

TDK-dolgozat

Márton Tarnay
2024.



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Qualitative Reasoning-Based Fault Diagnosis in Distributed Systems

Author:

Márton Tarnay
Fifth Year (V.)

Advisor:

András Földvári,
PhD student

Bertalan Zoltán Péter,
PhD student

December 19, 2024

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Microservice Architecture and Observability	3
2.1 What are Microservices?	3
2.2 Observability	4
2.3 What can be observed?	4
2.3.1 Monitoring performance metrics	4
2.3.2 Logging	5
2.3.3 Distributed Tracing	7
2.4 Research Direction	7
2.5 Related Work	8
3 Logic Reasoning	10
3.1 Qualitative Reasoning	10
3.2 Answer Set Programming	11
3.3 Inductive Logic Programming	12
4 Modeling and Fault Diagnosis	15
4.1 Request Metamodel	15
4.2 Log classification	16
4.3 Quantization	17
4.4 Model extraction process	17
4.5 Model Aggregation	18
4.5.1 Operation Graph	19
4.5.2 Events and Metric Anomalies	19
4.5.3 Aggregate Model	19

4.6	Fault Diagnosis	20
4.6.1	Diagnosis Example	21
5	Rule Extraction with Inductive Learning	23
5.1	Task definition	23
5.2	Workflow	24
5.3	Rule Extraction Examples	25
5.4	Scalability	26
6	Evaluation	27
6.1	Evaluation results	28
6.1.1	Rule extraction	28
6.1.2	Rule evaluation	28
7	Conclusion	30
	Acknowledgements	32
	Bibliography	33
	Appendix	36

Kivonat

A mikroszolgáltatás architektúra elterjedése, a felhő-natív fejlesztések, és a DevOps, valamint a Site Reliability Engineering gyakorlatok megjelenése fundamentálisan átalakította a modern szoftverfejlesztés és üzemeltetés stratégiáit. A mikroszolgáltatások lazán csatolt szoftver komponensek, melyek pontosan leírt csatornákon kommunikálnak, így megengedve, hogy minden szolgáltatás önállóan funkcionálhasson, autonóm skálázódhasson, valamint kicserélődjön anélkül, hogy megzavarná a teljes rendszer működését. A monolitokkal szemben a mikroszolgáltatások alkalmazásával moduláris és skálázható architektúra alkotható meg. Azonban ezek az előnyök jelentősen megnövelt rendszerek komplexitással járnak, amely leginkább a rendszer viselkedésének megértése során válik nyilvánvalóvá, különösen üzemzavarok és hibák esetén.

E kihívás kezelésére az üzemeltetési csapatok a megfigyelésről az észlelhetőségre (observability) tértek át. Az észlelhetőségi eszközök célja, hogy értelmezhető adatokat és hasznos információkat nyújtsanak a rendszerről, ezzel segítve az incidensek kezelési idejének csökkentését és a termelékenység növelését. A szervezetek sokkal kifinomultabb hibadetektálási és lokalizációs megközelítéseket igényelnek, amelyek képesek kezelni az ilyen összetett rendszerek bonyolultságát. A pontos és értelmezhető diagnosztika jelentősen csökkentheti a mérnökök munkaterhelését.

Ez a dolgozat egy új modell-alapú diagnosztikai keretrendszert mutat be, amely egy közös metamodellel reprezentálja a multimodális adatokat. Ez a megközelítés lehetővé teszi a modellfűzést különböző információforrásokból az elosztott rendszerben az OpenTelemetry ipari szabvány felhasználásával, amely magában foglalja az elosztott nyomkövetést, eseménynaplózást és teljesítmény monitorozást. A bemutatott skálázható (span szinttől a telepítési szintig) diagnosztikai modell észleli és pontosan lokalizálja a hibákat és azok okait a hibaterjedés analízis felhasználásával. Egy Answer Set Programming (Prolog-szerű) logikai következtető motort alkalmazva az egyesített adatok elemzésére, amely lehetővé teszi a hibák azonosítását és terjedés vizsgálatát az összekapcsolt komponensek között. A kifejlesztett módszereket egy hibákkal injektált benchmark adathalmazon értékeltem.

A javasolt megközelítés hatékonyan kezeli a hibalokalizáció kihívásait a felhő-natív mikroszolgáltatás alapú rendszerekben. Logikai következtetés alkalmazásával a hibák elemzésére és terjesztésére a keretrendszer olyan megoldásokat generál, amelyek jobban érthetőek a mérnökök számára, elősegítve az átláthatóságot és további döntéshozást. A keretrendszer által nyújtott skálázható diagnosztika hasznos betekintésekkel látja el az üzemeltetési csapatokat, javítva a rendszer megbízhatóságát és a döntéshozatali folyamatokat, miközben egyidejűleg csökkenti az üzemeltetési terheket. Ezek a hozzájárulások nemcsak az incidensek kezelésének hatékonyságát növelik, hanem új megközelítést kínálva a multimodális adategyesítésre, előmozdítják az észlelhetőség területét a mikroszolgáltatás alapú rendszerekben.

Abstract

The widespread adoption of the microservice paradigm, cloud-native developments, and the adoption of DevOps and Site Reliability Engineering practices have fundamentally transformed modern software development and operational strategies. Microservices are loosely coupled software components that interact through well-defined APIs, allowing each service to function independently, scale autonomously, and be updated or replaced without disrupting the overall system architecture. Compared to monoliths, microservices enable modular and scalable architectures, allowing for greater flexibility in these large, distributed systems. However, these benefits come at the cost of significantly increased system complexity, which is most evident in understanding system behavior, especially during outages and failures.

To tackle this challenge, operations teams shifted from monitoring to observability. Observability tools aim to yield actionable insights and meaningful data about the system, which helps improve incident response times and increase productivity. Organizations require much more sophisticated fault detection and localization approaches that can handle the intricacies of these complex systems. Accurate and interpretable diagnostics can significantly decrease the workload for engineers.

This report presents a novel model-based diagnostics framework using a joint metamodel representing multimodal data. This approach allows model fusion from different information sources in the distributed system using the OpenTelemetry industry standard, which includes distributed traces, event logs, and performance monitoring. The presented scalable (from span to deployment-level) diagnostics model detects faults and localizes root causes accurately with the use of error propagation analysis. An Answer Set Programming (Prolog-like) logic reasoning engine is employed to analyze the fused data, enabling the identification and propagation of faults across interconnected components. The developed methods were evaluated on a benchmark dataset with injected faults.

The proposed approach effectively tackles the challenges of fault localization within cloud-native microservices systems. By employing logic reasoning to analyze and propagate faults, the framework generates solutions that are more comprehensible to engineers, facilitating better understanding and reasoning. The scalable diagnostics provided by this framework empower operations teams with actionable insights, significantly improving system reliability and decision-making processes while simultaneously reducing operational workloads. These contributions not only enhance the effectiveness of incident response strategies but also advance the field of observability in microservice systems with a new approach to multimodal data fusion.

Chapter 1

Introduction

While being highly scalable and flexible, microservice systems introduce significant challenges in observability and understanding system behavior due to their highly complex, decentralized, and modular design [1]. The heterogeneous nature of the architecture and constantly changing topology make traditional monitoring and fault localization techniques difficult. The reliability and availability requirements in these systems necessitate approaches that can tackle multimodal observability data and provide operators with explainable diagnostic results. This report aims to tackle two challenges [2] in the field of observability and fault diagnosis of microservice systems:

Challenge 1: Multimodal data model

Traditional methods often focus on one data source only, like logs or metrics, which can miss important information. By combining different data types, such as logs, metrics, and system topology, new approaches can provide a more comprehensive understanding of the state of the system.

Challenge 2: Explainable diagnostic results

Several current methods use complex machine learning models that can be hard for operators to understand. Making these models easier to explain can build trust and confidence in the diagnostic results, helping with better decision-making.

The report presents a novel model-based diagnostics framework for microservice systems, utilizing a joint metamodel representing multimodal data from various sources such as distributed traces, logs, and performance metrics. The framework uses OpenTelemetry, an industry standard, to collect and integrate these data sources, enabling a comprehensive understanding of system behavior. The paper leverages Answer Set Programming [3], a logic-based reasoning engine, and Inductive Learning of Answer Set Programs [4], an Inductive Logic Programming approach, to analyze this fused data, allowing for the identification of faults across interconnected microservice components. This approach effectively tackles the challenges of fault localization in microservice systems by generating explainable diagnostic results and providing actionable insights for operations teams, ultimately improving system reliability and reducing operational workload.

Report Structure

Chapter 2 presents the advantages and the challenges of the *microservice architecture*, focusing on increased system complexity and the difficulty of understanding end-to-end flows. The concept and importance of *Observability* is introduced, as well as state-of-the-art approaches and solutions for monitoring and system telemetry. Finally, it presents the problem of fault diagnosis in complex systems and current research directions.

Chapter 3 introduces the concepts of *Logic Programming*, *Qualitative Reasoning*, *Answer Set Programming* and *Inductive Logic Programming*. These fundamental approaches and techniques lay the groundwork for the contributions presented in the following chapters.

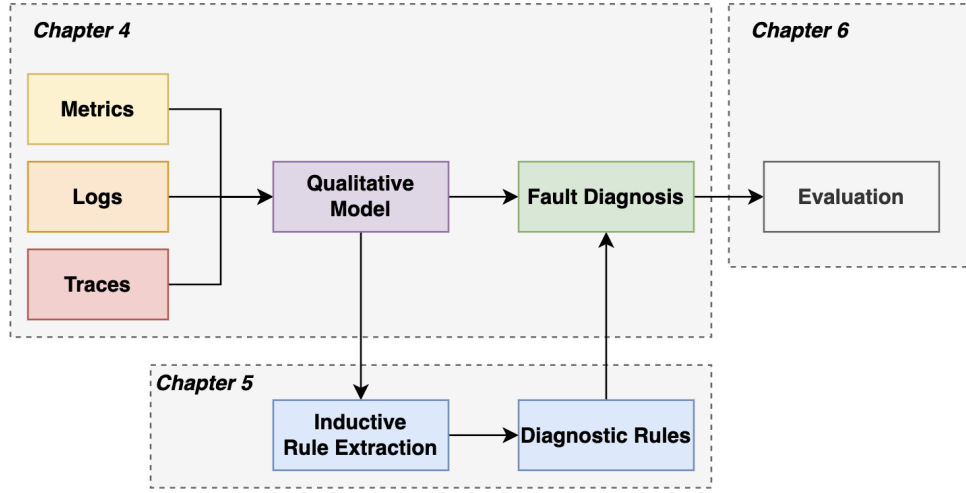


Figure 1.1: Contributions

Research Contributions are shown in Figure 1.1. The following paragraphs describe the chapters containing the proposed methodology and the developed solution.

Chapter 4 addresses *Challenge 1 (Multimodal data model)* by introducing a model-based framework for fault diagnosis in microservice systems, using a metamodel that represents multimodal observability data. This approach enables the integration of heterogeneous telemetry sources such as logs, metrics, and traces, leading to a more comprehensive understanding of system behavior.

Chapter 5 addresses *Challenge 2 (Explainable diagnostic results)* by extracting explainable diagnostic rules using Inductive Logic Programming of Answer Set Programs (ILASP). These rules aim to capture the relationships between observed anomalies and specific faults in the microservice system.

Chapter 6 uses a benchmark microservice system to evaluate the performance of the fault diagnosis framework and the power of inductive rule learning. The chapter assesses the accuracy of the developed approach to diagnose and localize faults within this system. The results are compared to several contemporary approaches as well.

Chapter 2

Microservice Architecture and Observability

2.1 What are Microservices?

Microservice architecture is a software development paradigm that has emerged as the prominent way to design rapidly evolving cloud-native applications. Microservices are small, independently deployed services that communicate over lightweight APIs and are built around business domains [5]. The simplified logical structure of microservices can be seen in Figure 2.1.

