AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FIELD OF SCIENCE: ENGINEERING AND TECHNOLOGY

SCIENTIFIC DISCIPLINE: INFORMATION AND COMMUNICATION TECHNOLOGY

DOCTORAL THESIS

Root Cause Analysis for Large-scale Cloud-native Applications

Author: Bartosz Żurkowski, M.Sc.

First supervisor: Professor Krzysztof Zieliński, Ph.D., D.Sc.

Assisting supervisor: Kazimierz Bałos, Ph.D.

Completed in: AGH University of Science and Technology,
Faculty of Computer Science, Electronics and Telecommunications,
Institute of Computer Science

Krakow, 2022

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

DZIEDZINA: NAUKI INŻYNIERYJNO-TECHNICZNE

DYSCYPLINA: INFORMATYKA TECHNICZNA I TELEKOMUNIKACJA

ROZPRAWA DOKTORSKA

Analiza źródeł defektów działania aplikacji dużej skali w chmurze obliczeniowej

Autor: mgr inż. Bartosz Żurkowski

Promotor rozprawy: prof. dr hab. inż. Krzysztof Zieliński

Promotor pomocniczy: dr inż. Kazimierz Bałos

Praca wykonana: Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie,
Wydział Informatyki, Elektroniki i Telekomunikacji,
Instytut Informatyki

Kraków, 2022

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Krzysztof Zieliński, for his helpful support, guidance, and motivation in carrying out the research described in this dissertation.

I also express heartfelt gratitude to my family and friends for their patience, understanding, and support.

The research was part of the "Industrial Ph.D." program developed in cooperation with Samsung R&D Institute Poland.

Table of Contents

Chapter 1

Introduction

This chapter introduces the research problem considered in this dissertation. Followed by presenting the problem motivation, the dissertation thesis is formulated. Then, the research scope and assumptions are defined. Finally, the chapter discusses the dissertation contribution and outlines the organization of subsequent chapters.

1.1 Motivation and Thesis Statement

In the last decade, microservices architecture has become the mainstream for building enterprise applications. Promoted as a cost-effective and efficient solution capable of addressing business complexity and scale, it dominates as a remedy for expanding business needs, advancing user load, and more challenging demand for quality of service. Moreover, the current needs of end users show that they crave higher performance and reliability of offered services entailing more extensive distribution and complexity of newly created IT systems than ever before.

Indeed, over the years, modern cloud applications built around principles of microservices and service mesh architectures evolved into large distributed systems consisting of hundreds of services deployed across geographically dispersed clusters. They comprise independent technology stacks (i.e., polyglot databases, frameworks, programming languages), use mixed communication protocols, and integrate with numerous cloud services (object storage, message queues, load balancers). They also get managed by advanced orchestration platforms functioning on top of the multi-layered cloud infrastructure. Due to the growing complexity, they are more and more challenging to observe and maintain.

Notably, the volume and diversity of telemetry data generated by these systems increase exponentially, making their practical analysis a critical challenge. Furthermore, the problem is aggravated by the number of dependencies between system elements, their interference across cloud layers, and the system evolution rate derived from continuous integration, auto-scaling, and self-healing mechanisms that are commonly applied in the cloud.

Due to the coupling between system elements, fault in a single point often triggers cascading fault propagation throughout its neighborhood. Typically, fault trajectories span multiple cloud layers and impact hundreds of elements, each of which yields an alert that must be analyzed by an operator. Manual inference of the primary fault from hundreds of reported symptoms in a large system - the root cause of system failure - is unmanageable considering the amount of telemetry data to inspect and the number of symptom dependencies to recognize.

Increasingly, operators cannot timely localize the root causes of failures emerging in large-scale enterprise systems. That leads to frequent quality-of-service violations constituting a barrier to achieving predictable performance guarantees and often comes with significant cost implications.

Observability tools in regular use are insufficient to effectively diagnose the root causes of failures. They do not aggregate complete information about the system behavior, nor do they implement algorithms to correlate them and give valuable guidance to an operator. Moreover, essential data is often spread between many telemetry sources. Consequently, operators must manually switch between several dashboards, filter out relevant information, and make prompt conclusions based on their system knowledge.

Considerable academic and industrial efforts were made to address the root cause identification in large-scale systems. Many of the proposed solutions leverage model-based, statistical, and machine-learning techniques and define numerous RCA problem dimensions. However, existing solutions concentrate on solving the problem in narrow domains that do not overlap with the cloud-native application area or ignore essential requirements resulting from the cloud-native application characteristics such as fault analysis across multiple cloud layers, isolation of parallel faults, or analysis transparency.

Thus, modern applications at scale pose a challenge of inventing an approach capable of providing a holistic insight into the multi-layered application tier with algorithms conducting root cause analysis (RCA) of failures with a clear failure explanation to an operator. Efficient analysis of system behavior becomes the key enabler to early or even proactively remediating system performance degradation and preventing failure exposure to end-users.

Based on the above motivation, a thesis is formulated:

> *With the synergy of symptom correlation methods and the continuous gathering of system structural and behavioral knowledge, it is possible to build an RCA solution capable of automatically identifying sources of complex defects emerging in large-scale cloud-native applications.*

The dissertation attempts to confirm the thesis statement by defining RCA system requirements and proposing the concept of root cause analysis for cloud-native applications. The concept followed by its realization emphasizes essential aspects of cloud-native application analysis. Furthermore, the dissertation comprises the implementation of a solution prototype and its comprehensive evaluation against failure scenarios in a dedicated cloud-native testbed.

1.2 Scope and Assumptions

The analysis of issues emerging in IT systems covers two aspects of behavioral system inspection, i.e., anomaly detection and root cause identification. Anomaly detection discovers system deviation from its normative state, while root cause identification aims to locate a system component and corresponding anomaly that initiated propagation of system failure across adjacent components. Due to the blurred boundary between these two analysis aspects addressed in the literature, a clear definition must be established for that research problem dimension.

This dissertation focuses solely on the root cause identification problem. Anomaly detection is not included in the dissertation scope. In other words, it is assumed that the implementation of the research study and evaluation of the proposed solution should use existing solutions for anomaly detection, if necessary. Moreover, the system subjected to analysis is assumed to expose an observability plane that enables reading metrics describing system operation and provides alerting mechanisms warning detected anomalies.

Further, cloud-native applications are the main subject of root cause identification assumed in this work. Thus, the developed solution should strictly analyze the structure and behavior of applications compliant with cloud-native properties. The analysis of application operation should be conducted from a holistic perspective. In particular, elements from essential layers of the multi-tier cloud infrastructure should be examined. At the same time, the developed solution must be flexible to an extent enabling its use in many cloud-native application scenarios.

Moreover, it is assumed that considered cloud-native applications function in a system environment of gray-box characteristics. The implication of this is the lack of ability to instrument or alter application code to collect additional facts for the root cause analysis. Instead, partial system knowledge can be obtained using interfaces exposed by system management and observability planes.

The final assumption established in the dissertation is minimizing the involvement of human administrators in the analytical process, i.e., the proposed root cause analysis solution should be able to produce a valuable problem diagnosis without requiring significant expert supervision.

1.3 Thesis Contribution

The main contributions of this dissertation are as follows:

- Definition of root cause analysis requirements for cloud-native applications established based on identified problem dimensions and review of common root cause analysis techniques.
- Critical analysis of the current state-of-the-art related to the subject of root cause identification for cloud-native applications. The analysis confronts existing solutions against defined RCA requirements. Important solution capabilities and limitations are identified and used as a foundation for proposing a new approach to the research problem.
- The abstract concept of root cause analysis decomposing the analysis problem into a high-level inference process comprising critical RCA solution elements. The concept outlines the boundaries of the RCA model and inference algorithm. Furthermore, it introduces a novel methodology for approximating causal symptom dependencies by consolidating selected symptom correlation methods.
- Realization of the symptom correlation framework that covers the elaboration of symptom correlation methods and outlines the process of causal symptom dependency approximation. Further, the framework realization provides the answer to the problem of temporal symptom consistency by proposing an improvement to symptom timestamp identification using changepoint detection techniques.
- Realization of the RCA model that submits a set of structures providing a holistic view of system structure and behavior, including fault symptom ingestion and symptom semantics. In addition, the realization presents a formal definition of algorithms that enable the construction of individual model structures adaptively.
- Realization of the inference algorithm. The algorithm utilizes the symptom correlation framework to approximate causal symptom dependencies based on system knowledge obtained from the RCA model. In addition, the algorithm contributes by implementing a methodology for estimating failure causes and effects, extracting fault trajectories, and ordering trajectories according to scoring criteria.
- Implementation of solution prototype covering the complete fault reasoning process, including RCA model construction and inference algorithm stages.

Developed using modern cloud-native technology and rich integration with cloud observability tools, the prototype proves the feasibility of creating a reference RCA system implementation from the solution concept.

- Functional evaluation of implemented solution prototype in a dedicated cloud-native testbed. The evaluation ensures the correctness of methods employed in the inference algorithm and confirms that the solution can be successfully applied to real-life application scenarios.

1.4 Dissertation Organization

This dissertation is organized in the following manner. Chapter 2 presents background aspects related to the dissertation scope. It characterizes cloud-native applications and introduces the RCA system abstraction. Moreover, it defines root cause analysis requirements for cloud-native applications and exhibits the limitations of existing solutions in the literature review. Further, Chapter 3 introduces the abstract concept of root cause analysis for cloud-native applications. The concept outlines the boundaries of important RCA system elements, i.e., symptom correlation framework, RCA model, and inference algorithm. Following the overview of the solution concept, the contribution of subsequent chapters focuses on the realization of individual concept elements. First, Chapter 4 presents the realization of the symptom correlation framework. It covers selected symptom correlation methods and organizes the process of consolidating correlation results into causal symptom dependency approximation. Second, Chapter 5 discusses the realization of the RCA model, taking into account structures representing the topology and semantics of the analyzed system. In addition, it explains algorithms enabling the automated construction of model structures. Third, the realization of the inference algorithm is described in Chapter 6. It formalizes subsequent stages of the fault inference process, including integrating symptom correlation methods, constructing the fault view causality graph, estimation of failure causes and effects, fault trajectory extraction, and ordering trajectories according to adopted scoring criteria. Chapter 7 details the implementation of the solution prototype. In particular, it motivates the selected cloud-native technology stack and explores the object-oriented decomposition of the analysis problem. The evaluation of the implemented prototype is presented in Chapter 8. Finally, Chapter 9 concludes the dissertation contribution and discusses possible directions for future research.

Chapter 2

Background and Related Work

Prior to explaining the concrete solution concept and its implementation, several research aspects have to be tackled in the context of technological background and related work. First of all, cloud-native applications are characterized and debated in the context of potential sources of defects and difficulties imposed on the root cause analysis problem. Further, existing root cause identification techniques are reviewed and confronted against established analysis dimensions. Together with the review of existing analysis approaches, the dimensions constitute a basis for defining RCA challenges for cloud-native applications relevant to the dissertation scope. Finally, related literature is analyzed to prove the novelty of the proposed solution concept.

2.1 Cloud Native Applications

Cloud-native applications deployed in the multi-layered cloud environment are the primary subject of failure analysis considered in this dissertation.

The term *Cloud-Native Application* (CNA) dates back to the birth of the first public cloud providers such as Amazon Web Services (2006) or Google Cloud Platform (2008) [1]. At that time, it posed a question concerning architectural and software design patterns that must be applied to existing applications to leverage the elastic and dynamic nature of cloud resources. Over the last decade, with the emergence of new cloud services and the systematization of design techniques, the term accumulated answers to fundamental questions and gradually evolved into a more specific and ubiquitous form. Ultimately, it converted into a set of rules establishing the state-of-the-art guideline for building cloud applications focusing on scalability, performance, and reliability.

Referring to the comprehensive survey conducted by Kratzke et al. [1], a CNA can be defined as a distributed, elastic, and horizontally scalable system decomposed into loosely coupled microservices that isolates state in a minimum of stateful components. Furthermore, each component should be self-contained, designed according to cloud-focused design patterns, and operated on a self-service elastic cloud platform. Similarly, Fehling et al. [2] summarize CNA with the acronym IDEAL describing that applications should have [i]solated state, be [d]istributed into multiple components, be [e]lastic in terms of horizontal scaling, have [a]utomated life-cycle management, and be [l]oosly coupled, i.e., application components must not influence each other in terms of availability, scaling, design and technological stack.

Following the general definition of CNA, individual application properties are presented and discussed in the context of challenges introduced to the problem of root cause analysis.

Microservices decomposition Due to expansive business complexity, advancing user load, and more challenging demands for quality of service, enterprise cloud applications are being increasingly decomposed into loosely coupled microservices [3]. Each microservice builds around a distinct business capability and takes responsibility for a subset of processes in the narrowed business domain. Fragmenting an application into smaller services improves its modularity, affects the ease of understanding, and enables independent development, testing, deployment, upgrade, and scaling of individual application components by autonomous teams [4, 5]. In addition, the fine-grained structure of microservices enables scaling the application more efficiently by scoping the scaling operation to only parts experiencing a heavier load.

Decomposition of monolithic applications to microservices architecture brings many benefits and is often perceived as a mandatory remedy for avoiding application performance and availability issues. However, at the same time, the transition introduces extensive complexity emerging from the architectural characteristics of a distributed system. Notably, the introduced complexity poses additional challenges for failure analysis.

Unlike failure analysis in monolithic applications, in the case of microservices, diagnostic data is distributed across many independent service units and concerns related components located in underlying infrastructure layers. The data must be orderly collected from each element involved in causing or being affected by the failure and then logically and chronologically correlated to estimate the failure root cause. Due to the potentially large number of microservices and the fact that they may be scaled to hundreds of instances [6], the data collection and correlation require integrating advanced observability and statistical approaches [7]. Moreover, complex service communication paths and varying communication characteristics create a large space of possible fault trajectories. As a result, discovering service dependencies and inferring valid causal relations requires precise system observation and analysis over extended periods.

Distinct technological stacks Each component in a microservice-based application can be developed based on a distinct technological stack adapted to its function. Components can use a variety of programming languages, runtimes, frameworks, and design techniques and can employ independently selected cloud platform elements such as databases or queuing systems. The stack flexibility enables choosing the right technology options to meet the performance requirements of a component and support the structure of its data model [5]. For instance, performance-critical components can be implemented using efficient programming languages such as C++ or Go, while other components can be implemented using Python or Ruby that, although they offer lower performance, require less development effort. Similarly, cloud applications are polyglot in terms of data persistence. They use database technology appropriate for the model structure and the required performance of data operations. An application can simultaneously utilize several database technologies, e.g., relational databases for tabular data with strong integrity constraints, document stores for schemeless structures, and graph databases for modeling complex associations between entities.

Technological diversity aggravates the complexity of failure diagnosis. Each programming language, framework, or runtime selected into the technology stack creates a potential source of software defects that may impact application performance

and behavior. An error introduced in the latest framework version or the use of a backward-incompatible library interface may cause the application to respond with errors upon user requests. More critically, a memory leak caused by a corrupted application runtime may result in a gradual memory overfilling until the operating system crashes or terminates the application. Given the number of microservices and differences in used programming environments, defects due to errors in dependency software are expected to occur at a high frequency. More importantly, they require dedicated teams specializing in these programming environments to detect and address failures effectively.

A similar problem involves platform elements such as databases, load balancers, or queuing systems. Each element may introduce software defects that cause a negative change in its functioning. Errors may also result from incorrect or sub-optimal settings in configuration parameters. Moreover, some platform elements, e.g., databases, are deployed in many instances, i.e., clusters, causing further fragmentation of the failure domain.

State isolation Statefulness is the key factor influencing the design of a distributed system. Prominently, cloud-native applications aim to isolate the entire state in a minimum number of stateful components. A stateful component stores an intermediate state with each processed transaction and requires the accumulated state to process subsequent ones. Typically, The state is critical for component runtime, and as such, the component is sensitive to changes in its life cycle. Sudden removals or restarts of a stateful component may cause permanent failure, data loss, or irreversible termination of processed transactions. Proper handling of the life cycle requires time-consuming data synchronization and often reelecting components with specialized roles to form a data cluster.

Due to the dynamic nature, cloud applications, where services are massively scaled and transferred between machines at high frequency, cannot be composed of too many stateful components. Therefore, most application parts are designated for stateless transaction processing. Their state is separated into a smaller group of stateful components, typically databases, which are not subjected to intense life-cycle changes and implement appropriate state handling mechanisms. Stateless components use atomic transactions to store required information and are designed to recover after failures or restarts seamlessly.

Strict state handling requirements and the technological diversity of microservices result in the emergence of many heterogeneous database components in a system. This phenomenon, known as polyglot persistence [8], has been observed to translate into an increase in operating expenditures. Each database engine introduces a

specific administrative and failure domain requiring expertise in performance tuning, data modeling, and failure recovery. Such expertise is not common to all database technology, and typically several database experts are required to maintain the persistence layer.

Regarding failure analysis, each database technology defines a dedicated set of metrics and configuration parameters that must be inspected. They may cause errors of different nature and symptoms that require a distinct diagnosis strategy [9]. Furthermore, databases are scaled to hundreds of nodes and set up into mixed topologies to support the required throughput, capacity, and data availability. That, in turn, poses a challenge in analyzing a broader database context, taking into account dependencies between database nodes, data transfer paths, and roles of individual cluster members.

Containerization Applications deployed in the cloud are expected to respond dynamically to varying runtime conditions such as increased user traffic or failure of a physical host. Therefore, to control variable load and handle failures gracefully while guaranteeing an optimal cost model, it is crucial to ensure that new instances of application components start with a minimal time overhead and minimal use of system resources. In addition, due to technological diversity and the need for fast and frequent delivery, applications must be compact and highly portable.

Light virtualization technology implementing isolation at the operating system level has dominated the ecosystem of cloud runtime environments, with containers constituting the standard workload unit. Containers are self-contained by packaging all application dependencies into a lightweight binary image that can be easily transferred between machines [10]. Further, as isolated userspace processes, containers share the kernel of a host operating system driving higher performance and application portability across system platforms. Finally, they start and stop in milliseconds satisfying scale-in and scale-out responsiveness required in the cloud.

Containers introduce required dynamics to the life-cycle of application components, but at the same time, containerization affects the failure analysis. Specifically, rapid starting and stopping of containerized component instances or transferring them between machines imply a continuous change in application structure. That, in turn, makes it difficult to track failure trajectories as abnormal component instances can be quickly replaced with new ones. Thus, information about abnormal component instances is not always complete as it may get partially lost or unobserved.

Orchestration Since many modern enterprise applications are being decomposed into independent microservices and deployed as containerized workloads scaled to hundreds of containers, automated container orchestration becomes a critical element

of efficient application management [11]. The primary responsibility of a container orchestrator is scheduling application containers to available physical nodes based on node capacity and affinity or resource policies. In addition, the orchestrator supervises the container life-cycle to preserve the defined application state.

The presence of an orchestrator enhances application dynamics. In particular, auto-healing and auto-scaling mechanisms built into cloud platforms make the rotation of component instances more frequent [12, 13]. Moreover, most orchestrator interventions are transparent for application users and administrators. Consequently, failure fragments are addressed automatically, blocking the emission of some symptoms that are essential for determining the root cause. Thus, the analysis must assume that some failure information may be missing, and the root cause itself may not be disclosed due to mechanisms of system self-organization.

CI/CD Cloud applications are built in accordance with agile techniques, whose primary goal is to provide small increases in functionality continuously and evaluate them on an ongoing basis, preferably in staging and production environments [14]. Each change in the application code triggers integration and deployment cycles, during which the application is tested, its artifact and configuration are built, and a new application version is deployed into the remaining part of the system using continuous deployment methodologies such as Blue/Green deployments or Canary releases. The entire process is seamless and fully automated to offload developers from the burden of manual testing and enable them to focus more on software development.

Continuous integration and deployment constitute another facilitator of cloud-native application dynamics. They enable the ordered deployment of subsequent application versions, implying frequent application structure and behavior changes. At the same time, continuous deployment is often one of the first failure triggers. If the software is undertested or contains a hidden bug, defects get revealed soon after installing the new application version.

Cloud infrastructure Cloud-native applications are designed to take advantage of elastic cloud resources [15]. They utilize compute, storage, and network resources abstracted from the underlying hardware, hypervisors, storage backends, and network controllers. Launched as microservices running inside virtual machines or containers and communicating using a private virtual network, they use virtual storage volumes and employ additional cloud services such as load balancers, DNS, or security groups. All required infrastructure elements can be provisioned on-demand using well-defined cloud APIs and optionally preserved in the form of high-level descriptors supporting the Infrastructure as a Code paradigm.

Running an application within the cloud implies multi-layered architecture that increases the variety of elements that may be subject to failure [16]. Specifically, each layer provides a set of elements, dependencies, and failure symptoms that must be recognized to identify the failure context and form a complete fault trajectory. Dependencies between elements and symptoms across layers are difficult to determine and evaluate semantically.

2.2 RCA Concepts and Terminology

This section discusses the main terminology and concepts used in the domain of root cause analysis. In particular, essential terms and abstractions are provided that will be used throughout the dissertation.

2.2.1 Basic Terminology

This dissertation employs terminology commonly used in the fault management field [17, 18, 19]. The most essential terms are defined in the following paragraphs.

Event An *event* is an exceptional condition occurring in the operation of a system. It concerns the deviation of the system from its normative state or denotes specific system operations.

Fault A *fault*, also referred to as *problem* or *root cause*, is an event that can cause other events but is not itself caused by other events. Faults can be categorized by duration as *permanent*, *intermittent*, or *transient*. A permanent fault is a severe system degradation that persists until system administrator intervention. Intermittent faults constitute discontinuous and recurring system degradation. In turn, transient faults cause temporary and minor system degradation. Typically, intermittent and transient faults are remediated automatically by platform orchestration mechanisms.

Effect Contrary to a fault, an *effect* is an event that is caused by other events but does not itself cause other events. Typically, effects reflect faults in the upper system layers.

Error An *error* is a discrepancy between the observed system state and the correct state that is expected from the theoretical perspective. Errors are a consequence of system faults. Notably, faults may cause one or more errors. Errors, in turn, may cause subsequent errors in dependent system regions. Errors do not necessarily translate into externally visible system degradation, and they often do not require special intervention.

Failure A *failure* is an error that is observable from outside the system.

Symptom A *symptom* is an external manifestation of failures observed in the form of alerts generated by the system observability plane.

Fault trajectory A *fault trajectory* is an ordered sequence of symptoms observed in a system. It connects symptoms manifesting the root cause and effect of failure. As such, the trajectory represents a failure explanation.

2.2.2 Abstraction of RCA System

Most RCA systems share a similar diagnostic workflow illustrated in the conceptual diagram of Figure 2.1 [17]. According to the presented abstraction, the workflow can be decomposed into two indispensable parts, namely, the *RCA model* and *inference algorithm*.

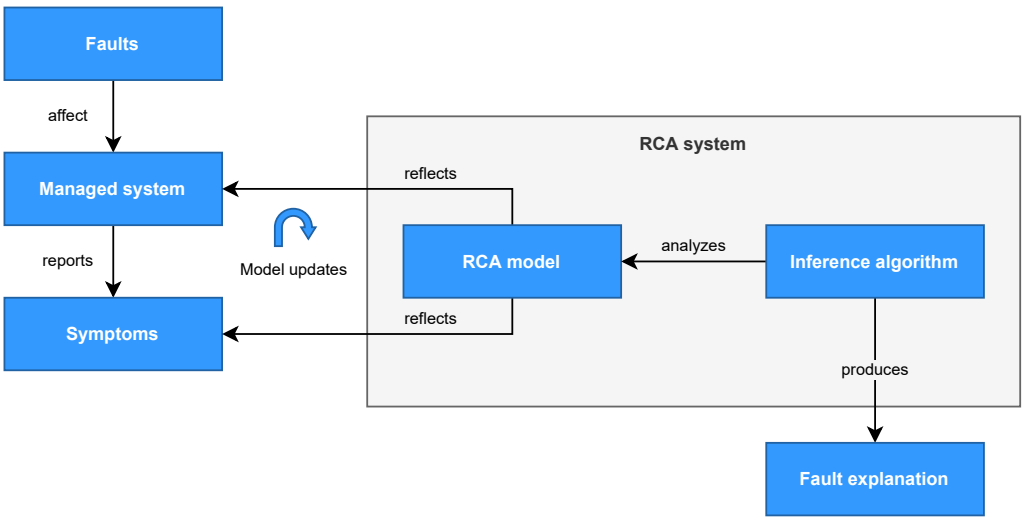


Figure 2.1: RCA system abstraction

The RCA model is a structural and behavioral reflection of the managed system. It encompasses structures and interfaces ensuring access to essential data required in the inference process. In a sense, the model represents the knowledge of the system administrator that would be normally used in manual failure diagnosis. The model is constructed based on domain knowledge provided by domain experts and complemented with knowledge accumulated by the RCA system by system observation. Often the RCA model results from the synergy of data collected from multiple system information sources. Their proper acquisition and correct processing pose one of the main challenges of model maintenance.

