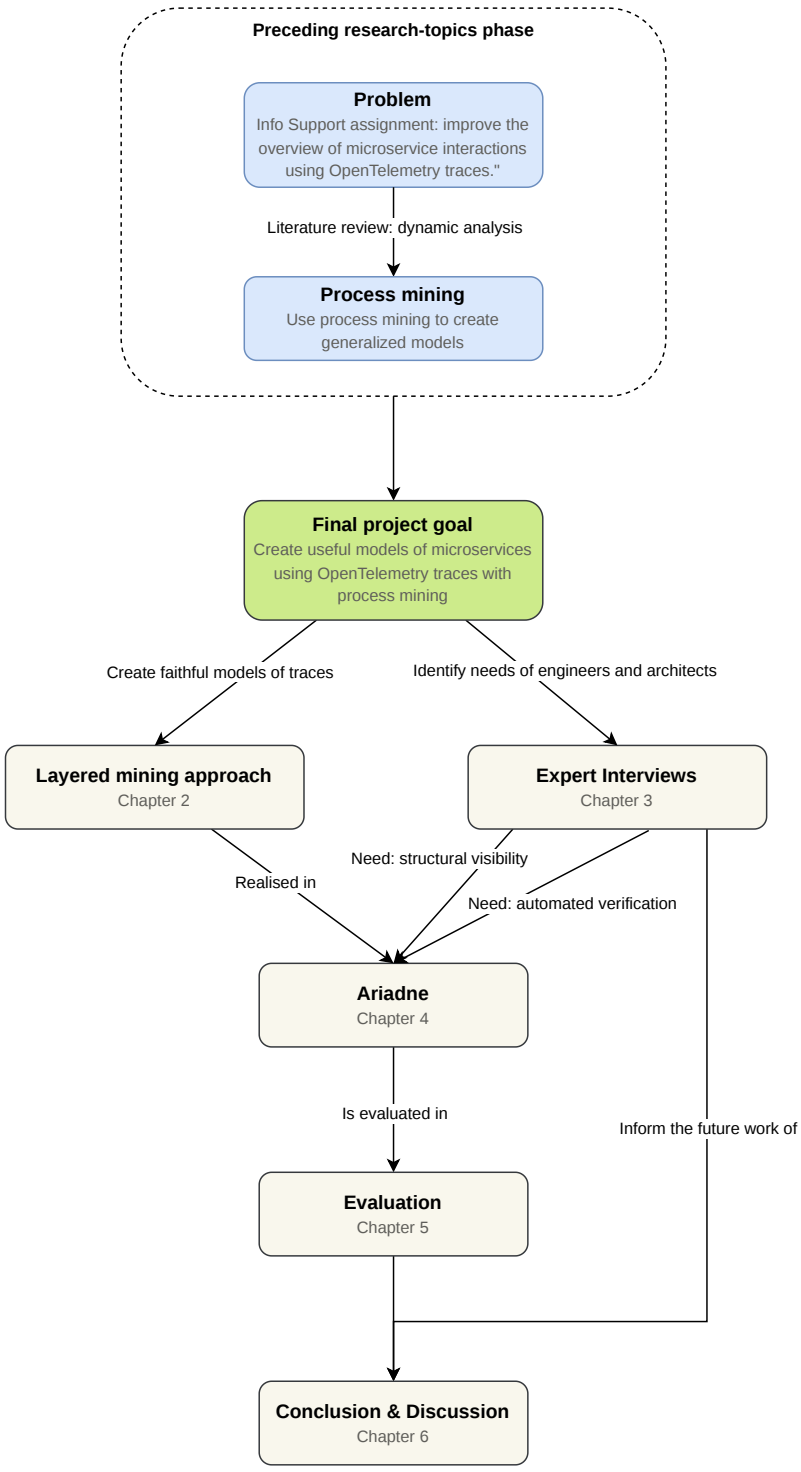

FIGURE 1.1: Overview of the project. The research-topics phase (top, dashed) preceded this thesis and is assessed separately; it framed the goal that Chapters 2–6 address. Arrows show how each chapter feeds the next.

Chapter 2

Layered Mining

Microservice traces are not flat sequences of events. They are trees. Each span, except for the root, has a parent. This structure is different from the situations where process mining is normally applied, with a flat event log where events are peers. This chapter describes how we approach this. Before doing so, Section 2.1 introduces distributed tracing and the OpenTelemetry framework which we investigate. Section 2.2 introduces the process mining concepts the approach builds on: event logs and discovery. Section 2.3 then identifies the problem with applying these tools directly to traces, and Section 2.4 presents the layered approach itself.

2.1 Distributed tracing in OpenTelemetry

Distributed tracing captures the journey of a single request as it propagates through a distributed system, providing a unit of observation that crosses service boundaries and is more informative than per-service logs or aggregate metrics [33]. Our approach takes as input traces produced by OpenTelemetry, the open-source standard that has become the dominant choice for instrumenting distributed systems [9]. OpenTelemetry covers three telemetry signals: traces, metrics, and logs. Only tracing is relevant to the work in this thesis. What matters for the remainder of the chapter is how OpenTelemetry represents a single execution.

A *trace* records one end-to-end execution of a single request as a tree of *spans*. A span captures a single unit of work, typically a remote procedure call, a database query, or an internal operation. Each span carries a start timestamp, a complete timestamp, an operation name, a unique span identifier, and a reference to its parent span; every span in the same execution also shares a single *trace identifier*. The root span has no parent and represents the entry point of the request, so the whole trace is the tree rooted at that span. A useful way to read the tree is by containment: an edge from a parent span to a child span means the parent delegated that unit of work to the child and does not finish until the child does. This tree structure is what distinguishes microservice traces from the flat event logs that most process mining algorithms are designed for, and it is the property our approach is built to preserve.

A running example. To make the rest of the chapter concrete, we draw all examples from a single source: the OpenTelemetry Demo [4], the reference application maintained by the OpenTelemetry project. The demo simulates an online astronomy retail shop composed of twenty services written in twelve languages, communicating over gRPC and HTTP

and backed by PostgreSQL, Kafka, and Valkey. We use the demo for illustration only; quantitative evaluation of the approach will be done in Chapter 5.

The user action used throughout this chapter is the *checkout flow*: a request to place an order. Figure 2.1 shows the resulting span tree. The request enters at the frontend proxy, reaches the `checkout` service, and fans out to seven downstream services. Some execute concurrently, others are strictly ordered: `cart/GetCart` must complete before `product-catalog/GetProduct`; `shipping/PostOrder`, `email/send_order_information` and `shipping/get-quote` all execute concurrently.

Three properties of this trace recur throughout the chapter. First, the hierarchy is structural: a child span represents work the parent delegated, and the parent's duration encloses the child's. Second, siblings can execute concurrently: the three calls to `product-catalog/GetProduct` at the end of the trace overlap in time. Third, the same operation can be invoked many times within a single parent: `product-catalog/GetProduct` is called once per item in the cart. These are not specific to the demo; they are the structural features that any process model of a microservice system must accommodate.

2.2 Process mining

A single trace tells us what happened in one request. What an engineer typically wants to understand is what happens across thousands of them: the patterns of service interaction the system exhibits when it is doing its job. Process mining is the discipline that automates this generalisation.

Process mining derives structured process models directly from event logs [39]. Where traditional business process management builds models by hand through interviews and document analysis, process mining starts from the data that systems already produce. The discipline distinguishes three activities: *discovery* produces a model from an event log without any prior reference; *conformance* compares a log against an existing model to identify deviations; and *enhancement* extends a model with additional perspectives such as time or resources.

2.2.1 Event logs

Process mining operates on *event logs*. Each event in a log records at minimum four pieces of information: a *case identifier* grouping events that belong to the same process instance, an *activity name* identifying what happened, a *timestamp* that establishes order, and a unique *event identifier*. Additional attributes may be attached for further analysis. Every event carrying the same case identifier is treated as part of one execution of the process, and discovery generalises a model from the set of all such cases. A *case* is therefore exactly the sequence of events that share a case identifier, and choosing the case identifier is what defines what counts as "one run" of the process.

Table 2.1 shows a slice of an event log derived from the checkout trace of Figure 2.1. Each row is one lifecycle event of one span; the case identifier groups events that share an invocation of `checkout/PlaceOrder`, and the activity name is the operation of the corresponding child span. Two details are already visible in the timestamps and will matter later. The start and complete intervals of `cart/GetCart`, `product-catalog/GetProduct`, and `currency/Convert` overlap: these spans run concurrently, a property a process model should preserve.

TABLE 2.1: Event-log slice for one `checkout/PlaceOrder` invocation, derived from Figure 2.1. Timestamps are in microseconds; the full log contains many such cases.

case_id	activity	timestamp	lifecycle
d113...	frontend/POST /api/checkout	13:05:26.023000	start
d113...	frontend/oteldemo.CheckoutService/PlaceOrder	13:05:26.023000	start
d113...	frontend-proxy/router frontend egress	13:05:26.023030	start
d113...	checkout/oteldemo.CheckoutService/PlaceOrder	13:05:26.024427	start
d113...	checkout/prepareOrderItemsAndShippingQuoteFr...	13:05:26.024490	start
d113...	checkout/oteldemo.CartService/GetCart	13:05:26.024503	start
d113...	cart/POST /oteldemo.CartService/GetCart	13:05:26.024737	start
d113...	cart/valkey-cart:6379	13:05:26.024810	start
d113...	cart/valkey-cart:6379	13:05:26.025133	complete
d113...	cart/POST /oteldemo.CartService/GetCart	13:05:26.025180	complete
d113...	checkout/oteldemo.CartService/GetCart	13:05:26.025335	complete
d113...	checkout/oteldemo.ProductCatalogService/GetP...	13:05:26.025365	start
d113...	product-catalog/oteldemo.ProductCatalogServi...	13:05:26.025570	start
d113...	product-catalog/astronomy-db	13:05:26.025640	start
d113...	product-catalog/astronomy-db	13:05:26.026011	complete
d113...	product-catalog/oteldemo.ProductCatalogServi...	13:05:26.026065	complete
d113...	checkout/oteldemo.ProductCatalogService/GetP...	13:05:26.026137	complete
d113...	checkout/oteldemo.CurrencyService/Convert	13:05:26.026161	start
d113...	currency/oteldemo.CurrencyService/Convert	13:05:26.026415	start
d113...	currency/oteldemo.CurrencyService/Convert	13:05:26.026992	complete
	...		

2.2.2 Discovery and model quality

Given an event log, *process discovery* attempts to derive a model that captures the behaviour the log records. The discovered model is typically a Petri net, that we introduce in Section 2.3.1 when we need it to discuss what existing approaches produce.

The quality of a discovered model is conventionally assessed along four competing criteria [39]. *Fitness* is the degree to which the model can replay the events in the log; perfect fitness means every observed trace is accepted. *Precision* is the degree to which the model restricts behaviour not in the log; low precision indicates overgeneralisation. *Generalisation* is the model's ability to admit plausible behaviour beyond the log, recognising that any log is incomplete. *Simplicity* is the structural compactness of the model. A good model balances all four; the right balance depends on the use to which the model will be put.

Tools. Several mature implementations of process mining algorithms exist. The ProM toolkit [12] is the most widely used in research and offers the widest selection of algorithms, but it is built around an interactive desktop application and does not expose a publicly documented, stable programming interface for embedding its algorithms into another tool. Since Ariadne needs to call discovery and conformance routines programmatically as part of an automated pipeline, rather than drive them through a graphical interface, ProM was a poor fit. We therefore implement the framework in Python and use PM4Py [11], a process mining library with a documented API that exposes the discovery and conformance algorithms we rely on.

2.3 The problem with flat mining

The most direct way to apply process mining to OpenTelemetry traces is to ignore their hierarchy. Two strategies suggest themselves immediately. The first is *global flat mining*: collect every span from every trace into a single event log and mine the result, treating each span as a peer-level event [35]. The second is *top-level trace mining*: treat each top-level, or user-facing, activity as unit of partitioning traces and produce one model per top-level trace type, again flattening the spans within each trace into a linear sequence ordered by timestamp. Both strategies are simple to implement, and both produce misleading models. To see why, we first need to explain what discovery algorithms produce.

2.3.1 Petri nets and workflow nets

A *Petri net* [30] is a directed bipartite graph consisting of two kinds of nodes: *places*, drawn as circles, and *transitions*, drawn as rectangles. Edges, called *arcs*, connect places to transitions and transitions to places, but never two nodes of the same kind. Places hold zero or more *tokens*; the distribution of tokens across places defines the net's *marking*, its current state. A transition is *enabled* when each of its input places contains at least one token, and firing an enabled transition consumes one token from each input place and produces one token in each output place. In a process mining context, transitions correspond to activities in the event log, places represent the conditions linking those activities, and the flow of tokens models a single process execution.

A transition need not correspond to an observed activity. *Silent* transitions, also called hidden or τ-transitions, carry no activity label and emit no event when they fire. They exist only to route tokens so that the net can express control-flow constructs such as skipping an optional branch or closing a loop. In the figures throughout this thesis, silent transitions are drawn as solid black rectangles, whereas labelled transitions that correspond to an activity are drawn as outlined rectangles bearing the activity name.

A *workflow net* (WF-net) is a Petri net constrained to model processes with a single, well-defined start and end [38]. The net has exactly one source place with no incoming arcs, exactly one sink place with no outgoing arcs, and every node lies on a directed path from source to sink. A workflow net is *sound* [39] when it satisfies four conditions: from any reachable marking the final marking remains reachable, so there are no deadlocks or livelocks; no reachable marking places more than one token in any single place; when the sink is reached no other place holds a token; and every transition can fire in some execution.

2.3.2 What flat mining produces

Consider what global flat mining produces for the checkout flow of Figure 2.1. The event log is a single bag of spans drawn from many trace executions: `frontend/PlaceOrder`, `checkout/PlaceOrder`, `cart/GetCart`, `valkey/HGET`, `product-catalog/GetProduct`, `currency/Convert`, `shipping/GetQuote`, `quote/GetQuote`, `payment/Charge`, `shipping/ShipOrder`, `kafka/publish`, and `email/SendOrderConfirmation` all appear as peers. The discovered Petri net (Figure 2.2) places transitions for each of these operations at the same level. A Valkey `HGET`, a leaf operation that exists only because the cart implementation happens to use a key-value store, becomes structurally indistinguishable from `checkout/PlaceOrder`, the top-level orchestrator of the entire flow.