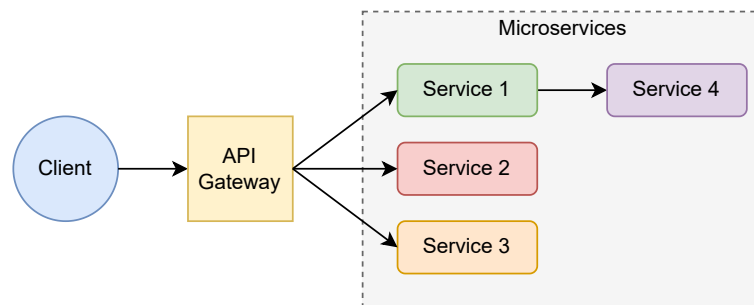


Figure 2.1: Microservice Architecture

The loose coupling of these services enables greater flexibility in development, scaling, and deployment. Services centered around business domains are developed by cross-functional teams, have end-to-end ownership of their domain, and are able to work independently with the means best suited to their needs. The decentralized, separated nature of microservices allows scaling only those services that experience high loads. Services can run on virtual machines selected for their performance needs and scale horizontally when needed, in turn optimizing infrastructure costs [6].

While microservices provide great advantages in scaling and flexibility, these advantages come at a cost. Moving from monolithic paradigms to the use of distributed systems significantly increases system complexity. As the number of services increases, it becomes more challenging to understand end-to-end flows. Technical debt, exception handling, service execution, and communication pose significant difficulties in modern microservice systems [7]. Verifying system behavior and testing also pose a challenge, as requests span across multiple services. Frequent, dynamic changes make maintaining an accurate mental model of the system architecture very difficult. During operation, a fault in a single

service can propagate and cascade into other dependent services. The loose coupling of these components makes it difficult to identify root causes [5, 6].

2.2 Observability

The Cloud Native Computing Foundation [8] describes observability as:

"Observability is a system property that defines the degree to which the system can generate actionable insights. It allows users to understand a system's state from these external outputs and take (corrective) action."

Giving these kinds of insights and understanding into the system behavior to developers allows for higher developer productivity and decreased incident response times. Increasing extra-functional properties such as reliability and availability is critical in systems where outages can negatively impact the user experience and pose a significant loss of revenue.

Interconnected context of different aspects allows to see how individual components affect the behavior of the system. By providing a comprehensive, unified view of the system, operators gain a deeper awareness of the larger architecture. Intuitive and understandable dashboards allow faster exploration of telemetry data and insights, reducing the time needed to detect and resolve issues.

Another key aspect is maintaining a single source of truth, fusing different kinds of monitoring data into a unified platform. Ensuring all collected data is available for analysis and reporting. Beyond data collection, an effective observability system requires the ability to correlate anomalies with observations to facilitate root-cause analysis, allowing a holistic understanding of issues and their impacts [9].

One of the most widespread standards in observability is the OpenTelemetry framework. OpenTelemetry is designed to create and manage telemetry data, providing a common standard for high-quality, portable telemetry [10].

2.3 What can be observed?

Observability is essential for understanding the health and performance of complex systems, particularly in microservice architectures. At its core, observability is built upon three main pillars [9] (Figure 2.2): logs, metrics, and traces. These data types provide a multi-aspect approach to monitoring and diagnosing system behavior. Each one offers unique insights that come together to give a complete picture of how a system works.

The following subsections describe the three pillars of observability in detail.

2.3.1 Monitoring performance metrics

Metrics monitoring is a crucial building block for creating observable systems, providing a quantifiable view of system performance over time by tracking request rates, error counts, resource utilization, and latency. Monitoring infrastructure metrics is easier compared to application metrics, but the most popular metrics to monitor are resource usage, for example CPU and memory utilization [11].

The collected data is often used as a basis for alerting systems when significant thresholds are reached, or relevant signals are detected. By taking advantage of early warnings, teams

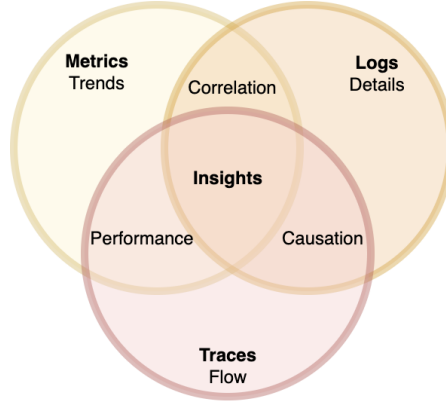


Figure 2.2: Three Pillars of Observability

can proactively address potential problems before they cause a failure in the overall system. By monitoring key metrics, the impact of new changes and releases can be assessed [1]. By employing metrics monitoring, organizations can gain a deeper understanding of their systems, address bottlenecks, and react to anomalous behavior.

The most common open-source metrics monitoring tool is Prometheus. Prometheus is used to collect and query time-series data on CPU usage, memory usage, network performance, and more. It boasts a powerful multi-dimensional data model, supports built-in alerting, and has the ability to scrape metrics from multiple sources.

Table 2.1 shows an example based on data collected from the system presented in 6.1. The first column of the table shows that the data is sampled every sixty seconds, with the timestamp being recorded for each entry. The metrics are collected from the frontend pod. At the 1661140459 timestamp, a possible anomaly can be seen, with all three metrics rising at a rapid rate.

TimeStamp	Pod	CpuUsageRate (%)	MemoryUsageRate (%)	Latency P90 (s)
1661140339	frontend	28.64	12.31	0.81
1661140399	frontend	25.73	12.82	0.69
1661140459	frontend	80.54	69.92	2.09
1661140519	frontend	81.35	20.55	2.31

Table 2.1: Example Metrics

2.3.2 Logging

Logging is the practice of generating, collecting, centralizing, and parsing log data, such as errors and failures. Logs contain data about events as seen, errors, and failures in the system components [1]. This data is produced by application servers, network devices, and other components. Log management is common for both microservice-based and traditional systems. Scanning and monitoring the collected data can be used to detect various problems, from exceptions to HTTP response codes. As microservices often run on virtual machines, which are dynamically created and destroyed, proper log management approaches, such as centralized logging, are necessary to identify service failures and causes [11]. Usually, logs carry information about the location of the log generation in virtualized systems (e.g. nodes, pods, and containers).

```
{
  "message": "[CreateQuoteFromFloat] received request",
  "severity": "info",
  "timestamp": "2022-08-22T04:01:35.509258491Z"
}
```

Listing 1: Example of an *Info* Log Entry

```
{
  "error": "product id not specified",
  "http.req.id": "7e7635d6-47f4-431c-8816-357545ee2f19",
  "http.req.method": "GET",
  "http.req.path": "/product/L9ECAV7KIM",
  "message": "Request error",
  "session": "ddc4280b-2a14-4833-bf3f-4766d33ae9d9",
  "severity": "error",
  "timestamp": "2022-08-22T04:02:15.08603417Z"
}
```

Listing 2: Example of an *Exception* Log Entry

Logs may contain information about an event in the component. These logs can be used to measure service usage, to verify the correct functioning of a component, or to help audit the security properties of the system. Information logs may even help track down bottlenecks, by showing requests stuck in queues waiting for resources. The log in Listing 1 shows that the service received a request.

Some logs may contain exceptions and errors thrown by the applications. These logs are invaluable for debugging and troubleshooting during development and operation. Error logs can also be used to generate error metrics, or as alerting when a specific error is found. The log in Listing 2 shows a request error, as the product *id* was not specified. This could point to a software bug being present in the calling service.

Logging tools also collect the HTTP responses seen commonly in these systems. These codes help quickly identify issues using a standardized system, from client to server errors. These response codes provide insight into error identification and user interactions. A sudden rise in 500 errors can imply that a service is unavailable. The HTTP 400 error seen on 3 shows a bad request.

```
{
  "http.req.id": "7478c494-d93b-4970-9836-63467c881302",
  "http.req.method": "GET",
  "http.req.path": "/product/1YMWWN1N40",
  "http.resp.status": 400,
  "message": "Request complete",
  "session": "59e62a54-a595-42b0-8450-647e563cfa33",
  "severity": "info",
  "timestamp": "2022-08-22T04:02:33.536472327Z"
}
```

Listing 3: Example of a *HTTP Response Code* Log Entry

2.3.3 Distributed Tracing

Distributed Tracing is a monitoring technique centered around the capture of detailed execution of casually related activities in a distributed system. Unlike other tracing techniques, distributed tracing is focused on observing the end-to-end execution of individual requests performed by many different services and processes, typical in microservice-based systems [12]. Distributed traces not only cover the path that a request travels through in the system, they also record information about the timing of operations. This can facilitate the identification of bottlenecks and latency issues [1]. A simplified view of the trace hierarchy and timing telemetry can be seen in Figure 2.3.

In Google’s technical report about their large-scale distributed tracing tool Dapper [13], they highlight the value of effectiveness of this monitoring approach. Dapper’s success is attributed to its incredible ability to provide a comprehensive understanding of system behavior to developers and operations teams.

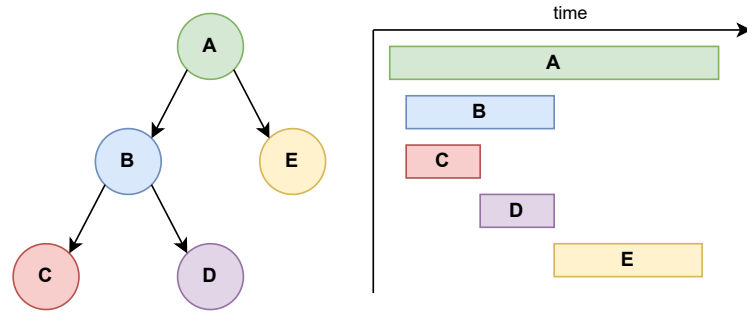


Figure 2.3: Example Trace

A trace represents a single request to the overall system. Traces are made up of spans, which are individual operations executed in a tree hierarchy. Each span contains its parent span (or root), the start and end time, attributes, and context, such as operation name, related pod, and service. Spans may also contain events, links, and status information about the operation. With more in-depth instrumentation more information can be collected and evaluated in these traces, however, in this report, the solution will be based on the bare minimum of information contained in spans, parent span link, timing data, executing pod, and operation name. An example span from the OpenTelemetry Documentation [10] can be seen in Listing 4. It is important to note that the parent ID is null, meaning this is the root of the trace.

2.4 Research Direction

As stated by the Cloud Native Computing Foundation [8], the goal of observability is to generate actionable insights. Robust and capable fault diagnosis tools can improve the system’s overall stability and reliability by localizing the root cause of failures, allowing for swift intervention to prevent cascading failures. Providing accurate and concise information about the system state in failure situations enables quicker recovery times, thus reducing downtime. Automated tooling is able to reduce the workload of development and operations personnel, allowing them to focus on providing value instead of troubleshooting and debugging. These advantages of diagnostics tools help minimize financial losses stemming from downtimes and increase user experience quality [2].

```

{
  {
    "name": "hello",
    "context": {
      "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
      "span_id": "051581bf3cb55c13"
    },
    "parent_id": null,
    "start_time": "2022-04-29T18:52:58.114201Z",
    "end_time": "2022-04-29T18:52:58.114687Z",
    "attributes": {
      "http.route": "some_route1"
    }
  }
}

```

Listing 4: Example Span Entry

A 2024 survey [2] highlighted key research directions in failure diagnosis for microservice systems. The solution presented in this report aims to tackle two of these key directions:

Explainable Diagnostic Results Many current approaches rely on complex machine learning and deep learning models, which can prove difficult to interpret for operators. By improving the explainability, trust and confidence in diagnostic results can be increased, leading to more informed decision-making.

Multimodal Data Fusion Traditional techniques are often focused on single data sources like logs or metrics, potentially overlooking valuable information. By fusing multiple data sources, including logs, metrics, and topology, it is possible to provide a more comprehensive understanding of the operational state of the system.