Further, the inference algorithm is responsible for conducting the diagnosis based on the system knowledge accumulated in the RCA model. As such, the RCA

model constitutes a prerequisite and input to the inference process. The purpose of the inference algorithm is to discover the fault that generated the observed set of symptoms. Depending on administrative system requirements, the diagnosis may be limited to identifying the root cause. However, a complete failure explanation may be required in some cases, i.e., a cause-effect sequence of symptoms indicating the propagation path from failure root cause to failure effect.

2.2.3 Classification of RCA Dimensions

Due to the large number of domains requiring root cause analysis and the extensive space of failures of different characteristics, many criteria emerge, based on which the classification of RCA techniques can be made [17]. This section summarizes the most relevant dimensions affecting the nature of the root cause analysis.

Analysis objective The first important RCA system dimension is the *analysis objective*. Depending on administrative requirements, it may be acceptable to indicate only a set of root cause candidates. Based on the root cause, the system administrator can quickly remove the fault at a critical system point and, consequently, stop further failure propagation. On the other hand, root cause information may be insufficient as effects of failure propagating throughout the system may have caused severe damage and violation at several system locations. In such a case, it is crucial to analyze the complete fault trajectory, identify system regions affected by the failure, and apply remediation strategy individually for each system component. The RCA output in the form of fault trajectory constitutes a failure explanation that allows estimating failure scope and evaluating system response at each component affected by the failure.

Analysis accurracy The *analysis accuracy* represents the criterion deciding whether the inference algorithm returns the exact or approximate failure cause. The exact result produces a cause whose manifestation is scoped to a concrete system component or anomaly constituting the correct origin of system failure. Conversely, the inference algorithm may return an approximate root cause or a hint based on heuristics employed to reduce the search space. In addition, some algorithms use iterative methods that converge to correct results with time.

Analysis time Another dimension is the *analysis time*. It is a non-functional criterion determining the maximum amount of time the inference algorithm can consume conducting the analysis based on available data. Typically, within this criterion, two algorithm categories are distinguished. First, *online* algorithms are dedicated to systems where the time of root cause identification is critical. An

algorithm in this category must generate a fault diagnosis proactively or within seconds while the system transits or shows a transition towards the failure. Online diagnosis is challenging, especially when the system subjected to analysis is complex while the volume of data needed to perform an accurate diagnosis is limited. Thus, it is impractical to provide an online diagnosis in some cases. In order to speed up calculations, online inference algorithms use heuristics or require helper structures comprising precomputed parts of the inference process.

Offline algorithms constitute the second algorithm category. They are intended for systems with a fault tolerance that allows diagnosing the failure without strict temporal constraints. More precisely, offline algorithms can spend additional time collecting diagnostic data and analyzing more failure aspects. Consequently, they often yield better analysis accuracy.

System size Further, *system size* determines the number of components in the system and the number of semantic dependencies between them. This criterion is essential in building and maintaining the RCA model, as building the model for a large system requires significant time and memory resources.

Fault trajectory length *Fault trajectory length* is a derivative of the system size, system structure complexity, and failure semantics. The criterion specifies the maximum number of components involved in the failure, forming a path between failure root cause and failure effect. The path length depends on the number of system abstraction layers and the susceptibility of horizontal failure propagation at each layer.

Effect propagation time Failure propagation between subsequent system components may show different temporal dependencies. The *effect propagation time* between a components pair, i.e., the amount of time between subsequent symptoms recorded by the observability plane, may vary depending on many factors, such as failure type, system dynamics, or component neighborhood. The propagation time is also related to symptom observability and reporting efficiency, e.g., telemetry data sampling.

System evolution rate The subsequent criterion, *system evolution rate*, describes the frequency and extent of changes occurring in the diagnosed system. The more often the system changes and the more system components are involved in a change, the more challenging it is to reflect system information about system components and their dependencies in the RCA model. Some systems have a low evolution rate, such as hardware systems. Others, in turn, are constantly changing, e.g., a self-organizing distributed IT system controlled by orchestration mechanisms can experience hundreds of changes every minute. Importantly, the evolution intensity

depends on the model granularity. For instance, the human body is an approximately static system at the level of organs or bones, while it is highly dynamic at the cellular and systemic levels.

Parallel faults Some systems tend to be vulnerable to *parallel faults*. In particular, if the system is complex and has many possible fault vectors, it is not uncommon for more than one system component at the same time to be a failure source. Faults can be semantically dependent or independent. The criterion decides whether a fault management system should isolate parallel faults, determine individual root causes and provide distinct failure explanations.

Knowledge availability Building an RCA model and conducting a practical analysis requires *knowledge availability*. Domain knowledge, including system purpose, building blocks, and primary system operation rules, can be provided by domain experts. However, additional system knowledge about the system structure and semantics is often needed. In such a case, it must be obtained directly from the system. Depending on the level of knowledge availability a given system offers, systems can be divided into *black-boxed* and *white-boxed*. The former group does not provide access to system knowledge as system components function in a closed manner and do not expose interfaces enabling reading the required information. The latter group, in turn, guarantees access to all internal information, often with the possibility to alter the system to possess additional facts. Practically, in terms of knowledge availability level, systems range between black-box and white-box and, as such, are referred to as *gray-box* systems.

2.2.4 Classification of RCA Techniques

Following the elaboration on root cause analysis dimensions, this section categorizes leading analysis approaches. Subsequent paragraphs define key properties and limitations of individual method categories.

Rule-based techniques Rule-based techniques rely on expert knowledge embedded in rules that can be used to make problem-related deductions and choices. Rules are maintained in the form of symptom matching conditions, named *premises*, followed by corresponding problem conclusions.

Many legacy fault management systems, such as IBM Tivoli or HP OpenView [20], are based on rules defined manually based on domain expert experience, service desk responses, or recording resolutions of new failures.

Codebooks are the dominant rule-based inference method [21]. They map problem signatures, i.e., sets of known symptoms, to specific problem causes. Typically,

codebooks are implemented as a matrix where rows represent observable symptoms, whereas columns represent problems. Codebooks diagnose problems by finding the closest symptom match across matrix rows and identifying the column with the predefined result.

Rule-based systems are characterized by fast inference and, to some extent, the ability to conduct diagnoses based on incomplete data. However, a significant limitation of rule-based methods is their inability to self-adapt and learn from new experiences. In particular, rule-based diagnosis cannot deal with problems not covered by available rules, even if problems are insignificantly different from those written in the rules.

Model-based techniques Model-based techniques operate on a mathematical representation of the managed system. They are constructed based on domain knowledge and have arbitrary semantics dependent on the selected inference approach. Typically, techniques in this category employ graph structures as graphs constitute a natural reflection of systems in terms of their taxonomy. In addition, the graph structure guarantees the efficiency of exploring dependencies between model elements which constitutes the essential part of the inference.

The model may take the form of a *dependency graph* [6, 22, 23, 24, 25, 26, 27, 28] representing system components and dependencies important from the analysis perspective. By following the dependencies, starting with the component that triggered the first observed symptom, the inference algorithm can identify correlations with symptoms emitted on adjacent components and thus determine the fault propagation. An interesting enhancement to this method is combining fault localization with testing. More precisely, during the model traversal, individual components affected by failure are checked to clarify their operational state and collect additional diagnostic information.

A particular group of model-based techniques uses the *failure propagation model* (FPM) to represent failure propagation throughout the system [29, 30, 31, 32]. Instead of focusing solely on the system structure, FPM comprises a *causality graph* consisting of known symptoms and cause-effect symptom dependencies with optional edge labels describing the probability of causal implication. Explanation of symptoms observed in emerging failures is produced based on mapping symptoms into FPM nodes and inspecting symptom dependencies in the graph.

Model-based techniques encapsulate semantic system knowledge, allowing deterministic localization of failure causes. Contrary to rule-based techniques, the model reflects failure propagation, thus strengthening the explainability element of root cause analysis. Moreover, the model is partially adaptable in determining fault trajectories, i.e., concrete failure propagation paths are not defined in advance but

designated dynamically by the inference algorithm based on the dependency knowledge enclosed in the model. At the same time, model construction constitutes a substantial challenge as it requires a deep understanding of system structure and behavior. In many cases, the manual building of the model is impractical and requires automation which is not always feasible.

An important limitation of model-based techniques is the level of diagnosis detail. The diagnosis determines a system region where the root cause is located or indicates a system component that reported the root cause symptom. However, reported symptoms often do not indicate exact system errors. Instead, they manifest observable error effects in recognized system abstraction layers. In other words, the diagnosis depth is limited by observability plane capabilities and the level of detail assumed in the model. For instance, the model can localize a problem to a single hardware node but cannot determine the deeper root cause, like a specific error in the kernel of an operating system.

Statistial techniques Statistical approaches dominate the literature on failure diagnosis. They find use in many system categories, including black-boxed systems, as they rarely require domain knowledge. Instead, they analyze empirical data, usually obtained from the observability plane, using tools such as correlation, outlier detection, and elements from probability theory. As data-centric methods, they need a considerable volume of input data to produce accurate results.

Statistical correlation analyzes historical system data to find dependencies between metrics or events generated by the system. For instance, Pearson and Spearman correlation methods are used to find relationships between system metrics based on analyzing simultaneous changes in linear trends of corresponding time-series [33, 34]. Further, event correlation methods based on time windows or probability distribution estimation are used to discover temporal event dependencies [35, 36, 37, 38, 38, 39, 40, 41].

Discovered correlations between pairs of metrics or events are used to hypothesize causal dependencies. However, it must be emphasized that correlation does not imply causality. Thus, additional statistical methods and domain knowledge are required to confirm obtained results.

The main difficulty imposed by statistical techniques is the necessity of collecting and processing large datasets. Otherwise, they yield inaccurate or invalid results, which may negatively impact the output of the root cause diagnosis. Typically, collecting sufficient data on system failures takes much time, especially if failures are expected to emerge in natural conditions. Further, it is crucial to validate and normalize the data before processing.

Moreover, since statistical methods do not incorporate semantic knowledge about the observed system, they often cannot differentiate accidental dependencies from dependencies resulting from actual failures. Thus, it is challenging to confirm causal dependencies based on discovered correlations.

Machine-learning techniques Machine learning (ML) is a computer science discipline studying algorithms that can improve themselves automatically by processing solution samples. ML algorithms build models based on sample data, known as training data, to tune model parameters so that the model returns correct solutions for new problem instances. ML-based methods are data-centric and extensively use elements from the statistics field.

ML algorithms are commonly categorized into two groups, namely *unsupervised learning* and *supervised learning*. Algorithms from the former group identify patterns in unlabeled data through clustering using a selected distance measure. For instance, observed symptoms can be translated into vectors, grouped into clusters based on vector similarity, and associated with corresponding root causes localized using other diagnostic methods [42]. Then, the inference algorithm can analyze new problem instances by matching observed symptoms to formed clusters and reading the associated root cause.

The latter algorithm category, supervised learning, uses labeled data to train model parameters. The data comprises samples, each of which includes an input vector and corresponding output label. For instance, the model can be trained with failure signatures labeled with root causes from a database of known system problems [43]. Subsequently, the model enables classifying new failure instances to learned problem causes.

ML methods can learn the system operation profile automatically by processing training datasets. Moreover, to a large extent, they can deal with incomplete data and new failure types that were not diagnosed in the past. On the other hand, ML methods require large training sets to build an efficient model. Significant changes in system behavior require additional data samples and time-consuming model retraining. Furthermore, ML techniques suffer from the curse of dimensionality, affecting diagnosis performance and accuracy. Hence, dimensionality may be a blocking factor in complex systems with a high evolution rate. ML models are also prone to overfitting and underfitting, i.e., a situation in which they cannot adequately capture the underlying structure of the data resulting in oversensitive or too generic results conditioning. In root cause analysis, ML methods offer a low failure explainability level as they directly classify a given set of symptoms to the root cause, ignoring aspects of intermediate system structure and semantics.

2.3 RCA for CNA Solution Requirements

According to RCA system dimensions presented in Section 2.2.3, RCA system specialized in diagnosing causes of cloud-native application failures must identify both failure root cause and failure explanation that are supportive for the system administrator. At the same time, to enable effective fault remediation, the inference algorithm must timely produce accurate results scoped to specific system components. Otherwise, the system is vulnerable to service degradation, resulting in severe revenue losses. For instance, according to Ibidunmoye et al. [44], Amazon experiences a 1% decrease in sales for an additional 100 ms delay in request response time, while Google reports a 20% drop in traffic for a 500 ms delay in response time.

Further, the size of cloud systems can be measured in millions of components, while the fault trajectory length may range from one to tens of elements. Inter-service communication significantly impacts the trajectory length dimension, especially when the application is decomposed into many microservices. To visualize the problem scale, Thalheim et al. [7] report that Netflix and Uber record 20 and 500 million metrics, respectively, across hundreds of components scaled to thousands of instances. Likewise, Wang et al. [6] indicate that the monitoring system at eBay generates 20 million metrics from 5000 services.

Moreover, the cloud environment is characterized by a high evolution rate due to the employment of numerous orchestration, continuous delivery, and chaos testing mechanisms. The extent and frequency of application changes imply the high likelihood of parallel faults occurring within the same system regions.

Finally, it must be assumed that applications running in a cloud environment operate in the gray-box model. Thus, although obtaining knowledge through well-defined cloud platform interfaces is possible, application code cannot be instrumented to gather additional facts at the software level.

In addition to the above RCA problem dimensions, the following paragraphs highlight RCA system requirements imposed directly by the cloud environment and cloud-native applications. These requirements constitute the main criteria for related research discussion and make the foundation for introducing the solution concept.

Fault analysis across multiple cloud layers Deployment of enterprise applications in the cloud implies multi-layered architecture imposed by the complexity of modern cloud infrastructure. Layers define successive levels of abstraction, enabling effective use of cloud resources and flexible adaptation of technology stack depending on the business use case. Upper layers depend on the implementation and perfor-

mance of lower layers. Hence, it is not uncommon for defects emerging in lower layers to heavily affect upper layers. For instance, an anomaly in the CPU operation at the hardware layer may translate into response latency in a user-facing service at the application service layer.

Therefore, to make accurate diagnoses, a fault management system must integrate fault diagnosis across multiple layers of the managed system. That goal is difficult to achieve because symptoms propagating across layers originate at mixed component classes and differ semantically, i.e., they have meanings associated with specific system subdomains and are described by means of distinct terms. Furthermore, it is challenging to construct an RCA model that reflects the complexity of cloud applications at each complexity layer with a balanced amount of detail. Notably, mining valid component dependencies across layers is not always possible due to the limited amount of information provided by the system.

Existing approaches concentrate on fixed diagnosis contexts. Typically, they consider correlations between anomalies observed in application services or hardware resources. However, narrowing the root cause identification down to a single layer poses a risk of ignoring essential failure aspects and producing inaccurate diagnoses.

Automated construction of RCA model The multi-layered nature of faults emerging in the cloud environment makes it mandatory to recognize system components and symptoms at each layer and include them in the RCA model.

However, since enterprise applications migrate toward the microservices architecture, they become more complex as they get fragmented into tightly related functional units, often scaled to hundreds of instances and distributed over many physical machines. Thus, constructing an RCA model that accurately and efficiently describes the live system poses a challenge that requires collecting and processing a large volume of information from numerous sources.

The challenge is aggravated by self-organization mechanisms increasingly employed in modern cloud platforms. Auto-scaling and auto-healing processes continuously influence system structure, imposing frequent and prompt model updates. Furthermore, companies that matured enterprise application development use agile practices dictating regular deployment of new application releases, which may occur hundreds of times a day depending on business requirements. New application versions are deployed automatically using continuous integration and continuous delivery tools. Finally, chaos testing techniques are used to purposely inject faults into running applications and test them against fault tolerance.

The above elements demonstrate that cloud applications progressively deviate from the static scheme and become dynamic entities with a high evolution rate. Therefore, the construction of the RCA model should be fully automated to enable timely model updates required in root cause inference.

Recognition of symptom semantics An important element of failure analysis is the recognition of symptom semantics, i.e., understanding their meaning in the domain of a given system layer. From the perspective of the system administrator who diagnoses the failure, symptom semantics are easily determined as their understanding results from the experience of past failure observations and cognitive intelligence that allows inferring symptom dependencies based on the symptom context or specific symptom attributes. However, a similar analytical instrument is difficult to achieve artificially as semantic symptom dependencies create an extensive space hardly possible to maintain manually or encompass in a machine learning model. At the same time, the automation of semantic symptom differentiation is key to improving the performance and accuracy of the root cause analysis.

One way to obtain information about symptom semantics is to observe system operation over a long period. Symptoms that are semantically related tend to co-occur consistently in time. Temporal dependencies between symptoms can be used to detect semantic symptom correlations as part of the root cause analysis.

Holistic insight into system operation Due to the high complexity of cloud applications described in previous sections, faults may emerge within various system elements and thus may manifest themselves differently. Therefore, a fault management system must recognize symptoms relating to various aspects of system operation. Key system performance indicators describing system performance and availability in the context of defined SLO constitute the foundation for symptom identification. Complementary, additional system operations must be considered, such as changes in system configuration, anomalous user behaviors, or system security breaches.

Failure explainability Advanced understanding of failures emerging in distributed and dynamically evolving cloud environments fulfills a crucial role in effective fault remediation. Therefore, besides localizing root causes of active problems, a fault management system must provide an extended failure view, including failure context and fault trajectory, i.e., a vector of subsequent symptom occurrences and associated system components affected by the failure. That constitutes a minimum amount of information that allows the administrator to estimate the problem scale objectively and better assess associated risks and losses. In addition, a complete understanding of fault trajectory enables evaluating the system in terms of fault tolerance and fault

mitigation. Importantly, the information enables identifying missing fault handling mechanisms and eliminating single points of failure.

Isolating simultaneous faults Another challenge of root cause analysis in the cloud and distributed systems, in general, is the isolation of parallel faults. Due to the multi-layered architecture, a large volume of system components, and symptom diversity, a relatively frequent phenomenon observed in cloud systems is the emergence of simultaneous faults in a short period. Faults can be semantically dependent or independent. More importantly, faults can occur within the same system region and affect the same subset of system components. The objective of a fault management system is to isolate faults and explain them by means of separate root causes and distinct fault trajectories.

Application-agnostic analysis Preliminarily, applications and cloud infrastructure provide limited information required in root cause analysis. Knowledge of system component dependencies and symptom semantics is not defined in advance. It must be obtained manually by requesting it from domain experts or automatically based on system observation and processing data collected from available data sources.

Some existing solutions require instrumenting application code to obtain the necessary information. These tools record the execution path of a program and locate defects in the source code that are causing problems. However, despite the diagnosis accuracy at the application level, the approach is inconvenient for system administrators and developers as it requires the integration of appropriate SDK. Considering that many businesses close the source code for altering due to security reasons, this category of tools is not always preferable.

It is more pragmatic to assume that application and infrastructure elements are black-boxed, i.e., there is no insight into their detailed operation, particularly into the source code. Acquiring knowledge about the system must be orthogonal to the application implementation.

2.4 Related Research

This section performs a critical review of approaches that are closely related to the main scope of this dissertation. Specifically, the most distinctive solutions to root cause identification are presented and confronted with RCA system dimensions (Section 2.2.3) and RCA system requirements (Section 2.3) established for cloud-native applications.

Wu et al. [23] propose an application-agnostic approach to root cause analysis, referred to as MicroRCA, designed for container-based microservices environments. Based on service mesh and system-level metrics, the solution constructs an attributed graph comprising host and application service nodes that form a base for modeling failure propagation throughout the system. Then, key performance indicators, i.e., service response times or host resource utilization, are inspected to determine sets of abnormal components. Dependencies between anomalous components are quantified using the Pearson correlation coefficient and averaged to calculate aggregated anomaly scores for each component. Finally, candidate root causes are localized using the PageRank centrality algorithm.

Although functional evaluation indicates high precision of returned diagnoses compared to existing solutions, considered evaluation scenarios are simple and consider only the most common failure situations. In addition, the graph model reflects only two types of components and two categories of metrics, thus neglecting in-depth aspects of system operation. Further, fault explanation is limited to identifying single abnormal components as candidates for the root cause of failure. Produced results do not indicate the fault trajectory. Last, the authors of the study do not discuss the problem of handling parallel faults.

Lin et al. [26, 27] introduce the Microscope solution that offers root cause localization in microservices applications that is orthogonal to application implementation. In order to perform the analysis, the solution comprises three stages, namely data collection, causality graph building, and root cause identification. As part of the data collection stage, system calls associated with network connections are analyzed to discover dependencies between communicating services, where dependency directions are determined by distinguishing system calls that initiate data transmission (e.g., *send* and *recv* system calls). In addition, SLO metrics, specifically average response time, are collected to identify abnormal services. Further, building the causality graph is based on processing network information and creating a deterministic network of service connections. Here, the study authors assume that a path between a pair of communicating services implies causal dependence of service abnormalities. Complementary to communicating service dependencies, authors define non-communicating service dependencies as service pairs sharing the same allocation of computational resources. This class of dependencies is determined using the PC algorithm applied against collected SLO metrics. It is based on the observation that SLO metrics of services colocated in the same compute node react simultaneously to local problems, e.g., CPU resource occupied by one of the services. At its output, the graph-building algorithm produces a causality graph that meets DAG characteristics. Last, the root cause inference is initiated by detecting an SLO anomaly in user-facing frontend

services. The inference algorithm traverses the causality graph in the opposite edge direction until it encounters services whose neighbors are not abnormal. The last abnormal services constitute root cause candidates. Candidates are sorted by computing the Pearson correlation coefficient between SLO values of frontend and root cause candidates. Authors perceive this technique to address the problem of differentiating parallel faults.

The discussed Microscope solution by Lin et al. presents powerful inferencing capabilities supported by a functional evaluation conducted on a live microservices application. However, several issues can be identified. First, the solution is limited to analyzing the failure in the context of one type of system component, i.e., application services, neglecting other component classes that are potential sources of application defects. Likewise, only one SLO metric is considered, i.e., response time. Consequently, the solution performs failure analysis in the context of services and does not leverage insight into lower system layers. Further, the authors consider service dependency calculated based on network communication to be equal to the causal dependence between service anomalies. In practice, the assumption is incorrect because anomalies emerging in services may be semantically-independent despite dependence in the communication plane. Another solution limitation is poor failure explainability. Only a list of root cause candidates is produced at the analysis output, while the fault trajectory and affected system regions are not included. Finally, the use of Pearson correlation for ranking root cause candidates is insufficiently refined. The approach is based on the assumption that anomalies across root cause and frontend services co-occur simultaneously. That is not always true, as anomalies across subsequent services in the service chain emerge with a delay. Usually, the delay is mild, but it can grow to several seconds or minutes. Proper fragmentation of SLO metrics is important for calculating the Pearson coefficient. Ignoring this aspect may result in coefficient disruptions and the risk of omitting essential anomaly dependencies.

Wang et al. [6] present GRANO as a root cause analysis solution dedicated to cloud-native applications and evaluate it against production data provided at eBay. GRANO consists of three functional stages: anomaly detection, anomaly graph construction, and root cause identification. First, GRANO detects anomalies in system metrics using ML models and statistical methods. Then, it constructs the anomaly graph by localizing detected anomalies in the system topology. Anomalies are manifested as alerts describing rich diagnostic information. Finally, in root cause identification, a relevance score aggregating properties of associated alerts, e.g., alert severity, is calculated for each component, and aggregated results are propagated to neighbor components until reaching the end of the graph. The component that

aggregates the highest score in the propagation indicates the most probable failure cause.

GRANO combines knowledge of system topology with the behavioral aspect examined in terms of alerts reported by the system observability plane. The system topology covers a holistic system view, including physical, virtual, and logical elements. However, the process of topology acquisition is not disclosed. Further, the inference algorithm is based on a relevance score comprising only two alert properties, i.e., alert severity and the frequency of alert occurrence on a given component in a given time window. Semantic alert dependencies are not considered, potentially producing false-positive alert dependencies in case of semantically-independent faults. Moreover, the algorithm focuses on identifying the most likely root cause and cannot extract fault trajectory from the anomaly graph. Finally, the authors do not attempt to discuss the problem of differentiating parallel faults.