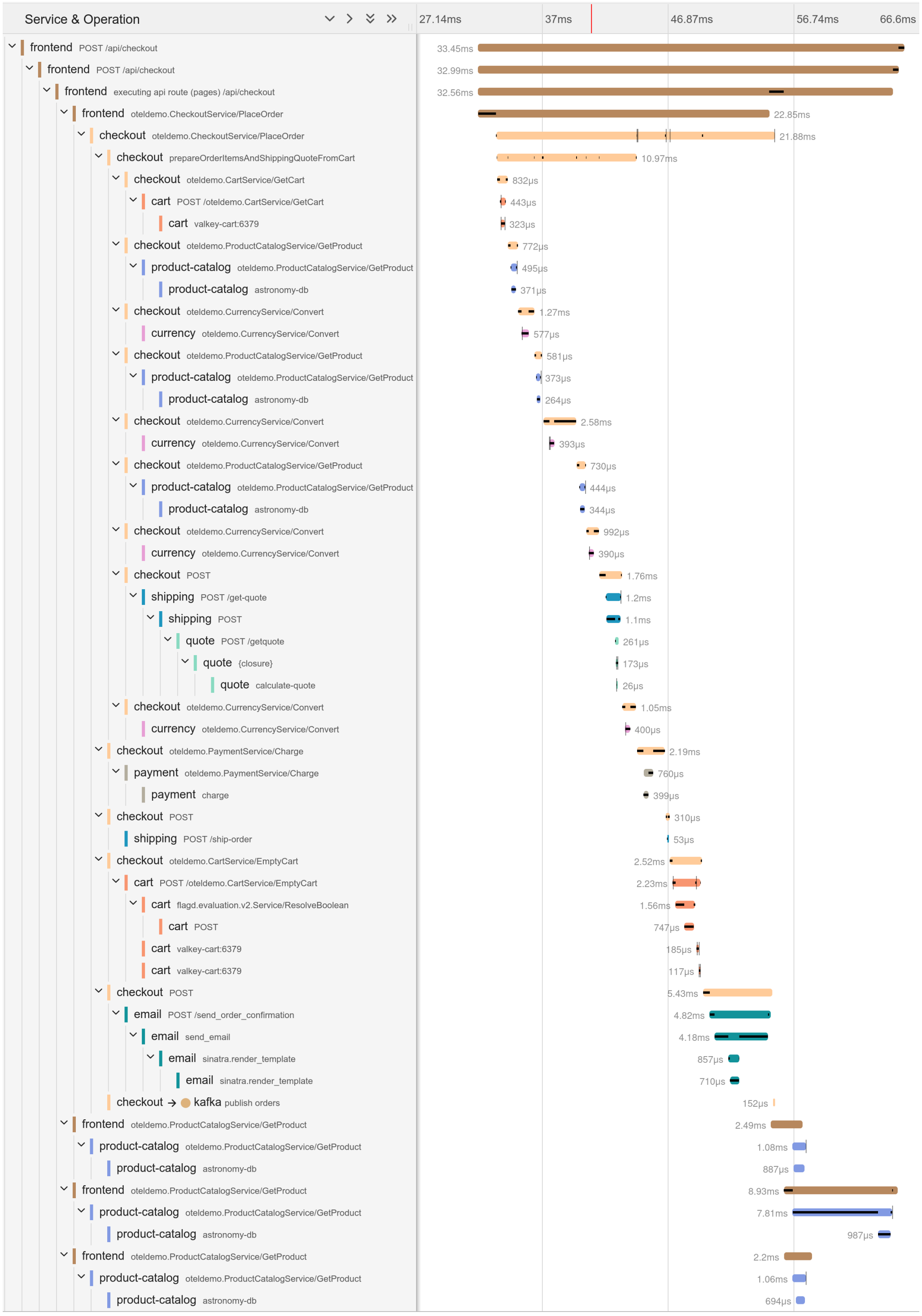

FIGURE 2.1: The span tree for a single `PlaceOrder` request in the OpenTelemetry Demo. The screenshot is taken from Jaeger.

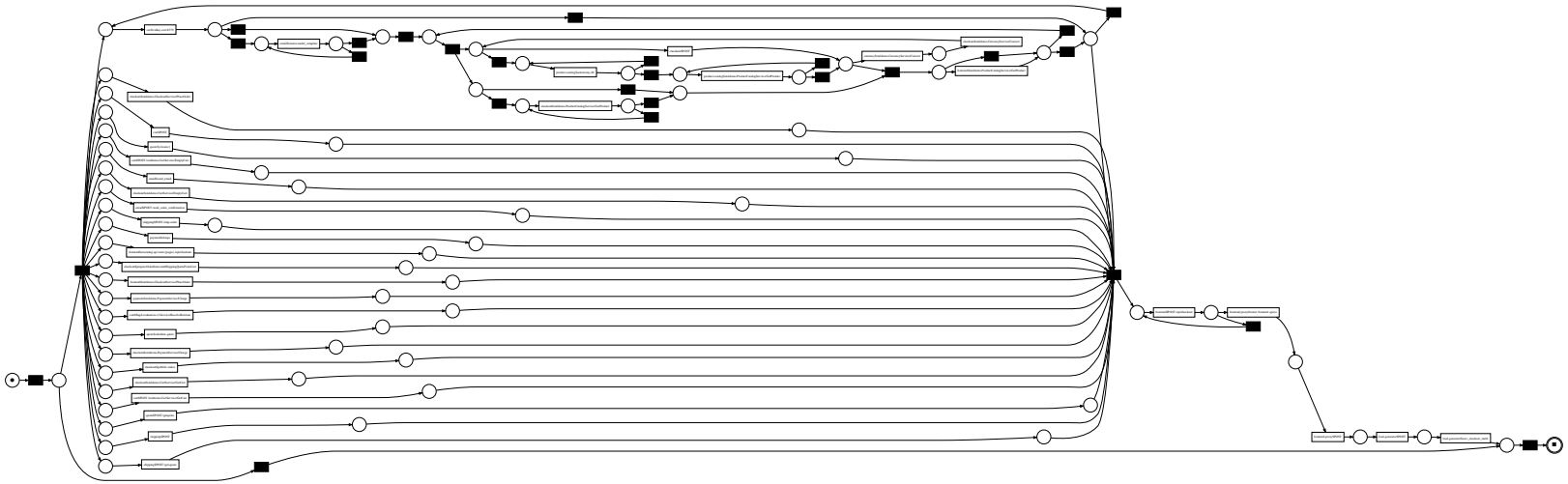

FIGURE 2.2: Petri net produced by top-level flat mining of the checkout flow. Many operations are concurrent. It is difficult to read and structurally incorrect.

This model is not merely ugly; it is structurally incorrect. A parent–child relationship between two spans is not a coincidence of ordering: it is a containment relation. A child span's work is part of its parent's work. `valkey/HGET` is part of how `cart/GetCart` is implemented; `quote/GetQuote` is part of how `shipping/GetQuote` delivers a quote. Flat mining treats these as independent events that just happen to occur near each other in time, discarding the subprocess semantics that make the trace interpretable in the first place.

The conclusion is straightforward. Microservice traces carry structural information that flat mining destroys, and no amount of post-processing can recover hierarchy from a flattened log. A mining approach that preserves the parent–child structure must operate on the structure directly.

2.4 Layered process mining

To preserve the structure that flat mining destroys, we propose a layered process mining approach. The core idea is simple: rather than mining a single global model that flattens all spans, we mine one process model per service operation, using only the direct children of that operation's spans as events. Child operations are then represented as subprocess references within the parent's model. The result is a hierarchical process model that mirrors the structure of the traces it was derived from.

2.4.1 The pipeline

For a given set of complete traces, the layered approach executes the following pipeline:

1. **Sublog construction.** The event log is partitioned by parent span operation. For each operation o that appears as a parent somewhere in the log, a sublog L_o is constructed containing the events derived from spans that are direct children of an o-span.

2. **Lifecycle enrichment.** For each span in L_o, two events are derived from its recorded timestamps: a *start* event and a *complete* event. The motivation is given in Section 2.5.2.

3. **Empty traces.** If an operation has variants with no child spans, these are not by default included in the partition, as there are no spans for that trace. Before the models are mined, a trace variant list is built. If the total amount of traces with operation o is bigger than the amount of traces in L_o, the empty variant is added. This way, a silent transition directly from source to sink is added if there are empty traces for an operation.

4. **Per-operation discovery.** Each L_o is mined independently using the Inductive Miner with lifecycle extensions, producing a sound workflow net N_o. The choice of algorithm is justified in Section 2.5.1.

5. **Referenced composition.** The per-operation nets are edited so that a transition for a child operation o' inside N_o is replaced by a reference to $N_{o'}$.

Figure 2.3 shows the result for the checkout flow. The top-level view consists of the model for `frontend/POST /api/checkout`, which consists of the sequential `checkout/PlaceOrder` and `product-catalog/GetProduct`. Each of those transitions is itself a reference to its own model: the engineer can drill into any subprocess they care about and ignore the rest. For illustration, the subprocess for `checkout/PlaceOrder` is also shown.

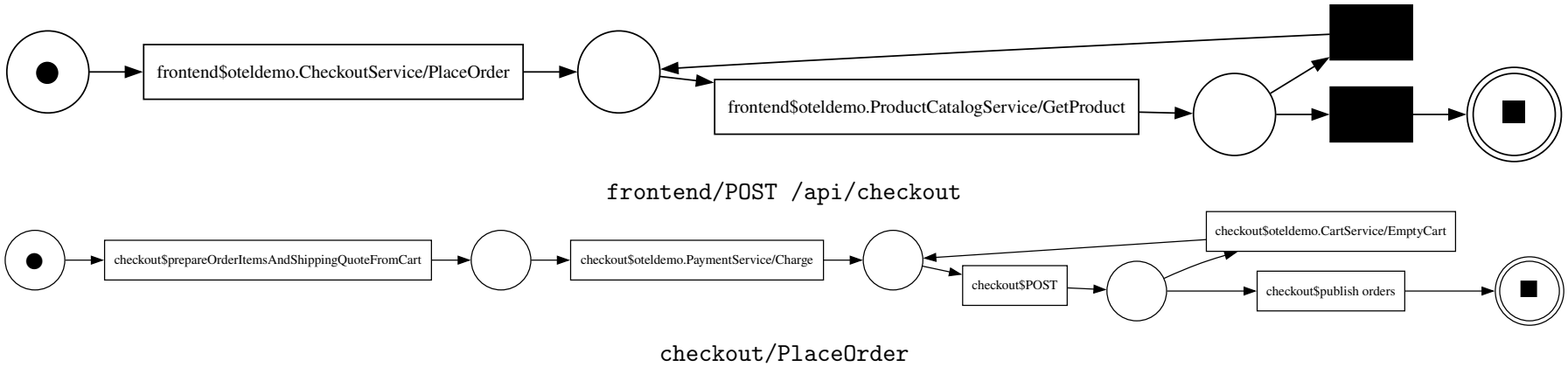

FIGURE 2.3: The same checkout flow mined with the layered approach. The first model shows the structure of `frontend/POST /api/checkout`: First, the order is placed; then, all items that have been ordered are queried using `product-catalog/GetProduct` to display in the result page. The second model is the subprocess `checkout/PlaceOrder` in the first model.

2.4.2 Case identifier

The case identifier is the pair of trace and parent span, rather than the trace alone. This matters whenever an operation can be invoked more than once within the same trace. In the checkout flow, `product-catalog/GetProduct` is called once per cart item: a trace for a three-item order contains three invocations as children of `frontend/POST /api/checkout`. If the case identifier were trace ID alone, the children of all three invocations would collapse into a single case, which is incorrect. Pairing the trace with the parent span cuts the sublog at the right boundary: one parent span instance, one case. Each case in L_o therefore corresponds to exactly one invocation of o within one trace, and its events are the children spawned by that invocation.

2.4.3 Advantages of the layered approach

Three properties follow from this construction.

The first is *structural integrity*: the hierarchy in the original traces is preserved end-to-end. A child operation that represents work done on behalf of a parent appears as a subprocess of that parent in the model, not as a peer. The model reflects the system as it is.

The second is *navigability*: regardless of how many operations the system contains, no single view of the model grows beyond a manageable size. The top-level view of the checkout flow shows the structure of `checkout/PlaceOrder` and nothing else; deeper structure stays nested within subprocesses, surfacing only when a subprocess is expanded. A flat model of the same system is a single net containing every span operation across every trace, well past the threshold of human readability.

The third is *independence*: because each L_o is mined in isolation and shares no state with any other, the per-operation discovery tasks can run in parallel. A system with hundreds of operations can be mined as quickly as its slowest single operation, given enough cores. This is a consequence of the design rather than a primary motivation, but it is useful in practice and helps the production-scale evaluation in Chapter 5.

2.5 Design decisions

Three concrete choices shape the pipeline of Section 2.4.1. Each is forced by a property of microservice traces that flat mining could afford to ignore but layered mining cannot.

2.5.1 The Inductive Miner

Step four of the pipeline mines each operation sublog L_o into a workflow net. Several discovery algorithms could be used here; the layered approach itself is agnostic to the choice. The downstream uses of the mined model — conformance checking and visualisation — impose a hard requirement, however: each N_o must be sound. An unsound model contains deadlocks or unreachable transitions by construction, and conformance checking against such a model returns meaningless results.

We use the *Inductive Miner* [25]. Unlike earlier algorithms such as the Alpha-miner, which infer a Petri net from local ordering relations between activities, the Inductive Miner operates top-down. Given an event log, it identifies a *cut*: a partitioning of the activities and the log according to one of four operators — sequence, exclusive choice, parallel composition, or loop. Each operator corresponds to a fragment of a *process tree*, a hierarchical representation of the process. The cut splits the log into smaller sublogs, one per branch of

the operator, and the algorithm recurses on each. Recursion terminates when a sublog can be modelled by one of the defined base cases or a fall-through, at which point the process tree is translated into a Petri net.

Several variants of the Inductive Miner exist, adding features such as noise filtering and support for incomplete behaviour [27, 26]. The variant we actually use is the lifecycle-aware variant introduced next.

2.5.2 Lifecycle enrichment

Standard process mining algorithms treat each event in the log as an atomic occurrence: an activity either happened or did not, and ordering is determined by event timestamps. This abstraction is reasonable when activities are short relative to the gaps between them, but it loses information when activities can overlap in time.

In microservice traces, overlap is the norm rather than the exception. A miner that orders activities by completion timestamp cannot capture this property well. If one of the concurrent processes is significantly slower than the others, a lifecycle-unaware miner will produce a sequential model, and on smaller samples infer phantom sequential dependencies that do not exist.

Lifecycle-aware mining addresses this by representing each activity not as a single event but as a pair: a *start* event marking the beginning of the activity and a *complete* event marking its end. The XES standard for event logs defines a controlled vocabulary of lifecycle transitions [13], of which *start* and *complete* are the most commonly used. With this richer representation, two activities whose execution intervals overlap can be better distinguished from two activities that occur strictly in sequence.

OpenTelemetry spans carry start and complete timestamps natively, so deriving lifecycle events is straightforward. The lifecycle-aware variant of the Inductive Miner [24] consumes start/complete pairs and produces Petri nets in which the start and complete transitions of each activity are explicitly distinguished, allowing the model to represent concurrent execution faithfully rather than by inference.

2.5.3 Parent–child timing mismatches

The third design decision addresses a problem that does not occur in classical process mining settings: parent and child timestamps that are not strictly nested.

In principle, a child span is fully contained within its parent in time: the parent starts before the child starts and completes after the child completes. In practice, the OpenTelemetry SDKs across different languages do not produce timestamps with identical precision, and they do not always read from identical clock sources. The OpenTelemetry Demo is an instructive case: `checkout/PlaceOrder` runs in a Go service, `shipping/GetQuote` in a Rust service, and `payment/Charge` in a JavaScript service. Even in a single-host deployment, subtle clock differences between language runtimes can cause a child span's recorded start timestamp to precede that of its parent, or its complete timestamp to follow. In production deployments spanning multiple hosts, the effect is more pronounced.

If the parent's timestamps were included directly in L_o, such mismatches would produce malformed lifecycle sequences in which a child appears to start before its parent or complete after it. The Inductive Miner would treat the parent and its children as activities of a single process and infer relations between them that violate the actual containment.

We avoid the problem by excluding the parent span from its own sublog entirely. L_o contains only the children of o-spans; the parent is not represented as an event in the

sublog at all. The approach eliminates timing inconsistencies as a source of malformed input without requiring timestamp correction or normalisation across SDKs.