2.5 Related Work

Sieve [14] is a platform built to derive actionable insights from monitored metrics in distributed systems. Sieve obtains metrics and call graphs (similar to traces), reduces the number of data points by filtering and clustering, selecting representative metrics to best reflect the system behavior. This reduces the number of metrics to analyze, all while preserving information about the overall state. By identifying causal relationships and building a dependency graph, it helps developers better understand the system’s behavior. This tool can be applied for various management tasks such as auto-scaling and root cause analysis.

iSQUAD [15] presents a framework to diagnose the root causes of intermittent slow queries in cloud database services. This framework is noteworthy, as it aims to create a highly interpretable and accurate model for operators. Identified cases are presented with concrete examples to aid understanding for operators. By highlighting anomalies in influential key performance metrics, the most significant factors are emphasized. Engineers presented with these representations showed significantly better performance in identifying correct root causes.

Nezha [16] introduces a novel approach for root cause analysis in microservice systems. The proposed solution uses application-level metrics for anomaly detection, triggering further analysis. A common event graph is created by integrating multimodal data from tracing, metrics, and logs. Nezha leverages Pattern Mining from these graphs for high-impact root causes and runs these results, focusing on results that indicate a deviation from expected execution paths.

The proposed solution leverages multimodal data sources (e.g. metrics, logs, traces) for anomaly detection, and uses similar approaches to identify causal relationships between services. However, by creating a heterogeneous metamodel and using Answer Set Programming for logic reasoning, the approach allows for greater flexibility in different applications and a more modular architecture. Using inductive learning explainable diagnostic rules can be generated, allowing for a more in-depth understanding of the system behavior and providing an insight into the fault diagnosis process.

Chapter 3

Logic Reasoning

Logic programming is a programming paradigm driven by the idea that a single formalism can be used for both logic and computation, with logic encompassing computation [17]. Unlike imperative programming, where the program specifies how to solve a problem, in logic programming, the program describes the problem at hand, and the system will solve it utilizing the power of logic and leveraging and expressing knowledge and computation as facts, rules, and queries.

3.1 Qualitative Reasoning

To simplify the diagnostic process and aid in creating interpretable models, this report uses qualitative reasoning over monitoring data. Qualitative reasoning is an abstraction technique that aims to help understand and predict the behavior of complex systems without relying on precise numerical data and computations. Using qualitative descriptions to model these systems can simplify thinking and can work even in situations where quantitative data may be incomplete or unavailable. The process is shown in Figure 3.1. The following paragraphs describe the process steps in further detail [18].

The continuous model represents a system using continuous variables and equations that describe the relationships between these variables. For example, these might be differential equations representing the dynamics of a physical system, such as the flow of fluids and water levels.

Discretization involves turning these continuous variables into discrete categories. This abstraction reduces the complexity of the models, by grouping ranges of values into qualitative states. The qualitative state is created by defining thresholds at critical values in terms of system behavior. For example, in a water tank, the following qualitative states could be defined for the water level: *Empty*, *Normal*, *Overflowing*.

The qualitative model uses discretized variables and captures the relationships between these variables in a qualitative manner. This offers two major advantages: the generalization allowed by the discrete variables makes the model applicable to a broader range of situations, while it is also more robust, as it is less sensitive to measurement errors or uncertainties. The model is made up of qualitative variables, qualitative relationships, describing how variables influence each other, and also causal relationships.

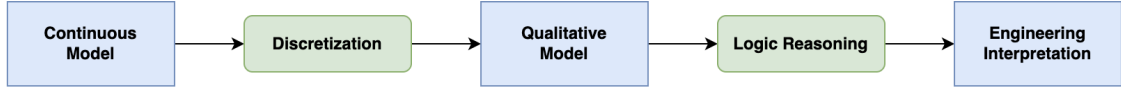


Figure 3.1: Qualitative Reasoning Workflow

Logic reasoning applies logic inference rules to the qualitative model to predict the system behavior. The main tool for this reasoning is qualitative simulation, which aims to explore the state space defined by the qualitative model, predicting possible behaviors. Constraint satisfaction aims to find inputs that can satisfy all qualitative constraints. While rule-based methods use ASP rules created from expert knowledge to infer knowledge from observations.

Engineering Interpretation is the final step, allowing engineers to make decisions or take action. The aim of qualitative reasoning is to give actionable insights, which can be explained and understood even for highly complex systems. The main application areas are diagnosis, design, and control of these systems.

3.2 Answer Set Programming

Answer Set Programming (ASP) is a form of declarative programming oriented towards difficult search problems [3]. The approach originates from the field of logic programming and non-monotonic reasoning. ASP utilizes the concept of stable model semantics to represent and solve problems. Problems are encoded as a logic program, and the solutions to the problem correspond to the answer sets, also known as stable models, of that program. An answer set is a consistent set of literals that represent a possible solution, which satisfies all rules and constraints of the program.

The workflow of ASP solving can be seen in Figure 3.2. The starting point is a real-world problem that needs to be solved. This could range from a scheduling task or puzzles to any kind of problem with a set of constraints. Modeling this problem involves encoding the facts, rules, and constraints into the logic program. The grounder transforms the logic program with variables into a variable-free, ground program. The solver processes this ground logic program and computes the stable models that satisfy the rules and constraints. The set of literals representing the solutions are extracted from the stable models and are interpreted to the solution.

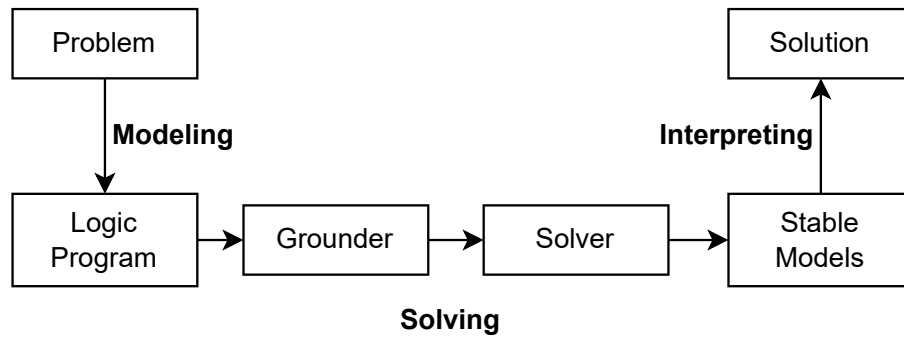


Figure 3.2: Solving Process [19]

```

1 wears_red(nick).
2 has_beard(nick).
3 not is_short(nick).

```

Listing 5: ASP Atoms

```

1 santa(Person) :- wears_red(Person), has_beard(Person).
2 elf(Person) :- wears_red(Person), is_short(Person).

```

Listing 6: ASP Rules

Atoms are the basic building blocks of ASP programs, they state the facts of the problem. Literals consist of atoms or their negations. A positive literal is the atom itself, while a negative literal is the negation of an atom. A simple example of atoms and literals be seen in Listing 5. The literals in this code are considered facts, as they define the known information about the problem.

Rules define how new information can be inferred from known facts and other rules. They are made of a head and body, where the head is a single literal, and the body is a conjunction of literals. An example rule can be seen in Listing 6. These rules mean that if a `Person` wears a red coat and has a beard, they are Santa, while if they wear red and are short, they are an Elf.

Disjunctive rules are able to model the nondeterministic behavior of the system. This can help include more knowledge in the model, even when the precise behavior cannot be determined. Listing 7 shows a disjunctive rule, which states that someone who is Santa is either giving presents or making presents. For example, if the literal `santa(nick)` is true, the program would return two possible stable solutions: `{giving_presents(nick)}` and `{making_presents(nick)}`.

```

1 1 {giving_presents(Person), making_presents(Person)} 1 :- santa(Person).

```

Listing 7: ASP Disjunctive Rules

3.3 Inductive Logic Programming

The paradigms of Inductive Logic Programming (ILP), and Inductive Learning of Answer Sets (ILASP) are used in this report to support inductive diagnostic rule generation described in detail in Chapter 5.

The advantages of ILP systems is their ability to learn rules that can be explained and interpreted [20]. Compared to other state-of-the-art artificial intelligence and machine learning approaches, which function like black-boxes, ILP systems work transparently with algorithms that can be fully explained. These systems are also able to generalize from a small set of examples, yielding a faster and cheaper training process [21]. Inductive Logic Programming has seen recent use in novel research from weather forecasting [22] using the FastLas tool to repair Petri nets [23] with ILASP.

Formally, an ILP task is a tuple $T = \langle B, S_M, \langle E^+, E^-, C \rangle \rangle$ where B is an ASP program, S_M is a set of ASP rules, E^+ , E^- are the finite sets of positive and negative partial interpretations, and C is the optional context of the example. $H \subseteq S_M$ hypotheses is an inductive solution of T if and only if:

$$\forall e^+ \in E^+ \quad \exists A \in AS(B \cup H \cup C) \text{ such that } A \text{ extends } e^+$$

Meaning that for all positive examples, there must be a rule that covers the example in the set of answer sets of the union of the background knowledge and the hypothesis.

$$\forall e^- \in E^- \quad \nexists A \in AS(B \cup H \cup C) \text{ such that } A \text{ extends } e^-$$

Meaning that for all negative examples, there must not be any rule in the answer sets that covers the negative example [4].

Inductive Learning of Answer Set Programs (ILASP) is a novel system for machine learning of ASP programs from examples [4] ILASP works by creating a meta-level representation of the candidate hypotheses, rules to be learned, and the meta-predicates, possible rule bodies. These mode declarations specify the allowed structure of rules, which are used to generate all possible rule structures, the hypothesis space. Examples and partial interpretations are translated into constraints. These constraints are used to test the candidate's hypotheses, as the solution must satisfy positive examples and not violate the negative examples.

In the following paragraphs, an example ILASP program is showcased to aid in understanding the methods used in this report. This scenario aims to learn rules that decide whether an individual is Santa or an Elf based on three attributes: whether the subject wears red, is short, or has a beard. To learn these rules, three examples are given, Nick, Lucy, and Goat. These individuals are the dataset used for learning, and they were labeled for Santa and an Elf.

The mode declarations guide ILASP in constructing hypotheses with only the specified predicates, as shown in Listing 8. The `#modeh` declarations show the predicates to learn the possible heads of the rules. The `#modeb` declarations are the predicates that can appear in the body of the rules. In this scenario, the head can be `santa` or `elf`, as our objective is to get rules that are true if an individual is Santa or an Elf, respectively. As our only reference for making this decision is the fact that the individual wears red, has a beard, or is short, the body of the rules must contain these literals.

```

1 #modeh(santa(var(person))).
2 #modeh(elf(var(person))).
3 #modeb(wears_red(var(person))).
4 #modeb(has_beard(var(person))).
5 #modeb(is_short(var(person))).

```

Listing 8: Mode declarations of the ILASP Program

The three examples are shown below. The first example is shown in Listing 3.3a. This example, named `e1`, contains Nick, a reliable example, as the penalty for not covering the example in the final rules is 5. The name and penalty of the example are shown at the start of the example, here `e1@5`. The example states that `santa(nick)` should be true,

meaning Nick is Santa, written in the *inclusions* part. It also states that `elf(nick)` should be false, meaning Nick is not an Elf, written in the *exclusions* part of the example. The context for this example is that Nick wears red and has a beard.

The second example shown in Listing 3.3b contains Lucy, for whom the `elf(lucy)` predicate should be true, and the `santa(lucy)` predicate should be false, meaning she is an Elf and not Santa. The context for this example is that Lucy wears red and is short.

The third example describes a goat, shown in Listing 3.3c. This goat is short and also wears red. In this example the labeled dataset shows that this goat was identified as Santa, incorrectly, for this reason the data point is should be considered noise. This example is given a low penalty compared to the others at a value of 1. This could be for a multitude of reasons in real applications, from heuristic approaches to labeling confidence values.