Groot constitutes another solution to root cause identification in cloud-native applications. Wang et al. [30] propose an approach comprising three stages: dependency graph construction, causality graph construction, and root cause ranking. The dependency graph comprises application services and is constructed automatically based on analyzing data from application logs and distributed tracing. Importantly, the graph is updated once a day, posing a notable limitation for systems with a high evolution rate. Further, Groot builds the causality graph based on several types of observability data. The authors take a holistic approach by collecting symptoms emitted based on performance metrics, application logs, and developer activity events (e.g., code deployment). Connections in the causality graph are identified based on a rule set defined in advance by the SRE team. After constructing the causality graph, Groot uses a modified PageRank algorithm to identify the failure root cause.

Like GRANO, Groot combines knowledge of system topology with the knowledge of system behavior. The system topology does not cover all essential abstraction layers and is limited to application services. Consistently with established RCA requirements, the solution adopts a holistic approach to causal analysis by processing metrics from various system operation aspects. The most notable solution limitation is the process of constructing the causality graph that involves the SRA team manually defining alert correlation rules.

Qiu et al. [32] present an RCA solution for cloud-native applications built based on microservices architecture. Like many other solutions discussed in this section, the solution leverages the synergy of knowledge about system topology and behavior. The causality graph constitutes the main solution element, where nodes represent anomalies detected in key system performance indicators (KPIs), while edges represent

causal dependencies inferred using statistical methods. The graph is constructed using the PC algorithm. Specifically, a directed acyclic graph (DAG) skeleton is built, where edges are designated using conditional independence testing from the probability theory, followed by edge orientation based on d-separation rules. Then, Pearson linear correlation is used to compute edge weights as KPI correlation strength. Interestingly, prior to calculating the Pearson coefficient, the authors translate KPI time-series into ternary sequences indicating substantial value changes. The sequences are obtained from the t-test proposed in Luo research [34]. In addition, the authors introduce the knowledge graph as a helper structure maintaining information about system component dependencies. It is employed to optimize the causality graph construction by excluding causal dependency computations for KPI pairs for which associated system components are unrelated. Finally, fault trajectories are designated by applying the Breadth-first search (BFS) algorithm on the constructed causality graph. Then, transversed paths are ranked according to the average dependency strength and total path length.

The solution proposed by Qiu et al. belongs to the category of RCA systems based on the causality graph concept. Conducted solution evaluation in cloud-native application testbed confirms solution effectiveness for basic failure scenarios. Unfortunately, some assumptions adopted in work forecast invalid results in some failure scenarios. First, determining causal KPI dependencies based solely on a linear correlation is insufficient as a detected correlation for a given failure instance may be transitive or accidental. Notably, strengthening causal dependency requires understanding the semantics of system operation. However, the aspect of anomaly semantics is not considered in the study. Second, the authors do not designate proper time-series fragments for correlation, ignoring the effect propagation time. Thus, in some cases, the solution may ignore significant anomaly dependencies.

Brandon et al. [28] combine the dependency graph approach with matching anomalous graph regions to a repository of failure patterns labeled with failure causes. The dependency graph is constructed by processing attributes of system metrics and analyzing TCP traces obtained from system kernel instrumentation. Next, system operation anomalies are detected, mapped into the dependency graph, and extracted as a subgraph representing a local system failure. The subgraph is then compared to subsequent failure patterns stored in the pattern repository by computing a graph similarity measure based on the node attributes in the graph. The root cause associated with the most similar pattern is produced as an algorithm result.

Despite high performance and low resource consumption compared to other methods, the solution shows significant limitations. First, the solution introduces an entry

overhead in the form of the necessity to label failure subgraphs with causes manually. Thus, the root cause analysis process is manual over a long period until a complete pattern repository is available. Second, the solution does not consider symptom semantics with negative consequences elaborated in the description of previous solution examples. Last, the authors do not attempt to discuss the problem of parallel faults.

Cai et al. [22] propose a trace-based root cause analysis solution based on data obtained from large-scale tracing and logging systems in the Alibaba cloud environment. The solution allows offline discovery of normative request trace patterns and performance metrics. Then, in the online root cause identification phase, anomalous traces are detected in the proximity of reported anomaly events and matched to normative patterns. Finally, the failure root cause is localized via relative importance analysis.

2.5 Summary

This chapter presented important background aspects required to discuss the concept and realization of the proposed RCA solution.

The study illustrates cloud-native applications and the major challenges they pose in root cause analysis, such as the dispersed nature of microservices architecture, diversity of application elements, multi-layered cloud infrastructure, high evolution rate, and the possibility of parallel faults. Moreover, the chapter indicates the importance of recognizing symptom semantics and adopting a holistic analysis perspective as significant dimensions influencing the diagnosis accuracy.

Evaluated against established RCA dimensions for cloud-native applications, current approaches to root cause identification show several significant limitations. Some limitations include focusing on the analysis within a single system layer, limited symptom space, lack of discussion related to parallel fault isolation, or insufficient failure explainability. The novel approach presented in the following chapters attempts to address them.

Chapter 3

Concept of RCA for Cloud Applications

The main goal of this chapter is to decompose the root cause analysis problem into a high-level process comprising critical RCA solution elements. In particular, the concept proposes a methodology for approximating causal symptom dependencies based on the consolidation of selected symptom correlation analyses. Furthermore, based on the adopted symptom correlation method, it outlines the boundaries of the RCA model and inference algorithm derived from the introduced RCA system abstraction.

The chapter does not provide concrete solution realization techniques but points out the most crucial aspects that must be considered when designing and implementing an RCA system capable of satisfying established RCA system requirements. Proposed concepts fulfill dissertation goals in subsequent chapters, presenting concrete realizations of root cause analysis for cloud-native applications.

3.1 Concept Overview

RCA challenges established in Section 2.3 of the previous chapter shape the proposed solution concept. First of all, the developed fault management system must adopt a holistic diagnosis approach that evaluates symptoms originating from many aspects of system operation and correlates them considering system components across essential layers of the multi-tier cloud architecture. Moreover, the analysis must recognize the semantics of observed symptoms to discover symptom dependencies with confidence. The comprehensive insight into system structure and symptom semantics must be represented in an RCA model, the construction of which should be fully automated and aligns with the frequency of changes imposed by the cloud-native environment. Further, due to the significant evolution rate of cloud systems and the extensive space of possible fault vectors, the RCA solution should support failure scenarios comprising parallel faults and conduct their precise isolation capable of identifying distinct root causes. In addition, depending on administrative requirements, failure diagnosis should indicate complete fault trajectories illustrating fault propagation throughout the system. Finally, the solution should be orthogonal to application implementation, i.e., due to the gray-box nature of cloud applications, it is impossible to instrument application code to gather facts for failure analysis.

The concept outline is reinforced by the proposed implementation of RCA problem dimensions defined in Section 2.2.3. In terms of analysis intent, the diagnosis should produce both failure root cause and relevant fault trajectory explaining symptom propagation. Furthermore, produced results should be accurate, i.e., they should be scoped to concrete system components and related anomalies, as the analysis accuracy is crucial to remediate failure causes effectively. Moreover, the RCA solution must handle diagnoses in large-scale environments built on millions of interconnected components and assume that the system is characterized by a high evolution rate. Finally, the solution should assume that parallel faults may emerge due to the complexity and multi-layered nature of cloud applications. The length of fault trajectories can, in turn, grow up to tens of elements and span across several system layers.

Based on the review of root cause analysis techniques presented in Section 2.2.4, the model-based approach appears to be the most adequate for fulfilling requirements demanded by cloud-native applications. First, the structure of the observed system constitutes the prominent subject of the root cause analysis. It clearly shows a relational component scheme, while its prompt recognition is essential to enhance the accuracy of the failure diagnosis. Second, the required ability to recognize symptom semantics forces constructing a model representing symptom dependencies and

serving as a basis for in-depth analysis. Third, model-based techniques demonstrate the potential of effective diagnosis in the cloud environment, as indicated in selected research works.

As part of the solution concept, a methodology for detecting causal symptom dependencies must be selected prior to conceptualizing RCA system elements. It is the most significant element of the root cause identification process and is widely discussed in relevant research. This dissertation encloses a unique study of symptom causality in the symptom correlation framework described in Section 3.2. Then, following the RCA system abstraction presented in Section 2.2.2, concepts of the RCA model and inference algorithm are presented, constituting a derivative of the developed symptom correlation methodology. They are discussed in Section 3.3 and Section 3.4, respectively.

3.2 Symptom Correlation Framework

The selection of symptom correlation methodology is fundamental for conceptualizing subsequent elements of the RCA solution.

Most studies on root cause identification wrongly assume that causal dependencies between observed symptoms can be discovered by correlating associated system metrics. That is a typical contradiction to the principle of *correlation does not imply causation*. While correlations may, in some cases, imply causation, they do not constitute a sufficient argument to conclude a causal relationship between variables. By definition, correlation is only a statistical indicator of variable relationship, but it does not guarantee that variables influence each other in a direct cause-effect manner. In particular, a third confounding variable may affect both variables, making them appear causally related when they are not.

Accordingly, the dissertation deviates from the common direction of examining symptoms using plain correlations and instead attempts to establish causal symptom dependencies. However, establishing causal dependencies that concern the behavior of cloud applications is impractical, as determining such dependencies requires performing repeated experiments in a controlled environment. Due to intrinsic system variability, conditions for accurate failure reproduction cannot be satisfied. Therefore, the concept proposes a novel approach in which causal dependencies between symptoms are approximated by consolidating multiple correlation methods. In other words, while a single correlation method does not imply causation with high certainty, the concept assumes that performing multiple correlation analyses, each inspecting a distinct aspect of system operation, increases the certainty of the causal

link. The correctness of the established assumption will be evaluated throughout this dissertation.

Thus, the key difficulty of the symptom examination problem is elaborating a set of correlation methods that approximate causal symptom dependencies. Methods should cover the analysis of independent system aspects and adhere to the limitations of knowledge acquisition imposed by the gray-boxed environment of cloud-native applications. Primarily, the analysis should focus on system structure and symptom semantics, as explained in the following paragraphs.

Locating symptoms in the system enables determining symptom dependencies based on structural properties. Specifically, it is possible to assess the distance between symptom sources and check if symptoms are located within the same system region. If symptom sources are located in direct or close proximity, it can be assumed that there is a significant likelihood of symptoms being correlated. Conversely, if symptom sources are located in distinct regions or are distant from each other, it can be assumed that the correlation is mild or unlikely. Importantly, to infer symptom dependencies from system structure effectively, the hierarchy of system components should be identified based on system taxonomy that reflects hints regarding causal component dependence.

The introduced analysis produces deterministic symptom dependencies based on the system structure. However, the method cannot handle failure scenarios involving parallel faults. It discovers redundant correlations for symptom sources that are topologically related even if semantically they contribute to independent faults. Hence, symptom semantics constitute the subsequent aspect that should be considered in analyzing symptom dependencies.

In the context of failure analysis, semantics is defined as a study of recognizing the meaning of symptoms given the occurrence of other symptoms. System administrators recognize symptom semantics by inspecting symptom attributes and recalling past experiences of failure observation. Intuitively, a similar cognitive process is difficult to imitate artificially, e.g., using rules or machine-learning models, due to the extensive space of possible symptom combinations.

The concept proposes obtaining semantic symptom dependencies based on event co-occurrence analysis, a statistical data mining technique that searches for event rules and patterns in an event stream. When applied to the symptom space, it quantifies the consistency of symptom pairs occurring in a given order. The more consistently symptoms co-occur, the greater the probability of symptoms being semantically correlated. Many techniques to event co-occurrence analysis can be applied to symptom dependency analysis because symptoms, as carriers of information

warning discovered system anomalies, can be seamlessly translated into the event space. Notably, the co-occurrence analysis requires a substantial volume of historical symptom data to produce objective results.

Semantic symptom correlations should be complemented by checking whether time lags between observed symptom pairs for which semantic correlation was hypothesized coincide with time lags appearing in historical symptom occurrences. This approach enables isolating parallel faults that are semantically dependent but originate from independent fault instances. If a time lag for a given symptom pair converges to the time lag mean determined based on symptoms registered in the past, the pair shows a valid temporal relationship and can be considered correlated. Otherwise, the symptom pair does not reflect characteristics of regular symptom co-occurrence and should be excluded from correlation.

The last analysis aspect proposed by the dissertation concept is the correlation of time-series associated with symptoms. Time-series correlation allows analyzing the mutual dynamics of system metrics. The method is motivated by the observation that time-series trends of correlated symptoms tend to respond simultaneously to anomalous system events.

Conclusively, the symptom correlation framework defines symptom correlation methods whose consolidation aims to approximate causal symptom dependencies. According to the framework, symptom pairs are evaluated in terms of structural, semantic, temporal, and metric data aspects. The following chapters of the research study verify whether the selected methods meet the expected approximation accuracy required for root cause inference in cloud systems.

3.3 RCA Model

The above characteristic of RCA solution requirements and dimensions for cloud-native applications, together with the outline of the symptom correlation framework, clearly illustrate that one of the main challenges to be addressed in the solution realization is recognizing the complex structure of the observed system. According to the requirements description, the representation of system structure should include essential cloud layers and recognize diverse system components in each of them. Notably, system components are not only interconnected at the layer in which they are located but also depend on components present in adjacent layers. For instance, the performance of virtual machines in the virtualization layer may depend on the resource capacity of physical servers in the hardware layer. Similarly, communicating application services depend on each other, creating chained dependencies derived from

implemented request patterns. Thus, reflecting the system structure visibly implies many dependencies that must be included in the constructed RCA model. Moreover, due to the significant evolution rate of cloud environments, these dependencies may be frequently updated.

The complexity of component dependencies can be effectively addressed by the graph structure as it enables mapping data with strong relational characteristics. In such a case, components and inter-component dependencies can be translated into graph nodes and edges, respectively, creating a dependency graph. An edge between components in the dependency graph indicates the logical relationship between components according to the system taxonomy. In addition, it may indicate a hint of causal dependency between symptoms reported on connected components. Importantly, adding and removing edges representing component dependencies is an efficient graph operation. Moreover, graph theory accumulates performant algorithms that enable graph traversal, path searching, and calculating metrics such as centrality or PageRank, which may be utilized to enhance the diagnosis process.

The literature review presented in Section 2.4 shows that dependency graphs constitute a practical tool supporting the root cause identification. However, existing works overlook some significant aspects.

Contrary to most existing works, this dissertation proposes a dependency graph that follows a holistic approach. Specifically, the proposed dependency graph covers multiple system layers instead of centralizing on a single one, e.g., the application service layer. The implication is the increased analysis accuracy as the identified root cause is not scoped to a high-level application service but to a component that initiated the failure at the system layer impacted by a fault. In addition, the concept imposes that the structure of the dependency graph is elastic, i.e., the set of system layers and components must not be strictly assumed. Instead, it should be selected arbitrarily based on the attributes of a given system class. Therefore, the concept realization should introduce proper generalizations and mechanisms enabling seamless extension of graph structure with new elements.

Most existing solutions are limited to the root cause analysis in a single layer due to the difficulty of constructing the dependency graph across multiple layers. Indeed, the concept of a holistic dependency graph poses a challenge in collecting the data required to construct the graph structure. In this matter, the concept assumes that complete dependency information can be obtained by integrating available knowledge sources located across considered cloud system layers. The assumption is supported by the observation that, over the years, subsystems such as network controllers, hypervisors, or orchestrators developed management APIs that allow

retrieving information about relevant system objects. In addition, some subsystems provide event systems capable of streaming event notifications about object changes to configured receivers. Such an event system can be leveraged to update the dependency graph structure incrementally.

Another aspect addressed by the solution concept is providing sufficient insight into system failures. More precisely, the RCA model should incorporate instances of symptoms warning anomalies in a broad spectrum of system operation aspects. Besides inspecting high-level anomalies related to inter-service communication, the model should ingest symptoms warning anomalies detected in lower system layers. For instance, anomalies related to hardware resources, virtualized workloads, or user activity should be considered. Similar to the problem of constructing the dependency graph, the concept assumes that obtaining symptoms providing a rich failure perspective is possible through the integration of available knowledge sources. In particular, the integration concerns observability tools.

Further, to make the dependency graph useful for the root cause identification process, the concept exhibits the need to associate system components with observed symptoms. That way, symptoms acquire a location property that can be used to determine topological symptom dependencies. As denoted in the symptom correlation concept, symptom dependencies can be quantified by examining distances between system components affected by the failure. The obtained dependencies are deterministic as they are based on the knowledge of system structure. Furthermore, the necessity of measuring the distance between graph nodes submits another argument in favor of modeling system structure as a graph.

3.4 Inference Algorithm

Symptom correlation framework and RCA model allow composing complete root cause identification process and enclosing it in the inference algorithm.

Given symptoms manifesting emerged system fault, the algorithm should take a set of active symptoms on its input and produce estimated fault trajectories on the output. Each fault trajectory should be a path consisting of symptoms ordered in a cause-effect manner, where the first symptom marks the failure cause, whereas the last symptom denotes the failure effect. In addition, trajectories should be ordered according to the confidence of inferred failure propagation.

Conceptually, the inference algorithm encompasses the symptom correlation framework as diagnostic apparatus. It incorporates correlation methods defined in the framework to evaluate symptoms against several aspects of system operation. Indi-

vidual correlation methods utilize the RCA model to access required structural and behavioral system knowledge.

Consolidation of obtained correlation results is a crucial failure diagnosis challenge. Here, the inference algorithm should use the synergy of correlation results produced by individual symptom correlation methods to construct a unified fault view in the form of a causality graph. In such a graph, nodes represent active symptoms, while edges represent approximated causal symptom dependencies. The structure of the causality graph forms the basis for conducting the subsequent failure analysis. First, however, a strategy for comparing and consolidating correlation results is required. The study of the result aggregation method presented in this dissertation constitutes contribution to the research area.

Interestingly, the literature debates the adoption of causality graphs in root cause inference. Nevertheless, no method analyzes symptom dependencies in a multi-faceted manner and does not approximate causal dependencies as proposed in the discussed concept.

Further, due to the potentially large number of symptoms resulting from system size and variability, the causality graph structure may be significant. Thus, reducing the graph complexity may be required to minimize the computational scope. For instance, the graph can be reduced by filtering out weak symptom dependencies, or a heuristic can be elaborated to extract relevant causality subgraph.

Given the causality graph, it is assumed that it is possible to estimate failure cause and effect candidates by analyzing the properties of nodes representing symptoms in the graph. For instance, symptoms whose nodes indicate the lowest ingress degree constitute potential root causes since, according to the causality graph structure, they are not caused by other symptoms. Analogously, symptoms whose nodes have the lowest egress degree constitute potential failure effects as they do not cause other symptoms. Alternatively, symptom effects can be selected by the system administrator as part of the semi-automated diagnosis.

After determining the root cause and effects symptoms, fault trajectories should be extracted using graph algorithms to search for paths connecting them in the causality graph. The structure of the causality graph, combined with the knowledge provided by the RCA model and leveraged using the substantial spectrum of algorithmic approaches offered by graph theory, allows for determining failure paths elastically and comprehensively, opening space for additional research contributions.

In the last diagnosis stage, the algorithm should order found fault trajectories by the confidence of mapping the correct propagation of symptoms during a failure.

The concept dictates sorting trajectory paths using two evaluation criteria, i.e., total trajectory length and average dependency strength, as, at the minimum, these two pieces of information can be obtained by inspecting trajectory properties extracted from the causality graph. Moreover, shorter trajectories are more likely to return a high average strength as they comprise fewer path edges holding potentially weak dependencies. At the same time, they show a tendency to omit intermediate symptoms in failure propagation. Therefore, the strongest and longest fault trajectories should be produced on the output of the inference algorithm.

3.5 Summary

The solution concept outlines the proposition of the RCA system capable of identifying causes of defects emerging in cloud-native applications.

The correlation of symptoms observed in the system constitutes the primary aspect discussed in the concept. Contrary to the current research direction of identifying symptom dependencies using plain correlations, the concept proposes a novel approach to discovering causal symptom dependencies. Considering that determining causal dependencies in the cloud environment is impractical, an approximation approach is presented based on consolidating several correlation methods, each analyzing a distinct aspect of system operation. Thus, the key research problem is selecting correlation methods that approximate symptom causality with sufficient accuracy. The concept addresses the problem by defining complementary correlation methods that examine symptoms in terms of structural, semantic, temporal, and metric data perspectives. The following dissertation chapters evaluate that concept.

The presented symptom correlation framework creates a basis for conceptualizing elements of RCA system abstraction, namely the RCA model and the inference algorithm.

In the matter of the RCA model, the main challenge is representing the knowledge of system structure and behavior. A dependency graph is proposed for representing inter-component dependencies. Moreover, the concept submits graph enrichment by associating observed symptoms with components affected by the failure. Consequently, active failure symptoms can be localized in system structure, allowing deterministic structural correlations. The use of the graph structure is motivated by the relational properties of system data, the efficiency of graph updates imposed by the evolution rate of cloud applications, and the availability of graph algorithms that can be used to enhance the diagnostic process.

Further, the symptom correlation framework and RCA model allow composing the complete root cause identification process and enclosing it in the form of an inference algorithm. According to RCA solution requirements for cloud-native applications, the algorithm should produce failure root cause and fault trajectory explaining symptom propagation throughout the system. The main concept idea is consolidating symptom correlation results into a symptom causality graph that provides a unified fault view that can be used to extract fault trajectories using elements from graph theory. Contrary to existing works, the proposed causality graph is constructed based on a holistic approach comprising analysis across several system operation planes.

Chapter 4

Realization of Symptom Correlation Framework

The abstract concept of root cause analysis introduced in the previous chapter leaves an open space for defining many possible realizations of symptom correlation framework, RCA model, and inference algorithm. This chapter discusses realizing the first element of the proposed root cause analysis concept, i.e., the symptom correlation framework. There are several issues that need to be addressed in the scope of the framework definition. The first issue is elaborating a set of symptom correlation methods, the consolidation of which provides an approximation of causal symptom dependencies essential in the root cause inference. Second, the realization must define a skeleton of the symptom correlation process that leverages the synergy of correlations produced by elaborated correlation methods to determine properties of symptom dependencies, i.e., dependency existence, direction, and strength. The last issue is identifying mechanisms capable of ensuring the temporal consistency of received symptoms.

4.1 Realization Overview

Referring to the solution concept, obtaining causal symptom dependencies is a difficult challenge. It comprises repeated experiments conducted in a controlled environment and requires the possibility of recreating past environment states in the moments of failure occurrence. Usually, these expectations are impossible to satisfy in the cloud due to the dynamic and complex nature of cloud applications and infrastructure.

Therefore, the concept proposes that instead of identifying causal dependencies between symptoms, one should focus on approximating them using available correlation methods. In this matter, the proposed solution shifts beyond existing approaches that frequently misinterpret the definition of causality. Instead of using a single correlation method to discover causal dependency, the concept proposes a synergy of correlation results obtained from several distinct correlation strategies, each examining system from a different operational perspective. In other words, the dissertation assumes that holistic and differentiated analysis can effectively guide the designation of symptom dependencies for failure diagnosis.

However, the selection of complementary correlation methods and the accuracy of dependency identification depend strongly on the amount of information that can be obtained from the system and reported symptoms. Thus, employed correlation methods are a derivative of available system knowledge.

By definition, symptoms are carriers of information about exceptional situations occurring in the system, e.g., a decrease in service performance, an increase in latency, or the emergence of application error. They are emitted based on changes in the associated telemetry data, i.e., time-series or temporal events. A time-series is a sequence of real-valued data points measured at successive time points with a uniform time interval (e.g., CPU load, network throughput, service latency, error rate), whereas an event is used to record the occurrences of a specific message indicating that something has happened in the system (e.g., application error or specific user action). In the former case, a symptom is reported if a time-series reaches a value threshold for a certain tolerance period or an anomaly is detected in its trend, e.g., value increase by expected value decrease. In the latter case, a symptom may be equivalent to an event or be emitted in response to anomalous event frequency.

Given the black-box nature of cloud systems, a reasonable minimum of information for both symptom categories that can be assumed to be guaranteed by most observability vendors are label and timestamp. A label identifies the type of system failure or

anomalous system situation that occurred, whereas a timestamp specifies the exact time point when the symptom rule was violated, e.g., a moment in time when CPU usage exceeded the threshold and was sustained for the 5-minute tolerance period. Further, observability tools provide symptom annotations storing information such as severity, tolerance period, or reference to a system component that reported the symptom. Additionally, it can be assumed that observability tools guarantee insight into time-series values in a requested time range for symptoms associated with time-series data.