2.6 Conclusion

This chapter has presented the layered process mining approach of this thesis. Microservice traces are hierarchical, but process mining algorithms assume flat event logs. Flattening that hierarchy, whether globally or per-trace, destroys the subprocess semantics in parent–child span relationships and produces shallow, structurally incorrect models. The layered approach addresses this by mining one process model per service operation, using only the direct children of that operation's spans as events, and representing child operations as subprocess references. The resulting model mirrors the hierarchical structure of the traces themselves: structurally faithful, navigable at any level, and amenable to per-operation parallelism during mining. Three design decisions shape the concrete pipeline. The Inductive Miner is used for each per-operation sublog because it provides a soundness guarantee that the downstream conformance checker depends on. Lifecycle-aware mining is used to distinguish concurrent child spans from sequential ones, since completion-order alone cannot make that distinction. And the parent span is excluded from its own sublog to sidestep the most common clock-skew failure mode in distributed tracing.

The remainder of this thesis instantiates and evaluates this approach. Chapter 3 reports on expert interviews for insight in real engineering needs and surfaced two themes: structural visibility and automated verification. These themes shaped the framework around the mining core. Chapter 4 presents Ariadne, the implementation of the approach, covering trace collection, hierarchical discovery, conformance checking and visualisation. Chapter 5 evaluates Ariadne for correctness on the MediaMicroservices benchmark and for scalability on production traces from Uber.

Chapter 3

Expert Interviews

Chapter 2 derived the layered approach from the structure of traces. This chapter investigates the possible use cases for the models generated using this approach. Here, we present the results of the expert interviews conducted at Info Support for the trace mining framework. The interviews had two objectives: to investigate the observability challenges that architects and engineers face in microservice systems, and to identify use cases where trace-derived process models can help address them.

This chapter starts with the interview setup and methodology. The results section then presents the six use cases that emerged, discussing for each its description by the interviewees, its position relative to related work, and an assessment of how well it can be served by trace-derived process models. The chapter concludes by grouping the use cases into the two themes that emerged across the interviews.

3.1 Methodology

Six Info Support professionals participated in the interviews, employed as engineer, senior engineer, or software architect. The participants volunteered after an announcement on the company's internal messaging board. The interview process was approved by the faculty ethics board. A semi-structured interview format was used [14], guided by a predefined set of topics and questions. Each interview lasted approximately one hour and was conducted in Dutch.

The interviews began with a brief personal introduction. The participants were then asked to describe their experience with microservice-based systems and observability. This was followed by a discussion of their use of current observability tools, including logging, monitoring, and tracing, as well as the challenges they face in this context. Subsequently, participants were asked about their familiarity with and use of process modelling techniques and notations, such as BPMN. Next, the research goal of automatically generating process models from microservice traces was introduced. The final part of each interview focused on potential use cases of these process models in the participants' professional contexts, exploring how these models might help address the previously discussed challenges.

All interviews were transcribed afterwards. Each transcript was read once to develop a set of analytical codes, all of which related to potential use cases for the process models. In a second reading, relevant quotes were systematically coded using this set. The resulting codes were then grouped into thematic categories, the use cases, which are described in the following section. All participants have been anonymised and personally identifiable information has not been included in this thesis.

The study uses a small sample of six participants, which is appropriate for a qualitative

inquiry. The use cases reported here should be understood as patterns observed within this group, not as a definitive list of needs in the industry.

3.2 Results

This section describes the use cases raised by the participants. For each use case, we first introduce the concept and summarise the participants' comments about it. All participants reported having worked on systems with limited or no tracing capabilities. Therefore, for each use case, we also discuss whether and how it can be handled by introducing OpenTelemetry, and explore whether the approach in this research introduces new possibilities.

Table 3.1 shows the six use cases that were discussed during the interviews. The use cases range from developer-oriented tasks such as detecting deprecated APIs to broader architectural and business-level analyses. Some were raised by multiple participants independently, while others emerged from only the specific project context of a single participant. Not all use cases are equally suited to the process mining approach proposed in this thesis. For some, existing tools or simpler trace-based analyses may be more appropriate. For each use case, we describe the core problem as articulated by the participants, position it relative to existing work, and assess how well it can be served by trace-derived process models, including whether process mining adds anything beyond simply introducing tracing. A more detailed explanation of the features that were implemented is provided in Chapter 4.

	P1	P2	P3	P4	P5	P6
API deprecation	✓		✓			
Architecture analysis	✓					✓
Client aggregation					✓	
Living documentation		✓	✓	✓	✓	✓
Architecture alerting			✓	✓		
Business KPIs		✓	✓	✓		

TABLE 3.1: The six use cases discussed with participants

3.2.1 API deprecation

Two participants mentioned working in systems where API deprecation was difficult. There was no clear overview of which API versions were still in use and which services were still utilising them. Consequently, teams needed to support and maintain many versions of an API, which costs more resources. Therefore, the participants suggested seeking a systematic approach to handle API deprecation.

Related work

Adding tracing to an application can already provide a way to handle API deprecation. Bonorden and Van Hoorn [16], for example, provided a tool which can automatically detect deprecated API usage using API specifications and OpenTelemetry. Their approach involves adding custom headers to requests to indicate deprecation for a specific endpoint. If no headers are present but a link to a service name is present, it checks if the API description of that endpoint contains a deprecation notice. They use the OpenTelemetry

Collector to provide the detection, and if a deprecation is found, it includes a list of the deprecations.

There are many approaches to web API versioning in practice. Serbout et al. showed that the vast majority of web APIs use the URL, HTTP Headers, or use an API specification, such as OpenAPI, for versioning [36]. They also showed that, in cases with multiple API versions in production, a majority used URL-based versioning. That is, having major identifiers in the URL to differentiate between API versions, under `/v1/` and `/v2/`, for example, or in the DNS name with `v1.api.example.com` or `v2.api.example.com`.

Approach

Dedicated tools such as the one proposed by Bonorden and Van Hoorn are more suitable for API deprecation: they check every individual trace and can therefore catch every occurrence. Trace-derived process models, by contrast, are generalisations, so infrequently called deprecated endpoints may not appear in a mined model at all. Process mining is therefore poorly suited to exhaustive deprecation detection, but it can complement such tools by providing a structural overview of where deprecated API versions appear in the service interaction flow, helping teams assess the scope of a migration and prioritise their efforts. OpenTelemetry has support for service versioning, in the `service.version` attribute[1], so this information can be collected within OpenTelemetry.

3.2.2 Architecture analysis

Two participants mentioned an approach to apply software architecture metrics to a system. They noted that individual codebases have metrics, which indicate the quality of the code. Similar metrics should be possible on an architecture level, creating metrics that analyse the 'quality' of the architecture.

Related work

Codebases and software projects have many metrics that are often used, such as cyclomatic complexity, the number of lines of code per function, or the coupling between components [17]. Metrics for microservice architectures, and software architectures in general, are not adopted in the industry [17]. However, there are some promising metrics, such as measuring coupling between services [43].

Approach

Having a complete overview of a software architecture that updates dynamically could undoubtedly help popularise microservice architecture metrics. A hierarchical process model derived from traces carries a great deal of structural information and could be enriched with further detail as needed, making it a plausible substrate for such metrics. Any analysis of this kind would, however, be limited to what can be derived from the mined models themselves, that is, from the structure of service interaction rather than from the source code that codebase-level metrics rely on.

3.2.3 Client aggregation

One participant worked in a system which was deployed to multiple clients. They introduced the concept of client aggregation, which allows for the comparison of different

[1] https://opentelemetry.io/docs/specs/semconv/resource/service/#service

deployments of the same architecture. With this, the processes between different clients can be compared.

Related work

We are not aware of work that specifically addresses cross-deployment process comparison in this form.

Approach

Client aggregation can be approached by mining a separate process model for each client deployment and comparing the models to identify structural or behavioural differences. Comparing models in this way can reveal common process paths as well as client-specific variations, and can highlight deviations, bottlenecks, or underutilised components across deployments. A practical advantage of working at the level of mined models is data sensitivity: because the comparison operates on models rather than raw traces, only the mined models, and not the underlying request data, need to leave each deployment.

3.2.4 Living documentation

All but one participant mentioned the need for improved documentation or visualisation of system interactions. Documentation in microservice architectures was often found to be outdated, inconsistent, or incomplete, particularly across teams. Participants suggested that automatically generated documentation could ensure consistency and reduce manual effort, especially if it provided an up-to-date overview of inter-service communication.

For a visualization, participants wanted an interactive view that allows drilling down into specific parts of the architecture. They all indicated at least some familiarity with BPMN, having used them to model business processes or encountered them in their work. Additionally, they wanted an overview of how services interact at different levels of granularity, rather than a single monolithic diagram. Another use case was being able to pinpoint a specific trace, such as one that errored or one that a user reported problems with, and seeing the path it took through the network.

Related work

Documentation in microservice architectures is a well-known challenge. Li et al. surveyed microservice tracing practices across ten companies and found that visualisation and basic statistical metrics are the most common forms of trace analysis [29]. Engineers primarily rely on tools such as Jaeger and Zipkin, which show individual traces rather than generalised process models.

Existing documentation solutions fall into two categories. The first is manually maintained documentation, which requires developer updates and tends to become stale. The second is auto-generated documentation derived from code artefacts, such as OpenAPI specifications [8]. These document individual service interfaces, but not the inter-service behaviour observed at runtime. Process models generated from traces offer a third option: documentation that reflects actual runtime behaviour. Unlike static API specifications, these models capture the ordering, concurrency, and conditional branching of service interactions as they occur. Unlike manually maintained diagrams, they update when the underlying traces change.

Approach

This is among the use cases best matched to trace-derived process models. A mined hierarchical model can be presented as an interactive, navigable view in which engineers browse services and operations and expand subprocess references to move through the hierarchy at the level of granularity they need, rather than reading a single monolithic diagram. Because the models are derived from traces, such documentation reflects how the system actually behaves at runtime, and it updates whenever the underlying traces change, addressing precisely the staleness that participants reported. The familiarity participants expressed with BPMN also suggests that rendering the models in a notation close to BPMN would lower the barrier to adopting them as documentation.

3.2.5 Architecture alerting

Two participants discussed detecting unexpected architectural changes. One described a test-driven approach: model the intended architecture first, then verify after deployment that the live system matches it. This could prevent unplanned dependencies or misconfigurations.

Related work

Zdun et al. proposed constraints and metrics for assessing conformance to microservice decomposition patterns, enabling semi-automatic detection of deviations from intended designs [41]. Ntentos et al. extended this with coupling-related metrics and demonstrated their applicability on open-source systems [32]. Both approaches operate on static architecture models rather than runtime traces. They detect structural deviations in the design, not behavioural deviations during execution. In the infrastructure domain, tools such as Terraform's drift detection monitor whether deployed resources match their Infrastructure-as-Code definitions [23]. These operate at the infrastructure level and do not capture service interaction patterns.

To our knowledge, no existing work combines runtime trace analysis with process mining to detect architectural drift through conformance checking in microservice systems.

Approach

This use case maps naturally onto conformance checking, a standard process mining technique. A model mined from traces collected during a period the team considers representative can serve as a reference for the intended architecture; newly collected traces can then be replayed against it to identify deviations such as unexpected service calls, missing components, or altered execution orderings. The test-driven variant one participant described, modelling the intended architecture first and verifying the live system against it afterwards, fits the same mechanism with a hand-authored rather than a mined reference. Running such a check periodically, or in a CI/CD pipeline, would enable early detection of architectural drift, which is exactly the gap left open by the static-analysis and infrastructure-level tools discussed above.

3.2.6 Business KPIs

Three participants discussed linking system behaviour to business-level key performance indicators (KPIs). They wanted to measure whether architectures met performance and

reliability goals. However, they noted that OpenTelemetry often samples traces, which may limit accuracy.

Related work

Process enhancement, as described by Van der Aalst [39], overlays additional attributes from event logs onto mined process models to derive performance insights. Commercial tools such as Celonis [3] provide KPI dashboards that track cycle time, throughput, and SLA compliance from event logs.

In the microservice domain, KPI monitoring is typically handled by observability platforms such as Prometheus, Grafana, and Datadog [34, 6, 18]. These aggregate metrics like response time, error rate, and availability per service. They operate on metrics rather than process models, so they report on individual service performance but not on how performance distributes across the end-to-end process flow.

Combining process mining with microservice traces for KPI analysis is largely unexplored. Span timestamps can be used to compute per-activity latencies, error attributes to calculate failure rates per endpoint, and trace-level aggregations to approximate SLA compliance.

Approach

Layered process models could serve as a foundation for KPI analysis by overlaying timing and error information from the underlying traces onto the mined models. Because each span records start and end timestamps, per-activity latencies can be computed and mapped onto the corresponding parts of the process model, revealing where time is spent across the end-to-end flow. Similarly, error attributes on spans can be aggregated per operation to identify failure-prone components. The fundamental limitation is the one the participants themselves raised: when tracing is sampled, as it commonly is, aggregate metrics such as throughput or SLA compliance computed from the retained traces may not be representative. Full KPI dashboards and process enhancement therefore go beyond what trace-derived models provide directly, but the models and their associated trace data are a reasonable basis on which such analyses could be built.

3.3 Conclusion

The interviews revealed six use cases for trace-derived process models. They differ in how well process mining serves them. Some, such as API deprecation, are better served by dedicated tools that inspect individual traces, since a generalised model may omit rare occurrences. Others, particularly living documentation, architecture alerting, and architecture analysis, benefit directly from the generalised, hierarchical process models that the layered approach produces, because what they need is precisely a faithful overview of how services interact rather than a record of any single request.

Underlying these six use cases, two broader needs emerged across most interviews, shown in Figure 3.1. First, participants lacked structural visibility into their systems. They could inspect individual traces or read outdated documentation, but had no way to see the general patterns of service interaction. Second, several participants wanted automated verification: confirming that the system behaves as intended rather than relying on manual inspection.

These two needs, visibility and verification, are the strongest and most broadly shared signal in the interviews, and they are also the two that generalised process models are best

placed to meet: a navigable view of mined models speaks to visibility, and conformance checking against a reference architecture speaks to verification. They are the findings this chapter contributes to the rest of the thesis.

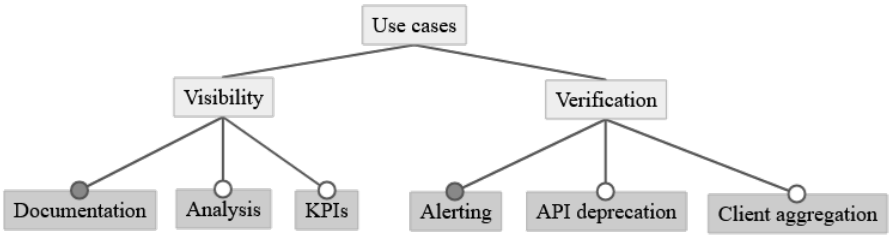

FIGURE 3.1: Feature model of the six use cases, grouped by the two needs that emerged across the interviews: structural visibility and automated verification. Use cases implemented in the framework are marked.

Chapter 4

Framework

This chapter describes the framework developed to implement the layered process mining approach presented in Chapter 2. The framework, named Ariadne, is a Python-based library with a command-line tool that takes OpenTelemetry traces as input, mines hierarchical process models per service operation, and presents the results through an interactive web-based visualisation. Its design is shaped by the two needs that emerged from the expert interviews in Chapter 3: structural visibility into how services interact, and automated verification that a system behaves as intended. These map onto two of the framework's modules, the visualisation interface (Section 4.4) and the conformance checker (Section 4.3), respectively. Each functionality is a separate command in the application, and commands can be chained together. Figure 4.1 gives an overview of Ariadne's pipeline and commands. The chapter is structured based on the framework's processing pipeline: trace collection, process discovery, conformance checking, and visualisation.