1	<code>#pos(e1@5,</code>	<code>#pos(e2@5,</code>	<code>#pos(e3@1,</code>
2	<code>{</code>	<code>{</code>	<code>{</code>
3	<code> santa(nick)</code>	<code> elf(lucy)</code>	<code> santa(goat)</code>
4	<code>},</code>	<code>},</code>	<code>},</code>
5	<code>{</code>	<code>{</code>	<code>{</code>
6	<code> elf(nick)</code>	<code> santa(lucy)</code>	<code> elf(goat)</code>
7	<code>},</code>	<code>},</code>	<code>},</code>
8	<code>{</code>	<code>{</code>	<code>{</code>
9	<code> wears_red(nick).</code>	<code> wears_red(lucy).</code>	<code> wears_red(goat).</code>
10	<code> has_beard(nick).</code>	<code> is_short(lucy).</code>	<code> is_short(goat).</code>
11	<code>}</code>	<code>}</code>	<code>}</code>
12	<code>).</code>	<code>).</code>	<code>).</code>

(a) Nick Example
(b) Lucy Example
(c) Goat Example

Figure 3.3: ILASP Examples

After running this ILASP program, the output will return the following learned rules:

```

1 { santa(V1) :- has_beard(V1). }
2 { elf(V1) :- is_short(V1). }
```

These rules mean, that we should consider an individual Santa if they have a beard, and consider an individual an Elf if they are short. The property of wearing red is not written in the rules, as ILASP aims for the simplest rules to solve the given learning task. In the creation of these rules ILASP prioritizes the higher penalty interpretations over ones with a lower weight. Effectively excluding the goat example in order to create rules that can satisfy the higher cost Nick and Lucy examples.

Chapter 4

Modeling and Fault Diagnosis

In this chapter, I present two contributions of the report. The heterogenous modeling approach I created for multimodal observability data and the developed qualitative diagnostic framework. These approaches aim to address Challenge 1, the use of multimodal data in understanding the behavior of microservice systems.

4.1 Request Metamodel

To represent the collected multimodal observability data, I created a metamodel to represent operations in the system. This metamodel provides a structured foundation for further analysis. Systematic representation simplifies interpretation and gives a cohesive framework as the basis for logic programs. The solutions explored in the subsequent chapters leverage the structure of the metamodel (Figure 4.1).

As the metamodel is heavily grounded in the structure of distributed traces, it is able to represent the relationships between the topological elements of the system (e.g., Services, Hosts). This can provide greater insight into the system.

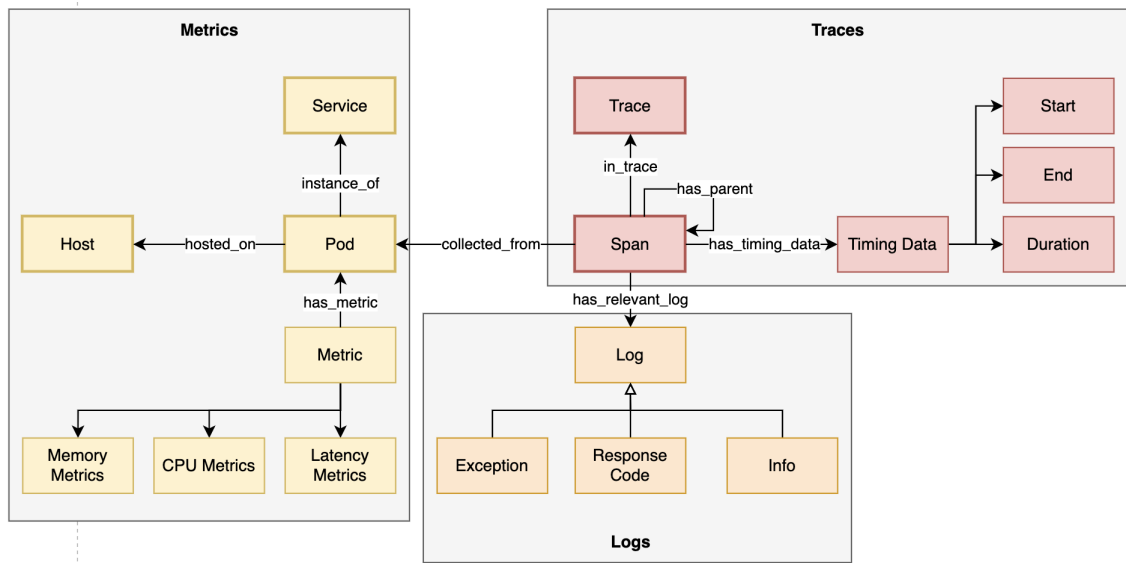


Figure 4.1: Request Metamodel

Metrics and timing data are derived using the quantization process described in Section 4.3. The metric atoms are sourced from the monitored metrics, providing insights into the system and service performance. Timing data is collected from tracing data, able to capture temporal relations and quality of service anomalies visible to users. Relevant logs are collected from monitoring and organized into specific categories (exceptions, HTTP response codes, info). The classification of logs makes the underlying logic programs more robust and portable, as the classification system is easier to adapt to the specific toolchain and requirements.

The approach presented in this report only illustrates one use of this metamodel. However, during experimentation, it was clear that the versatility of this metamodel extends beyond this single application. This unified structure opens up numerous possibilities for logic reasoning over observability data. The model extraction framework developed for this report enables further research into these potential applications. For example, this framework could be applied to analyze user behavior, allowing for a deeper understanding of user interactions based on trace data, and their relation to experience metrics and logs. It could also facilitate auto-scaling, providing predictive insights into system demand trends. The metamodel also has the potential to model extra-functional requirements, such as performance, reliability, and security attributes, which could be used to assess and ensure these qualities.

4.2 Log classification

In most systems, logs have three primary types, this chapter describes their use from their relevance to modeling system behavior and from a diagnostic standpoint.

Information logs are able to provide general insights into routine operations, mostly for tracking normal system behavior; they are most useful in well-instrumented systems. From a diagnostic standpoint, they can serve as a baseline for typical activity, distinguishing it from potential failures. Faults may also be ruled out if information logs can provide confirmation of the correct operation of a service, such as success messages, heartbeat signals, and status updates.

HTTP response logs help detect operation issues, capturing the success and failure of HTTP requests. This standardized format requires basically no instrumentation while containing significant diagnostic data. Response logs can shed light on issues from application level issues, network problems, or resource bottlenecks. However, these logs often lead to false positives due to fault hiding, possibly indicating failures in correctly functioning services that depend on a faulty service.

Error logs can pinpoint detailed, localized exceptions, thus providing fine-grained information on issues in the system. Even though they require some degree of instrumentation, such as exception handling, they are commonly implemented because of their utility and simplicity. The presence of error logs serves as an important indicator of issues in the system, and the information encapsulated in these logs can be invaluable in root cause analysis.

4.3 Quantization

Metrics provide quantitative data on system performance, which can signal the presence of faults in the system. The deviation of key metrics from their normal operational range can signal the presence of faults in the system. The current approach classifies metrics between two categories, normal and anomalous. The algorithm is described below, where $D_{training}$ is the training dataset, containing a tuple of the metric (m) and the timestamp (t). θ is the calculated threshold for quantization. This threshold is calculated before operation, based on the training dataset.

$$\theta = P_{95}(\{m \mid \forall(m, t) \in D_{training}\})$$

During operation, the collected metric, represented with a set of tuples as $D_{measured}$, is labeled based on the pre-calculated threshold (θ). The result is $D_{labeled}$, a set of tuples containing the discrete variable (m_l) and the timestamp.

$$D_{labeled} = \left\{ (m_l, t) \mid m_l = \begin{cases} \text{"high"} & \text{if } m \geq \theta \\ \text{"low"} & \text{if } m < \theta \end{cases}, \forall(m, t) \in D_{measured} \right\}$$

Labeling anomalous metrics in a discrete quality space allows the use of qualitative logic reasoning [24] with these key properties of the system. The selected statistical approach was chosen based on the distribution of key metrics and their relation to faults. This decision was made after analyzing the distribution of metrics in the datasets used in the report. An example of the distribution for a specific metric can be seen in Figure 4.3.

The span duration time can also be used to gain insight into the behavior of individual operations in the system, revealing signals, that may otherwise be undetectable, allowing for greater diagnostic ability [25]. Similar to monitored metrics, these values also follow a similar distribution, thus the same approach can be used to create a quality space and labeling rules based on statistical properties.

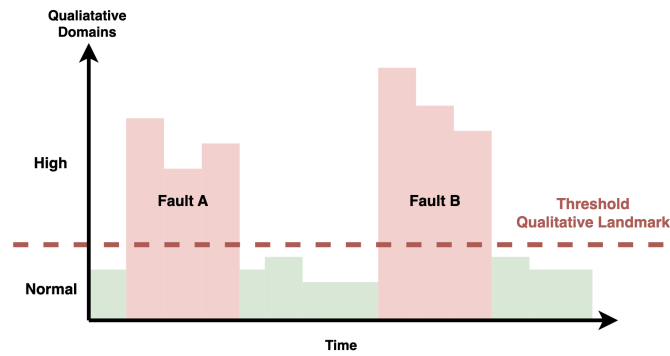


Figure 4.2: Metrics Relation to Faults

4.4 Model extraction process

For this metamodel to be viable, an efficient, automatic process is required for the creation of models from the collected telemetry data. The properties are extracted from log, metric, and trace data. The current process directly generates ASP code from the traces. However,

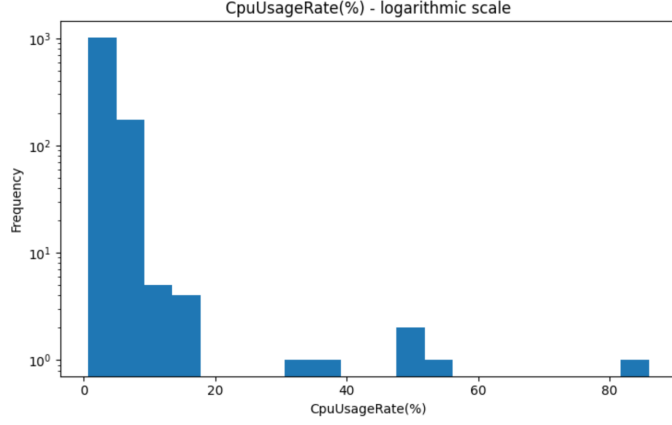


Figure 4.3: Metric Distribution on a Logarithmic Scale

there is a possibility for an intermediate representation allowing for the possibility for the use of other toolchains. These ASP request models can then be used with other ASP programs for analysis, as described in Section 4.5 The main workflow of model generation is shown in Figure 4.4. An example of these extracted request models can be found in the Listing 13.

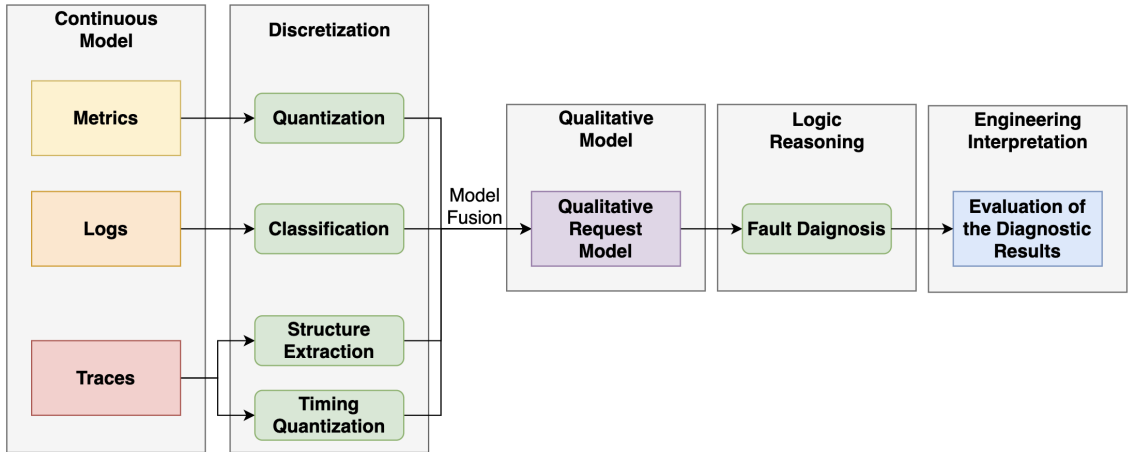


Figure 4.4: Model Extraction and Reasoning

4.5 Model Aggregation

Given the vast amount of requests handled by modern applications, applying large, complex diagnostic models to each request model would require significant resources. To improve the scalability of the solution, these request models are aggregated to focus on relevant anomalies and telemetry information. Summarizing key metrics and keeping pertinent data points, trends, and issues can be revealed without losing essential details. Subsection 4.5.1 describes how the proposed solution handles the reduction of span relationships, while Subsection 4.5.2 showcases the method created to represent important anomalies in the system.