It is crucial to emphasize that symptom timestamps typically do not reflect the actual failure start but are influenced by thresholds and tolerance periods enforced by symptom triggering rules. While it is not relevant for symptoms based on temporal events as events are generated instantly when a specific failure occurs in the system, the correlation accuracy may be negatively affected for symptoms associated with time-series data. For instance, a symptom rule may specify that a symptom of *high memory usage* should be triggered after exceeding 80% of available memory for at least 5 minutes. If the memory usage increased gradually due to the memory leak introduced in the latest application version and exceeded the threshold after 3 hours, the difference between symptom emission and the moment when the anomaly started would be over 3 hours. Such inadequacies make it impractical to assess the correct correlation direction. Therefore, mechanisms should be employed to correct activation timestamps for ingested symptoms.

Moreover, it can be assumed that the system subjected to analysis allows reading the context of emitted symptoms in a structure plane. Specifically, the system is assumed to show a certain level of self-descriptiveness, offering a management API or an event stream that exposes information about system components and inter-component dependencies. Component information integrated with symptom annotations enables localizing symptoms in the system and creates a basis for discovering symptom dependencies based on structural system properties.

The above elaboration shows a range of information that can be obtained through system inspection and observing reported symptoms. Timestamp and label information transported with symptoms enable localizing symptoms in time, which, combined with a repository of historical symptom occurrences, provides a basis for inspecting temporal symptom characteristics. That, in turn, opens the way to co-occurrence and time lag analyses, which implement the conceptualized semantic analysis perspective that allows recognizing the meaning of symptoms in the context of other symptoms. The semantic perspective is reinforced by detecting simultaneous trend changes in time-series associated with applicable symptoms. Finally, symptom

annotations localize symptoms in the hierarchy of system components and allow examining dependencies between active symptoms from a structural perspective.

Selected correlation methods form a symptom correlation framework that enables approximating causal dependencies between observed symptoms. Given a symptom pair on its input, the framework attempts to recognize three properties of symptom dependency [34].

- *Correlation existence* indicates whether the appearance of symptom A implies the appearance of symptom B, i.e., whether probabilities of symptom occurrences are not independent.
- *Temporal direction* determines the cause-effect order between correlated symptoms. For instance, if symptoms of type B always occur after symptoms of type A, one can infer a direction $A \rightarrow B$ and propose a hypothesis that A may be the cause of B.
- *Dependency strength* quantifies the correlation by assigning a standardized value to each correlated symptom pair.

In addition, the framework defines a mechanism for consolidating results of individual symptom correlation methods into an aggregated correlation coefficient, which approximates the properties of causal symptom dependency.

4.2 Symptom Correlation Process

Following the high-level overview of the symptom correlation framework realization, this section decomposes the framework into essential building blocks and organizes them into a complete symptom correlation process.

Figure 4.1 illustrates symptom correlation methodology adopted in the framework. In essence, the framework takes pairs of symptoms reported by the observability plane as input. Then, for each symptom pair, it performs subsequent correlation analyses, namely *Symptom Co-occurrence Analysis*, *Symptom Time Lag Analysis*, *Symptom Time-series Analysis*, and *Symptom Topological Distance Analysis*, and consolidates obtained correlation results into the aggregated correlation coefficient that approximates causal symptom dependency.

Since the correlation process applies to symptoms related to various types of telemetry data, prior to performing correlation analyses, it is essential to preprocess them, ensuring that all of them consistently and precisely describe failure situations emerging

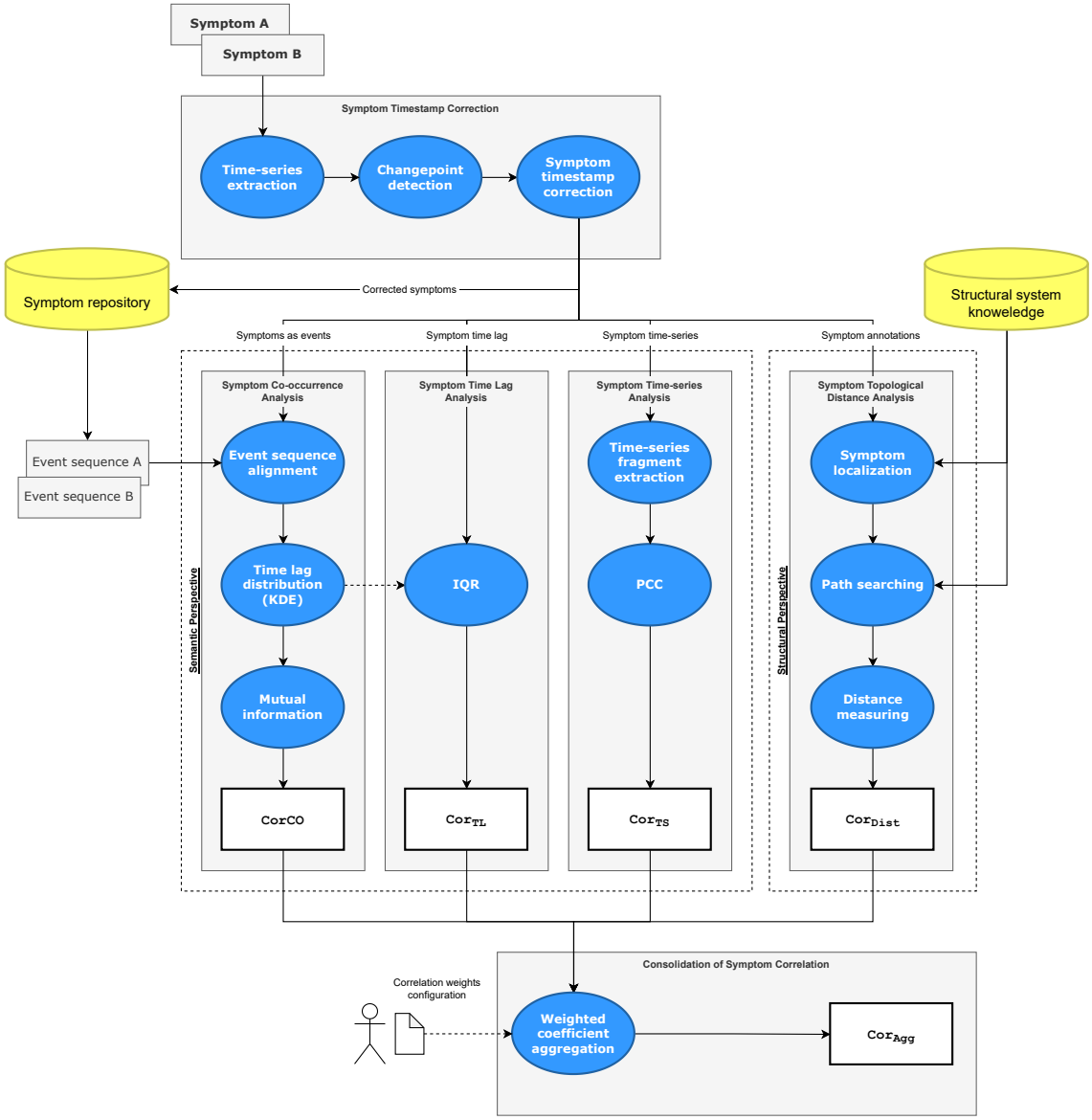


Figure 4.1: Symptom correlation process adopted in the symptom correlation framework

in the system. In particular, the preprocessing concerns symptoms emitted in response to changes in time-series data. Due to the mentioned alerting configuration impact, timestamps of time-series symptoms do not always reflect failure start and thus cannot be effectively used to determine valid correlations with other symptoms. Therefore, this work contributes by employing changepoint detection techniques to find a significant change in trend or change in statistical properties (mean or variance) of associated time-series as a correction to symptom emission time. It is assumed that a changepoint closest to the symptom emission time reflects the start of failure with a high likelihood. Based on that assumption, in the first stage of the correlation process, each symptom associated with time-series data is preprocessed by detecting changepoints and determining the most accurate imprint of failure start. The estimated changepoint constitutes a correction to the original timestamp

transported in emitted symptom. Then, all processed symptoms are stored in the symptom repository for future inspection of historical symptom occurrences.

In subsequent method stages, symptoms are processed in the event space. More precisely, event correlation techniques are used to investigate event co-occurrence and time lag.

Determining event co-occurrence is a black-box approach based solely on analyzing event timestamps and labels without knowing the semantics of system operation. Intuitively, if symptoms of type B always occur after symptoms of type A with an approximately constant time lag, one can mark symptoms as correlated. Conversely, if sequences of symptoms of two types do not co-occur consistently, one can mark the lack of symptom correlation.

Elements of the Iterative Closest Events (ICE) method are used to analyze event co-occurrence. The method models temporal event dependency as a time lag probability distribution based on finding alignment across event sequences. The dissertation extends the method by intermediate data preprocessing and normalization techniques comprising outlier detection and clustering to counterpart non-deterministic behavior of systems subjected to the analysis.

For an input pair of events, sequences of historical event occurrences are retrieved from the symptom repository. First, an optimization problem is solved to align event sequences by minimizing distances between corresponding event instances. Then, based on the assignment, time lags between matched event pairs are calculated, and lag signs are evaluated to determine the causal order in event dependency. Last, a time lag probability distribution is estimated, and correlation strength is quantified using Mutual Information (MI) from the information theory.

Further, the time lag probability distribution obtained in the symptom co-occurrence analysis is reused in the process of time lag analysis. Its primary objective is excluding symptom correlations for which the time lag does not match the distribution, e.g., due to symptoms originating from parallel faults or being mixed between different fault instances. Time lag calculated for a new event pair is compared to the time lag distribution modeled from historical event occurrences. Outlier detection is conducted based on the interquartile range technique (IQR), and a standardized correlation coefficient is calculated based on the lag distance from the distribution mean. If the considered time lag is significantly distant from the distribution mean, it is marked as an outlier, and the correlation coefficient converges to zero. In other words, the analysis aims to check if lag between symptoms is expected for a given symptom pair. For instance, if symptom instances of two types indicated strong co-occurrence over the past year with the time lag concentrated strictly around the

average of 5 seconds, an event pair with a time lag of 30 seconds may be excluded as an outlier with high confidence.

Another piece of symptom information that is used to enrich discovered correlations is time-series data associated with some symptoms. If time-series for a pair of symptom instances show dependent trend characteristics, i.e., both trends show an increase or decrease at a similar pace in close proximity, it can be concluded that symptoms are correlated. However, correlated trends are not obligated to originate at the same point in time. Usually, the metric for one symptom shows a delayed reaction related to changes in metrics for other symptoms. Consequently, it is crucial to designate time-series fragments for correlation to include only trend changes responding to the inspected fault. Otherwise, the correlation coefficient is likely to be distorted. Similar to symptom co-occurrence, changepoint detection methods are employed to determine valid time-series fragments for the analysis.

In the last correlation stage, symptoms are inspected in the context of system structure, assuming that the adopted RCA model provides insight into structural system knowledge comprising dependencies between system components represented in a dependency graph. Then, using symptom annotations, it is possible to identify system components affected by the failure. If components are located in direct or close proximity, it can be concluded that symptoms are correlated. Otherwise, when components are located in distinct system regions or are significantly distant from each other, the correlation is mild or missing. A graph distance measure is defined to quantify the dependency.

The elaborated symptom correlation methods yield a tuple of coefficients corresponding to four analytical approaches to symptom correlation. Obtained coefficients can be interpreted as answering four questions about symptom dependency. First, symptom co-occurrence analysis answers whether types of given symptom instances indicate a statistically strong temporal dependency based on historical observations. Second, the time lag analysis informs whether a time lag calculated for a symptom pair is valid and does not qualify as an outlier. Third, the time-series analysis checks whether metrics associated with symptoms react interdependently. Last, the topological distance analysis localizes symptom sources in the system and inspects whether they are structurally related.

In order to provide a unified view of symptom dependency, the framework consolidates results obtained from individual correlation methods into an aggregated correlation coefficient computed using a weighted average strategy.

Details of listed correlation building blocks are elaborated in the following sections. Section 4.3 explains changepoint detection techniques required to correct failure

onsets for time-series symptoms. Section 4.4 describes techniques adopted to correlate symptoms through event co-occurrence analysis. Section 4.5 discusses analysis of symptom time lags. Section 4.6 describes symptom correlation technique based on time-series analysis. Section 4.7 elaborates on symptom topological distance analysis examining the structural symptom correlation. Last, Section 4.8 presents a method for consolidating correlation results into an aggregated correlation coefficient.

4.3 Changepoint Detection

As pointed out in the previous chapter, symptoms do not always transport accurate time information required to determine a valid relationship with other symptoms. Symptom triggering rules based on thresholds and tolerance periods enforced by business and infrastructure SLOs significantly affect the moment of symptom emission. In practice, symptom emission does not coincide with failure start but is distant by the configured tolerance period and delayed by the time needed by the time-series to reach the value threshold. As a result, the sequence of emitted symptoms is often opposite to the cause-effect symptom order. Thus, retrospection of the associated symptom information is necessary to correct the relationship. In particular, the retrospection concerns time-series data as changes in its trend can be examined to estimate failure start and hypothesize causal symptom order.

Moreover, when computing a correlation between a pair of time-series as part of the time-series analysis, proper designation of time-series fragments subjected to correlation is important. If fragments are too long, they contain numerous fluctuations in trends or may contain changes linked to past failures unrelated to analyzed symptoms. The correlation of such fragments is prone to distortions. For instance, time-series can be highly correlated at values close to the symptom ingestion time, but the correlation coefficient is underestimated due to the impact of remaining values in designated fragments. Therefore, it is crucial to narrow down the correlation scope only to fragments that manifest trend changes associated with specific symptom instances, i.e., a significant increase or decrease closest to the moment of symptom ingestion.

This work proposes using changepoint detection (CPD) methods to address the symptom timestamp correction and designate time-series fragments for time-series analysis.

4.3.1 Problem Statement

Changepoint detection is a process of detecting specific points in time-series where abrupt shifts in trends occur that divide the time-series into stationary fragments. Formally speaking, given a time-series $M = (m_1, ..., m_n)$, a changepoint occurs at time τ if distributions of $(m_1, ..., m_\tau)$ and $(m_{\tau+1}, ..., m_n)$ differ significantly with respect to some statistical criterion (e.g., variance or mean). The trend of a time-series can change repeatedly at K instants $\tau_0 < \tau_1 < ... < \tau_K < \tau_{K+1}$ where $\tau_0 = 1$ and $\tau_{K+1} = n$. Consequently, the time-series is split into $K + 1$ fragments with the ith segment containing data points denoted by a subsequence $M_{\tau_{i-1}+1:\tau_i}$. The objective of the CPD algorithm is to calculate indexes of individual changepoints. The number of changepoints K may be unknown or provided in advance.

4.3.2 Selection of Changepoint Detection Method

There exists a large body of research examining approaches to changepoint detection [45, 46]. Most of them cast the detection to the model optimization problem of minimizing the sum of costs for time-series subsequences with respect to the number of changepoints K and changepoint locations $\mathrm{T} = (\tau_1, ..., \tau_{K+1})$, i.e.,

$$\arg\min_{K,\mathrm{T}} \sum_{i=1}^{K+1} [C(M_{\tau_{i-1}+1:\tau_i})] + pen(K)\,. \tag{4.1}$$

Here, C is a cost function that determines the measure of segment homogeneity. Intuitively, the cost function returns low values for homogeneous segments, i.e., segments with no significant statistical deviations, thus indicating no further changepoints. Contrary, the function returns high values for heterogeneous segments, i.e., segments for which the statistical criterion for stationarity is not satisfied. That, in turn, indicates the existence of intermediate changepoints in the segment. Further, function *pen* is a measure of penalty constraint related to segmentation granularity. It enables the avoidance of model over-fitting if the number of changepoints is unknown. For low penalty values, many changepoints are detected, including those resulting from noise. Higher penalty values allow detecting only the most significant changes in trend.

Over the years of research, many options have emerged for cost and penalty functions [47]. The vast majority of cost functions derive from maximum likelihood estimation, which embodies a large class of statistical properties such as mean and scale shifts of normally distributed data. Equation 4.2 demonstrates the mean shift model expressed as the quadratic error loss function. For a time-series subsequence $M_{a:b}$,

it computes the squared sum of absolute distances of individual points from the subsequence mean. If the points converge to the mean, the value returned by the cost function is low, indicating the subsequence stationarity, while if the points diverge, the function returns higher values indicating intermediate changepoints.

$$C_{L2}(M_{a:b}) = \sum_{t=a+1}^{b} \left\| M_t - \overline{M_{a:b}} \right\|_2^2 \tag{4.2}$$

Regarding penalty function *pen* in the Equation 4.1, the most common choice is linear penalty, often referred to as l_0. Equation 4.3 shows an example of linear penalty pen_{l_0}. The smoothing parameter β determines the balance between changepoint density and model fitness measured as a sum of costs for a given combination of segments. Low β values favor the detection of many changepoints, while high β values keep the number of change points to a minimum.

$$pen_{l_0}(K) = \beta \left| K \right| \tag{4.3}$$

Last, given the cost and penalty functions, a method is needed to search through the space of possible changepoint locations in order to fit the defined optimization model. The naive approach of testing all possible changepoint locations is hardly practicable due to the large space of possible time-series segmentations, i.e., for n data points, there are 2^{n-1} possible solutions. Therefore, computationally efficient search methods are required. Methods established in changepoint literature can be organized into two general categories: *optimal* methods that yield the exact solution to the optimization problem and *approximate* methods that yield an approximate solution.

Among approximation methods, the Binary Segmentation (BS) method proposed by Scott and Knott (1974) [48, 49] is the most common. Basically, the method provides a divide-and-conquer algorithm that detects single points of change in narrowing subsets of time-series. At each step, the algorithm attempts to introduce an extra changepoint if and only if it reduces the sum of costs in the Equation 4.1. First, an initial changepoint is detected by evaluating the entire sequence of data points. Then, the sequence is split into two smaller fragments at the location corresponding to the detected changepoint. The same operation is repeated several times for divided fragments until the stopping criterion is met. The advantage of the binary segmentation method is that it is computationally efficient with $O(n \log n)$ calculations for a time-series consisting of n data points. However, the accuracy of finding the global minimum in the Equation 4.1 is not guaranteed due to the

estimation of the first changepoint not always being accurate and affecting the selection of subsequent changepoints.

In the category of optimal methods, the Pruned Exact Linear Time (PELT) method provides the most computationally efficient algorithm for calculating the exact solution to the Equation 4.1 when the penalty function is linear (Equation 4.3). The method is an extension of the Optimal Partitioning (OP) method of Jackson et al. (2005) [50]. In essence, it conditions the last changepoint and redefines the optimization problem as minimizing the sum of the optimal partitioning cost for the fragment before the changepoint and the cost for the fragment spanning from the changepoint to the data end. The obtained formula allows solving sub-problems recursively relying on the output yielded by previous iterations. Combined with elements of dynamic programming, the optimal partitioning method results in quadratic computational cost $O(n^2)$. Pruning rules introduced by the PELT extension authored by Killick et al.(2012) [51] enable discarding calculations for changepoint locations that can never provide the minimum in the Equation 4.1. Consequently, the effectiveness of the method gets significantly improved, resulting in an average computational cost of $O(n)$.

As the state-of-the-art method with high accuracy and balanced performance, the PELT method was selected for further research and application in the prototype implementation.

4.3.3 Estimating Number of Changepoints

When searching for a changepoint to estimate failure start, the past fragment of the associated time-series is analyzed starting from the moment of symptom reporting. The fragment has a fixed width of, e.g., 5 or 30 minutes, within which the onset of failure is expected. There can be many changepoints within such a fragment, but only one can be designated to correct the failure start. The exact number of changepoints is unknown in assumed failure scenarios, and thus, the number must be estimated by inspecting relevant time-series data.

The number of changepoints in a fragment is determined based on linear penalty parametrization in the PELT method, i.e., the lower the penalty, the more sensitive changepoint detection to variations in time-series values, whereas the higher the penalty, the more strict and limited selection.

Haynes et al. [52] propose the Changepoints for a Range of PenaltieS (CROPS) method to efficiently compute time-series segmentation for a selected range of penalty values using an arbitrary changepoint detection method. The method derives from

the fact that different penalty values translate into different changepoint numbers and locations and, consequently, different costs. By matching obtained changepoint numbers to the corresponding costs, it is possible to analytically estimate the number of changepoints above which subsequent ones do not cause a significant cost reduction. Typically, cost reduction is significant for initial changepoint numbers, but from a certain point adding new changepoints produces a marginal cost decrease, distorting the result by excessive changepoint granulation.

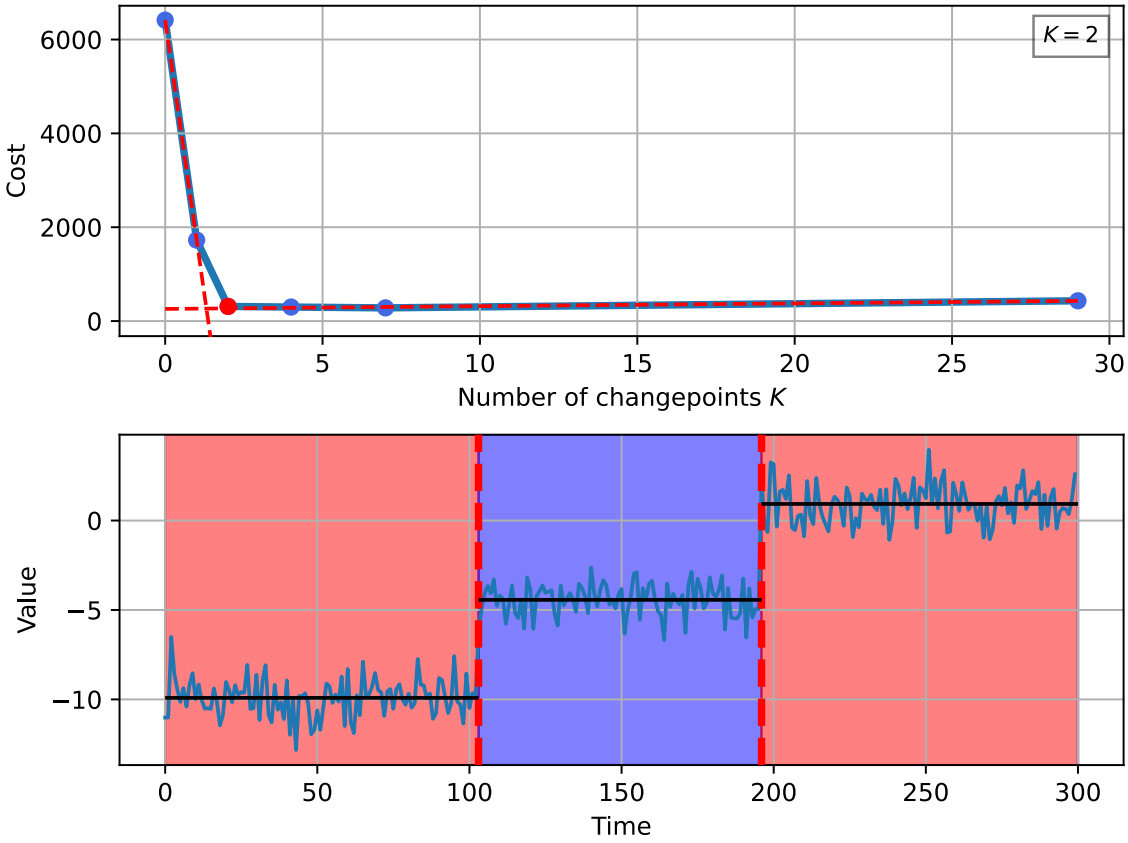


Figure 4.2: Estimated number of changepoints ($K = 2$) and detected changepoint locations for synthetic time-series with true number of changepoints $K = 2$

Figure 4.2 shows a graph that plots segmentation cost against changepoint numbers. Notably, the space of changepoint numbers is not continuous, i.e., there exists a wide range of changepoint numbers for which there is no useful time-series segmentation. For instance, there is no segmentation with three changepoints in the presented scenario due to segmentation with two or four changepoints always returning a lower cost within all considered penalty values. In its shape, the graph is similar to an inverted logarithmic curve. According to the CROPS method, the optimal number of changepoints is located on the bend of the curve, where changes in cost between subsequent changepoint numbers start converging to zero. This visual method of changepoint estimation is often referred to as the *elbow* method. In the considsered scenario, the optimal number of changepoints $K = 2$ is marked with a red circle.