4.1 Trace collection

The framework supports two main methods of ingestion:

The first method uses the `collect` command, which runs an OTLP (the OpenTelemetry Protocol)-compatible gRPC endpoint that receives traces directly from an OpenTelemetry Collector. A command-line argument sets a maximum duration; the collector stops when this limit is reached or when interrupted from the terminal. Output is written in CSV, which is Ariadne's canonical trace format, with one row per span. CSV was chosen because aggregating by trace ID means having to keep uncompleted traces in memory, as the OpenTelemetry Collector does not necessarily report all spans of a trace at once.

The second method, using the `import` command, is to convert spans from a different source into the same format as `collect` produces. In this way, other trace solutions can also be used. These tracing solutions often have a slightly different format for their traces. `import` bridges these differences so that downstream commands can assume a single input shape. For example, the Elastic APM tracing format is supported via this conversion mechanism. Instead of having a start and end time like OpenTelemetry, it has a start time and a duration column. Other tracing solutions use completely different formats; traces exported from the Jaeger UI, for example, are saved as a JSON file which groups the spans per trace, as was the case for both evaluation datasets in Chapter 5.

Listing 4.1 shows a typical end-to-end session: collecting (or importing) traces, mining the per-operation models, and producing the interactive viewer.

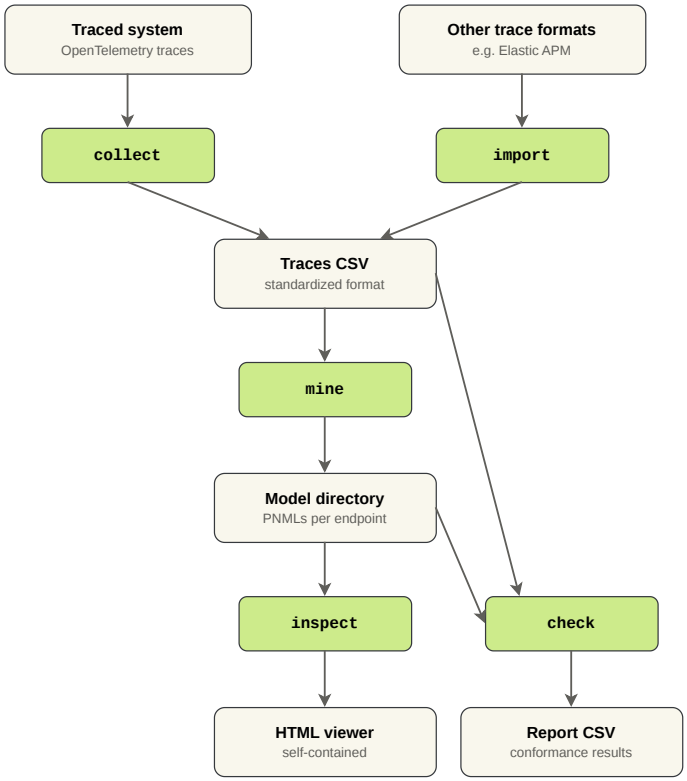

FIGURE 4.1: Overview of Ariadne's pipeline and commands. Commands in the CLI are marked as green.

Listing 4.1 A typical Ariadne pipeline, from trace collection to interactive viewer

```
# Option A: collect live traces from an OpenTelemetry Collector for 5 minutes
$ ariadne collect --duration 5m --out traces.csv

# Option B: import an existing Jaeger JSON export into the same CSV format
$ ariadne import --format jaeger traces/ --out traces.csv

# Mine one hierarchical model per service operation
$ ariadne mine traces.csv --out models/

# Render the models as a single self-contained, interactive HTML file
$ ariadne inspect models/ --out report.html
```

4.2 Process discovery

Once traces have been collected, the framework partitions the span data into per-operation sublogs as described in Section 2.4. An operation is defined as a combination of the service name and the operation name. Whatever its origin, every span has by this point been normalised into one row of Ariadne's canonical CSV, whose schema is shown in listing 4.2: each row carries the identifiers needed to reconstruct the trace tree (`trace_id`, `span_id`, `parent_span_id`), the operation label, and the two timestamps from which lifecycle events are derived.

Listing 4.2 Ariadne's canonical trace CSV, one row per span

```
trace_id,span_id,parent_span_id,service,operation,start_time,end_time
d113a..,9f2c..,,frontend,POST /api/checkout,1700..023000,1700..046870
d113a..,5a71..,9f2c..,checkout,PlaceOrder,1700..024427,1700..046120
d113a..,1bd0..,5a71..,cart,GetCart,1700..024503,1700..025335
```

For each operation o, the framework collects all spans whose parent span belongs to o. These child spans form the sublog for o. Each span in the sublog generates two lifecycle events — a *start* event derived from the span's start timestamp and a *complete* event derived from its end timestamp — as described in Section 2.5.2.

Each sublog is structured as a pandas [10] dataframe. The case identifier is the concatenation of the trace ID and the parent span ID (Section 2.4.2); the activity label is the child operation name.

Each operation is mined independently using the Inductive Miner with lifecycle extensions. The lifecycle-aware variant of the Inductive Miner is not implemented by PM4Py, at the time of writing. Ariadne therefore includes its own implementation, following the description by Leemans [24]. This implementation is shipped as a plugin (Section 4.5) so that future PM4Py releases or alternative implementations can replace it without modifying the core framework.

The output of this stage is a collection of Petri nets (WF-nets), one per operation. Each net models the behaviour observed for that operation's direct children. Activities in these nets that correspond to calls to other operations serve as subprocess references. Each net is saved as a PNML file. PNML, the Petri Net Markup Language [15], is a standardised XML-based interchange format for Petri nets: it records the places, transitions, arcs, and initial marking of a net as XML elements, so that a net can be written by one tool and read back by another without loss. Using a standard format here means the mined models are not locked into Ariadne; any PNML-aware process mining tool can open them. The PNML files are written into a directory structure that mirrors the service hierarchy for easier navigation. Listing 4.3 shows the structure of the output directory of `mine`.

Listing 4.3 Model directory produced by `mine`: one PNML, dot per operation, grouped by service

```
models
 cast-info-service
    MongoInsertCastInfo
       model.dot
       model.pnml
    WriteCastInfo
        model.dot
        model.pnml
 compose-review-service
    UploadMovieId
       model.dot
       model.pnml
    ...
 ...
 user-review-service
    MongoFindUser
       model.dot
       model.pnml
    ...
```

```
user-service
    MmcGetUserId
       model.dot
       model.pnml
    ...
```

4.3 Conformance checking

The framework includes a conformance checking module that verifies whether observed trace behaviour matches a previously mined reference. This module addresses the *architecture alerting* use case and, more broadly, the need for automated verification identified in Chapter 3: it lets a reference architecture be checked against live behaviour rather than inspected by hand. The module is exposed through the `check` command, which takes a directory of mined PNML files and a set of traces in Ariadne's CSV format, and produces a CSV report describing how well the traces conform to the reference.

The reference model is itself a model directory previously produced by the `mine` command. A typical workflow is to mine a model from traces collected during a period the team considers representative of intended behaviour, store the resulting directory under version control, and then check newly collected traces against it as part of regression testing or periodic monitoring.

Conformance is checked per operation, mirroring the layered structure of the mined models. For each operation o for which a reference net exists, the framework constructs the sublog of o from the input traces using the same partitioning logic as the mining stage (Section 4.2). The sublog is then doubled by creating separate activities for the *start* and *complete* events, such that the *start* events are also checked: the PM4Py implementation only uses *complete* events by default. Similarly, each operation Petri net is changed by replacing each transition with a *start* and *complete* transition, with a *pending* place between the two. In this way, both the *start* and *complete* events can be conformance checked, instead of only the *complete* event.

Each trace in the sublog is then replayed against the reference net for o using token-based replay [39], and the fitness is added to the report. Because the per-operation checks share no state, they are executed concurrently to reduce the total runtime on systems with many operations.

The output is a CSV report with one row per operation, containing the operation name, the number of traces replayed, and the percentage of those traces that conformed. This report format was kept intentionally minimal: it gives a single comparable number per operation that can be diffed against previous runs or used as a pass/fail signal in a CI/CD pipeline. Listing 4.4 shows an example of the CSV report.

Listing 4.4 Example CSV output produced by `check`

```
service_name,operation_name,total_traces,fitness
movie-id-service,RegisterMovieId,1552,1
movie-info-service,WriteMovieInfo,1336,1
compose-review-service,UploadText,132,1
compose-review-service,UploadUniqueId,12,1
...
```

Richer diagnostics, such as per-trace deviation reports or visual highlighting of non-conforming transitions in the model, are left as future extensions.

4.4 Visualisation

The visualisation module addresses the need for structural visibility, and in particular the *living documentation* use case, identified in Chapter 3: it turns the mined models into documentation that engineers can navigate and that reflects how the system actually behaves. It presents the mined models through an interactive web-based interface, exposed through the `inspect` command. The command produces a single self-contained HTML file with all model data embedded inline, which can be opened in any browser without a server. Figure 4.2 shows the interface applied to the MediaMicroservices benchmark used in Chapter 5.

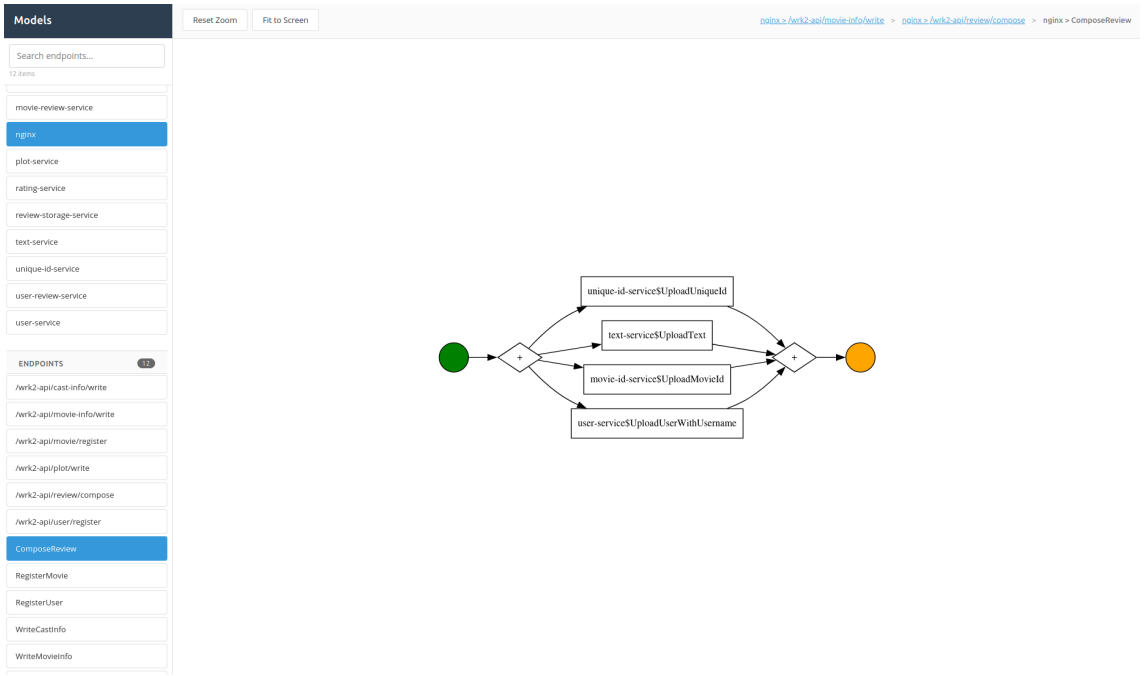

FIGURE 4.2: Ariadne's interactive visualisation, applied to the MediaMicroservices benchmark. Each operation is rendered as a BPMN diagram; activities labelled with another operation's name are clickable and open that operation's model. The sidebar (left) supports text search across all services and operations.

Models are rendered as BPMN diagrams rather than Petri nets, in line with the familiarity expressed by all interview participants in Chapter 3. The Business Process Model and Notation (BPMN) [2] is a graphical notation for describing business processes, standardised by the Object Management Group. Whereas Petri nets are a low-level formalism aimed at analysis, BPMN is designed for human readability and is widely used by business analysts, software architects, and engineers to model and communicate processes. BPMN diagrams are composed of activities, events, and gateways connected by sequence flows, with conventions for parallelism, choice, looping, and asynchronous communication. Translations between BPMN and Petri nets are well-established [19], allowing processes mined as Petri nets to be rendered as BPMN for end-user-facing tools. The translation from Petri net to BPMN happens during mining: each call to the mining plugin returns both a PNML representation of the sound WF-net and a DOT representation of the corresponding BPMN diagram.

The hierarchical composition described in Section 2.4 is realised at the visualisation layer rather than in the mined models themselves. Each per-operation BPMN is rendered

independently, and activities whose label matches a known operation are clickable: clicking opens that operation's model. This UI-level convention preserves the navigability of the layered approach without requiring a single composed model file. A sidebar lists all services and operations and supports text search, allowing engineers to jump to a specific operation directly.

4.5 Plugin system

Ariadne provides a plugin system for its process mining parts, such that different implementations can be used interchangeably. These plugins can be defined through Python entry points [5]. Ariadne scans for packages in the Python environment with these entry points, and lists them to the user. In this way, new implementations of mining algorithms do not need changes to the Ariadne source code, and can be installed using the Python package manager. Table 4.1 contains an overview of the types of plugins currently supported, and their input and output.

Name	Entry point	Input	Output
Mining algorithm	ariadne.mining_algorithms	Traces	OperationModel
Conformance checker	ariadne.conformance_checkers	OperationModel, Traces	ConformanceReport

TABLE 4.1: Overview of plugin types in Ariadne

The output of the mining algorithm is an OperationModel, which is available in the core package of Ariadne and contains two fields: a PNML of the petri net, and a DOT representation of the model. Listing 4.5 shows how a mining algorithm is registered as a plugin through an entry point and the shape of the `OperationModel` it returns.

Listing 4.5 Registering a mining-algorithm plugin via a Python entry point

```
# pyproject.toml -- declares the plugin to Ariadne
[project.entry-points."ariadne.mining_algorithms"]
pm4py-imlc = "ariadne_pm4py.mining:PM4PyMiningAlgorithm"
```

The implementation of Ariadne in this thesis provides two plugins, one for each type, in a separate Python package with an AGPL-3.0 license. The mining algorithm plugin, PM4PyMiningAlgorithm, can use any inductive miner implemented in PM4Py, plus the inductive miner lifecycle (IMLC) implemented for this thesis. The conformance checker plugin, PM4PyConformanceChecker, uses the token-based replay implementation in PM4Py.

4.6 Implementation limitations

This section lists the limitations of the framework as currently implemented. These are properties of Ariadne itself, distinct from the more general caveats of trace-based analysis discussed in Section 4.7.

4.6.1 Synchronous communication only

The framework currently models only *synchronous* service interactions. In synchronous communication, the caller issues a request and blocks until the callee returns a response: the caller and callee are active at the same time, and the caller's own work is suspended until

the call completes. A direct HTTP or gRPC request between two services is the typical example. In *asynchronous* communication, by contrast, the two parties are decoupled in time: a producer hands a message to an intermediary, most commonly a message queue, and continues immediately, without waiting; a consumer picks the message up and processes it at some later point, potentially seconds later and on a different machine. The producer and consumer are never necessarily active simultaneously, and the producer does not wait for, or even observe, the consumer's work.

This distinction matters because the layered approach depends on the synchronous case. In OpenTelemetry's data model, a child span represents work performed on behalf of its parent, and the parent does not complete until all of its children complete. This parent–child relationship is the structural assumption on which the layered mining approach in Chapter 2 rests: sublogs are partitioned by parent operation, and child spans become the events of that operation's sublog.