4.5.1 Operation Graph

Traces contain the path of requests through the system. To simplify the massive amounts of trace telemetry data, I aggregate the spans by their operation, while keeping the parent-child relations between these aggregated groups, all taken request paths can be extracted from the system. These paths can show the normal interactions between services and even anomalous executions. In the next sections of the report, this new aggregated relationship graph will be called the operation graph.

The creation from traces to operation graph can be seen in Figure 4.5. Service graphs are commonly used for similar purposes in microservice development, however, by taking into account spans and operations, a more fine-grained model of the system can be created. The main differences between service graphs and the proposed operation graph can be seen in Figure 4.5

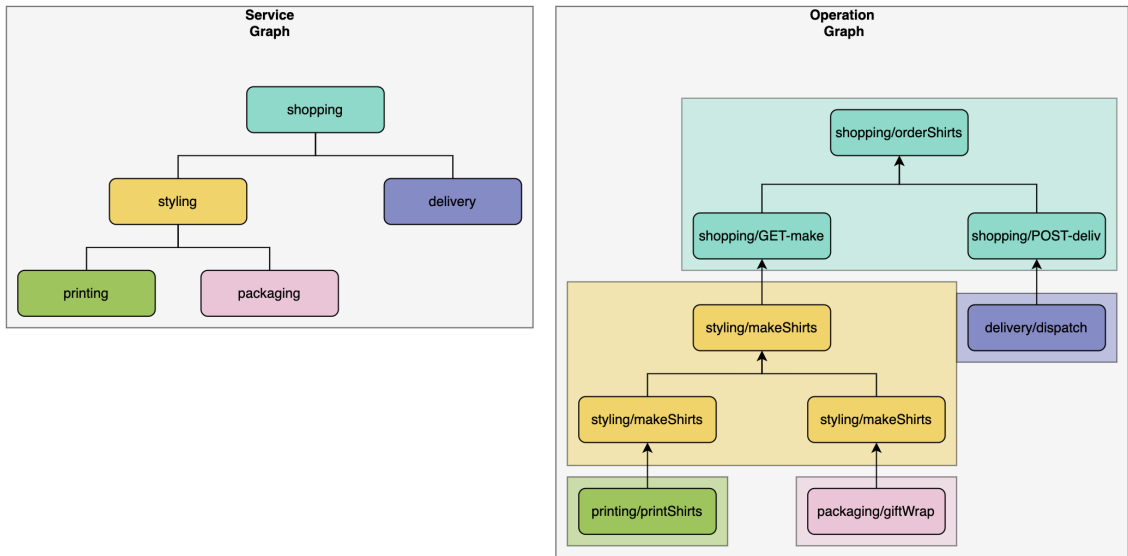


Figure 4.5: Service and Operation Graphs

4.5.2 Events and Metric Anomalies

Similar to the approach to tracing data, it proved necessary and advantageous to aggregate logs and metrics for inductive reasoning. Log events that appeared at least once in a fault injection timeslice can already signal the presence of a fault in the system. Metrics were also aggregated to facilitate simpler diagnostic rules.

4.5.3 Aggregate Model

An example of the reduced and aggregated model can be seen in Figure 4.6, as a visual representation of the ASP program of the model. This graph includes the operation graph, highlighting important differences to span relation graphs. The aggregated logs and metrics are displayed, summarizing the significant anomalies in the time period. A complete aggregate model can be found in Listing 14.

This aggregated data model allows for faster inductive learning, without sacrificing diagnostic performance. Using this method also allows for greater scalability, both in the rule extraction and operation phase.

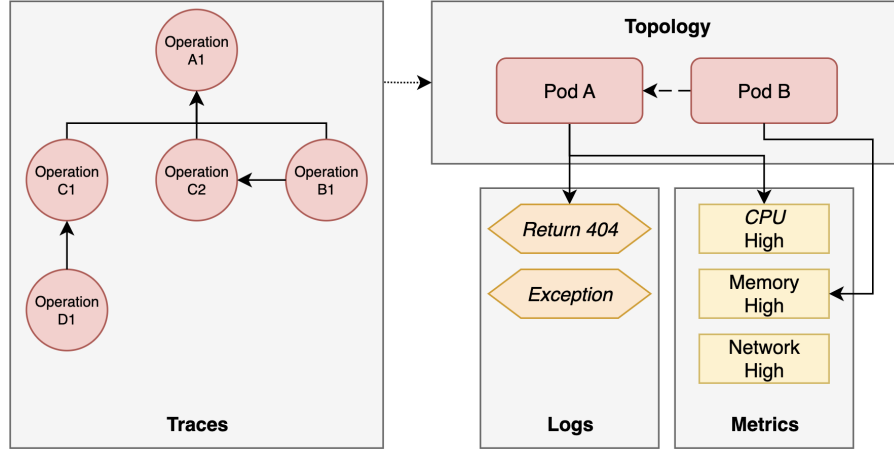


Figure 4.6: Aggregate Model of the Telemetry Data

4.6 Fault Diagnosis

The developed qualitative models can be leveraged for use in fault diagnosis. The nature of ASP programs, along with the structure of the created qualitative metamodels, makes this approach particularly well-suited for the incomplete, evolving, and inconsistent data typical of microservice systems. Encoding rules to model the possible fault conditions in the system allows logic reasoning to find patterns in the data that indicate potential faults, allowing it to infer specific issues within the system and identify likely fault origins. The presented approach facilitates robust and interpretable fault diagnosis.

The proposed diagnostic approach works by applying diagnostic rules to the qualitative aggregate models. Each rule head specifies the fault mode and the fault location in the system. The fault location can either be a variable suited for general rules that can be evaluated over any service in the system or a concrete reference to a specific service, enabling more precise, in-depth diagnostic details. This creates a highly adaptable framework, suitable for both broad and highly targeted fault localization. The body of each diagnostic rule consists of literals that describe the observed anomalies and relevant properties derived from the aggregate model, modeling the conditions under which a potential fault should be detected. The following is an example of a possible diagnostic rule:

```
fault(some_fault, Service):- cpu_metric(high, Service), has_log(error, Service).
```

Combining the ASP program of the aggregate model with the diagnostic rules yields a result that contains a set of fault literals representing the identified faults in the systems. These fault literals capture the root cause of the diagnosed fault, while the relevant anomalies are also provided for an interpretable result.

The complete workflow of the created diagnostic approach can be seen in Figure 4.7. Metric thresholds are created from previously collected data, and the rules are defined before operation. During operation, the system periodically extracts the aggregate model based on the request models representing collected data. The diagnostic rules are evaluated over the aggregate model, the resulting diagnosis is relayed to the operators in case a fault is detected. The diagnosis may also be used to execute automated corrective actions on the system.

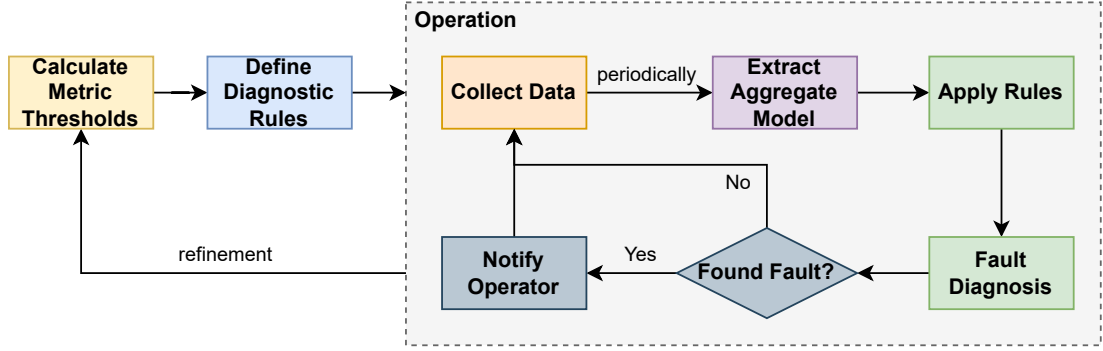


Figure 4.7: Diagnostic Workflow

4.6.1 Diagnosis Example

This section presents an example scenario to illustrate and explore the workings of the fault diagnosis framework. In this scenario, the system is comprised of several microservices, one of which is the *AuthService* microservice. The system employs performance monitoring, logging and distributed tracing for monitoring.

Metric thresholds are calculated for each service based on historical data. For the *AuthService* component, the following thresholds were created, shown in Table 4.1.

Metric	Threshold
CPU Usage [%]	85%
Memory Usage [%]	75%
Response Time [ms]	500 ms

Table 4.1: Metric Thresholds

The second step is to define the diagnostic rules to capture the potential fault conditions. The two diagnostic rules created for this scenario can be seen in Listing 9.

```

1 fault(cpu_consumed, Service) :-
2     cpu_metric(high, Service),
3     memory_metric(high, Service).
4
5 fault(endpoint_exception, auth_service) :-
6     has_log(400, auth_service),
7     has_parent(fallback_authenticate, authenticate).

```

Listing 9: Diagnostic Rules

The first rule describes the *CPU Consumed* fault, which is diagnosed when the CPU consumption and memory usage metrics of the service are high. The second diagnostic rule is specific to the *AuthService* microservice, the rule addresses the *Endpoint Exception* fault, which is defined to alert if the *AuthService* has 400 HTTP responses and the operation graph has a parent relationship between the *fallback_authenticate* and *authenticate* operations.

Timestamp	CPU [%]	Memory [%]	Response Time [ms]
1661140459	30	11	37
1661140519	76	40	42
1661140579	90	22	203

Table 4.2: Recorded Metrics from AuthService

Timestamp	Pod	Type	Value
1661140472	Frontend	Error	Invalid parameter
1661140523	AuthService	HTTP Response	400
1661140567	AuthService	HTTP Response	400

Table 4.3: Collected Logs

After defining the rules and calculating the thresholds, the next step is to observe the system in operation. Table 4.2 shows the metrics and Table 4.3 shows the logs in the scenario.

Periodically, the collected logs are classified into categories, and the performance metrics are labeled based on the thresholds. The qualitative request models are created after discretization and then aggregated into the aggregate model. The aggregate model can be seen in Listing 10, which shows the operation graph extracted from distributed traces.

```

1  cpu_metric(high, auth_service).
2  has_log(error, frontend).
3  has_log(400, auth_service).
4  has_parent(root, login).
5  has_parent(authenticate, login).
6  has_parent(fallback_authenticate, authenticate).
```

Listing 10: Aggregate model

The diagnostic rules are applied to the aggregate model. The rules check for the defined patterns and match diagnosis conditions to the aggregated data. The first rule, for *CPU Consumption*, is not fulfilled, as while the CPU metric is high, the memory metric remains normal. The second rule, for *Endpoint Exception* specifically in the AuthService module, is evaluated to be true, as the `has_log(400, auth_service)`, and `has_parent(fallback_authenticate, authenticate)` literals are present in the aggregate model. With the diagnosis made for the Endpoint Exception fault mode at the AuthService location, the alert would be sent to the operations team.

After taking the necessary steps for further analysis and needed corrective actions for the system, the operator may refine and modify the defined rules based on the lessons learned from this scenario.

Chapter 5

Rule Extraction with Inductive Learning

This chapter presents the contributions made for creating an automated workflow to learn diagnostic rules using inductive learning. The creation of interpretable diagnostic rules rooted in collected observations is necessary to solve Challenge 2, explainable diagnostic results. My solution also improves the adaptability of the diagnostic framework described in the previous chapter, while decreasing manual workload.