The graph at the bottom visualizes time-series fragments based on the estimated number of changepoints and found changepoint locations. Red and blue mark time-series fragments based on true changepoint locations used to synthesize the time-series. Changepoints discovered using the PELT method are marked with red vertical lines. Black horizontal lines illustrate the means for obtained segments. The graph confirms that estimated changepoints match the true ones. All fragments with distinct means are correctly separated.

In order to estimate the valid number of changepoints automatically, Lung et al. [53] suggest an extension to the CROPS method. Iteratively, each number of changepoints is considered optimal. First, sets of changepoint numbers are determined before and after the considered changepoint to reflect graph regions where the segmentation cost decreases rapidly and where the reduction is marginal. Then, least-squares linear regression is computed, and sums of residuals are calculated for each set. Eventually, the number of changepoints for which the sum of calculated residual sums is the lowest is selected as optimal. Due to the characteristic of cost distribution for subsequent changepoints, linear regression returns the best fit when the expected number of changepoints is located around the bend of the curve.

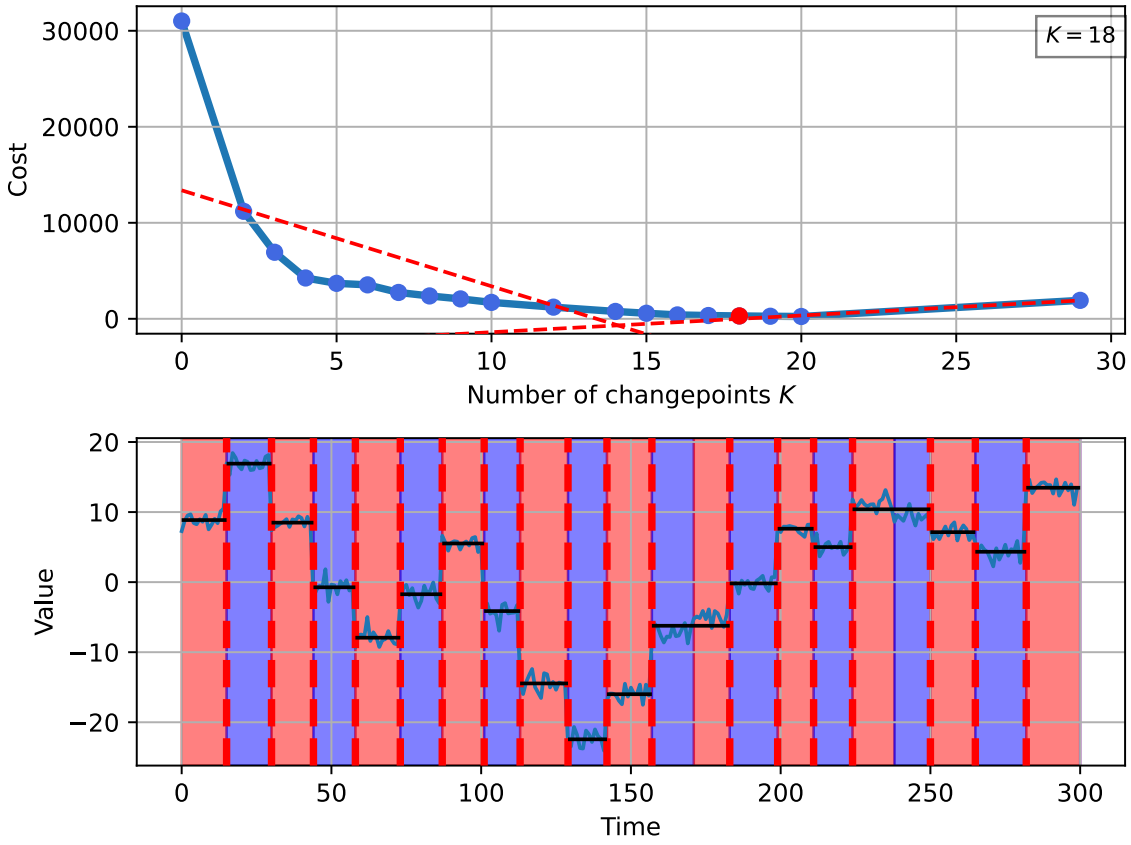


Figure 4.3: Estimated number of changepoints ($K = 18$) and detected changepoint locations for synthetic time-series with true number of changepoints $K = 20$

Both Figure 4.2 and Figure 4.3 denote number of changepoints estimated based on the regression method. The estimated number of changepoints is marked with a red circle in the graph. Further, the graph illustrates a pair of regression lines that produced the best separation of costs around the curve bend. The estimated number of changepoints $K = 2$ matches the true number in the former example. However, in the latter example, the estimated number of changepoints $K = 18$ is underestimated compared to the true number of changepoints (20). Changepoints are missing at around 170th and 240th second of the time-series. The underestimation results from the fact that mean shifts in these fragments are insignificant in the context of the considered period.

4.3.4 Symptom Timestamp Correction

The primary objective of changepoint detection is the estimation of failure onsets for symptoms based on time-series data. As explained in Section 4.1, symptom timestamp, i.e., point in time at which observability tool detects system anomaly and recognizes it as a failure, usually does not equal failure start but is enforced by a value threshold and tolerance period included in the symptom triggering rule. Therefore, correcting symptom timestamps is mandatory to ensure the correct cause-effect order in symptom dependencies discovered in failure diagnosis.

Figure 4.4 illustrates context of a symptom. It plots values of a time-series M over a period containing a rapid value increase in response to a failure. The increase occurs at approximately 260th second, denoted by changepoint $\tau_{Failure}$. Then, the symptom triggering rule is activated at about 280th second, denoted by $\tau_{Activation}$, indicating that the threshold v_{max} has been exceeded, and the monitoring system observed threshold violation. Lastly, after the configured tolerance period $\Delta_{Tolerance}$ elapses, the symptom is emitted at around 500th second, marked by $\tau_{Emission}$.

Depending on the symptom handling strategy offered by a given observability tool, the symptom timestamp subjected to the correction may be equivalent to symptom activation time $\tau_{Activation}$ or symptom emission time $\tau_{Emission}$ presented in the failure context. Regardless of that, the symptom timestamp may be significantly distant from the failure start $\tau_{Failure}$.

As part of the symptom timestamp correction, the symptom triggering rule can be utilized. It allows approximating failure start by subtracting the configured tolerance period from the symptom emission time, i.e.,

$$\tau_{Failure} \approx \tau_{Activation} = \tau_{Emission} - \Delta_{Tolerance}\,. \tag{4.4}$$

In many cases, failure onset is located shortly before the value threshold is violated. However, the approximated failure start is not the exact failure starting point. For instance, rules related to CPU utilization tend to activate symptoms within seconds after a high load is initiated. Even if the time difference between symptom activation and failure start is insignificant, correction is still necessary. Differences of several seconds between symptom timestamps must be taken into account due to limitations of signal sampling by monitoring systems. Furthermore, it cannot be assumed that the tolerance period is always available in the information transported by a symptom.

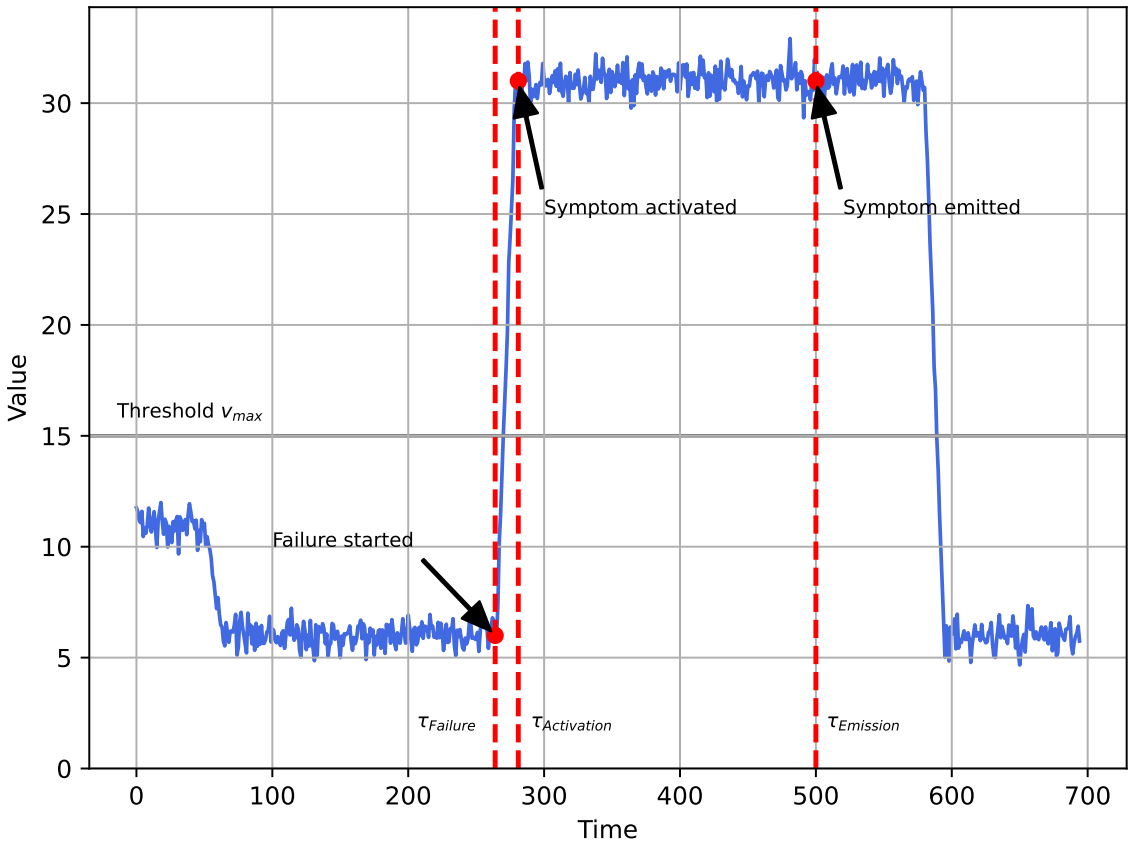


Figure 4.4: Visualization of time-series symptom context denoting moments of failure start, symptom activation and symptom emission

The estimated symptom activation time may constitute a hint for the changepoint detection method, i.e., the method should search for the changepoint closest to the symptom activation time. In turn, if the hint is not available, the changepoint can be searched in the proximity of symptom emission time $\tau_{Emission}$.

Formally, let T be the set of all changepoints detected in time-series M, $\tau_{Activation}$ be the symptom activation time, and $\tau_{Emission}$ be the symptom emission time. If the symptom activation hint is available, the failure start $\tau_{Failure}$ can be estimated as the changepoint

$$\tau_{Failure} = \arg\min_{\tau \in \mathrm{T}} |\tau_{Activation} - \tau| \ , \tag{4.5}$$

whereas, if the symptom activation hint is not available, the failure start can be driven by the symptom emission time, i.e.,

$$\tau_{Failure} = \arg\min_{\tau \in \mathrm{T}} |\tau_{Emission} - \tau| \ . \tag{4.6}$$

Failure onset $\tau_{Failure}$ computed using Equation 4.5 or Equation 4.6 constitutes correction to the symptom timestamp.

4.3.5 Impact of Data Sampling on Changepoint Detection

Since changepoint detection operates on time-series data, the way in which monitoring systems sample time-series may significantly affect the results of the analysis. Currently, no monitoring system collects time-series data in a continuous space. Instead, monitoring systems use time-series sampling to reduce consumed memory and disk space for metric recording and save network bandwidth for metric transmission. The sampling interval is set from a few seconds to several minutes, depending on requirements. For most analytical scenarios, an interval of 15 seconds provides a reasonable balance between resource consumption and time-series accuracy.

A consequence of using sampling for time-series data collection is the risk of missing significant changes in its trends. While this is not a problem for most standard analytical use cases, the issue becomes apparent when statistical analysis is used and an attempt is made to determine the correct cause-effect relationship between time-series and associated symptoms. A short period between trend changes may occur in the middle of the sampling interval, leading to loss of information due to mutual trend changes being folded into single data points at the sampling time. The issue emerges because many metrics in IT systems respond almost simultaneously, e.g., within half a second. A notable example is the relationship between CPU usage and service latency metrics. The sudden increase in application CPU usage typically causes high latency in related services in less than one second.

Figure 4.5 illustrates the sampling problem. Two time-series M^i and M^j are marked in blue and plotted in continuous space, while values recorded by the monitoring system according to the sampling configuration are marked red. Additionally, points scattered in red indicate sampling times according to the sampling interval of 15 seconds. Around the 30th second, a failure occurs, causing a rapid increase in the values of M^i at the time $\tau^i_{Failure} = 35$, and after 5 seconds, an increase in the values of M^j at the time $\tau^j_{Failure} = 40$. Thus, the delay between time-series is 5 seconds, which allows putting a hypothesis on the cause-effect relationship between time-series with the direction $M^i \rightarrow M^j$. However, the significant change in time-series trends occurs in the middle of the sampling period. Time-series values collected by the monitoring system indicate simultaneous increase at the time $\tau^i_{Failure}{}' = \tau^j_{Failure}{}' = 30$. Recorded failure starts are imprecise and folded into a single time point, making it impossible to infer the correct time-series dependency direction.

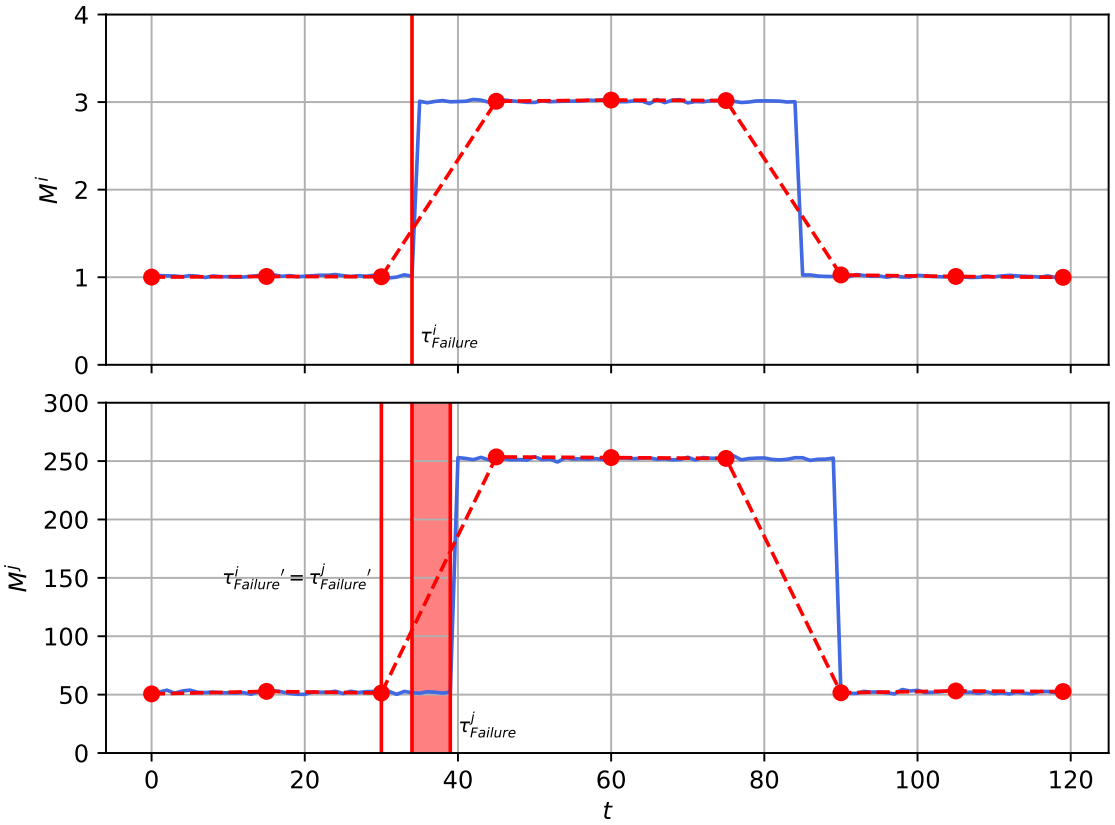


Figure 4.5: Impact of time-series sampling on changepoint detection

Due to the sampling, changepoints for time-series that are interdependent in a short period cannot precisely reflect actual failure starts. As a result, it is infeasible to determine the valid dependency direction for symptoms associated with such time-series. Notably, the limitation is imposed by the monitoring system configuration, which, in turn, results from the practical approach to time-series data collection.

Intuitively, the impact of sampling on changepoint detection can be minimized by shortening the sampling interval. However, in large systems, that would lead to performance degradation and time-series value losses because some values could not be accumulated due to a lack of resources. An interesting alternative could be the use of adaptive monitoring [54], i.e., proactively detect time-series anomalies and temporarily collect more telemetry data only for specific application components.

4.4 Symptom Co-occurrence Analysis

After correcting timestamps for time-series symptoms using changepoint detection techniques described in Section 4.3, each symptom, regardless of the associated telemetry data type, can be translated into the event space. The translation allows applying state-of-the-art techniques to event correlation to discover dependencies between symptoms based on the statistical analysis of historical symptom occurrences. The co-occurrence analysis supported by processing event sequences over long periods and inspecting the consistency of event time lag constitutes an entry point for discovering semantic symptom dependencies motivated in the solution concept.

Detecting event co-occurrence is a challenging task due to the nature of event generation in IT systems. Related events usually do not occur simultaneously but are separated by intervals known as *temporal lag* or *time lag*. Depending on the fault scenario type, the lag can vary from seconds or minutes to hours or days. Furthermore, the time lag may oscillate with noise as the generation of events is influenced by external factors such as the varying characteristics of system operation, the precision of data sampling by the monitoring system, fixed interval of event rule evaluation, or host clocks out of sync. Additionally, it was observed that the number of event instances on both sides of the relation is rarely equal. This effect can be caused by several factors, including orchestration engine response, an administrator performing a repair, or event fluctuation due to anomalous system behaviors.

Importantly, the problem of event correlation centers around processing a large volume of historical data rather than inspecting the properties of single event occurrences in real-time. While the occurrence of events group at a given time point may suggest certain event relations, only the analysis of events over extended periods allows for discovering strong correlations and formulating a hypothesis about causality. Examining event co-occurrence from a single system failure is prone to errors due to random co-occurrence. Conversely, a consistent interweaving of event instances of the same types over weeks or months constitutes a solid base for establishing a correlation with high confidence.

4.4.1 Problem Statement

Mathematically, the problem of event correlation can be formulated as follows. Let S be the sequence of all event instances $S = (e_1, e_2, ..., e_n)$ and E be a set of all possible event types. Further, let $S_A = (a_1, ..., a_m)$ be a subsequence of m event instances from S containing only events of type A with a_i being the timestamp of ith event. Correspondingly, let $S_B = (b_1, ..., b_k)$ be a subsequence of k events of type B. The objective of the correlation task is to determine all pairs of event types $A, B \in E$ that are not independent of each other, i.e.,

$$P(A, B) \neq P(A) \cdot P(B)\,, \tag{4.7}$$

where $P(A)$ and $P(B)$ are probabilities of events A and B, and $P(A, B)$ is their joint probability. Those pairs of event types are correlated.

Additionally, the objective of the task is to discover the temporal dependency between A and B. Equation 4.8 expresses the temporal lag between events where b_j and a_i are event timestamps from subsequences S_B and S_A respectively, t is the constant time lag, and ϵ represents random noise.

$$b_j = a_i + t + \epsilon \tag{4.8}$$

Due to the random noise ϵ, the overall time lag between event instances, i.e., $T = t + \epsilon$, is not constant and must be interpreted as a random variable. Thus, in order to discover the temporal dependency $A \xrightarrow{\mathrm{T}} B$ one needs to compute the distribution of random variable T:

$$P(B|A) = T(B|A) = T(b - a)\,. \tag{4.9}$$

Here, $T(B|A)$ is the conditional probability distribution of events A and B, and can be translated to the probability distribution of event time lag $T(b - a)$ with $b - a$ expressing the time lag between subsequent event pairs.

4.4.2 Selection of Temporal Event Correlation Method

Since the early 1980s, the problem of event correlation has been widely studied. The oldest approaches originate from the domain of expert systems and involve domain experts [55] to transfer the knowledge about event dependencies by manually

searching and defining correlation rules. The advantage of the approach is the high determinism of rules which, being produced by human experts, are verified by many years of experience and intuition. The disadvantage is the significant amount of time required to find valid event correlations. Moreover, the process is error-prone and hardly practicable for complex systems. Rules defined for one system cannot be easily reused in other system domains, and each time a new ruleset has to be created.

An important evolution of the event correlation approach utilizes time windows [56] to detect co-occurring event pairs. A time window of fixed size is shifted over the sequence of events, counting event instances at different window locations to discover frequent pairs. The major drawback of this method is the inability to determine a reasonable window size. If the window is too small, some event relationships will be missed, especially when the time lag between event instances is greater than the window size. Conversely, with a window that is too wide, many false-positive correlations will be returned.

Much attention was devoted to expressing temporal dependency as a fixed time interval $A \xrightarrow{[t_1,t_2]} B$ [57, 58]. It informs that event B occurs after event A within time driven by $[t_1, t_2]$. However, the approach does not allow distinguishing typical intervals from the ones occurring due to randomness, although approaches were proposed to address that issue using statistical methods.

The most recent approaches to event correlation attempt to model the dependency between events as a time lag probability distribution. For instance, the lagEM method proposed by Zeng et al. (2015) [59] models temporal lag as a normal distribution with the maximum likelihood of model parameters estimated by means of the Expectation-Maximization (EM). Further, Zöller et al. (2017) [35, 36] employ the energy distance correlation measure together with the Iterative Closest Points (ICP) algorithm from computer vision for time lag estimation and compute the probability distribution using Kernel Density Estimator (KDE). Finally, Huber et al. (2018) [37] redefine the temporal lag dependency as a binary integer optimization problem and propose the LpMatcher method for approximating the optimal solution using elements of linear programming. All three methods are considered current state-of-the-art.

Due to the high accuracy and relatively better performance than other methods, the ICE method was decided for further research and adoption in implementing the solution prototype. The following section describes the method with proposed extensions in more detail.

4.4.3 Iterative Closest Events

The Iterative Closest Events method consists of three complementary stages. First, event sequences having weak correlations are eliminated based on the energy distance measure to reduce the computational effort in further stages. Second, the optimal assignment of event instances between event sequences is determined using Iterative Closest Points, and the time lag between matched event pairs is calculated. Last, based on the obtained event assignment and time lag, a non-parametric probability distribution is estimated using the Kernel Density Estimator.

In detail, the first method stage reduces the space of possible event type correlations to minimize the computational effort, i.e., only perform heavy calculations for those pairs which show significant correlation. The reduction is implemented by performing a statistical test for event independence based on the energy distance measure [60]. The energy distance is inspired by the gravitational energy potential in physics. Like two objects attracting each other in a gravitational field, samples of two distributions affect each other depending on their distance. If equivalent distributions produced both sample sets, their potential energy would be zero. Accordingly, the more similar event distributions for a pair of event types are, the more event instances are correlated. Therefore, only pairs with low energy distance should be processed in subsequent method stages.

The key property of energy distance determining its use in the ICE method is the lack of need for knowing the pairwise event assignment across event sequences. The energy distance comprises only the analysis of distances between individual event instances and does not require the expensive computation of temporal lag distribution. Moreover, event sequences do not have to be of equal size.

Equation 4.10 defines the energy distance measure for a pair of event sequences S_A and S_B.

$$\begin{aligned} Cor_{ED}(S_A, S_B) = & \frac{2}{m \cdot k} \sum_{i=1}^{m} \sum_{j=1}^{k} \|a_i - b_j\| \\ & - \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} \|a_i - a_j\| \\ & - \frac{1}{k^2} \sum_{i=1}^{k} \sum_{j=1}^{k} \|b_i - b_j\| \,. \end{aligned} \tag{4.10}$$

In order to eliminate weak correlations between pairs of event sequences, a statistical test is performed based on the energy distance. The test is defined as

$$S(S_A, S_B) = T(S_A, S_A) + T(S_A, S_B) + T(S_B, S_A) + T(S_B, S_B)\,, \tag{4.11}$$

where

$$T(S_A, S_B) = \frac{m \cdot n}{m + n} Cor_{ED}(S_A, S_B)\,. \tag{4.12}$$

Based on the test, the p-value can be calculated as

$$p = \frac{1}{h} \sum_{i=1}^{h} \mathbf{1}(S(S_A, S_B) > S(S_A^*, S_B^*))\,, \tag{4.13}$$

where $\mathbf{1}$ is a predicate function returning 0 or 1 depending on the boolean condition under test.