Asynchronous communication via message queues breaks this assumption. When a producer publishes a message to a queue and a consumer processes it some time later, the consumer's work is not part of the producer's subtree. In OpenTelemetry, this relationship is represented through span links rather than parent–child references [9], precisely because the consuming process lies outside the producing process from a process-modelling perspective. Ariadne does not currently follow span links and therefore does not produce models that span the producer-consumer boundary. Asynchronous parts of a system are mined as if they were independent processes, which is correct in isolation but loses the end-to-end view across queues.

4.6.2 Clock skew beyond the parent–child case

Section 2.5.3 describes how Ariadne handles the specific clock-skew case in which a child span's recorded start time precedes its parent's: the parent is excluded from the sublog and its lifecycle events are inserted synthetically. This handles the most common failure mode but does not eliminate the broader issue. Two sibling spans recorded on different machines may also have inconsistent timestamps, and lifecycle-aware mining relies on those timestamps to determine concurrency. The consequence is a risk of misclassified concurrency for the affected operation. If skew makes two genuinely sequential siblings appear to overlap, the miner can infer parallelism that the system never exhibits. The error distorts only the model of the parent operation whose children were mistimed. Ariadne accepts the timestamps as recorded; pre-processing for clock alignment is out of scope.

4.7 Caveats and recommendations

The previous section listed limitations of Ariadne itself. This section discusses caveats that apply to any process mining analysis of microservice traces, regardless of the framework used, together with recommendations for how teams can mitigate them when adopting Ariadne.

4.7.1 Sampling bias

The mined models reflect the traces that reach the framework, not the traces that the system produces. When tracing is sampled, as is common in production deployments to control storage and processing costs, the sublog for each operation is a subset of the operation's actual invocations. If sampling is uniform and the sample size is sufficient, the

resulting models are still representative; if it is biased, the models will be biased in the same way.

It is common to probabilistically sample on normal traces, but always sample traces with errors. If these traces are included in the input to Ariadne, the model mined will include the error paths as well. This is intended, because if a service errors, fail-safe mechanisms of other services are also part of the operations. In the current version of Ariadne, this imbalance does not matter, as the standard Inductive Miner is insensitive to the frequency distribution of traces in the log. Other mining algorithms, such as the Inductive Miner - Infrequent [27], can filter out less frequent trace paths. So in these cases, sampling bias is important to consider.

Tail-based sampling on error-status spans is preferable to head-based sampling for trace mining, since it captures rarer behaviours.

4.7.2 Tracing completeness

Ariadne models the system as it is traced, not as it actually exists. If a service in the system is uninstrumented — meaning it has no tracing mechanism — it will not appear in the spans emitted by its callers as a child operation, and its outgoing calls to downstream services will not be linked to any incoming request. The framework will still produce a model from the available data, but that model will reflect a system in which the uninstrumented service has been silently excluded.

The recommendation is therefore that all services participating in a flow under analysis should be instrumented with OpenTelemetry. In practice this is rarely fully achieved: legacy services, third-party components, and infrastructure such as databases and message brokers are commonly partially instrumented or not instrumented at all. Engineers using Ariadne should treat the mined models as a view of the instrumented subsystem rather than of the system as a whole, and should interpret apparent gaps in the models in light of which components are known to be uninstrumented. OpenTelemetry's auto-instrumentation libraries cover common frameworks (Spring Boot, FastAPI, Express); using these as a baseline gives the broadest coverage with the least manual work.

4.8 Conclusion

Ariadne is a Python-based library and command-line tool implementing the layered process mining approach from Chapter 2. Its pipeline is exposed as five commands:

- `collect` and `import` ingest traces into a single canonical CSV format,
- `mine` partitions the spans into per-operation sublogs and produces a sound WF-net per operation,
- `check` replays new traces against a previously-mined reference model and reports per-operation conformance,
- and `inspect` produces a self-contained HTML viewer that renders the models as interactive BPMN diagrams.

A plugin system isolates the mining and conformance algorithms behind stable interfaces, so that alternative implementations can be swapped in without modifying the core framework. The framework's design reflects the two themes that emerged from the expert interviews in Chapter 3. The visualisation module addresses structural visibility, giving

engineers a navigable view of how services actually interact at runtime rather than how documentation says they do. The conformance checker addresses automated verification, allowing a reference architecture mined from a known-good trace set to act as a regression test against architectural drift in later traces. Both are first-class commands in the pipeline rather than ad-hoc analyses, and both can be driven from a CI/CD pipeline. Two classes of caveat apply to any deployment. The framework itself is limited to synchronous request-response flows and accepts span timestamps as recorded beyond the specific parent-child case it corrects for. Independently of Ariadne, any trace-based analysis is bounded by what is sampled and what is instrumented. These shape the interpretation of the results in the next chapter, where Ariadne is evaluated against a benchmark with full instrumentation and against a much larger production trace dataset.

Chapter 5

Evaluation

This chapter evaluates Ariadne along two dimensions, each addressing a different concern. Section 5.1 addresses correctness: does the framework produce a model that faithfully describes a system whose intended behaviour is known? Section 5.2 addresses real-world performance: does the framework remain practical when applied to production traces at scale, and what engineering work was required to get there? Both evaluations also describe development processes carried out alongside the framework itself. The MediaMicroservices benchmark served as the primary correctness target throughout development, and the Uber dataset surfaced performance bottlenecks that drove substantial optimisation work. Both are therefore presented not only as evaluation results but also as accounts of how the implementation reached its current state.

A note on scope before continuing: The framework was also exploratively applied to an internal Info Support project during the course of this work, with the aim of testing the approach against a closed-source production system. Findings from that exploration informed the limitations and design decisions reported below, but it is not presented as a separate evaluation case due to time constraints.

5.1 Correctness

To evaluate the correctness of the layered mining approach, we apply the framework to the MediaMicroservices application from the DeathStarBench benchmark suite [22]. This system is well-suited for correctness evaluation because its source code is fully available, allowing the mined models to be compared directly against the behaviour described in the implementation. MediaMicroservices served as the correctness target throughout the development of Ariadne. Mining runs on this benchmark were inspected after every substantial change to the framework. Discrepancies between the mined models and the expected behaviour drove iterative refinement of the mining pipeline, the lifecycle handling (Section 2.5.2), and the parent–child timing decision (Section 2.5.3). The evaluation reported here therefore describes the end-state of that development: a configuration in which the mined model matches the structure of the application as written.

5.1.1 Setup

MediaMicroservices simulates a movie content platform consisting of 14 services covering movie and cast information, user reviews, ratings, and supporting infrastructure such as a plot service and a pagination service. The services communicate synchronously over Thrift RPC [1], and the system is deployed via Docker Compose using the images provided by

the benchmark authors. An NGINX frontend exposes a small number of HTTP endpoints, which route requests to the corresponding Thrift services. Figure 5.1 gives an overview of the services of the application and how they communicate. The application is instrumented by the benchmark authors with OpenTracing, a predecessor of OpenTelemetry, and collects traces using Jaeger. We export the traces from the Jaeger instance and construct a CSV of traces conforming to Ariadne's canonical format described in Section 4.1. Load is generated from two sources. First, the benchmark's initialisation scripts populate the system: `register_users.sh`, `register_movies.sh`, and `write_movie_info.py` issue requests against the `RegisterUser`, `RegisterMovie`, `WriteMovieInfo`, and `WriteCastInfo` endpoints. Second, the `compose-review.lua` workload, executed via wrk2 [37], drives continuous traffic against the `ComposeReview` endpoint. wrk2 was run for 5 minutes with 8 threads totalling 29908 requests for the mining dataset, and 28710 for the conformance dataset. Together, these two phases produce traces covering all system operations.

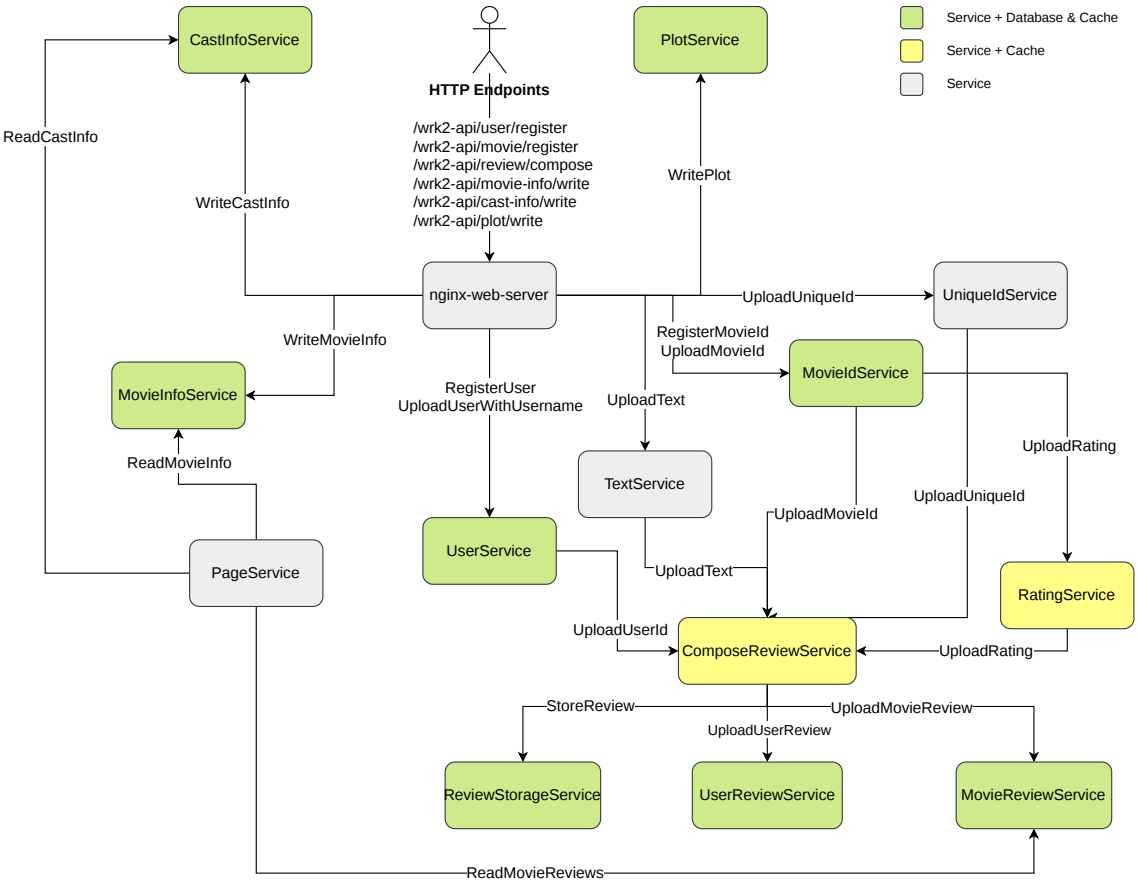

FIGURE 5.1: Overview of the MediaMicroservices application. Communication between services is marked with an arrow from the calling service to the callee.

5.1.2 Verification of the `ComposeReview` endpoint

Of the endpoints tested by the workload, the correctness evaluation focuses on `ComposeReview`. The registration and initialisation endpoints perform only a small number of downstream calls in simple sequence and offer little structural behaviour to verify against. `ComposeReview`, by contrast, is the endpoint with genuine large-scale service interaction: it coordinates calls across multiple services, exhibits both concurrency and a data-dependent branch, and is

therefore the endpoint whose mined model is substantive enough that structural correctness is a meaningful question.

Expected behaviour. According to the application source code, a `ComposeReview` request is handled in two steps. In the first step, the NGINX frontend spawns four concurrent calls, under a single parent span, to four independent services: `UserService`, `TextService`, `MovieIdService`, and `UniqueIdService`. Each of these services performs its work and reports its result back to `ComposeReviewService` through an upload call. `MovieIdService` additionally calls `RatingService`, which likewise reports its result back to `ComposeReviewService`. In total, therefore, five upload calls report into `ComposeReviewService`, each incrementing an internal per-request counter. The second step is triggered by whichever upload call is the last to complete. Rather than a designated coordinator initiating the next step, the request is advanced by whichever service happens to arrive last. When the internal counter reaches five, the thread handling that last upload issues the second step: it calls `UserReviewService`, `MovieReviewService`, and `ReviewStorageService` concurrently and waits on their completion. The remaining four upload calls perform no further work.

Mined model. The model mined by the framework for the `ComposeReview` endpoint is consistent with this description. At the top level, the endpoint is mined as a parallel composition of the first four service calls: `UserService`, `TextService`, `MovieIdService`, and `UniqueIdService`. This correctly reflects that these are independent and dispatched concurrently rather than sequentially ordered. The data-dependent branch is captured in the models of the upload operations.

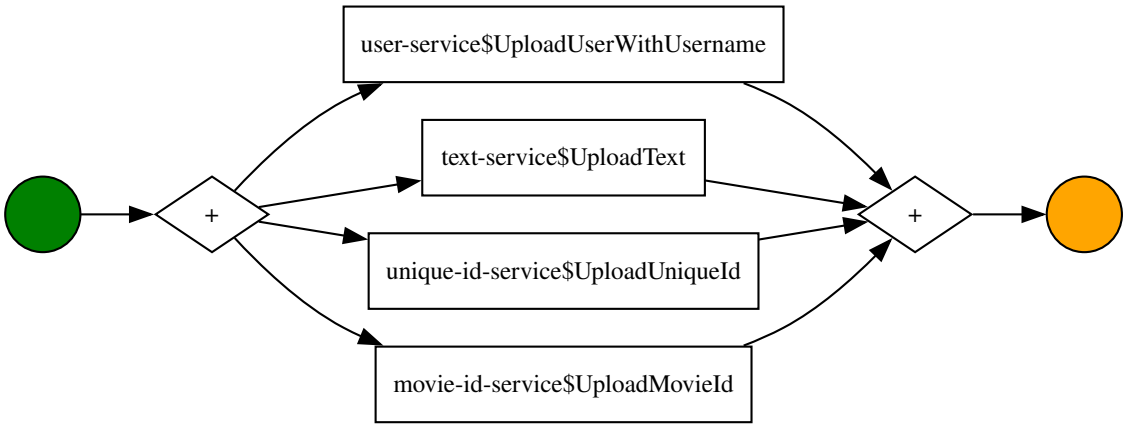

FIGURE 5.2: Mined model of the `ComposeReview` endpoint, as rendered by Ariadne's visualisation. The four first-step service calls appear as a parallel split. Each call is a subprocess reference that can be expanded; the model of the `UploadUserId` subprocess is shown in Figure 5.3.

Because any of the five upload calls may be the one that completes last, each upload operation is mined as an exclusive choice between two alternatives: a silent transition, representing the case in which the upload was not the fifth to complete and therefore performs no further work; and a parallel composition of the three second-step services, representing the case in which the upload was the fifth and triggered the remainder of the process. The mining aggregates over many traces in which different uploads win this race,

and the resulting choice construct correctly represents that any one of them may carry the continuation.

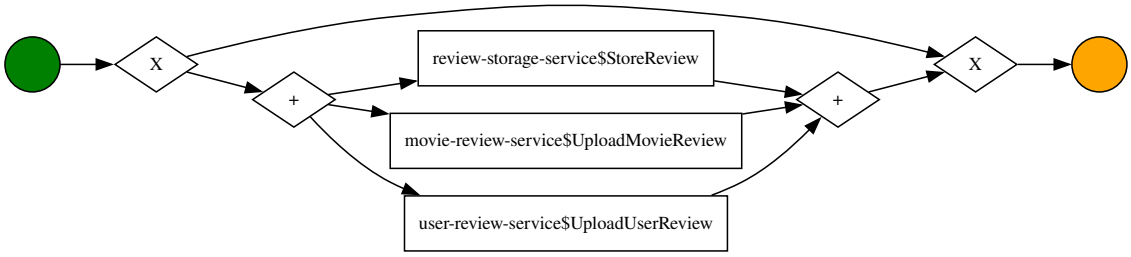

FIGURE 5.3: Mined model of the `UploadUserId` operation, one of the five upload operations of `ComposeReview`. The model is an exclusive choice between a silent transition (the upload was not the last to complete) and a parallel split into the three second-step services (the upload was the last to complete, and triggered the continuation). The other four upload operations are mined with the same structure.