Manually defining diagnostics rules is time-consuming and complex, as it requires deep knowledge of the system architecture and behavior, potential failure modes, and even operational nuances. Each rule must be carefully formulated to accurately identify specific faults while avoiding false positives, necessitating extensive system expertise and regular refinement as the system evolves. The manual approach is very labor-intensive and highly error-prone, especially in the ever-changing environments of microservice systems.

The application of Inductive Learning offers a solution to this challenge, by automatically creating diagnostic rules. By analyzing labeled historical data, inductive algorithms are able to identify the correlations and patterns needed for fault diagnosis in the system. This requires no human input for rule generation, reducing the need for expert knowledge. Not only speeding up the process and decreasing the workload on operators but also being able to create more accurate and adaptable diagnostic rules. The presented Inductive Rule Extraction approach is a scalable and flexible solution for maintaining a robust set of diagnostic rules with minimal manual intervention.

5.1 Task definition

Let the operation graph be OP_{graph} , a set of operation relationships derived from distributed traces. Let $o_{\text{child}}, o_{\text{parent}} \in OP$, where OP is the set of all operations within the system. Here, the relation $(o_{\text{child}}, o_{\text{parent}}) \in OP_{\text{graph}}$ represents an execution path going from the parent operation to the child operation.

Define the set of services *Services*, let $s \in \text{Services}$, where each s represents an individual service within the system.

Let *Metrics* be the set of metric observations in the system. Each metric observation is written in the form $(m, v, s) \in \text{Metrics}$, where $m \in \text{MetricType}$ is a observed metric (e.g., Memory, CPU), $v \in \text{MetricValue}$ is a qualitative value of the observation (e.g., Low, High), and $s \in \text{Services}$ is a service in the system.

Let $Logs$ be the set of logging observations extracted from the system. Each log $(l, s) \in Logs$, where $l \in LogTypes$ a log type (e.g., 404 HTTP error), and $s \in Services$ is a service in the system.

Define the set of faults $Faults$ in the system, let $(f, s) \in Faults$, with $f \in FaultTypes$ a fault type (e.g., CPU contention), and $s \in Services$ is a service in the system.

To create inductive rules, I propose to encode this problem as *ILP tasks* as follows. Let the rule extraction problem be ILP tasks tuples $T = \langle B, S_M, \langle E^+, E^-, C \rangle \rangle$.

Where S_M is the hypothesis space defined by the head and body mode declarations $M = M_{head} \cup M_{body}$.

The heads are defined as $\{head \in M_{head} \mid head \in Faults\}$. The bodies of the rules in the hypothesis space can be defined from the union of elements from the observation sets $M_{body} = Metrics \cup Logs \cup OP_{graph}$.

E^+ is the set of positive partial interpretations, where $E^+ = \{f_{active} \mid f_{active} \in Faults\}$ representing the active faults in the aggregation. Let the negative partial interpretation $E^- = \{(f, s) \mid (f, s) \in Faults \setminus f_{active}\}$. The context C contains all observations from the aggregation, as $C = Logs \cup Metrics \cup OP_{graph}$

5.2 Workflow

This section describes in detail the workflow of inductive diagnostic rule extraction, focusing on the creation of ILASP programs from a fault-labeled monitoring dataset. The complete workflow is shown in Figure 5.1.

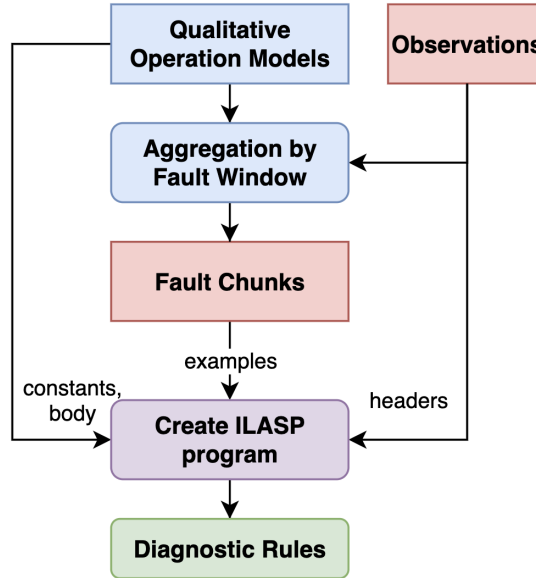


Figure 5.1: Rule Extraction Workflow

The first step in the process is extracting all faults, and their relevant data from a labeled dataset with faults in them. For each active fault, the impacted timeslice, which encompasses the duration in which the fault occurred, must be processed, request models shall be created and aggregated into an aggregate model. Following data collection and aggregate model extraction, an ILASP program is created for each microservice within the system.

Each ILASP program is tailored to a specific microservice in the system. The mode headers reflect only the faults injected into the particular microservice assigned to it, meaning that the results of the program will only create rules for that microservice relevant to the context of that service.

Each example in the program is a fault affected timeslice, all timeslices have to be included across all ILASP programs. This is critical because it allows for the analysis of examples where the inspected microservice is fault-free, providing a more robust dataset for inductive learning. The results of the execution of these ILASP programs are the fault diagnosis rules. These rules encapsulate the learned relationships and patterns identified by inductive learning.

5.3 Rule Extraction Examples

Listing 11 shows two examples from a simple ILASP program. The program contains a single fault, named `exampleFault`. There are two partial interpretations, `aggregate1` and `aggregate2`, the fault is active in `aggregate1` and inactive in `aggregate2`. The only context given is a metric, named `someMetric`, which has a `high` anomaly in `aggregate1`. After running this simple program, the result is a rule that states that the example fault is active when the `someMetric` metric is `high`. This demonstrates the basic principle of how partial interpretations work for diagnostic rules.

```

1 #modeh(fault(exampleFault)).
2 #modeb(someMetric(const(metricLevel))).
3 metricLevel(high).
4 metricLevel(low).
5
6 #pos(aggregate1,
7     {fault(cpuConsumed)},
8     {},
9     {someMetric(high)}
10 ).
11 #pos(aggregate2,
12     {},
13     {fault(cpuConsumed)},
14     {}
15 ).

```

Listing 11: Example ILASP program for fault rule extraction

The result of Listing 11 is:

```

fault(exampleFault) :- someMetric(high).

```

The excerpt from an ILASP program is shown in Listing 12 with two positive interpretations, `aggregate1`, and `aggregate2`. `aggregate1` has a weight of 10, this penalty value is calculated based on the relative relevance of the example in the diagnosis rule extraction. The value is higher if the example contains an actual fault in the service. The value is also influenced by the amount of anomalies present in the context aggregate model. The *inclusions* of the program is the `cpu_contention` fault localized in the `frontend` service. The *exclusion* is the `return` fault also localized in the `frontend` service. The *context* is the `high` CPU metric and `high` memory metric in the `frontend` service. `aggregate2` has a penalty of 15; this value is higher because it contains more anomalies in the aggregate

```

1  #pos(aggregate1@10,
2      {fault(cpu_contention, frontend)},
3      {fault(return, frontend)},
4      {
5          cpuMetric(high, frontend).
6          memMetric(high, frontend).
7      }
8  ).
9  #pos(aggregate2@15,
10     {fault(return, frontend)},
11     {fault(cpu_contention, frontend)},
12     {
13         log(400, frontend).
14         log(error, frontend).
15         netMetric(high, frontend).
16         has_parent(Frontend_ErrorHandler, Frontend_Open)
17     }
18 ).

```

Listing 12: Example ILASP Program for Fault Rule Extraction

model. The *inclusions* are the `return` fault in the `frontend` service, and the *exclusions* are the CPU contention fault also in the `frontend` service. The *context* describes two logs, a 400 HTTP response and an error log, a high network latency metric. It also contains an operation relationship, the `Frontend_ErrorHandler` operation is called by the `Frontend_Open` operation.

This example shows how different penalties can be set for partial interpretation based on how heavily they should influence the rule learning process. This is crucial, as real-world data usually contains noise, either from measurement errors or fault miss-labeling. The example also shows how all kinds of multimodal observability data can be used to give context to the fault slices, allowing for the generation of more accurate rules. A complete ASP program can also be found in Listing 15 and Listing 16.

5.4 Scalability

The time required to solve an ILP problem depends heavily on the size of the search space and the complexity of the background knowledge. For example, choice rules generate multiple models, each of which have to be evaluated. The number of examples also plays a smaller factor in the runtime, however this was barely noticeable at the scale of the benchmark datasets used in this report. The main concern in terms of scalability is increasing the complexity of the mode declarations. Using more types of variables and constants in these literals leads to an increase in the hypothesis space.

In the presented solution, the background knowledge does not affect the process since the program does not include choice rules. However, to decrease the number of constant types in the head and body declarations, I chose to decompose and extract rules based on services. This allowed for shorter execution times, facilitating rapid prototyping with different models and configurations necessary for research.

Chapter 6

Evaluation

The Online Boutique dataset was used for the evaluation of the developed framework. This dataset was chosen for its use of traces, metrics, and logs, as well as its previous usage for benchmarking contemporary diagnostic tools [16]. The dataset was extracted from a real, commonly employed benchmark system for microservice research and development.

Online Boutique is a microservices demo application created by Google to demonstrate their cloud-native tools and services, focusing on Kubernetes and Google Cloud Operations [26]. This application is built as an online store, the primary user journey in the application consists of the user selecting an item, adding it to their cart, and making the purchase. The system architecture can be seen in Figure 6.1.

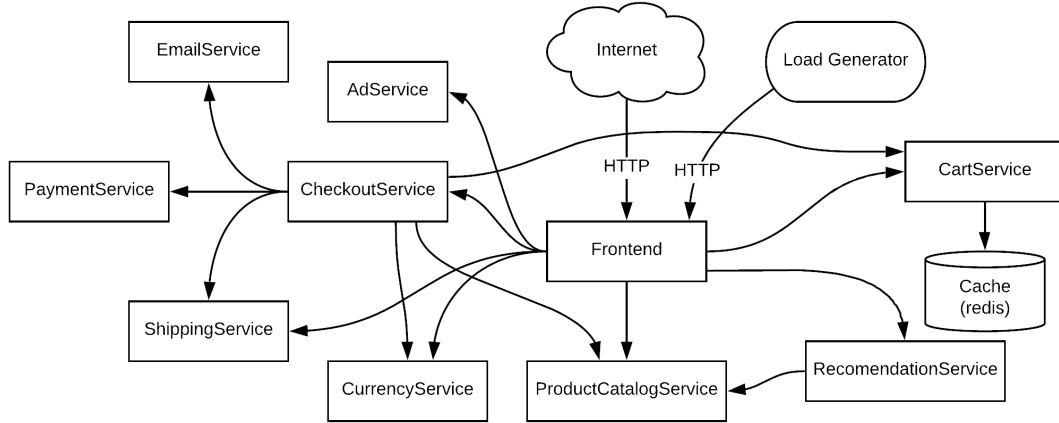


Figure 6.1: Architecture of the Online Boutique Application. Source: [26]

In this report, a pre-made, fault-injected dataset was used from Nezha [16]; as such, no deployment of this system was required. The augmented application is available on GitHub [27].

The dataset contains multiple kinds of faults injected into several different services. The following five fault modes are used:

- **CPU contention:** This fault occurs when multiple processes or services are competing for CPU resources simultaneously. It leads to decreased performance because the CPU is overloaded with tasks.

- **CPU consumption:** In this fault, a single process or service consumes an excessive amount of CPU resources. This can happen due to inefficient algorithms, infinite loops, or heavy computational tasks.
- **Network delay:** This fault introduces latency in network communication between services. It causes delays in data transmission, leading to slower response times and timeouts.
- **Exception:** This fault involves unexpected errors or exceptions occurring within a service’s execution. Examples include null pointer exceptions, array out-of-bounds errors, or any unhandled exceptions that can cause the service to crash or behave unpredictably.
- **Return:** This fault type refers to a service returning with unexpected behavior. It can be due to logic errors, incorrect configuration, or data corruption, affecting the functionality of dependent services that rely on the affected service.