In order to obtain the p-value, a permutation test is done with a hypothesis put for event independence. S_A and S_B are merged into a single event sequence, randomly permuted, and then split again into two distinct subsequences S_A^* and S_B^*. Then, the output of the test for original event subsequences is compared to the output of the test for subsequences obtained from permutation. The test preceded by permutation is repeated h times, and the ratio of comparison results is substituted as the value of p. If the p-value is larger than $\alpha = 0.05$, the null-hypothesis can be rejected implying event sequences are not independent

After discarding pairs of event sequences showing weak correlation, the next method stage aims to determine the temporal lag dependency. To quantify the dependency, event instances from both sequences must be matched into pairs, i.e., for each event instance $a_i \in S_A$, the corresponding event instance $b_i \in S_B$ must be found. However, considering that events may be shifted by a time lag oscillating with noise as pointed in the Equation 4.8 and some events may be missing in both sequences, determining the correct event pairing is not obvious.

In order to obtain an optimal match, the ICE method adopts the algorithm from the computer vision called Iterative Closest Points (ICP) [61]. The algorithm takes two sets of points on its input, namely the *data* set and the *reference* set. Then, it searches for the rigid transformation of points from the data set into points of the reference set so that the distance between points in two sets after applying the transformation to the data set is minimized.

The ICP algorithm looks for point transformation in three-dimensional space in its original form. Given that event sequences are one-dimensional vectors, the cost function can be simplified to

$$\frac{1}{m}\sum_{i=1}^{m}(a_i + t - b_{\alpha_i(t)})^2\,, \tag{4.14}$$

where $\alpha_i(t)$ is the index of the closest event of type B to the event a_i.

The optimal point transformation minimizing the cost function (Equation 4.14) is accomplished by iteratively executing two steps until the convergence is met.

In the first step, the algorithm searches for the closest neighbor in the reference set for each point in the data set. Typically, the closest neighbor is found by taking the squared Euclidean distance between a given point and all other points and taking the neighbor with the smallest distance, i.e.,

$$\alpha_i(t^{(l)}) = \arg\min_j (a_i + t^{(l)} - b_j)^2\,, \tag{4.15}$$

where $t^{(l)}$ is the transformation obtained in the lth iteration of the algorithm.

In the second step, given found point assignments, the algorithm attempts to find the most optimal transformation $t^{(l+1)}$ minimizing the cost function. Equation 4.16 describes the optimization problem where t is the searched transformation variable and m is the size of the event sequence corresponding to the source event type in the processed event relation. The problem can be solved by using an arbitrary solver, e.g., the least-squares method.

$$t^{(l+1)} = \arg\min_t \frac{1}{m}\sum_{i=1}^{m}(a_i + t - b_{\alpha_i(t^{(l)})})^2 \tag{4.16}$$

The presented optimization problem is sensitive to local minima. Therefore, it is necessary to estimate the initial guess guiding the algorithm to the correct solution. In this dissertation, the RANSAC algorithm [62] is used which consists of three steps. First, n elements from the S_A sequence are selected randomly. Second, for each element, the k nearest neighbors are searched in the sequence S_B, and one neighbor is selected at random. Finally, the transformation for elements selected from both sequences is computed according to the Equation 4.16. The above steps are repeated q times, and the transformation for which the cost function error was minimal is selected for the initial sequence assignment.

Based on the closest neighbor assignments $\alpha_i(t^*)$ between event sequences and the final transformation t^*, it is possible to determine the optimal time lag between each event pair. However, even with an optimal transformation found for event sequences, single outliers may emerge and disturb the modeling of temporal dependency. Therefore, a three-step outlier removal process was adopted to detect and eliminate abnormal time lags. First, obtained time lags are preprocessed by correcting their signs, i.e., signs of all time lags are inverted if most time lags are negative. For instance, considering a dependency $A \rightarrow B$ while the true event occurrence is $B \rightarrow A$ would result in most lags turning negative. Hence, the direction of temporal dependency must be inverted. Additionally, the first process step removes domain-inconsistent time lags, e.g., those that fall outside the allowed fixed value range.

In the second step, outliers are removed using the Density-based Local Outliers (LOF) method [63]. Contrary to traditional outlier detection methods that consider outliers from a global perspective, LOF searches for outliers locally by analyzing a restricted neighborhood of each sample and evaluating the sample against the densities of their neighborhoods. The method was employed based on the observation that some event sequences co-occur consistently with several different time lags of variable likelihood, e.g., 90% of time lags is 60 seconds on average, 8% is 30 seconds, while the remaining 2% are actual outliers. Globally, the 8% part of the population can be interpreted as statistically marginal. In such a case, methods similar to z-scores mark samples as outliers. The LOF method, in turn, allows removing only abnormal time lags while preserving valid time lags that occurred with lower likelihood.

Following the observation that some event sequences co-occur with different time lags, in the last step of the outlier removal process, time lags are clustered using the Density-Based Spatial Clustering (DBSCAN) algorithm [64, 65] to identify time lag clusters and select the strongest one to represent the temporal dependency. That is dictated by another observation, in which estimating the time lag distribution based on all time lag clusters produces a diffused distribution, i.e., a distribution with a normal curve that is flat and widely stretched due to distances between individual samples. Such a distribution results in temporal dependency weakening upon its quantification. Instead, the strongest time lag cluster is assumed for dependency quantification to maximize the dependency significance. In this dissertation, the assessment of cluster strength is based on evaluating the number of samples in the cluster.

Given time lags filtered through the proposed outlier removal process, the temporal lag distribution is calculated in the last stage of the correlation method. Time lags are used as samples to train the time lag distribution using Kernel Density

Estimator (KDE). KDE is a common statistical instrument for estimating the unknown distribution based on a random sample set. In a general form, the lag distribution can be described as

$$\hat{f}_{lag}(t^*) = \frac{1}{m}\sum_{i=1}^{m} K(\frac{a_i - b_{\alpha_i(t^*)}}{h}) \,, \tag{4.17}$$

where $K(\cdot)$ is Gaussian kernel function

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} \tag{4.18}$$

and h is a bandwidth parameter determining the smoothness of the distribution curve. The bandwidth selection exhibits a strong influence on the resulting distribution estimate. Too small bandwidth results in distribution containing much noise, whereas too large bandwidth generalizes the distribution, omitting essential features. In this dissertation, the bandwidth is selected using the rule-of-thumb bandwidth estimator [66]

$$h = \left(\frac{4\sigma^5}{3n}\right) \,, \tag{4.19}$$

where n is the number of samples and σ is standard deviation.

4.4.4 Quantification of Temporal Event Correlation

According to the adopted symptom correlation framework, the correlation method must provide three correlation properties, i.e., correlation existence, temporal direction, and correlation strength. In the ICE method, correlation existence is determined by putting a statistical hypothesis on event type independence and performing a statistical test based on the energy distance measure. Depending on the p-value obtained from the statistical test, the hypothesis may be rejected, implying that event types are not independent, or cannot be rejected, implying insufficient evidence to reject the hypothesis. The former output establishes a correlation between event types, whereas the latter excludes pair of event types from further processing.

The second correlation property, i.e., the temporal direction, is determined based on inspecting signs of time lags between event pairs. After a correct event assignment between event sequences is found using elements of the ICP method (Equation 4.14), the time lag between each pair of events A and B is calculated, i.e., the timestamp

of A event is subtracted from the timestamp of the corresponding B event according to the event assignment. If the actual temporal direction in event dependency is $A \rightarrow B$, signs of time lags will be positive, while if the actual temporal direction is the opposite, then signs of time lags will be negative. Negative signs suggest that the dependency should be reversed. Hence, the dependency direction is reversed in the proposed correlation process if more than half of the calculated time lags have negative signs.

Last, the correlation strength can be calculated based on the probability distribution obtained using KDE in the last step of the ICE method (Equation 4.17). The distribution describes the probability of event co-occurrence for each time lag value. As such, it adequately represents the dependency between a pair of event types.

Mutual information [67] was selected as a dependency measure for event correlation. Its concept is intimately linked to the entropy of a random variable, a fundamental notion in information theory. However, in contrast to entropy, mutual information measures the information of one random variable about another one instead of considering the information of a single random variable. Figure 4.6 shows that correlation.

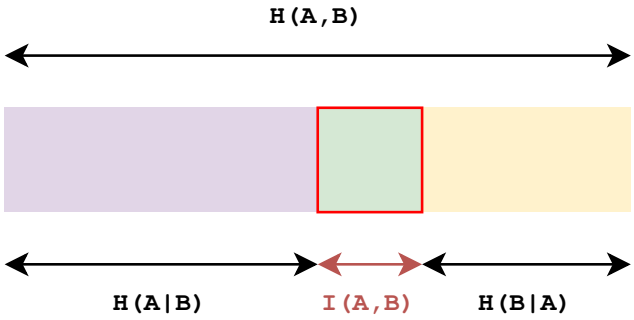


Figure 4.6: Additive and subtractive relationships of entropy and mutual information measures associated with correlated variables A and B

By definition, mutual information is a measure of mutual dependence between two random variables. Intuitively, mutual information measures how much information about B can be determined from knowing A, i.e., how much learning one variable reduces the uncertainty about the other. In the considered event correlation context, mutual information quantifies how knowing the timestamp of event A enables predicting the timestamp of event B.

For continuous probability distributions, such as the one produced at the output of the ICE method, the mutual information is defined as a double sum

$$\begin{aligned} I(A,B) &= H(B) - H(B|A) \\ &= \iint_{-\infty}^{+\infty} P(a,b) \log_2 \left(\frac{P(a,b)}{P(a) \cdot P(b)} \right) da\, db \\ &= \iint_{-\infty}^{+\infty} P(b|a) \log_2 \left(\frac{P(b|a)}{P(b)} \right) da\, db\,, \end{aligned} \tag{4.20}$$

where $P(A)$ and $P(B)$ are marginal probability distributions of event occurrence for events of types A and B, and $P(A,B)$ is their joint probability distribution. In fact, $P(A,B)$ is the time lag probability distribution $T(x)$. Then, considering that the time lag distribution depends only on the value of time lag, and realizations of marginal probabilities $P(A)$ and $P(B)$ are fixed, i.e.,

$$\begin{aligned} P(A) &= \frac{|S_A|}{|S|}\,, \\ P(B) &= \frac{|S_B|}{|S|}\,, \end{aligned} \tag{4.21}$$

the mutual information can be reduced to

$$I(A,B) = \int_{-\infty}^{+\infty} T(x) P(A) log_2 \left(\frac{T(x)}{P(B)} \right) dx\,. \tag{4.22}$$

This allows the calculation of the correlation coefficient

$$Cor_{MI}(S_A, S_B) = \frac{I(A,B)}{H(A,B)}\,, \tag{4.23}$$

where $H(A,B)$ is the joint entropy standardizing the coefficient [67, 68].

Given the Cor_{MI} coefficient, the correlation strength in the proposed event correlation method based on event co-occurrence can be defined as

$$Cor_{CO}(a,b) = Cor_{MI}(S_A, S_B)\,, \tag{4.24}$$

where a and b are analyzed event instances of types A and B, and S_A and S_B are sequences of historical event occurrences of the same types.

The coefficient value is adaptive in the sense that with each new pair of events, the coefficient produces a marginally different value towards potential changes in co-occurrence characteristics. Consequently, over long periods comprising many event instances, significant changes in co-occurrence, e.g., extending the time lag from 5 to 8 seconds, will be included. However, due to the slow pace of the adaptation process, performing heavy calculations related to discovering temporal event dependency for each new event pair is impractical. Therefore, coefficient values should be cached and read from a lookup structure updated according to a fixed rate.

4.4.5 Temporal Event Correlation Examples

This section presents examples of event co-occurrence scenarios with results produced by the adapted version of the ICE method. The objective is to verify subsequent method stages, i.e., event sequence alignment using ICP (Equation 4.14) and estimation of time lag probability distribution using KDE (Equation 4.17). Scenarios are simulated by synthetically-generated event sequences based on probability distributions obtained using various statistical parameterizations.

The following paragraphs describe input event sequences for each scenario and discuss obtained correlation results at each method stage. In addition, two diagram types are provided for each scenario. The first diagram type (e.g., Figure 4.7) enables evaluating the quality of event alignment between event sequences by illustrating event instances for a pair of event types in the time space. Event timestamps are denoted in seconds. Sequences marked in blue represent events of type A (top) and B (bottom). The sequence marked in red illustrates the A sequence shifted according to the transformation calculated using the ICP algorithm (Equation 4.16). Moreover, red lines connect the original A sequence with the corresponding closest neighbors in the B sequence. The more the shifted sequence of A events overlaps with the original sequence of B events, the better the obtained event sequence alignment. Finding a good match indicates that event instances show strong co-occurrence.

The second diagram type (e.g., Figure 4.8) illustrates the probability density distribution (PDF) obtained in the Equation 4.17 by training KDE with samples using time lags calculated from matched event pairs. The horizontal axis holds lag values in continuous time space denoted in seconds, while the vertical axis describes the probability density value. Additionally, the true event distribution is marked in blue, and the distribution estimated using the ICE method is marked in red. The more two distribution curves match together, the better the obtained estimation accuracy.

Scenario 1 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with a normally distributed time lag $\mathcal{N}(5; 1)$. Furthermore, no events are lost.

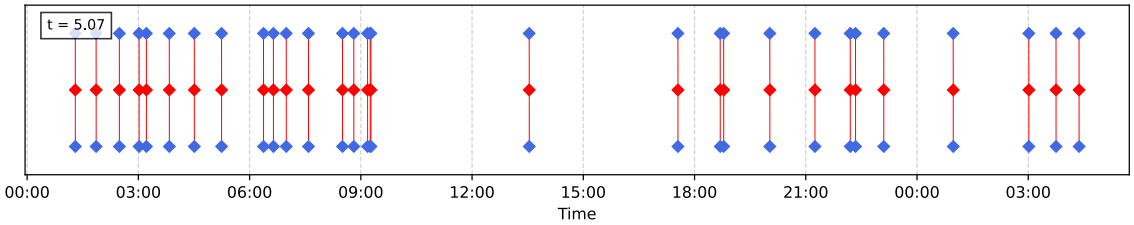


Figure 4.7: Temporal event alignment: normally distributed time lag ($\mu = 5$), no events lost

In this scenario, events co-occur orderly and there are no overlapping event pairs. As expected, the algorithm computes the correct event assignment between two sequences (Figure 4.7). The resulting translation $t = 5.07$ matches the actual event shift based on the normal distribution with $\mu = 5$. The estimated probability density function (Figure 4.8) accurately maps the true distribution.

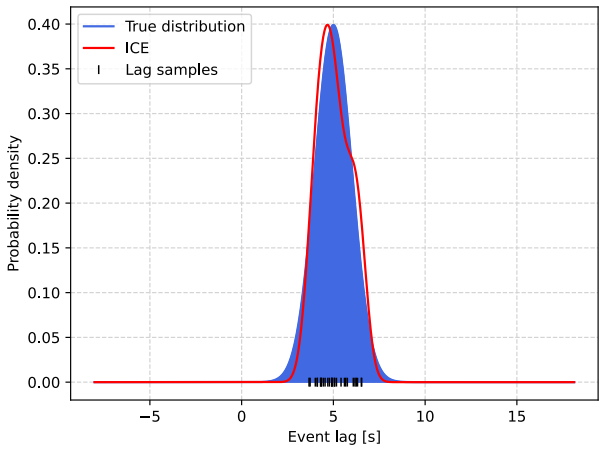


Figure 4.8: Estimated and true time lag probability distribution: normally distributed time lag ($\mu = 5$), no events lost

Scenario 2 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with a normally distributed time lag $\mathcal{N}(1800; 1)$. Furthermore, no events are lost.

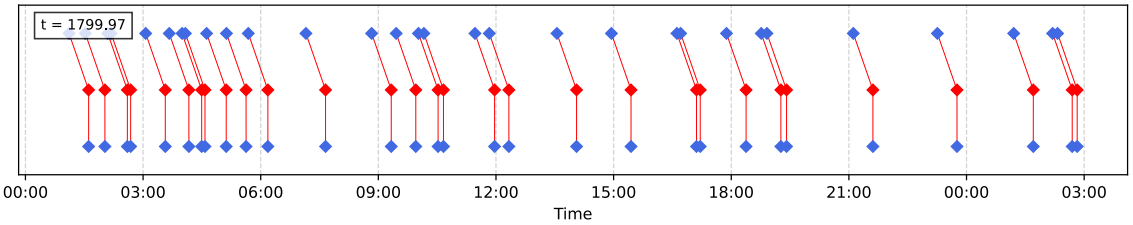


Figure 4.9: Temporal event alignment: normally distributed time lag ($\mu = 1800$), no events lost

Similar to the previous scenario, events co-occur orderly and there are no overlapping event pairs. The algorithm computes the correct event assignment between event sequences (Figure 4.9). The estimated translation $t = 1799.97$ matches exactly the actual event shift based on the normal distribution with $\mu = 1800$. The obtained probability density function (Figure 4.10) accurately maps the true distribution.

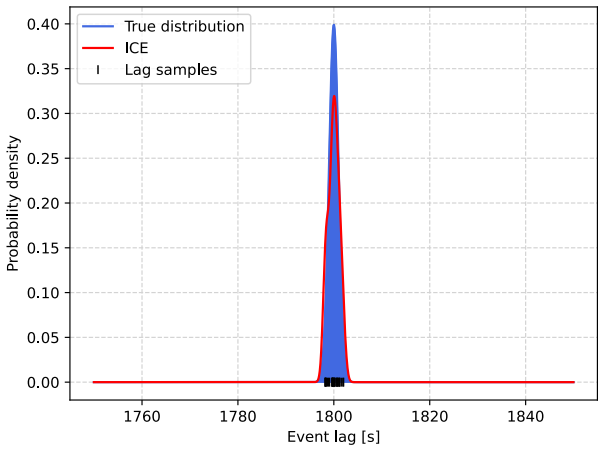


Figure 4.10: Estimated and true time lag probability distribution: normally distributed time lag ($\mu = 1800$), no events lost

Scenario 3 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with two normally distributed time lags $\mathcal{N}(5; 1)$ and $\mathcal{N}(15; 1)$. Furthermore, no events are lost.

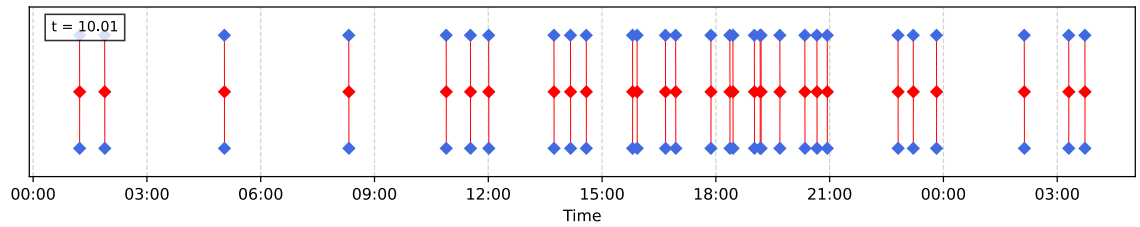


Figure 4.11: Temporal event alignment: two normally distributed time lags ($\mu_A = 5, \mu_B = 15$), no events lost

The scenario reproduces a situation where events are semantically related but co-occur with two distinct but consistent time lags. Events of type B follow events of type A according to two normal distributions with different averages $\mu_A = 5$ and $\mu_B = 15$. Events co-occur orderly and there are no overlapping event pairs. The found event assignment between sequences (Figure 4.11) is correct, and the estimated translation of $t = 10.01$, as expected, averages the means of both normal distributions. The obtained probability density function (Figure 4.12) accurately reflects the true distribution with two characteristic increases in the proximity of distribution means.

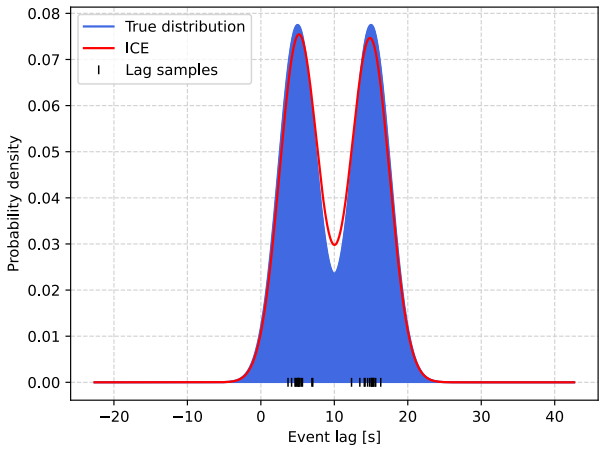


Figure 4.12: Estimated and true time lag probability distribution: two normally distributed time lags ($\mu_A = 5, \mu_B = 15$), no events lost

Scenario 4 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with a normally distributed time lag $\mathcal{N}(1800; 1)$. Furthermore, in contrast to previous scenarios, both event types are lost with a probability of 10%.

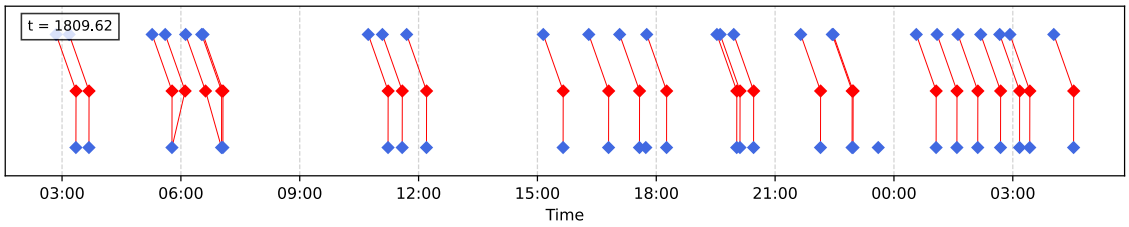


Figure 4.13: Temporal event alignment: normally distributed time lag ($\mu = 1800$), 10% of events lost in both event sequences

The scenario simulates a situation where events A and B co-occur consistently, but some event instances are missing. The missing events may be caused by errors in event transmission or varying system operation characteristics. Consequently, some event instances ended unpaired, e.g., the instance of event B at around 00:00. Moreover, missing events are compensated by sharing event instances across event sequences, e.g., at around 05:00 and 07:00, several event instances of type A are assigned to the same event instance of type B.

Despite missing event instances in both sequences, the found sequence assignment (Figure 4.13) is correct. The obtained translation $t = 1809.62$ confirms the convergence with the distribution mean $\mu = 1800$. The estimated probability density function (Figure 4.14) accurately maps the true distribution.

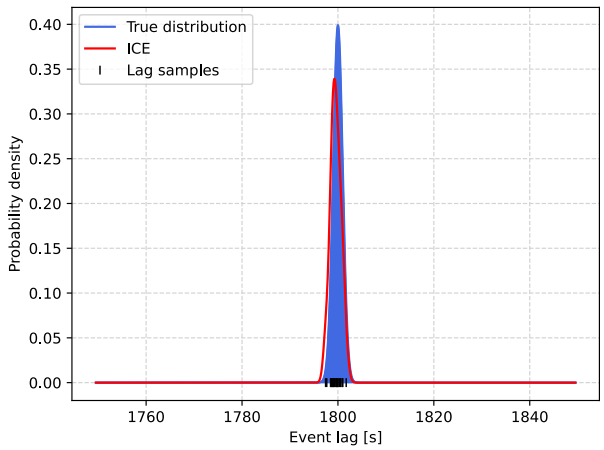


Figure 4.14: Estimated and true time lag probability distribution: normally distributed time lag ($\mu = 1800$), , 10% of events lost in both event sequences

Scenario 5 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with a normally distributed time lag $\mathcal{N}(1800; 1)$. Furthermore, both event types are lost with a probability of 50%.

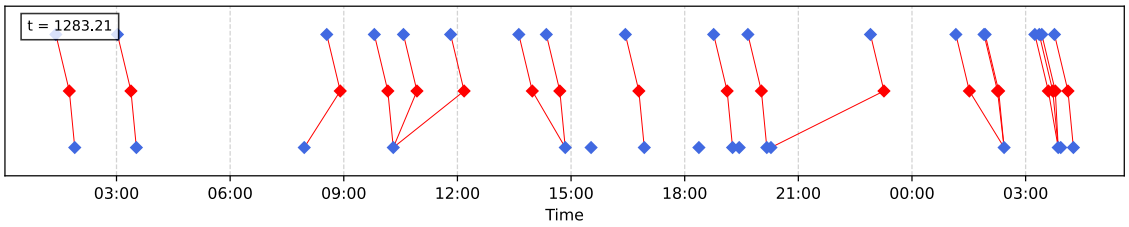


Figure 4.15: Temporal event alignment: normally distributed time lag ($\mu = 1800$), 50% of events lost in both event sequences

Similar to the previous scenario, event instances of both types are lost. However, the probability of losing an event is 50%. Such a severe loss of events may reflect a situation where the symptom triggering rule causes nondeterministic symptom emission, or symptoms are not correlated. As evident in the diagram (Figure 4.15), sequence alignment has lost accuracy. The translation $t = 1283.21$ is clearly underestimated. Multiple events remain unpaired, and some B events are shared among multiple A events. Notably, many lost events of type B at around 10:30 produce significant distances to assigned events of type A. That, in turn, underestimates the event alignment, i.e., sequence translation must be corrected to minimize the distances, but at the same time, the matching of other pairs gets broken. The probability density function (Figure 4.16) significantly diverges from the true distribution. The variety of obtained time lags results in a high standard deviation, lowering density values for lags in the proximity of the true distribution mean.