This structure matches the behaviour described in the source code: a parallel split over the first four services, followed in each upload operation by a choice between completing silently and fanning out into the three second-step services.

Concurrency and lifecycle-aware mining. The correct identification of the two parallel blocks is a direct consequence of the lifecycle-aware mining described in Section 2.5.2. The concurrently dispatched calls frequently differ in duration, and a mining approach based on the completion order of spans alone would infer spurious sequential dependencies between them. By deriving explicit start and complete events from span timestamps, the framework distinguishes genuine sequential dependencies from incidental completion ordering, and produces a model that faithfully represents the concurrency present in the endpoint's behaviour. The structural match was verified by inspecting the mined `ComposeReview` model, and the models of its upload operations, against the application source code; the framework's current state is the result of that iteration.

5.1.3 Quality of the mined models

The structural inspection of `ComposeReview` establishes that the layered approach produces a faithful model on the most behaviourally rich endpoint of the benchmark. To complement that account with a quantitative view across the whole system, we computed the four standard process mining quality dimensions [39] – fitness, precision, simplicity, and generalisation – for every per-operation model mined from the MediaMicroservices traces. Fitness and precision are computed using token-based replay; simplicity is the inverse-arc-degree metric; and generalisation is the standard activity-frequency-based estimator, all as implemented in PM4Py [11]. The mined models cover 30 operations across 14 services.

Table 5.1 summarises the distribution of each metric across these 30 operations. Fitness is 1.0 for every operation, as expected: the lifecycle-aware Inductive Miner used here does not filter infrequent behaviour, so it produces models that replay every trace in the training log by construction. Precision is 1.0 for 21 of 30 operations and has mean 0.92, indicating that the mined models are not only fitting but also tight around the observed behaviour for most of the system. Simplicity and generalisation are similarly high (mean 0.94 and 0.90 respectively), with simplicity at 1.0 for 16 of 30 operations.

	Fitness	Precision	Simplicity	Generalisation
Mean	1.00	0.92	0.94	0.87
Min	1.00	0.44	0.85	0.69
Operations at 1.0	30/30	21/30	16/30	0/30

TABLE 5.1: Distribution of model quality metrics across the 30 per-operation models mined from MediaMicroservices.

For completeness, the mined model of every one of the operations in the application is reproduced in Appendix A, so that the structural account given here for `ComposeReview` can be checked against the full set rather than the single endpoint examined in detail.

5.2 Real-world performance

The correctness evaluation establishes that the framework produces a faithful model on a small benchmark. This section addresses a complementary question: does the framework remain practical when applied to production traces at a scale much larger than any benchmark? We evaluate this using the publicly released trace dataset from Uber's CRISP project [42], which contains 100,000 traces sampled from Uber's production microservice architecture, totalling approximately 2 GB of trace data and spanning 18,386 distinct operations across 1,314 services. This evaluation also describes a substantial part of the framework's optimisation history. The initial mining implementation took over three hours on the full dataset, which is too slow for any practical workflow. Two rounds of optimisation brought the runtime down to its current value of 5 minutes and 38 seconds, with a standard deviation of 3.2 seconds, a speedup of roughly 37× over the initial implementation. The remainder of this section describes the benchmark setup, the optimisations applied, and the resulting runtimes.

5.2.1 Setup

The dataset is loaded as a CSV of trace spans in Ariadne's canonical format, derived from the Jaeger exports released by the CRISP authors. Mining is invoked via the `mine` command, with the resulting per-operation models written to a folder containing the service/operation directory hierarchy described in Section 4.2. All benchmarks were run on a workstation with an Intel Core i7-8700K processor (6 cores, 12 threads), 48 GB of RAM, and an M.2 NVMe SSD, running Linux with no other desktop applications active. The runtime of the final configuration was measured over ten runs after three runs of warmup; the earlier configurations were measured informally during development and are reported as approximate figures only.

5.2.2 Optimisations

The mining pipeline was profiled at three points during this evaluation, each surfacing a distinct bottleneck. The optimisations applied in response are described below.

Initial implementation. Mining the full dataset took over three hours. Profiling showed that the dominant cost was the repeated computation of the directly-follows graph (DFG) and the concurrency graph. Both graphs are recomputed on every sublog produced by

the recursive partitioning of the Inductive Miner, and both were initially implemented in a straightforward but inefficient way. The concurrency-graph computation maintained the set of currently active activities as a list, performing a linear scan to remove each completed activity, giving an effective per-trace cost quadratic in the number of events. The DFG computation performed a forward scan from each completed event, exhibiting the same worst-case behaviour.

Graph computation optimisations. The active-activities structure was replaced with a hash-based representation, reducing activity removal from a linear scan to a constant-time operation. In addition, the decoding of lifecycle events, the prefix check and string slice performed for every event, was extracted out of the inner loops by preprocessing each trace once into a sequence of typed events, rather than re-parsing the same strings at every level of the recursion. Together, these changes reduced total mining time on the full dataset from over three hours to approximately 30 minutes.

Parallel mining. The remaining bottleneck was that the per-operation mining tasks ran sequentially, despite being independent: each operation's sublog shares no state with any other. This was a bottleneck for two compounding reasons: the framework was not exploiting available cores, and the wide variance in per-operation mining times meant the slowest operation alone determined the critical path. The implementation was extended to dispatch the per-operation mining tasks to a pool of worker processes sized to the number of available CPU cores, with results collected and written to the model archive by the main process. This reduced total mining time to its final value of 338 seconds.

5.2.3 Results

In its final configuration, the framework mines the full CRISP dataset in a mean time of 338.1 seconds, with a standard deviation of 3.2 seconds over ten runs, and observed runtimes ranging from 334.0 to 343.4 seconds. Table 5.2 summarises the runtime across the three configurations.

Configuration	Runtime	Speedup
Initial implementation	> 3.5 hours	—
Graph computation optimised	≈ 30 minutes	$\approx 7\times$
Parallel mining (final)	338.1 ± 3.2 s	$\approx 37\times$

TABLE 5.2: Mining runtime on the full CRISP dataset across the three configurations. Speedup is relative to the initial implementation. The first two figures are approximate, measured informally during development; the final figure is the mean over ten runs.

The final runtime of under six minutes demonstrates that the framework can process production-scale trace data on a single desktop workstation within a timeframe suitable for interactive use or routine batch jobs, without requiring server-grade hardware. The two optimisation steps together account for a speedup of roughly 37× over the initial implementation, with the graph-computation optimisations and the parallelisation each contributing a comparable share.

5.2.4 Quality of the mined models at scale

The runtime measurements above establish that the framework scales to production trace volumes, but say nothing about the models it produces at that scale. To complement them, we computed the four standard quality dimensions – fitness, precision, simplicity, and generalisation – on every per-operation model mined from the CRISP dataset, using the same PM4Py implementations as in Section 5.1.3. 3,186 of the 18,386 operations are leaf operations; meaning that they do not have any child spans of their own and their corresponding model is one single silent transition from start to end. Because of their triviality, they have been excluded from the metrics. For 63 of the 15,200 remaining models, PM4Py's prefix-based precision computation exhausted available memory on the workstation described in Section 5.2.1; precision is reported for the remaining 15,137. The other three metrics were computed for all 15,200 non-leaf models. Here there was no split between training and validation datasets, as some operations have very few traces to train a model with.

Table 5.3 summarises the distribution of each metric. The most informative figures are that fitness and precision are 1.0 for a significant number of the operations. Additionally, the fitness, precision and simplicity means do not differ much from the MediaMicroservices evaluations, indicating that these are properties that generalise across systems. Also interesting are the outliers at the minimums, especially a fitness of 0.00.

	Fitness	Precision†	Simplicity	Generalisation
Mean	0.95	0.95	0.96	0.60
Min	0.00	0.03	0.50	0.00
Operations at 0.0	374	0	0	0
Operations at 1.0	14,014/15,200	12,927/15,137	11,962/15,200	0/15,200

TABLE 5.3: Distribution of model quality metrics across the 15,200 per-operation models mined from the CRISP dataset. †Precision is reported over the 15,137 models for which PM4Py's prefix-based estimator completed within the workstation's memory budget.

Two observations require separate discussion.

Imperfect fitness on the training log. The Inductive Miner used here does not filter infrequent behaviour and should therefore produce models that replay the training log with perfect fitness by construction; this is the property exploited in Section 5.1.3 for MediaMicroservices, where every operation indeed reports fitness 1.0. On the CRISP dataset, 1186 operations (7.8%) report fitness below 1.0, with 374 reporting fitness exactly 0.0.

For the large majority of these operations the cause is the same: the operation's sublog contains at least one trace in which an activity overlaps a concurrent instance of itself, that is, a fragment of the form $\langle a_s, a_s, a_c, a_c \rangle$ in which a second instance of an activity starts before the first has completed. This pattern arises naturally in microservice traces whenever an operation issues several identical child calls in parallel, such as a concurrent fan-out of requests to the same downstream operation. The lifecycle-aware Inductive Miner represents each activity as a single pair of start and complete transitions and produces a *collapsed* process model in which, by construction, no activity can be concurrent with itself [28]. A trace that exhibits self-overlap of an activity therefore cannot be replayed

against the discovered net, and token-based replay reports a fitness below 1.0. This mechanism accounts for 1175 of the 1186 sub-perfect operations, and for 372 of the 374 operations with fitness exactly 0.0. The phenomenon is intrinsic to lifecycle-aware discovery rather than specific to Ariadne: it follows directly from representing each activity once, which is precisely what makes the systematic start/complete distinction tractable.

For the remaining eleven sub-perfect operations, two of which have fitness 0.0, the cause could not be established within the time available for this evaluation. In one of them the miner produced a model containing a loop even though every activity in the operation's sublog occurs exactly once, behaviour that no fitting model should require; this points to a likely defect in the discovery implementation rather than a property of the input data, and is left as an item for future investigation.

Precision skips as a tooling limitation. The 63 operations for which precision could not be computed all have very large sublogs and produce nets whose prefix language exceeds the memory available on the evaluation workstation. This is a limitation of the specific precision algorithm used by PM4Py, not of the layered approach. For the purposes of this evaluation, the 63 skipped operations represent 0.41% of the total and do not affect the qualitative conclusions drawn from the remaining 15,137.

5.3 Conclusion

The two evaluations together demonstrate that the framework is correct, performant, and produces models of consistently high quality across systems of very different scale. The MediaMicroservices benchmark shows that the layered approach produces a model of the `ComposeReview` endpoint that matches the source code: both the concurrency among its downstream calls and the sequential composition step that follows are captured correctly. Across all 30 per-operation models mined from this benchmark, fitness is perfect by construction; precision is 1.0 for 21 of 30 operations. The Uber CRISP dataset shows that the framework scales to production-grade trace volumes on a single desktop workstation, once two specific optimisations have been applied. Quality holds up at that scale: 84.4% of the 15,200 per-operation models mined from CRISP achieve perfect fitness and precision simultaneously.

Chapter 6

Concluding remarks

This final chapter draws the thesis to a close. Section 6.1 revisits the research question and summarises the contributions made in answering it. Section 6.2 reflects on what the results mean and on the validity of the methods used to obtain them. Section 6.3 situates the limitations of the work, and Section 6.4 outlines directions for future research.

6.1 Conclusion

Modern microservice systems are difficult to understand because the behaviour of a single user request is distributed across many independently deployed services, and is nowhere visible in full. Distributed tracing, and OpenTelemetry in particular, records that behaviour one request at a time, but a collection of individual traces does not by itself reveal the general patterns of interaction that characterise a system. Process mining offers a route from individual observations to a structured model, but as established in Chapter 1, prior applications of process mining to microservice traces flattened traces into peer-level event logs, discarding the parent–child relationships between spans and producing models that are shallow and structurally incorrect. Against that background, this thesis set out to answer the following question:

> How can process mining techniques be applied to OpenTelemetry traces to support engineers and architects in understanding, validating, and improving microservice systems?

The thesis answers this question through four contributions:

- **A layered process mining approach** (Chapter 2) that mines one process model per service operation, using only the direct children of that operation's spans, and represents calls to child operations as expandable subprocess references. This preserves the hierarchical structure of OpenTelemetry traces end to end, yielding models that are structurally faithful to the systems they describe and that remain navigable regardless of system size.
- **An interview study** (Chapter 3) with six engineers and architects at Info Support, which surfaced six concrete use cases for trace-derived process models and, more importantly, two recurring needs: structural visibility into service interaction, and automated verification that a system behaves as intended.
- **Ariadne** (Chapter 4), a framework that implements the layered approach as a command-line tool. Ariadne collects or converts traces into a canonical format,

mines a hierarchical model per operation using a lifecycle-aware Inductive Miner, checks conformance through token-based replay, and renders the result as an interactive BPMN visualisation. A plugin system isolates the mining components from the core, so that additional and different mining implementations can be used without changing the core of the framework.

- **An evaluation** (Chapter 5) on systems using distributed tracing, establishing the correctness of the mined models against the MediaMicroservices benchmark and the performance of the framework on Uber's publicly released CRISP production trace dataset.

Taken together, these contributions show that process mining can support engineers and architects in understanding, validating, and improving microservice systems, provided that the hierarchical structure of traces is preserved rather than discarded. The layered approach addresses *understanding* by producing models that are both structurally faithful and navigable: an engineer can begin at a top-level operation and expand only the subprocesses relevant to the question at hand, rather than confronting a single unreadable global model. It addresses *validation* through conformance checking: by treating a previously mined model as a reference, the framework can detect when newly observed behaviour deviates from a known-good architecture, without requiring anyone to author a reference model by hand. It contributes to *improvement* more modestly: the mined models, enriched with the timing and error data already present in spans, provide a foundation on which performance and reliability analyses can be built, although such analyses are themselves left as future work. The evaluation supports these claims on two fronts. The correctness evaluation showed that the model mined for the `ComposeReview` endpoint of MediaMicroservices is structurally consistent with the application's source code, correctly capturing both the concurrent fan-out across four services and the data-dependent branch by which any of five upload calls may carry the continuation of the request. This match was only possible because of lifecycle-aware mining: a miner relying on span completion order alone would have inferred spurious sequential dependencies between the concurrently dispatched calls. The performance evaluation showed that, after two rounds of optimisation, Ariadne mines the full CRISP dataset in under six minutes on a single desktop workstation. The dataset comprises 100,000 traces spanning 18,386 operations across 1,314 services; six minutes is well within reach of interactive use or routine batch jobs.

6.2 Discussion

Use structural data. The central finding of this thesis is that the parent–child structure of a trace is not incidental metadata to be discarded before analysis, but is itself part of what the trace means. A child span represents work performed on behalf of its parent; the parent does not complete until its children do. Flattening a trace removes exactly that relationship, and with it the subprocess semantics that distinguish a checkout operation from the database calls and payment steps it delegates to. In this light the layered approach is less an optimisation than a correction: it mines the object that a trace actually is, a tree of nested operations, rather than a flattened projection of it. The structural errors of flat mining are not a matter of degree but of kind, and no amount of post-processing of a flat model recovers a hierarchy that was destroyed before mining began.