The first two faults represent node-level performance issues in a system. *Network delay* is intended to show faults in the communication network of the system. *Exception* and *Return* faults are application-level faults, which resemble configuration issues and software bugs. By using the five fault modes, the approach can be tested regarding its viability in diagnosing many areas of network, infrastructure, and application-level faults while minimizing the complexity of the results.

6.1 Evaluation results

6.1.1 Rule extraction

The inductive rule extraction workflow was done on the training dataset from the Online Boutique application. This yielded 13 service-specific rules and 4 general rules. The rules used in the evaluation can be seen in the Listing 18. There was significant variety in the execution times of the ILASP programs for specific services, ranging from around 10 to 100 seconds, depending on the complexity of the service and the examples.

An excerpt of the diagnostic rule learning ILASP program for the Frontend service can be found in Listing 15 and Listing 16. This program does not contain all constants and partial interpretations where there are no active faults in the Frontend service, for readability. The service specific rule learning results for the Frontend service can be found in Listing 17.

6.1.2 Rule evaluation

The diagnostic evaluation is done, by taking each fault injection window (the timeslice where a specific fault is active) in the dataset as the input for fault diagnosis. Specific diagnostic rules are learned for each service in the system, while general diagnostic rules are aggregated from the results of multiple services.

The results are evaluated using the *Accuracy@1* ($A@1$) metric, also known as Top-1 Accuracy, and *Accuracy@3* ($A@3$) statistics, commonly used to evaluate the performance of models in classification tasks, such as recommender systems and multi-class classification problems.

Accuracy@1, measures the percentage of test cases where the correct answer was the top prediction. *Accuracy@3* measures the percentage of test cases, where the top three predictions were correct. As the developed solution does not rank the possible diagnosis results but may present multiple answers, the correct *A@1* test cases are considered when the only given answer is correct. *A@3* test cases are considered correct if there are at most three possible diagnosis results, and one of them is correct.

After running the diagnostics framework on the benchmark dataset, the *A@1* result was 37.5%, while the *A@3* result was 50%. The diagnostic result of each injection can be seen in Table A.1. These results are compared to results [16] from other diagnostic and root cause analysis tools, using the same dataset, as seen in Table 6.1. The results are also visualized in Figure 6.2, where the darker column represents *A@1* and the brighter grey shows *A@3* accuracy. The developed solution is superior to MicroScope, MicroRCA, SBLD, LogFaultFlagger, TraceAnomaly in Top-1 accuracy. While the results are slightly inferior to PDiagnose, a multimodal approach. Nezha, a novel multimodal approach, shows a significant accuracy difference between the solution presented in the report, but also to other state-of-the-art approaches. It is clear, that there is a great room for further development, however these results prove, that the developed framework has potential, even in this early state.

Approach	A@1	A@3
MicroScope [28]	12.5	41.07
MicroRCA [29]	16.07	62.5
SBLD [30]	19.64	23.21
LogFaultFlagger [31]	19.64	21.42
MicroRank [32]	41.07	48.21
TraceAnomaly [33]	30.35	33.92
PDiagnose [34]	41.07	73.21
Nezha [16]	92.86	96.43
Presented Solution	37.5	50

Table 6.1: Results of the Online Boutique Benchmark. Comparison Data from Nezha [16]

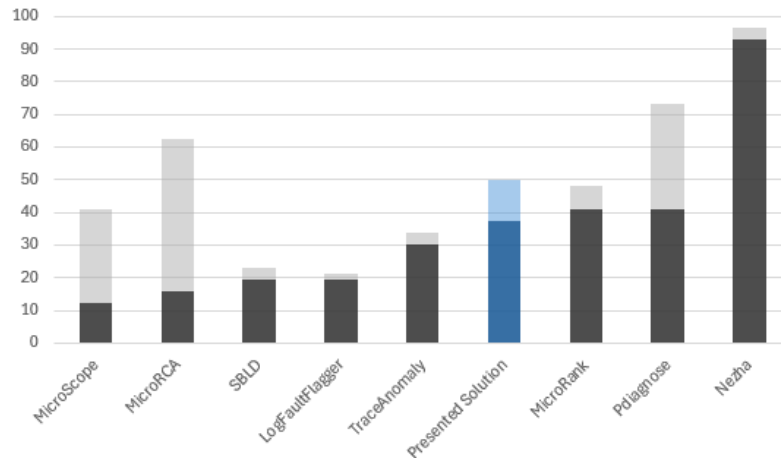


Figure 6.2: Results of the Online Boutique Benchmark

Chapter 7

Conclusion

This report aimed to address two main challenges concerning fault diagnosis and observability in microservice systems: multimodal telemetry data representation and creating explainable diagnostic results. To tackle these challenges, the following contributions were made:

Contribution 1: Metamodel for heterogeneous observability data

I created a request metamodel and aggregate model capable of handling heterogeneous observability data, integrating multiple types of observability data, including metrics, logs, and traces, to facilitate a comprehensive view of system behavior and enable further analysis.

Contribution 2: Model-based diagnostic framework

Leveraging the power of the created metamodel, I developed a modular diagnostic framework using the power of Answer Set Programming. This framework integrates the processing of telemetry data, qualitative model extraction, and the application of diagnostic rules, enabling precise and interpretable diagnostics in microservice environments.

Contribution 3: Inductive diagnostic rule learning

To automate diagnostic rule extraction, I devised a novel solution based on Inductive Logic Programming to learn rules using historical data. This technique is able to systematically derive diagnostic rules by analyzing fault-containing datasets, enabling the discovery of complex relationships and causalities.

The developed framework was evaluated using a fault-injected datasets based on Google's Online Boutique system. The solution was able to outperform several contemporary diagnostic tools, showing the potential for accurate, interpretable diagnostics in microservice systems. The developed models and framework lays the groundwork for further research in the field of observability and fault diagnosis.

Future Work

Other applications of the created metamodel, model extraction, and inductive reasoning framework should be explored. The developed approaches could have further uses ranging from auto-scaling, fulfilling extra-functional requirements, and even analyzing user behavior.

Improving the qualitative diagnostic process could yield more robust and accurate results. The use of clustering algorithms for metrics, qualitative knowledge discovery [35], and incorporating quantitative elements should be considered. The employing Error Propagation Analysis could enable the diagnosis of more complex faults [25].

Microservice test environment would allow for evaluating the solution, and further improvements over a more controllable and diverse set of data. A test bench with the necessary observability and fault injection tools was developed in the department [36].

Acknowledgements

I would like to express my sincere gratitude and appreciation to my advisors, András Földvári and Bertalan Zoltán Péter. This report would not have been possible without their continued support and encouragement. Their expertise, insightful knowledge, and feedback were invaluable during the writing of this report.

Bibliography

- [1] L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dinga, A. Koziolk, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin, and F. Simon Panojo. Monitoring tools for devops and microservices: A systematic grey literature review. *Journal of Systems and Software*, 208:111906, 2024.
- [2] Shenglin Zhang, Sibao Xia, Wenzhao Fan, Binpeng Shi, Xiao Xiong, Zhenyu Zhong, Minghua Ma, Yongqian Sun, and Dan Pei. Failure diagnosis in microservice systems: A comprehensive survey and analysis, 2024.
- [3] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI’08, page 1594–1597. AAAI Press, 2008.
- [4] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence - Volume 8761*, page 311–325, Berlin, Heidelberg, 2014. Springer-Verlag.
- [5] Sarah Wells. *Enabling Microservice Success*. O’Reilly, 2024.
- [6] Sam Newman. *Building Microservices, 2nd Edition*. O’Reilly, 2021.
- [7] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Aakash Ahmad, and Ali Rezaei Nassab. On the nature of issues in five open source microservices systems: An empirical study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE ’21, page 201–210, New York, NY, USA, 2021. Association for Computing Machinery.
- [8] Cloud Native Glossary Observability. <https://glossary.cncf.io/observability/>. Accessed: 2024-10-22.
- [9] Muhammad Usman, Simone Ferlin, Anna Brunstrom, and Javid Taheri. A survey on observability of distributed edge & container-based microservices. *IEEE Access*, 10:86904–86919, 2022.
- [10] What is OpenTelemetry. <https://opentelemetry.io/docs/what-is-opentelemetry/>. Accessed: 2024-10-22.
- [11] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners’ perspective. *Journal of Systems and Software*, 182:111061, 2021.
- [12] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*, chapter Chapter 3: Distributed Tracing Fundamentals. Packt Publishing, 2019.

- [13] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
- [14] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 14–27, New York, NY, USA, 2017. Association for Computing Machinery.
- [15] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. Diagnosing root causes of intermittent slow queries in cloud databases. *Proc. VLDB Endow.*, 13(8):1176–1189, April 2020.
- [16] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *ESEC/FSE 2023*. ACM, 2023.
- [17] Robert Kowalski. Logic programming. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 523–569. North-Holland, 2014.
- [18] Prof. Vinay K. Chaudhri, Stanford. CS 227: Knowledge Representation and Reasoning, Lecture 13. Qualitative Reasoning (spring 2011). <https://web.stanford.edu/class/cs227/Lectures/lec13.pdf>.
- [19] University of Potsdam Torsten Schaub. Answer Set Solving in Practice. <https://github.com/potassco-asp-course/course>, 2024.
- [20] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994. Special Issue: Ten Years of Logic Programming.
- [21] Andrew Cropper and Sebastijan Dumančić. Inductive logic programming at 30: a new introduction, 2022.
- [22] Talissa Dreossi, Agostino Dovier, Andrea Formisano, Mark Law, Agostino Manzato, Alessandra Russo, and Matthew Tait. *Towards Explainable Weather Forecasting Through FastLAS*, pages 262–275. Springer Nature Switzerland, 10 2024.
- [23] Francesco Chiariello, Antonio Ielo, and Alice Tarzariol. *An ILASP-Based Approach to Repair Petri Nets*, pages 85–97. Springer, 10 2024.
- [24] András Földvári and András Pataricza. Semi-automated model extraction from observations for dependability analysis. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 1–6, 2021.
- [25] Márton Tarnay. Managing operational uncertainties in deployed systems using logic reasoning. <https://tdk.bme.hu/conference/VIK/2023/sessions/distsys/paper/Mukodesi-Bizonnytalansagok-Kezelese-Elosztott>, 2023.
- [26] Google. Online boutique application. <https://github.com/GoogleCloudPlatform/microservices-demo>, n.d. Accessed: 2024-10-23.

- [27] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. Augmented online boutique. <https://github.com/IntelligentDDS/Augmented-OnlineBoutique>, 2023. Accessed: 2024-10-23.
- [28] Jinjin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 3–20, Cham, 2018. Springer International Publishing.
- [29] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2020.
- [30] Carl Martin Rosenberg and Leon Moonen. Spectrum-based log diagnosis. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [31] Anunay Amar and Peter C. Rigby. Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 140–151, 2019.
- [32] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinhong Sun, and Xiaoyun Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, WWW '21, page 3087–3098, New York, NY, USA, 2021. Association for Computing Machinery.
- [33] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (IS-SRE)*, pages 48–58, 2020.
- [34] Chuanjia Hou, Tong Jia, Yifan Wu, Ying Li, and Jing Han. Diagnosing performance issues in microservices with heterogeneous data source. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 493–500, 2021.
- [35] Gabriele Kern-Isberner, Matthias Thimm, and Marc Finthammer. Qualitative knowledge discovery. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, *Semantics in Data and Knowledge Bases*, pages 77–102, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [36] Weisz Apolló. Distributed-tracing-based diagnosis of microservices. Bachelor’s thesis, Budapest University of Technology and Economics, 2023.