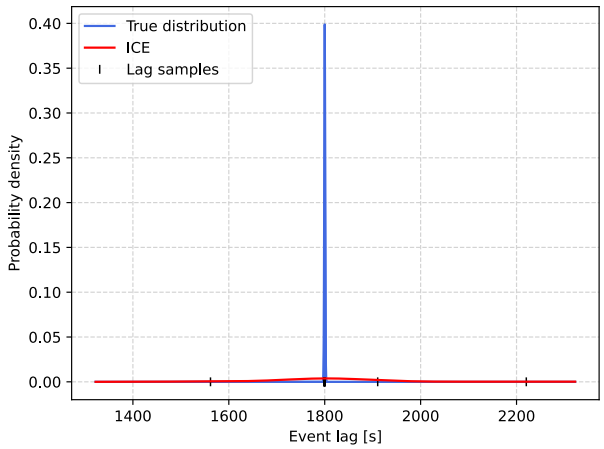


Figure 4.16: Estimated and true time lag probability distribution: normally distributed time lag ($\mu = 1800$), 50% of events lost in both event sequences

Scenario 6 The sequence contains 30 elements in total. Event A occurs based on an uniform distribution with $\mathcal{U}(1; 10000)$. Event B occurrs after event A with a normally distributed time lag $\mathcal{N}(1800; 1)$. Furthermore, events of type A are deleted with a probability of 50%.

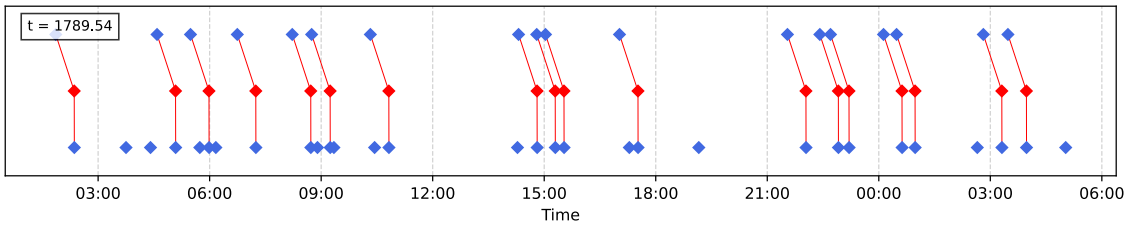


Figure 4.17: Temporal event alignment: normally distributed time lag ($\mu = 1800$), 50% of events lost in the A sequence

The scenario objective is to evaluate the correlation method when instances of two event types consistently co-occur, but one of the event sequences contains additional event instances, in this case, the sequence comprising events of type B. Additional events may be a consequence of correlation across several event types, e.g., events of types A and B are correlated, but another event C relates to event B as well, causing missing A event occurrences in situations when C and B events co-occur.

Despite redundant events in the B sequence, the resulting translation $t = 1789.54$ is close to the distribution mean $\mu = 1800$. Figure 4.17 visually confirms the accuracy of sequence assignment. The obtained probability density function (Figure 4.18) is well-positioned but has a lower standard deviation than the true distribution due to the strict outlier detection.

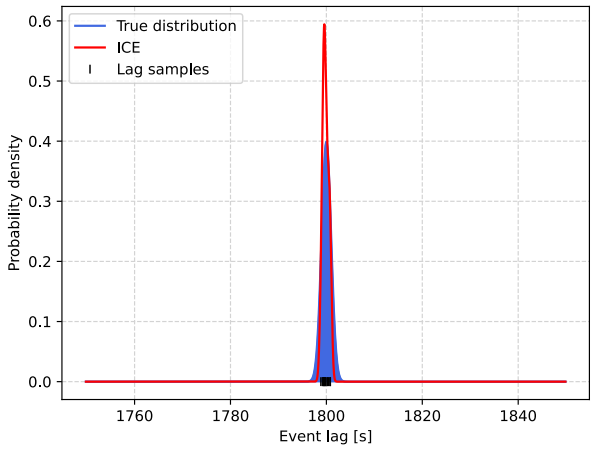


Figure 4.18: Estimated and true time lag probability distribution: normally distributed time lag ($\mu = 1800$), 50% of events lost in the A sequence

Scenario 7 The sequence contains 30 elements in total. Both event types occur based on an uniform distribution with $\mathcal{U}(1; 10000)$. Furthermore, no events are lost.

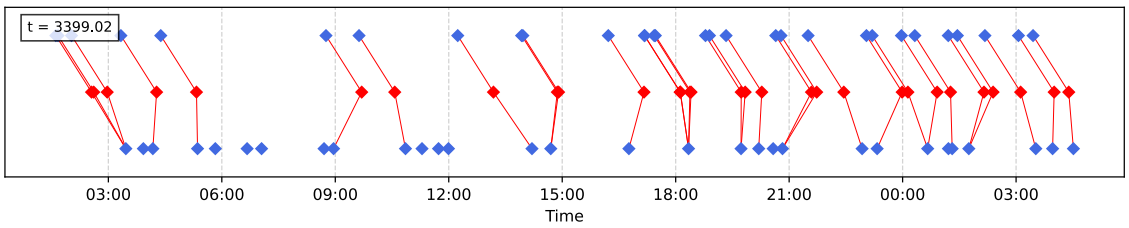


Figure 4.19: Temporal event alignment: time lag generated based on uniform distribution

The scenario reflects a situation where event sequences are not correlated. Both event sequences are generated using a uniform distribution without imposing a co-occurrence relationship. As in previous scenarios where events were lost, many events remain unpaired. Moreover, many events of type B are shared by events of type A. The found sequence assignment (Figure 4.19) does not provide a consistent event interleaving. Events are randomly matched with no consistent order of occurrence. The obtained translation $t = 3399.02$ is highly overestimated. Intuitively, the probability density function (Figure 4.20) is significantly flattened due to time lags spread over the wide range of uniform distribution.

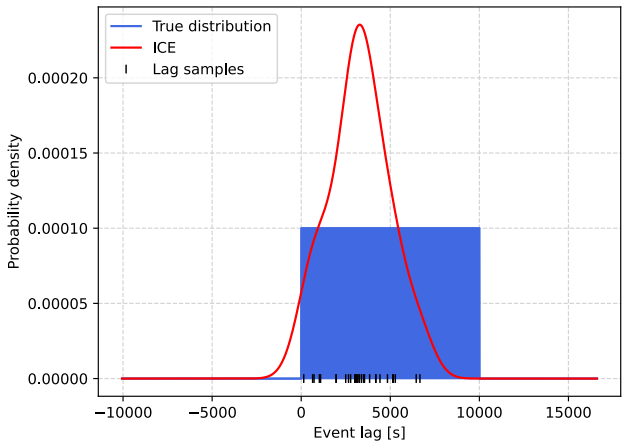


Figure 4.20: Estimated and true time lag probability distribution: time lag generated based on uniform distribution

4.5 Symptom Time Lag Analysis

The co-occurrence analysis introduced in Section 4.4 allows determining the existence, direction, and strength of the correlation between a given event pair. For a given pair of new event instances of types A and B, it answers whether events of types A and B showed statistically strong co-occurrence in the past. However, the answer is based solely on processing historical event data, and it does not take into account the properties of new event pairs. The properties of new event instances have a marginal contribution to the correlation coefficient value.

Symptom time lag analysis enables the inclusion of newly observed event pairs in the correlation process. It utilizes the time lag distribution computed as part of the event co-occurrence analysis to produce a distinct correlation coefficient. Specifically, the time lag calculated between events in a new pair is evaluated against the time lag distribution corresponding to processed event types, and the distance between the new lag and the distribution mean is used to estimate the correlation strength. The coefficient takes high values for time lags centered around the distribution mean and converges to zero as the lag value moves away from the mean. Intuitively, the more typical the time lag between event instances is compared to past event occurrences, the higher the coefficient value.

Assume the system emitted a pair of symptoms, one of which is a remainder after the previous fault, and the other reports the beginning of a new fault. The time lag between symptoms is 20 seconds, while in the past, the time lag for the same event types oscillated around 3 seconds on average. Based on the event co-occurrence analysis, symptoms would show a semantically strong correlation because, historically, they have occurred consistently with an approximately constant time lag. The time lag analysis, in turn, by comparing the time lag calculated for the new symptom pair (20 seconds) to the time lag distribution, would classify the time lag as an outlier and mark a lack of correlation between symptoms. Thus, the time lag analysis complements the event co-occurrence analysis, especially in scenarios involving parallel faults.

The calculation of the correlation coefficient is based on the interquartile range (IQR), a measure of statistical dispersion used to study data distribution characteristics. By definition, it is equal to the difference between the 75th and 25th percentile or between the third and first quartile, i.e.,

$$IQR = Q_3 - Q1\,. \tag{4.25}$$

In the literature, the IQR is often presented in a box plot to facilitate the analysis of the position, dispersion, and shape of the distribution. Figure 4.21 illustrates an exemplary box plot. The plot comprises elements that make up the statistical five-number summary, i.e., minimum, maximum, median, first quartile (Q_1), and third quartile (Q_3) of the dataset. Additionally, the diagram illustrates the IQR as a box spanning between the upper and lower quartiles.

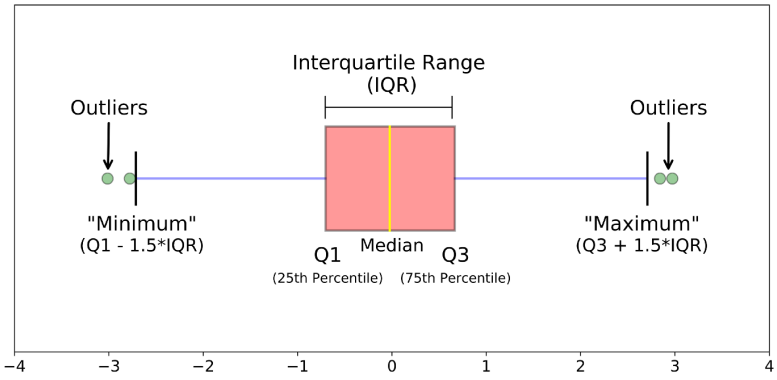


Figure 4.21: Example of Interquartile range boxplot

More importantly, IQR is widely employed to detect data outliers and thus can be used to differentiate valid event time lags. Outliers are observations that fall outside the range defined by thresholds:

$$\begin{aligned} \Theta_{Lower} &= Q_1 - 1.5 \cdot IQR\,, \\ \Theta_{Upper} &= Q_3 + 1.5 \cdot IQR\,. \end{aligned} \tag{4.26}$$

Both minimum and maximum of the data, marked as whiskers in the box plot, exclude outliers based on IQR thresholds. Outliers are plotted as green points.

Given lower and upper time lag thresholds calculated based on the time lag distribution, the correlation coefficient Cor_{TL} can be determined using the formula:

$$Cor_{TL}(a,b) = \begin{cases} 0 & \text{if } \tau < \Theta_{Lower} \\ 1 - \frac{|\mu-\tau|}{|\mu-\Theta_{Lower}|} & \text{if } \tau < \mu \\ 1 & \text{if } \tau = \mu \\ 1 - \frac{|\mu-\tau|}{|\mu-\Theta_{Upper}|} & \text{if } \tau > \mu \\ 0 & \text{if } \tau > \Theta_{Upper} \end{cases}, \tag{4.27}$$

where τ is the time lag between event instances a and b, while $(\mu, \Theta_{Lower}, \Theta_{Upper})$ are parameters of time lag distribution estimated for the corresponding event types.

The coefficient value takes 0 if the lag value exceeds the selected lower or upper threshold since lag values outside the range constrained by thresholds are treated as outliers. Contrary, the coefficient value takes 1 for a lag equal to the distribution mean μ. In other cases, the range defined by the lower and upper threshold is split into two subranges corresponding to lag values before and after the distribution mean. The coefficient is calculated as the lag distance from the distribution mean and standardized by the length of the corresponding lag range.

4.6 Symptom Time-series Analysis

Another piece of information that can be effectively used to infer correlations between symptoms as part of the symptom correlation framework is time-series data associated with some symptoms. A time-series describes a trend of a given metric (e.g., CPU usage or service latency) by recording its values at time points distributed evenly according to a uniform interval. Typically, when a failure occurs, the time-series trend responds by a sudden increase or decrease of values and reverts to its original characteristics once the failure is resolved. When a fault causes an extensive system degradation involving many system components and, as a result, many symptoms are emitted, the co-occurrence of changes in time-series trends provides an opportunity to determine the correlation between symptoms. For instance, if trends of time-series M^i and M^j start to increase simultaneously and at the same rate, it can be assumed that time-series, hence relevant symptoms, are correlated.

Pearson correlation coefficient (PCC) [69, 70] is the most common statistical method for calculating the linear correlation between a pair of time-series variables. Given a pair of time-series (M^i, M^j), where time-series $M^i = (m_1^i, m_2^i, \ldots, m_t^i)$ and $M^j = (m_1^j, m_2^j, \ldots, m_t^j)$ are sequences of discrete measurements of two system metrics over time, and m_t^i, m_t^j are their values at the most recent time point t, the formula for calculating the population correlation coefficient is the covariance of time-series variables divided by the product of their standard deviations, i.e.,

$$\rho_{M^i,M^j} = \frac{cov(M^i, M^j)}{\sigma_{M^i}\sigma_{M^j}}, \tag{4.28}$$

where *cov* is the covariance, σ_{M^i} is the standard deviation of time-series variable M^i, and σ_{M^j} is the standard deviation of time-series variable M^j.

The correlation coefficient ranges from -1 to 1. A value of 1 implies that all data points lie on a line for which M^j increases as M^i increases (Figure 4.22). A value of -1 implies that all data points lie on a line for which M^j decreases as M^i increases

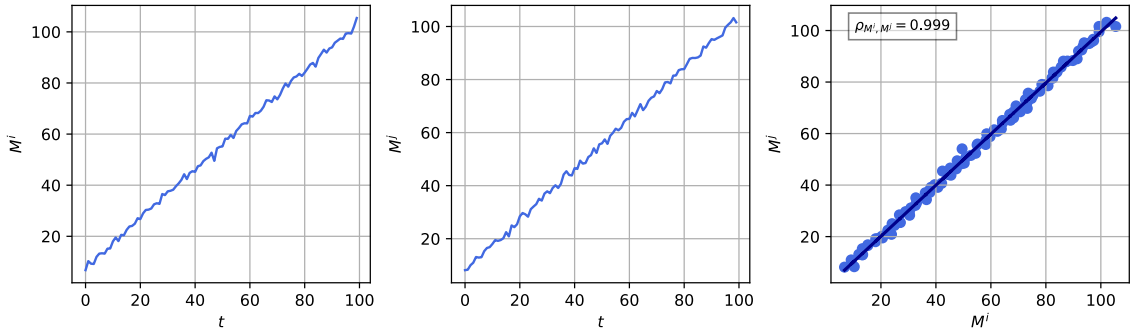


Figure 4.22: Linear time-series correlation: $\rho \approx 1$

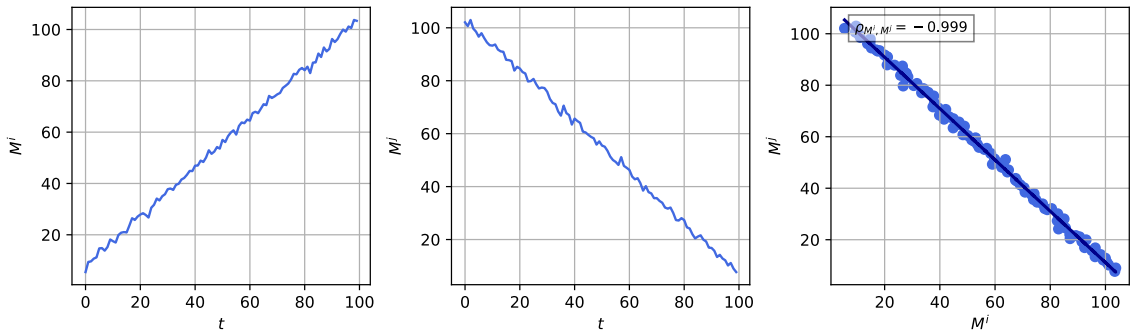


Figure 4.23: Linear time-series correlation: $\rho \approx -1$

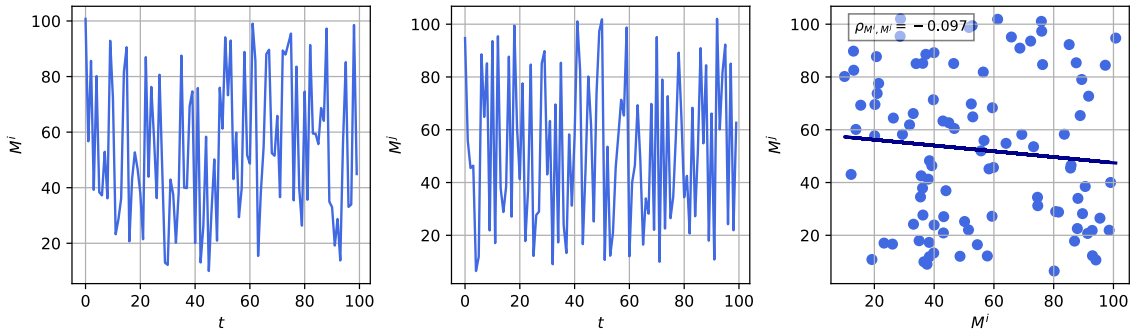


Figure 4.24: Linear time-series correlation: $\rho \approx 0$

(Figure 4.23). A value of 0 implies that there is no linear correlation between two variables (Figure 4.24).

When correlating time-series data associated with symptoms, it is not mandatory to correlate the entire population of time-series variables as denoted in Equation 4.28. Such an extensive correlation is inadvisable due to time-series showing many variations over wide time ranges. Their overall correlation would not be effective, resulting in numerous distortions. Therefore, the practical way to correlate time-series data is to correlate smaller fragments, i.e., samples, in the proximity of areas that show significant changes in characteristics.

A common approach to limiting fragments of correlated time-series is to use a window of fixed size [33]. Given a length w and a time point t, a fixed window for time-series M^i, denoted as $m^i_{t,w}$, is the sample $(m^i_{t-w+1}, \ldots, m^i_t)$. The formula for calculating the coefficient for a pair of time-series samples $m^i_{t,w}$ and $m^j_{t,w}$ is defined as

$$r^{ij}_{t,w} = \frac{\sum_{\tau=1}^{w}(m^i_\tau - \overline{m^i_t})(m^j_\tau - \overline{m^j_t})}{\sigma^i_{t,w}\sigma^j_{t,w}}, \tag{4.29}$$

where m^i_τ and m^j_τ are individual sample points indexed with τ, $\overline{m^i_t}$ is the sample mean, and $\sigma^i_{t,w}$ and $\sigma^j_{t,w}$ are standard deviations.

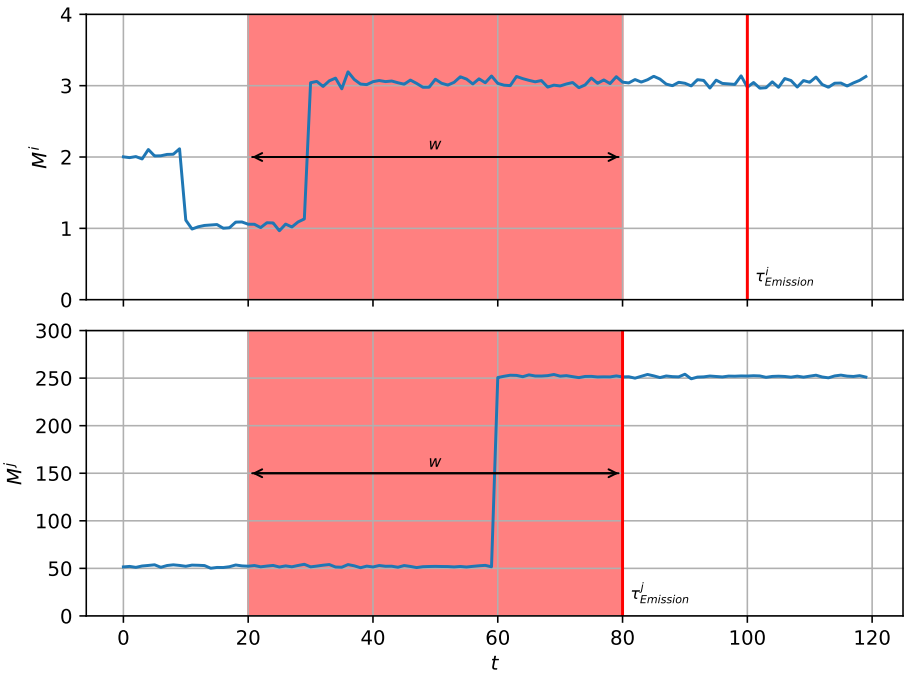


Figure 4.25: Time-series fragments for Pearson correlation designated by the fixed sliding window

Figure 4.25 plots a pair of time-series in a failure context. The time-series M^i shows a sudden increase at the 30th second of the experiment, while the time-series M^j responds with a similar increase at about 60th second, with a delay of about 30 seconds from M^i. Time-series fragments designated by the sliding window of fixed size $w = 60$ are marked red. Additionally, times of symptom emission, i.e., $\tau^i_{Emission}$ and $\tau^j_{Emission}$, are marked with vertical lines.

Pearson correlation coefficient for time-series fragments illustrated in Figure 4.25 is 0.54. Due to the delayed reaction in M^j, time-series are shifted by about 30 seconds. However, designated fragments do not take the time shift into account. Thus, as M^i values increase at the 30th second of the experiment, values of M^j remain relatively stable, lowering the coefficient value due to a lack of linear relationship. Furthermore,

with a wider time window, e.g., 5 minutes ($w = 300$), the coefficient value would be influenced by past changes in time-series, e.g., the increase in M^i occurring in the first 20 seconds of the experiment.

In order to improve the correlation accuracy in terms of designating time-series fragments for correlation, this work proposes employing changepoint detection techniques. Changepoint detection methods explained in Section 4.3 enable identifying time-series fragments most related to failures reported by the associated symptoms. The fragment beginning is denoted by the closest changepoint preceding the symptom emission, while the fragment end is denoted by the size of the selected window. The window is a shorter one out of windows constrained by changepoints and symptom emission times for a pair of analyzed time-series, i.e.,

$$w = min\left\{\tau^i_{Emission} - \tau^i_{Failure}, \tau^j_{Emission} - \tau^j_{Failure}\right\}, \tag{4.30}$$

where $\tau^i_{Failure}$ and $\tau^j_{Failure}$ are changepoints closest to symptom emissions, and $\tau^i_{Emission}$ and $\tau^j_{Emission}$ are symptom emission times. Window selection is based on the observation that a shorter window lowers the likelihood of including irrelevant time-series fragments in the analysis.

Given the window length w and a changepoint $\tau^i_{Failure}$, a changepoint window for time-series M^i, denoted as $m^i_{\tau_{Failure},w}$, is the sample $(m^i_{\tau_{Failure}}, \ldots, m^i_{\tau_{Failure}+w})$. Then, the formula for calculating the coefficient for a pair of time-series samples $m^i_{\tau_{Failure},w}$ and $m^j_{\tau_{Failure},w}$ is defined as

$$r^{ij}_{\tau_{Failure},w} = \frac{\sum_{\tau=1}^{w}(m^i_\tau - \overline{m^i_{\tau_{Failure},w}})(m^j_\tau - \overline{m^j_{\tau_{Failure},w}})}{\sigma^i_{\tau_{Failure},w}\sigma^j_{\tau_{Failure},w}}. \tag{4.31}$$

Figure 4.26 presents the same failure experiment as Figure 4.25 except that time-series fragments are designated using the changepoint detection method. The beginning of each fragment is denoted by the changepoint closest to the symptom emission time, accurately reflecting the failure start, while the fragment end is denoted by the size of the selected window determined by the symptom emission time $\tau^j_{Emission}$.