The role of lifecycle information. A second, narrower finding concerns concurrency. Microservice operations routinely dispatch several children in parallel, and those children

seldom complete in a fixed order. A miner that orders activities by completion timestamp alone will read incidental completion ordering as a sequential dependency. By deriving explicit start and complete events from span timestamps and mining with the lifecycle-aware Inductive Miner, the framework distinguishes the two. The correctly identified parallel split in the `ComposeReview` model is the concrete payoff of this decision. The lesson generalises: when traced operations have non-trivial duration and can overlap, lifecycle information is essential rather than optional.

Per-operation models are context-free. Applying the framework to the OpenTelemetry Demo surfaced a property of the layered approach worth naming explicitly. The demo's synthetic load generator labels its top-level spans by user action — `user_add_to_cart`, `user_checkout`, `user_browse_products` — but its child spans carry the more generic operation name `POST`, which resolves further downstream to `router frontend egress`. Because the layered approach mines one model per operation across all traces, the model of `router frontend egress` is the union of every endpoint the router serves: an engineer navigating downward from `user_add_to_cart` reaches a generic operation whose model contains `POST /api/cart`, `POST /api/checkout`, `GET /api/products`, and every other endpoint as concurrent alternatives. The cart-specific path is not selected, and not selectable, by the upstream context. This is a property of the approach rather than a bug in the framework: per-operation mining deliberately trades upstream context for generalisation across invocations of the same operation, which is what gives the layered approach its scalability and its small, navigable per-operation models. The trade-off is favourable when operation names are distinct and unfavourable when operation names are generic, as happens in this case. Producing per-context variants of an operation's model when operation names cannot be made specific is a more substantial extension and is discussed as future work in Section 6.4.4.

Fitting into existing workflows. Ariadne was designed to slot into the way engineering teams already work rather than to require new infrastructure around it. The `collect` command can run inside a CI/CD pipeline; the conformance report reduces to a single comparable number per operation that can serve as a pass/fail gate; and the visualisation is a single self-contained HTML file that opens in any browser without a server. Earlier attempts to make a more encompassing application, with a separate frontend and monitoring built in, stalled because it could not be generally adopted. Microservice architectures are very different and creating a one-size-fits-all application was deemed too big of a project. After realizing this, we pivoted to a design where adopting the framework should be a small, incremental step for a team that already produces traces, not a separate observability project in its own right.

Threats to validity. Several characteristics of the work bound the strength of its conclusions. The interview study involved six participants at a single company; as already noted in Chapter 3, this supports no quantitative or representativeness claims, and the six use cases should be read as indicative of problems engineers encounter rather than as an exhaustive or weighted survey. The evaluation establishes correctness and performance but not utility in practice: the correctness assessment focused on `ComposeReview`, the one endpoint with substantial structural behaviour to verify against, and the structural match was confirmed by source-code inspection by the author rather than independently. The performance figures were measured on a single workstation, and while the optimisation history is informative, the absolute runtimes are specific to that hardware. As discussed

in Section 5.2.4, 374 operations on the CRISP dataset reported fitness of 0.0. While most of these operations can be accounted for because of self-concurrent activities, a handful of models still are unaccounted for. This is probably a bug in the implementation. Most importantly, no part of the evaluation tested whether engineers working on their own systems find the mined models genuinely helpful for the understanding and verification tasks the framework targets.

6.3 Limitations

The framework's limitations and the broader caveats of trace-based analysis are set out in detail in Chapter 4; they are recalled here only at the level needed to frame the future work that follows. The most consequential limitation is that Ariadne models synchronous communication only. The layered approach rests on the parent–child relation between spans, and asynchronous communication through message queues does not satisfy it: a consumer's work is not part of the producer's subtree, and OpenTelemetry represents the relationship through span links rather than parent–child references. Ariadne does not currently follow span links, so asynchronous portions of a system are mined as if they were independent processes. Beyond this, the framework handles only the most common clock-skew case, in which a child span starts before its parent; inconsistent timestamps between sibling spans recorded on different machines are accepted as given. Two further caveats apply to any trace-based analysis regardless of framework: sampled tracing means the mined models reflect the traces that were retained rather than every invocation, and incomplete instrumentation means the models describe the instrumented subsystem rather than the whole system. Teams adopting Ariadne should interpret its output accordingly.

6.4 Future work

6.4.1 Asynchronous communication

Extending the approach to asynchronous communication is the clearest priority, since message-queue interaction is widespread in real microservice systems and is precisely what the current framework cannot model end to end. The natural starting point is OpenTelemetry's span links, which connect a consumer span to the producer span that enqueued the message it processed. Following these links would, in principle, allow producer and consumer subtraces to be stitched into a single end-to-end model. The difficulty is that a span link is not a containment relation. The consumer does not execute inside the producer, so it cannot simply be treated as a child in the existing layered scheme. Addressing this properly is likely to require extending the model rather than the partitioning logic alone: for example, coupling the producer's enqueue with the consumer's dequeue. BPMN already provides message throw and catch events for exactly this kind of decoupled communication, which makes it a promising target notation for the asynchronous case once the underlying mining model has been extended.

6.4.2 Framework extensions

Several extensions follow directly from use cases raised in the interviews but left unimplemented. The Business KPIs use case points to process enhancement: the per-activity latencies and per-operation error rates already derivable from span attributes could be overlaid onto the mined models, turning them from purely structural descriptions into a

basis for performance and reliability analysis. The Architecture analysis use case points to computing architecture-level metrics, such as service coupling, over the mined models. Conformance checking could also be made considerably more informative: the current report gives a single conformance percentage per operation, whereas per-trace deviation reports and visual highlighting of non-conforming transitions in the BPMN view would make it far easier to diagnose why a check failed.

A broader direction is to exploit other process mining techniques beyond discovery and conformance, which the plugin system of Section 4.5 is designed to accommodate. The most immediately valuable is performance-oriented analysis, in particular *bottleneck finding*: because every span carries start and complete timestamps, replaying a log on its mined model while measuring the waiting time accumulated in each place would locate, per operation, where requests spend their time and which downstream call dominates an endpoint's latency. This turns the structural models into a diagnostic tool for the performance questions engineers actually ask, and connects directly to the Business KPIs theme above. Other techniques worth exploring within the same plugin interface include variant analysis to compare the dominant execution paths of an operation across time windows or deployments (the basis of the client-aggregation use case), and frequency-aware discovery such as the Inductive Miner – Infrequent [27] to surface the common case separately from rare error paths. Because each of these consumes the same per-operation sublogs the framework already builds, they can be added as alternative analysis plugins rather than as changes to the mining core.

6.4.3 Evaluation with practitioners

The evaluation in this thesis establishes that the framework is correct and that it scales, but not that the models it produces are useful to the engineers they are intended for. A study in which practitioners use Ariadne on their own systems and report whether the mined models help understanding and verification in practice would improve the evaluation. Broadening the range of evaluated systems, in particular to industrial systems beyond the benchmark and public dataset used here, would also strengthen confidence that the layered approach generalises across architectures and instrumentation styles.

6.4.4 Context-aware layered mining

The layered approach as developed in this thesis is context-free per operation: the model of an operation aggregates every invocation of that operation, regardless of the upstream call chain that led to it. As discussed in Section 6.2, this is a deliberate trade-off but becomes costly when operation names are generic enough that semantically distinct activities share the same operation label. A context-aware variant of the approach would partition an operation's invocations by upstream context. This would produce, for instance, separate models of `router egress` per top-level user action, at the cost of the per-operation simplicity that motivates the layered design.

6.5 Closing remarks

This thesis began from a simple observation: that microservice traces are trees, and that treating them as flat event logs throws away the structure that makes them meaningful. The layered process mining approach, and the Ariadne framework that implements it, are an attempt to take that structure seriously from collection through to visualisation. The result is a framework that integrates cleanly into OpenTelemetry-traced architectures

and produces structurally faithful, navigable models of service interaction, subject to the limitations on synchronous communication and instrumentation completeness set out in Section 6.3. It is a demonstration that the idea is sound, feasible, and useful; and a foundation on which the asynchronous case, richer analyses, and a proper study of practical utility can be built. Ariadne is ready to use and available under an MIT-license under the following URL: `https://github.com/Michael-Janssen-dev/Ariadne/`, with mining plugins using PM4PY available under AGPL 3.0.

Appendix A

MediaMicroservices models

This appendix contains all models from the MediaMicroservices benchmark mined using Ariadne. The chapter is split per service.

Trivial cases. Leaf models, which appear at the leaves of the span tree and have no executions with child models, always model to the trivial case of a single silent transition. For the sake of space efficiency, this model will be shown once, and the endpoints which are leaf models will state so in the subsection.

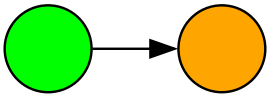

FIGURE A.1: Leaf model with only a single transition, which is silent.

A.1 cast-info-service

MongoInsertCastInfo is trivial.

A.1.1 WriteCastInfo

FIGURE A.2: cast-info-service – WriteCastInfo

A.2 compose-review-service

A.2.1 UploadMovieId

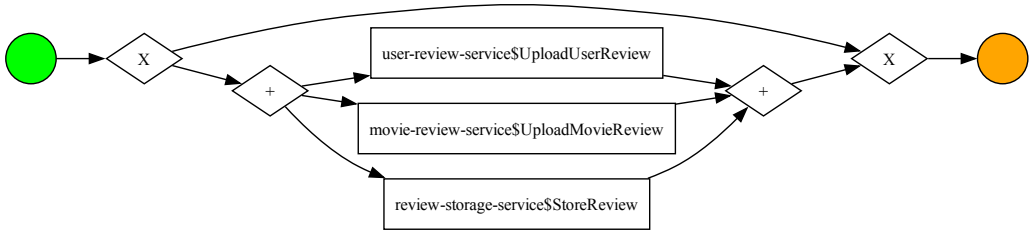

FIGURE A.3: compose-review-service – UploadMovieId

A.2.2 UploadRating

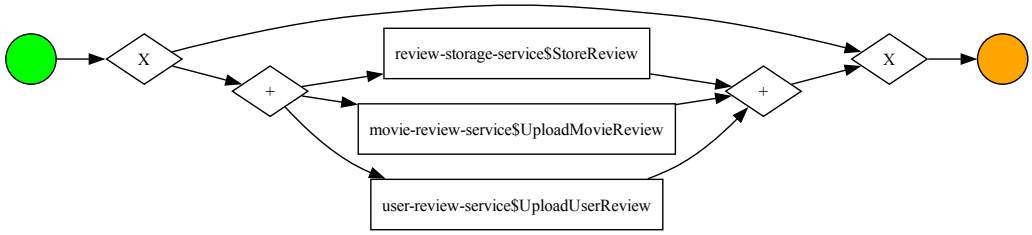

FIGURE A.4: compose-review-service – UploadRating

A.2.3 UploadText

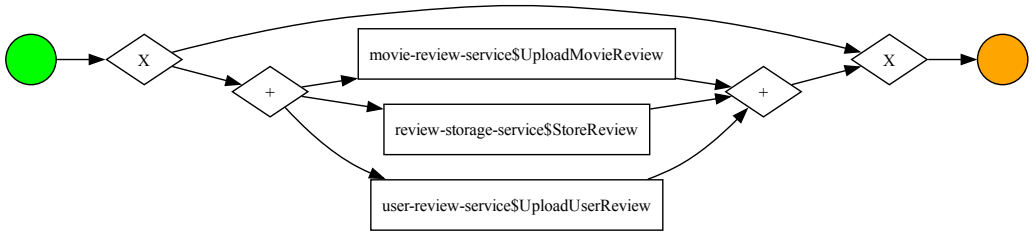

FIGURE A.5: compose-review-service – UploadText

A.2.4 UploadUniqueId

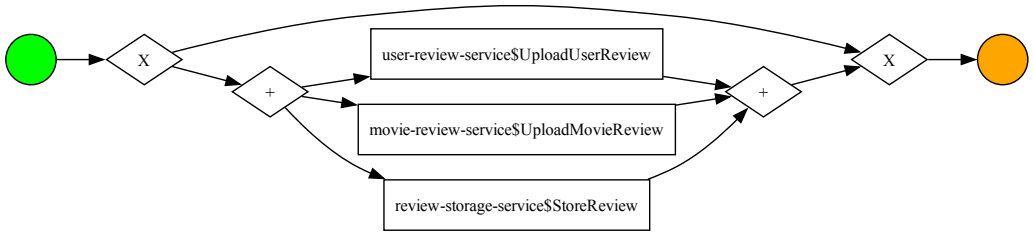

FIGURE A.6: compose-review-service – UploadUniqueId

A.2.5 UploadUserId

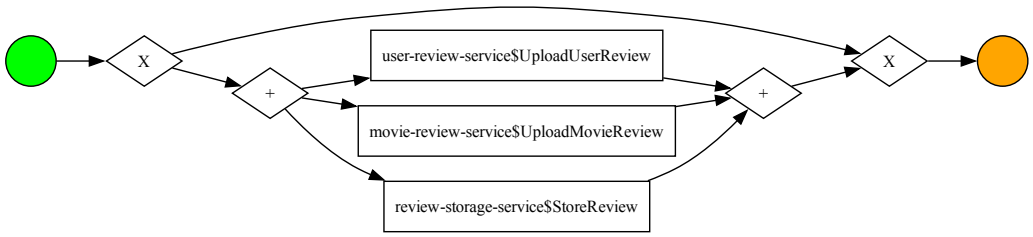

FIGURE A.7: compose-review-service – UploadUserId

A.3 movie-id-service

MmcGetMovieId, MmcSetMovieId, MongoFindMovieId, MongoFindMovie and MongoInsertMovie are trivial.

A.3.1 RegisterMovieId

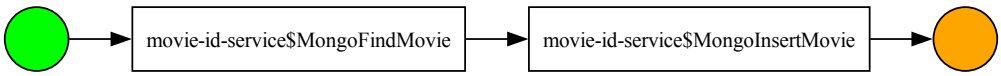

FIGURE A.8: movie-id-service – RegisterMovieId

A.3.2 UploadMovieId

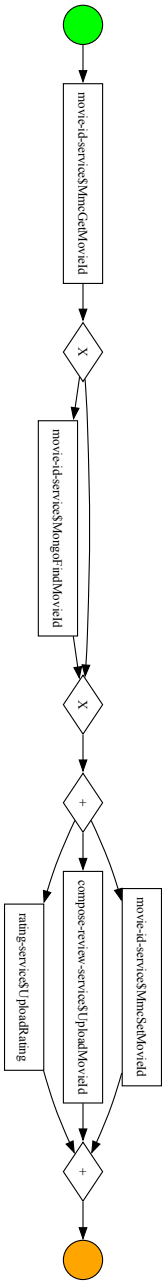

FIGURE A.9: movie-id-service – UploadMovieId

A.4 movie-info-service

MongoInsertMovieInfo is trivial.

A.4.1 WriteMovieInfo

FIGURE A.10: movie-info-service – WriteMovieInfo

A.5 movie-review-service

MongoFindMovie, MongoInsert, MongoUpdate and RedisUpdate are trivial.