Appendix

```
1 service(productcatalogservice).
2 service(frontend).
3
4 pod(frontend_579b9bff58_t2dbm).
5     instance_of(frontend_579b9bff58_t2dbm, frontend).
6 pod(productcatalogservice_668d5f85fb_jhwx9).
7     instance_of(productcatalogservice_668d5f85fb_jhwx9, productcatalogservice).
8 has_cpu_metric(productcatalogservice_668d5f85fb_jhwx9, high).
9
10 span(span_cb4e1f95f5bb1021, op_hipstershop_Frontend_Recv_).
11     has_response_code(span_cb4e1f95f5bb1021, 500).
12     has_exception(span_cb4e1f95f5bb1021, error).
13     in_trace(span_cb4e1f95f5bb1021, trace_00434254a6708f091a51a1e5147244f1).
14     has_parent(span_cb4e1f95f5bb1021, span_root).
15     runs_on(span_cb4e1f95f5bb1021, frontend_579b9bff58_t2dbm).
16     has_start(span_cb4e1f95f5bb1021, 1661267117693136314).
17     has_end(span_cb4e1f95f5bb1021, 1661267117697751351).
18
19 span(span_8340a8c9d8a3e136, op_hipstershop_ProductCatalogService_GetProduct).
20     in_trace(span_8340a8c9d8a3e136, trace_00434254a6708f091a51a1e5147244f1).
21     has_parent(span_8340a8c9d8a3e136, span_cb4e1f95f5bb1021).
22     runs_on(span_8340a8c9d8a3e136, frontend_579b9bff58_t2dbm).
23     has_start(span_8340a8c9d8a3e136, 1661267117693322018).
24     has_end(span_8340a8c9d8a3e136, 1661267117697438174).
25
26 span(span_d2d0aa53cae7dc26, op_hipstershop_ProductCatalogService_GetProduct).
27     has_exception(span_d2d0aa53cae7dc26, error).
28     in_trace(span_d2d0aa53cae7dc26, trace_00434254a6708f091a51a1e5147244f1).
29     has_parent(span_d2d0aa53cae7dc26, span_8340a8c9d8a3e136).
30     runs_on(span_d2d0aa53cae7dc26, productcatalogservice_668d5f85fb_jhwx9).
31     has_start(span_d2d0aa53cae7dc26, 1661267117694324289).
32     has_end(span_d2d0aa53cae7dc26, 1661267117696706807).
33
34 span(span_2efef040b8733d22, op_sql_conn_query).
35     in_trace(span_2efef040b8733d22, trace_00434254a6708f091a51a1e5147244f1).
36     has_parent(span_2efef040b8733d22, span_d2d0aa53cae7dc26).
37     runs_on(span_2efef040b8733d22, productcatalogservice_668d5f85fb_jhwx9).
38     has_start(span_2efef040b8733d22, 1661267117696181795).
39     has_end(span_2efef040b8733d22, 1661267117696641875).
```

Listing 13: ASP program of a Request Model

```
1 depends_on(productcatalogservice,frontend)
2 depends_on(checkoutservice,frontend)
3 depends_on(shippingservice,checkoutservice)
4 depends_on(recommendationservice,frontend)
5 depends_on(productcatalogservice,recommendationservice)
6 depends_on(productcatalogservice,checkoutservice)
7 depends_on(shippingservice,frontend)
8
9 log(500,frontend)
10 log(error,productcatalogservice)
11 log(error,frontend)
12 log(error,checkoutservice)
13
14 memMetric(high,productcatalogservice)
15 netMetric(low,shippingservice)
```

Listing 14: Aggregate Model

```

1  #modeh(fault(const(faultmode), var(service))).
2  #modeb(cpuMetric(const(metricLevel), var(service))).
3  #modeb(memMetric(const(metricLevel), var(service))).
4  #modeb(log(const(logValues), var(service))).
5
6  #pos(id1661140434@15,
7      {
8          fault(cpu_contention, frontend_579b9bff58_t2dbm)
9      },
10     {
11         fault(return, frontend_579b9bff58_t2dbm),
12         fault(exception, frontend_579b9bff58_t2dbm),
13         fault(cpu_consumed, frontend_579b9bff58_t2dbm)
14     },
15     {
16         cpuMetric(high, frontend_579b9bff58_t2dbm).
17         depends_on(checkoutservice_578fcf4766_9csqn,
18                     frontend_579b9bff58_t2dbm).
19         memMetric(high, frontend_579b9bff58_t2dbm).
20         netMetric(low, frontend_579b9bff58_t2dbm).
21     }
22 ).
23 #pos(id1661140927@15,
24     {
25         fault(return, frontend_579b9bff58_t2dbm)
26     },
27     {
28         fault(cpu_contention, frontend_579b9bff58_t2dbm),
29         fault(cpu_consumed, frontend_579b9bff58_t2dbm),
30         fault(exception, frontend_579b9bff58_t2dbm)
31     },
32     {
33         depends_on(checkoutservice_578fcf4766_9csqn,
34                     frontend_579b9bff58_t2dbm).
35         log(error, frontend_579b9bff58_t2dbm).
36         log(400, frontend_579b9bff58_t2dbm).
37         netMetric(low, frontend_579b9bff58_t2dbm).
38     }
39 ).
40 #pos(id1661141420@5,
41     {
42         fault(cpu_consumed, frontend_579b9bff58_t2dbm)
43     },
44     {
45         fault(cpu_contention, frontend_579b9bff58_t2dbm),
46         fault(return, frontend_579b9bff58_t2dbm),
47         fault(exception, frontend_579b9bff58_t2dbm)
48     },
49     {
50         depends_on(checkoutservice_578fcf4766_9csqn,
51                     frontend_579b9bff58_t2dbm).
52         cpuMetric(high, frontend_579b9bff58_t2dbm).
53     }
54 ).

```

Listing 15: Inductive Learning Program for the Frontend Service (Part 1)

```

1  #pos(id1661141915@10,
2      {
3          fault(exception, frontend_579b9bff58_t2dbm)
4      },
5      {
6          fault(cpu_contention, frontend_579b9bff58_t2dbm),
7          fault(return, frontend_579b9bff58_t2dbm),
8          fault(cpu_consumed, frontend_579b9bff58_t2dbm)
9      },
10     {
11         depends_on(checkoutservice_578fcf4766_9csqn,
12                     frontend_579b9bff58_t2dbm).
13         log(error, frontend_579b9bff58_t2dbm).
14         log(500, frontend_579b9bff58_t2dbm).
15     }
16 ).
17 #pos(id1661142423@5,
18     {
19     },
20     {
21     },
22     fault(cpu_contention, frontend_579b9bff58_t2dbm),
23     fault(return, frontend_579b9bff58_t2dbm),
24     fault(exception, frontend_579b9bff58_t2dbm),
25     fault(cpu_consumed, frontend_579b9bff58_t2dbm)
26 },
27 {
28     depends_on(checkoutservice_578fcf4766_9csqn,
29                 frontend_579b9bff58_t2dbm).
30     netMetric(low, frontend_579b9bff58_t2dbm).
31 }
32 ).

```

Listing 16: Inductive Learning Program for the Frontend Service (Part 2)

```

1  fault(cpu_contention, frontend_579b9bff58_t2dbm) :-
2      memMetric(high, frontend_579b9bff58_t2dbm).
3
4  fault(return, frontend_579b9bff58_t2dbm) :-
5      log(400, frontend_579b9bff58_t2dbm).
6
7  fault(exception, frontend_579b9bff58_t2dbm) :-
8      log(500, frontend_579b9bff58_t2dbm);
9      not log(error, checkoutservice_578fcf4766_9csqn).
10
11 fault(cpu_consumed, frontend_579b9bff58_t2dbm) :-
12     cpuMetric(high, frontend_579b9bff58_t2dbm);
13     not memMetric(high, frontend_579b9bff58_t2dbm).

```

Listing 17: Results of Inductive Learning for the Frontend Service

Fault Mode	Location	Top-1 Diagnosis	Top-3 Diagnosis
cpu_contention	frontend	(cpu_contention, frontend)	-
return	frontend	(return, frontend)	(return, frontend) (network_delay, currency) (network_delay, cart)
cpu_consumed	frontend	(cpu_contention, frontend)	(cpu_contention, frontend) (network_delay, productcatalog)
exception	frontend	(exception, frontend)	(exception, frontend)
network_delay	cart	-	-
cpu_contention	cart	(cpu_contention, cart)	(cpu_contention, cart)
network_delay	checkout	(cpu_contention, checkout)	(cpu_contention, checkout)
exception	checkout	(return, checkout)	(return, checkout)
return	checkout	(return, checkout)	(return, checkout) (network_delay, currency) (network_delay, cart)
cpu_consumed	checkout	(cpu_contention, checkout)	(cpu_contention, checkout) (cpu_consumed, checkout)
cpu_contention	currency	(cpu_contention, currency)	(cpu_contention, currency) (network_delay, productcatalog)
network_delay	currency	(network_delay, productcatalog)	(network_delay, productcatalog)
cpu_contention	email	(cpu_contention, email)	(cpu_contention, email)
network_delay	email	-	-
cpu_consumed	email	(cpu_contention, email)	(cpu_contention, email)
network_delay	payment	-	-
cpu_contention	payment	(cpu_contention, payment)	(cpu_contention, payment)
network_delay	productcatalog	(network_delay, cart)	(network_delay, cart) (network_delay, payment) (network_delay, email)
return	productcatalog	(return, checkout)	(return, checkout) (network_delay, productcatalog)
cpu_consumed	productcatalog	(cpu_consumed, productcatalog)	(cpu_consumed, productcatalog)
exception	productcatalog	(return, checkout)	(return, checkout) (cpu_contention, productcatalog)
network_delay	recommendation	-	-
cpu_contention	recommendation	(cpu_consumed, recommendation)	(cpu_consumed, recommendation) (cpu_contention, recommendation)
cpu_consumed	recommendation	(cpu_consumed, recommendation)	(cpu_consumed, recommendation) (cpu_contention, recommendation)
network_delay	shipping	(network_delay, cart)	(network_delay, cart) (network_delay, productcatalog) (network_delay, shipping)
cpu_contention	shipping	(cpu_contention, shipping)	(cpu_contention, shipping)
cpu_consumed	shipping	(cpu_contention, shipping)	(cpu_contention, shipping)
network_delay	ad	(exception, ad)	(exception, ad) (network_delay, productcatalog) (network_delay, cart)
return	ad	-	-
exception	ad	(exception, ad)	(exception, ad)
cpu_contention	ad	(exception, ad)	(exception, ad) (network_delay, productcatalog) (cpu_contention, ad)

Table A.1: Diagnostic Result from the Online Boutique Benchmark

```

1  fault(cpu_contention,S) :- memMetric(high,S).
2
3  fault(return,S) :- log(400,S).
4
5  fault(network_delay,S) :- not netMetric(low,S); depends_on(S, V1);
6                          netMetric(low,V1).
7
8  fault(cpu_consumed,S) :- cpuMetric(high,S); not memMetric(high,S).
9
10 fault(cpu_contention,frontend) :- memMetric(high,frontend).
11
12 fault(return,frontend) :- log(400,frontend).
13
14 fault(exception,frontend) :- log(500,frontend);
15                             not log(error,checkoutservice).
16
17 fault(cpu_consumed,frontend) :- cpuMetric(high,frontend);
18                             not memMetric(high,frontend).
19
20 fault(exception,adservice) :- log(error,frontend); not log(400,frontend);
21                             not log(500,frontend).
22
23 fault(cpu_contention,cartservice) :- cpuMetric(high,cartservice).
24
25 fault(return,checkoutservice) :- log(error,checkoutservice).
26
27 fault(cpu_contention,checkoutservice) :- cpuMetric(high,checkoutservice).
28
29 fault(cpu_contention,currencyservice) :- memMetric(high,currencyservice).
30
31 fault(cpu_contention,emailservice) :- memMetric(high,emailservice).
32
33 fault(cpu_consumed,productcatalog) :- cpuMetric(high,productcatalog).
34
35 fault(cpu_consumed,recommendation) :- cpuMetric(high,recommendation).
36
37 fault(cpu_contention,shippingservice) :- cpuMetric(high,shippingservice).

```

Listing 18: Rules Used in the Evaluation