A.5.1 UploadMovieReview

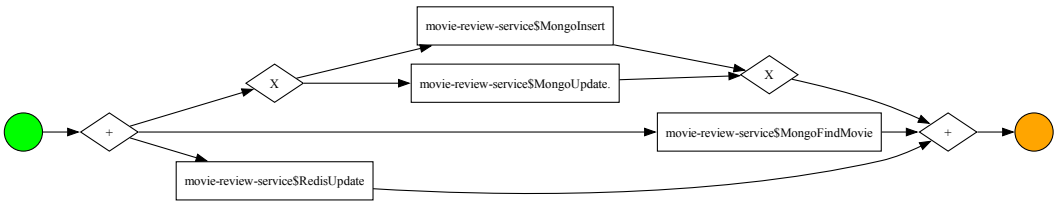

FIGURE A.11: movie-review-service – UploadMovieReview

A.6 nginx

A.6.1 ComposeReview

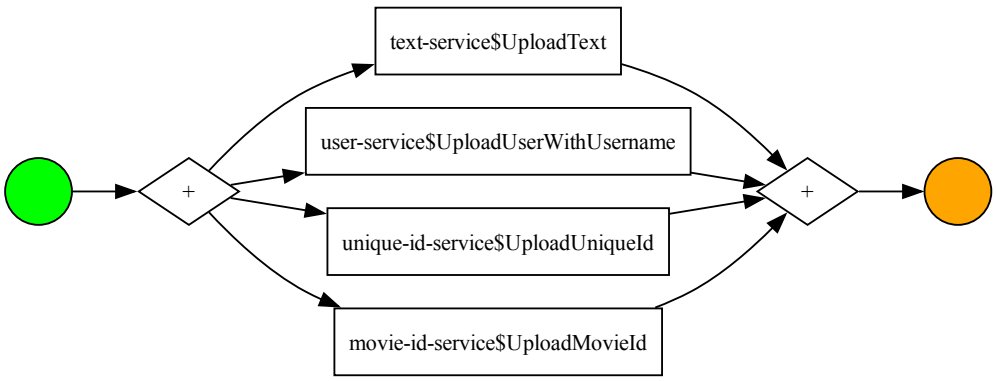

FIGURE A.12: nginx – ComposeReview

A.6.2 RegisterMovie

FIGURE A.13: nginx – RegisterMovie

A.6.3 RegisterUser

FIGURE A.14: nginx – RegisterUser

A.6.4 WriteCastInfo

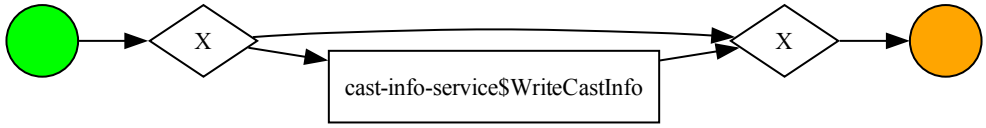

FIGURE A.15: nginx – WriteCastInfo

A.6.5 WriteMovieInfo

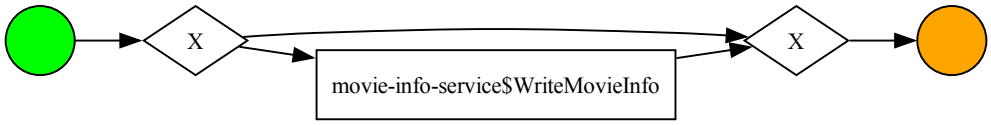

FIGURE A.16: nginx – WriteMovieInfo

A.6.6 WritePlot

FIGURE A.17: nginx – WritePlot

A.6.7 /wrk2-api/cast-info/write

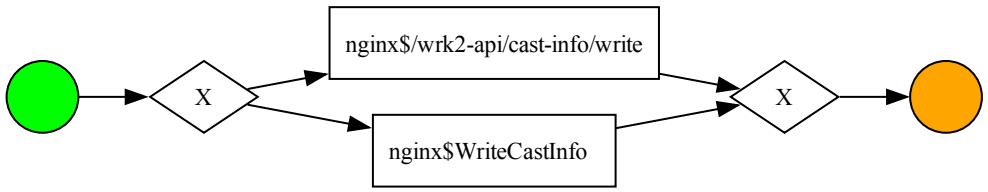

FIGURE A.18: nginx – /wrk2-api/cast-info/write

A.6.8 /wrk2-api/movie-info/write

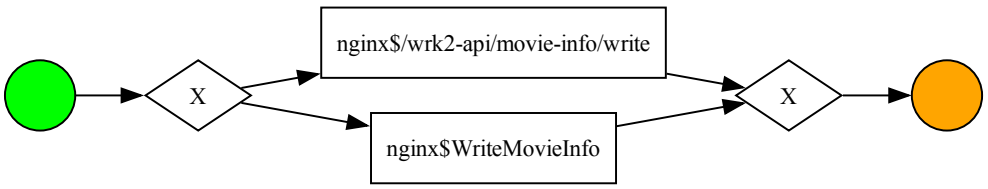

FIGURE A.19: nginx – /wrk2-api/movie-info/write

A.6.9 /wrk2-api/movie/register

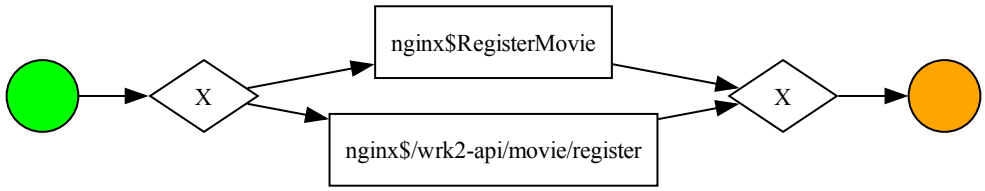

FIGURE A.20: nginx – /wrk2-api/movie/register

A.6.10 /wrk2-api/plot/write

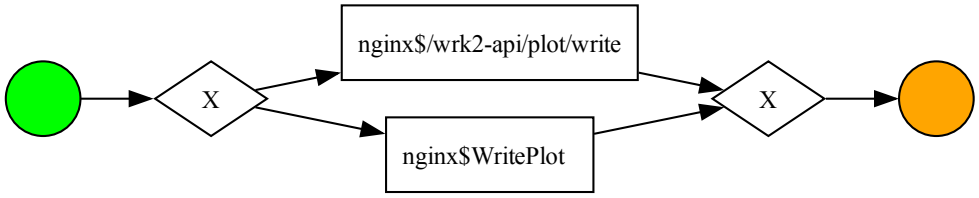

FIGURE A.21: nginx – /wrk2-api/plot/write

A.6.11 /wrk2-api/review/compose

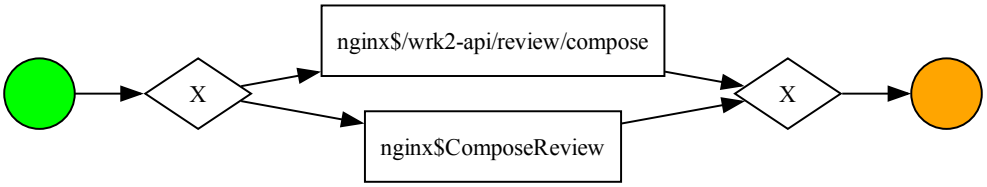

FIGURE A.22: nginx – /wrk2-api/review/compose

A.6.12 /wrk2-api/user/register

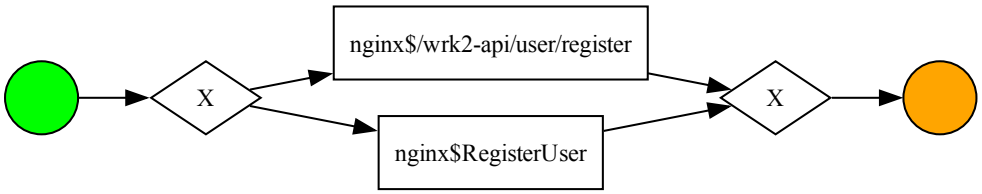

FIGURE A.23: nginx – /wrk2-api/user/register

A.7 plot-service

MongoInsertPlot is trivial.

A.7.1 WritePlot

FIGURE A.24: plot-service – WritePlot

A.8 rating-service

RedisInsert is trivial.

A.8.1 UploadRating

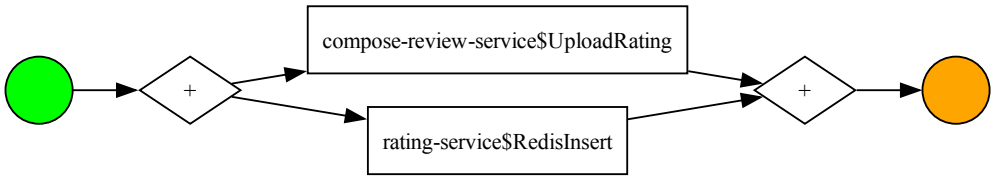

FIGURE A.25: rating-service – UploadRating

A.9 review-storage-service

MongoInsertReview is trivial.

A.9.1 StoreReview

FIGURE A.26: review-storage-service – StoreReview

A.10 text-service

A.10.1 UploadText

FIGURE A.27: text-service – UploadText

A.11 unique-id-service

A.11.1 UploadUniqueId

FIGURE A.28: unique-id-service – UploadUniqueId

A.12 user-review-service

MongoFindUser, MongoInsert, MongoUpdate and RedisUpdate are trivial.

A.12.1 UploadUserReview

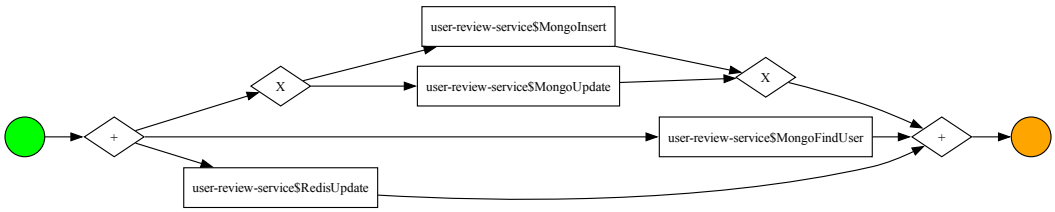

FIGURE A.29: user-review-service – UploadUserReview

A.13 user-service

MmcGetUserId, MmcSetUserId, MongoFindUser and MongoInsertUser are trivial.

A.13.1 RegisterUser

FIGURE A.30: user-service – RegisterUser

A.13.2 UploadUserWithUsername

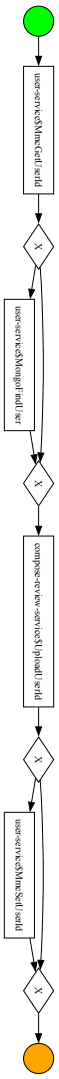

FIGURE A.31: user-service – UploadUserWithUsername

Bibliography

[1] Apache Thrift - Home. URL: https://thrift.apache.org/.

[2] BPMN Specification - Business Process Model and Notation. URL: https://www.bpmn.org/.

[3] Celonis | Process Intelligence and Process Mining. URL: https://www.celonis.com/.

[4] Demo Architecture | OpenTelemetry. URL: https://opentelemetry.io/docs/demo/architecture/.

[5] Entry points specification - Python Packaging User Guide. URL: https://packaging.python.org/en/latest/specifications/entry-points/.

[6] Full-stack observability for the agentic era | Grafana Labs. URL: https://grafana.com/index/.

[7] Jaeger: open source, distributed tracing platform. URL: https://www.jaegertracing.io/.

[8] OpenAPI Specification v3.2.0. URL: https://spec.openapis.org/oas/v3.2.0.html.

[9] OpenTelemetry. URL: https://opentelemetry.io/.

[10] pandas - Python Data Analysis Library. URL: https://pandas.pydata.org/.

[11] PM4PY - Process Intelligence Solutions GmbH (P.I.S.). URL: https://processintelligence.solutions/pm4py.

[12] ProM Tools. URL: https://promtools.org/.

[13] IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. *IEEE Std 1849-2016*, pages 1–50, November 2016. doi:10.1109/IEEESTD.2016.7740858.

[14] Omolola A. Adeoye-Olatunde and Nicole L. Olenik. Research and scholarly methods: Semi-structured interviews. *JACCP: Journal of the American College of Clinical Pharmacy*, 4(10):1358–1367, 2021. doi:10.1002/jac5.1441.

[15] Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In Wil M. P. van der Aalst and Eike Best, editors, *Applications and Theory of Petri Nets 2003*, pages 483–505, Berlin, Heidelberg, 2003. Springer. doi:10.1007/3-540-44919-1_31.

[16] Leif Bonorden and André van Hoorn. Detecting Usage of Deprecated Web APIs via Tracing. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pages 23–33, June 2024. doi:10.1109/ICSA59870.2024.00011.

[17] Théo Coulin, Maxence Detante, William Mouchère, and Fabio Petrillo. Software Architecture Metrics: a literature review, January 2019. doi:10.48550/arXiv.1901.09050.

[18] Datadog. Cloud Monitoring as a Service. URL: https://www.datadoghq.com/.

[19] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281–1294, November 2008. doi:10.1016/j.infsof.2008.02.006.

[20] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, Today, and Tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-67425-4_12.

[21] Martin Fowler and James Lewis. Microservices, 2014. URL: https://martinfowler.com/articles/microservices.html.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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, pages 3–18, New York, NY, USA, April 2019. Association for Computing Machinery. doi:10.1145/3297858.3304013.

[23] HashiCorp. Drift Detection for Terraform Cloud. URL: https://www.hashicorp.com/en/lp/drift-detection-for-terraform-cloud.

[24] Sander J. J. Leemans. *Robust Process Mining with Guarantees: Process Discovery, Conformance Checking and Enhancement*, volume 440 of *Lecture Notes in Business Information Processing*. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-030-96655-3.

[25] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency*, pages 311–329, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-38697-8_17.

[26] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In Niels Lohmann, Minseok Song, and Petia Wohed, editors, *Business Process Management Workshops*, pages 66–78, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-06257-0_6.

[27] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering Block-Structured Process Models from Incomplete Event Logs. In Gianfranco Ciardo and Ekkart Kindler, editors, *Application and Theory of Petri Nets and Concurrency*, pages 91–110, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07734-5_6.

[28] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Using Life Cycle Information in Process Discovery. In Manfred Reichert and Hajo A. Reijers, editors, *Business Process Management Workshops*, pages 204–217, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-42887-1_17.

[29] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering*, 27(1):25, November 2021. doi:10.1007/s10664-021-10063-9.

[30] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989. URL: https://ieeexplore.ieee.org/abstract/document/24143, doi:10.1109/5.24143.

[31] Sam Newman. Monolith to Microservices. ISBN: 9781492047841. URL: https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/.

[32] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, pages 3–20, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58923-3_1.

[33] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Inc., April 2020.

[34] Prometheus. Prometheus - Monitoring system & time series database. URL: https://prometheus.io/.

[35] Vinay Raj and G Punnam Chander. Monitoring of Microservices Architecture based Applications using Process Mining. In *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 486–494, March 2022. doi:10.23919/INDIACom54597.2022.9763155.

[36] Souhaila Serbout and Cesare Pautasso. How Are Web APIs Versioned in Practice? A Large-Scale Empirical Study. *Journal of Web Engineering*, 23(4):465–506, June 2024. doi:10.13052/jwe1540-9589.2341.

[37] Gil Tene. giltene/wrk2. URL: https://github.com/giltene/wrk2.

[38] W. M. P. Van Der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, February 1998. doi:10.1142/S0218126698000043.

[39] Wil Van Der Aalst. *Process Mining*. Springer, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49851-4.

[40] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, September 2015. doi:10.1109/ColumbianCC.2015.7333476.

[41] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol, editors, *Service-Oriented Computing*, pages 411–429, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-69035-3_29.

[42] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. pages 655–672, 2022. URL: https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou.

[43] Chenxing Zhong, He Zhang, Chao Li, Huang Huang, and Daniel Feitosa. On measuring coupling between microservices. *Journal of Systems and Software*, 200:111670, June 2023. doi:10.1016/j.jss.2023.111670.