The Pearson correlation coefficient for fragments designated by changepoints in Figure 4.26 is 0.99. The high coefficient value correctly reflects the strong linear relationship between increases in both time-series.

Figure 4.27 presents scatter plots showing the dependency between time-series M^i and M^j from experiment depicted in Figure 4.26. The horizontal axis holds values of time-series M^i, while the vertical axis holds the corresponding values of time-series

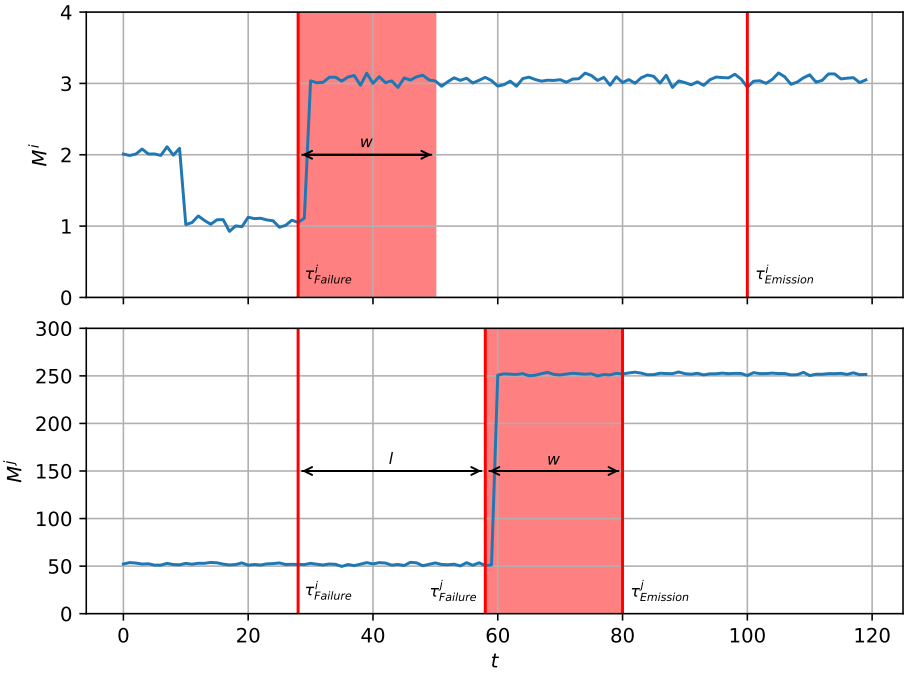


Figure 4.26: Time-series fragments for Pearson correlation designated by changepoint detection

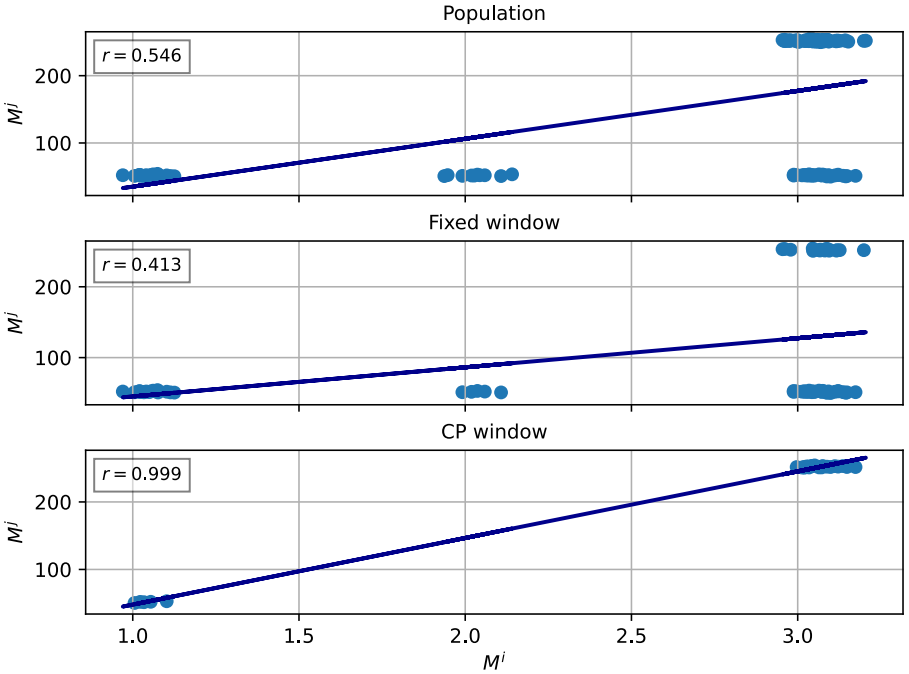


Figure 4.27: Linear dependency between time-series variables

M^j according to the shift imposed by used time-series fragments. The regression line illustrates the linear fit for both time-series variables. Further, each subplot shows point distribution and linear fit for a different method of designating time-series fragments for analysis. The first plot shows the result obtained by examining the entire population of time-series variables, the second by selecting fragments using a

fixed window, and the last by selecting fragments based on the changepoint detection method.

The best linear and symmetrical fit was obtained using the changepoint detection method. Thus, the method was employed in the solution implementation. For the other two methods, the fit is heavily disturbed by irrelevant time-series fragments. In both cases, the inaccuracy results from ignoring the shift between time-series.

4.7 Symptom Topological Distance Analysis

Discussed symptom correlation methods based on event co-occurrence, time lag analysis, and inspection of time-series trends realize part of the symptom correlation framework focusing on the semantical analysis perspective. Further, this section presents the realization of the structural perspective centering around symptom dependence in the context of system structure.

According to the solution concept described in Section 3.3, the RCA model constructs a dependency graph reflecting inter-component system dependencies and maps ingested symptoms into nodes representing components affected by the failure. As a result, individual symptoms can be localized in the hierarchy of system objects. That, in turn, enables analyzing structural symptom dependencies based on examining properties of paths connecting symptom sources in the graph.

Given that component dependencies in the dependency graph provide hints regarding causal component dependence, if symptom sources are located in direct or close proximity, it can be assumed that there is a significant likelihood of symptoms being correlated. Conversely, if symptom sources are located in distinct regions or are distant from each other, it can be assumed that the correlation is mild or unlikely.

There are many realizations of dependency quantification based on a distance measure. In particular, graph structure decided to represent the system structure arguments using graph metrics to enhance the measurement by recognizing the significance of subsequent graph elements on the path connecting components affected by a failure. For instance, graph centrality [71] or PageRank [72] algorithms assign node indicators describing the degree of node influence or importance in a broad graphical context. They could be tested as mechanisms identifying components that create main spots in fault propagation.

The adopted distance measure is a simple and intuitive heuristic constituting a replacement for an in-depth analysis approach that, considering the extensiveness of this study, goes beyond the scope of this work. The heuristic produces a correlation

coefficient Cor_{Dist} whose value decreases geometrically with the distance between analyzed symptom sources, as described in Equation 4.32.

$$Cor_{Dist}(p) = \begin{cases} 1 & \text{if } p \in \{0, 1\} \\ 2^{-(len(p)-1)} & \text{if } p > 1 \end{cases} \tag{4.32}$$

The coefficient is defined as a power of 2 with an exponent equal to the length of the shortest path p between a given pair of nodes in the dependency graph. Moreover, the coefficient gets a value of 1 for two exceptional cases when symptoms are emitted on the same system component or components localized in direct proximity. The shortest path between nodes in the graph can be calculated using Dijkstra's algorithm [73].

4.8 Aggregated Symptom Correlation

After computing correlation coefficients for adopted symptom correlation methods, in the final correlation step, the framework consolidates coefficient values into an aggregated correlation coefficient that provides a unified representation of approximated causal symptom dependency.

Due to the extensive research scope, a simple aggregation method based on a weighted average was employed. The method enables adjusting the contribution of individual symptom correlation methods, thus opening the way to testing different variants of causal symptom dependency approximation through experimentation. Elaborating on an intelligent correlation consolidation method constitutes the subject of future research work.

The aggregated correlation coefficient is calculated by multiplying obtained coefficient values by configured analyses weights and summing the components, i.e.,

$$Cor_{Agg}(a, b) = \sum_{k=1}^{n} w_k \cdot Cor_k(a, b)\,, \tag{4.33}$$

where $Cor_1, ..., Cor_n$ are correlation coefficient values obtained from individual correlation methods, whereas $w_1, ..., w_n \in [0; 1]$ are their weights such that

$$\sum_{k=1}^{n} w_k = 1\,. \tag{4.34}$$

Weights enable strengthening the impact of selected analyses or excluding those that return invalid or inaccurate results. It is assumed that the system administrator configures weights tuned for the managed system operation.

For the considered symptom correlation analyses, the Equation 4.33 can be instantiated to

$$\begin{aligned} Cor_{Agg}(a,b) = w_{Dist} \cdot Cor_{Dist}(a,b) + w_{CO} \cdot Cor_{CO}(a,b) \\ + w_{TL} \cdot Cor_{TL}(a,b) + w_{TS} \cdot Cor_{TS}(a,b) \end{aligned} . \tag{4.35}$$

4.9 Summary

The main objective of the symptom correlation framework is approximating causal dependencies between observed symptoms. Identifying these dependencies is essential for inference algorithm precision. Specifically, causal dependencies allow the construction of the symptom causality graph, designating fault trajectories, and determining failure root causes. Contrary to existing research, instead of utilizing a single correlation method, this work proposes an approach that leverages the synergy of several symptom correlation analyses, each employing a correlation technique inspecting a different aspect of system operation.

An important aspect discussed on the subject of symptom correlation is ensuring temporal symptom consistency, i.e., validating the correctness of failure time information transported in reported symptoms. In this matter, the dissertation proposes changepoint detection techniques to enforce symptom timestamp correction prior to performing correlation analyses. The correction derives from the observation that SLOs and alert rules adopted in enterprise systems often affect failure time information. Particularly, timestamp correction is crucial to improve the accuracy of time-sensitive symptom correlation methods.

The research on symptom correlation concludes with four symptom correlation analyses:

- *Symptom co-occurrence analysis* examines semantic dependence of observed symptoms through discovering temporal dependencies between sequences of past symptom occurrences.

- *Symptom time lag analysis* answers the question of whether the time lag between a given symptom pair converges to the mean of the estimated time lag distribution.
- *Symptom time-series analysis* measures the linear correlation between trends of time-series associated with processed symptoms.
- *Symptom topological distance analysis* localizes symptoms in system structure and inspects structural dependency between system components affected by the failure.

In addition, the framework defines a method for consolidating symptom correlation results into an aggregated correlation coefficient, providing a unified representation of approximated symptom dependency.

Chapter 5

Realization of RCA Model

Following the realization of the symptom correlation framework, this chapter elaborates on the realization of the RCA model. The model is the subsequent element of the defined RCA solution concept. It represents gathered system knowledge and constitutes input for root cause identification in the inference algorithm. The chapter submits a set of RCA model structures that provide a holistic view of system structure and behavior. In order to meet concept expectations, several aspects of model realization need to be considered. First, the model must reflect system structure, i.e., the hierarchy of system components and inter-component dependencies across essential cloud layers constrained by the taxonomy of a given system class. Further, the model must maintain information about active symptoms occurring in the system and their propagation throughout system components. Finally, the RCA model must incorporate the knowledge of symptom semantics, i.e., estimated temporal symptom dependencies discovered from past symptom occurrences. In addition, due to the scale and high evolution rate of cloud-native applications, the RCA model must be constructed automatically and show rapid adaptation to structural changes occurring in the system.

5.1 Realization Overview

According to the adopted RCA concept, the solution implements RCA system abstraction characterized in Section 2.2.2, in which the RCA system is decomposed into two indispensable parts, namely the RCA model and inference algorithm. RCA model is a structural and behavioral reflection of the analyzed system, whereas the inference algorithm utilizes knowledge enclosed in the model instance to explain the causes of observed failures using symptom correlation methods employed in the adopted symptom correlation framework

RCA model is a set of structures and interfaces ensuring access to essential data required in the inference process, i.e., information on the structure and behavior of the supervised system. To some extent, the model encapsulates knowledge and experience used in the manual process of failure diagnosis conducted by a system administrator. As such, the model instance constitutes an input for automated diagnosis by providing the context needed to localize symptoms, recognize dependencies between system objects, and support the diagnosis with knowledge of symptom semantics.

Importantly, due to the high evolution rate of cloud systems, the RCA model instance must be constantly updated according to changes emerging in system structure and behavior [17]. System changes must be instantly recorded in the model to enable online diagnosis of active faults and keep up with processes imposed by orchestration and continuous integration tools. Additionally, since many faults may occur at distinct time points, preserving historical model data and providing an interface enabling past failure introspection is also necessary. In order to address the above requirements, this dissertation proposes complete automation of RCA model construction and maintenance by integrating system knowledge sources exposed in the form of management APIs and event streams.

The adopted RCA model consists of four elements that complement each other to produce a holistic view of the analyzed system. The elements are depicted in Figure 5.1 and are shortly summarized in the following paragraphs.

System object taxonomy maintains the meta-information needed to recognize object types from raw system object data and link extracted object instances into a hierarchical structure (system object dependency graph).

System object dependency graph maintains the current and past information about system object instances, i.e., components and symptoms.

Fault symptom ingestion is a process of enhancing the system object dependency graph with failure symptoms to reflect fault propagation throughout the system.

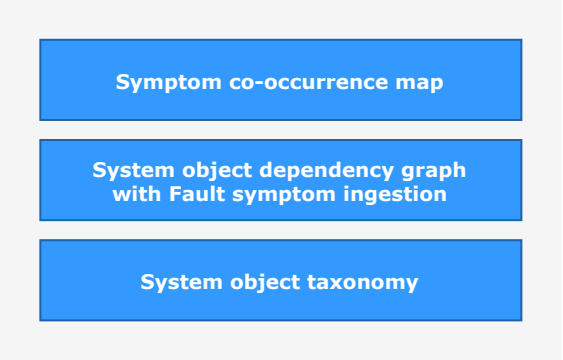


Figure 5.1: RCA model elements

The dependency graph enriched with active symptoms provides data required in topological distance analysis introduced in Section 4.7.

Symptom co-occurrence map represents knowledge of semantic symptom dependencies. The knowledge is obtained automatically by analyzing symptom co-occurrence based on historical symptom data. The co-occurrence map is primarily used in symptom co-occurrence and time lag analyses introduced in Section 4.4 and Section 4.5, respectively.

Formal definitions, construction algorithms, and usage examples of listed RCA model elements are detailed in the following sections. First, Section 5.2 introduces the system object taxonomy. Next, Section 5.3 details the system object dependency graph. Further, Section 5.4 discusses the process of fault symptom ingestion. Finally, Section 5.5 defines the symptom co-occurrence map.

5.2 System Object Taxonomy

System object taxonomy constitutes the main foundation for constructing the subsequent elements of the RCA model. Designed for the particular system class, it maintains the knowledge required to recognize and extract instances of system objects, i.e., components and symptoms, from raw system data and connect them into a hierarchical structure (system object dependency graph). Components represent application elements such as workload units, services, or storage volumes, whereas symptoms represent the external manifestation of system faults detected and reported by integrated observability tools.

The taxonomy, depicted in Figure 5.2, is a directed graph $G_T = (T, D_T, A, C)$, where T is a finite set of nodes t_i representing object types, and D_T is a set of edges (t_i, t_j) representing possible dependencies between them. Additionally, the taxonomy provides function $A(t_i)$, which returns a set of mandatory attributes for a given object

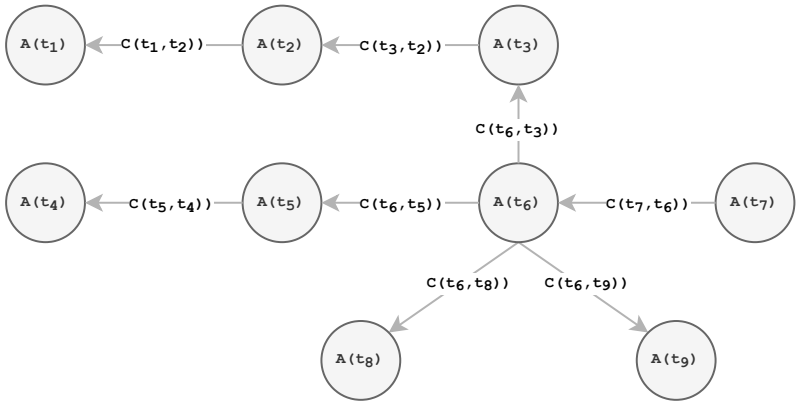


Figure 5.2: System object taxonomy

type. The attributes are required to determine essential information that must be extracted for object instances from raw system object data. Further, the taxonomy provides function $C(t_i, t_j)$. For each connection between a pair of object types, the function returns a set of attribute-based logical expressions allowing to test whether object instances of these types are related to each other and can be connected in the hierarchy. Formally speaking, given two sets of attributes $A(t_i) = \{a_1, ..., a_m\}$ and $A(t_j) = \{a_1, ..., a_n\}$ for a pair of neighbor object types (t_i, t_j) in the G_T, a generalized set of binding conditions for object instances is presented in Equation 5.1, where (a_k, a_l) is a pair of object attributes and $\cdot$ is a logical operator for comparing them, e.g., the equality operator.

$$C(t_i, t_j) = \{(a_k \cdot a_l) : a_k \in A(t_i) \wedge a_l \in A(t_j)))\} \tag{5.1}$$

Construction of System Object Taxonomy

Due to systems being built upon different data models and logical abstractions, no versatile system object taxonomy exists that describes a wide range of system classes. The object taxonomy is closely related to a given system class and must be designed based strictly on analyzing its objects. Each system requires a dedicated taxonomy structure.

At the time of writing, there exists no practice or convention for sharing system object taxonomy in a unified standard, although standards for describing taxonomies of complex systems are well-developed [74]. Most system vendors are limited to defining object types and inter-object dependencies as part of the system documentation. In addition, some systems vendors extend systems with a management API allowing obtaining object data with identifiers of related objects. Based on object identifiers, one can infer valid associations in the taxonomy structure.

Practically, logical conditions for connecting pairs of object instances in many systems are simple and based on identifier attributes imposed by the relational scheme of the underlying platform. For instance, a virtual machine object typically comprises a physical server identifier as it is on one side of the has-many relationship originating from the server entity. The server identifier provided by the virtual machine object type, together with the identifier of the server object type itself, can be used to formulate the connection rule between these two object types.

The above elements can be utilized to build system taxonomy manually. Typically, system object taxonomy can be created once and reused across system instances of the same class. Taxonomy is the only structure in the proposed RCA model that is not automatically acquired.

5.3 System Object Dependency Graph

The system object dependency graph maintains the current and past topology of system object instances extracted from raw system data based on attributes defined in the system object taxonomy. The primary purpose of the dependency graph is to localize symptoms in the system structure. Moreover, it allows identifying system regions affected by the failure and tracing vectors of failure propagation.

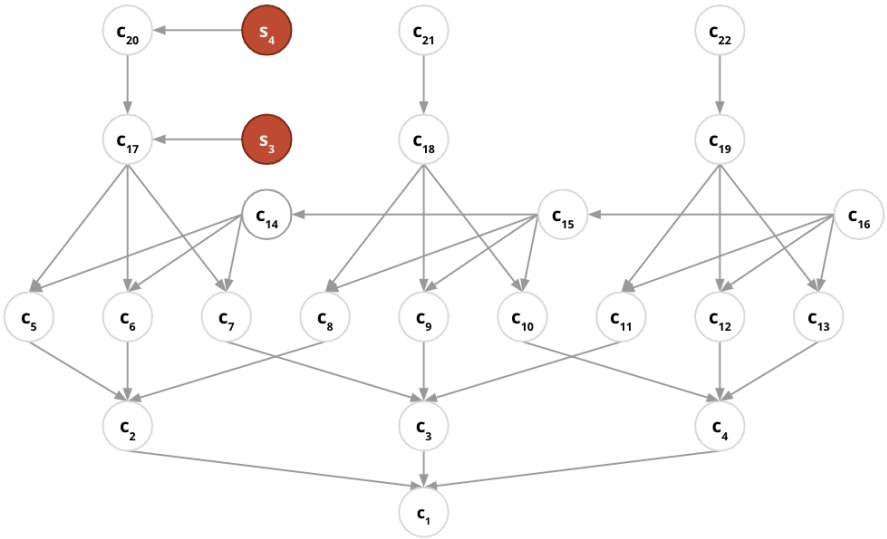


Figure 5.3: System object dependency graph

The dependency graph, depicted in Figure 5.3, is a directed graph $G_D = (O, D_O)$ where O is a finite set of system object instances and D_O is a set of edges representing dependencies between them. The set O is further subdivided into sets C and S (i.e., $O = C \cup S$) to distinguish instances of system components and symptoms accordingly.

Connections in the dependency graph are derived from attribute-based conditions enforced by the system object taxonomy. The directed edge $(c_i, c_j) \in D_O$, where $c_i, c_j \in C$, indicates hint regarding causal dependency between components, i.e., c_i causally depends on c_j and as such c_i may be affected by failures emerging at c_j. For instance, application workload may depend on a virtual machine, which in turn depends on a physical server. Further, the directed edge $(s_i, c_j) \in D_O$ where $s_i \in S$ and $c_j \in C$ is a particular case of causal relationship denoting that component c_j is the source of symptom s_i. In other words, the symptom s_i depends on the component c_j due to the fault affecting c_j being the direct cause of triggering the symptom.

Additionally, the dependency graph provides functions $\tau_{created}$, $\tau_{updated}$, and $\tau_{deleted}$ returning the creation, update, and deletion times, respectively, for an object or dependency in the graph. They constitute the main mechanism for preserving historical data about system components and symptoms. Graph elements are softly removed from the structure, i.e., a deletion time is set while the element information is preserved. Then, a snapshot of the dependency graph is constructed by filtering elements with the deletion time unset.

Given time t, the set of active system objects can be defined as

$$O^t = \{o \in O \colon \tau_{created}(o) <= t \wedge (\tau_{deleted}(o) = \emptyset \vee \tau_{deleted}(o) > t)\,, \tag{5.2}$$

whereas the set of active system object dependencies is denoted as

$$\begin{aligned} D_O^t = \{(o_i, o_j) \in D_O \colon o_i, o_j \in O^t \wedge \tau_{created}(o_i, o_j) <= t \\ \wedge (\tau_{deleted}(o_i, o_j) = \emptyset \vee \tau_{deleted}(o_i, o_j) > t)\,. \end{aligned} \tag{5.3}$$

Subsequently, the snapshot of the system object dependency graph at time t can be defined as

$$G_D^t = (O^t, D_O^t)\,. \tag{5.4}$$

Notably, sizes of component and symptom sets differ between the dependency graph G_D and the dependency graph snapshot G_D^t. The former retains a historical record of all objects and object dependencies that have ever existed in the system. Hence, it constitutes the foundation for introspecting past system states and acts as a repository of symptoms for statistical analysis. The snapshot graph, in turn, is a subgraph of G_D at a given point in time. Primarily, it constitutes an input for the

inference algorithm as it comprises system structure and symptoms active at the diagnosis time.

Construction of System Object Dependency Graph

Depending on the system subjected to analysis, different mechanisms can be provided for retrieving information about system objects and changes related to their configuration and life-cycle. Most systems provide mechanisms for retrieving a complete set of objects and a minimum collection of attributes enabling the identification of inter-object dependencies based on the system object taxonomy. In such a case, the dependency graph construction algorithm can periodically synchronize objects by comparing the current state of the dependency graph with the current state of objects retrieved from the system: new objects are added to the graph, existing objects are updated, and stale objects are removed. This approach will be referred to as *full* or *pull* update.

More advanced systems also manifest additions, updates, and removals of objects in the form of events that enable accurate tracking of object life-cycle and reconstructing object states in the past. The construction algorithm can process events from an event stream and apply changes to the dependency graph incrementally, i.e., apply only changes related to system objects transported in the event. This approach will be referred to as *incremental* or *push* update.

While the event processing approach is more efficient than the full system view synchronization regarding the number of processed objects, the event approach alone is insufficient. First, when the RCA model is initialized, the complete state of the dependency graph must be constructed prior to consuming system events. Moreover, system events could be occasionally delayed, corrupted, or lost, causing object and dependency malformation over time. Therefore, in the dissertation, a hybrid approach is adopted, depicted in Figure 5.4, in which the primary graph update strategy is based on the event-driven approach, but it is periodically corrected by full graph synchronization in longer time intervals, e.g., once in 30 minutes.

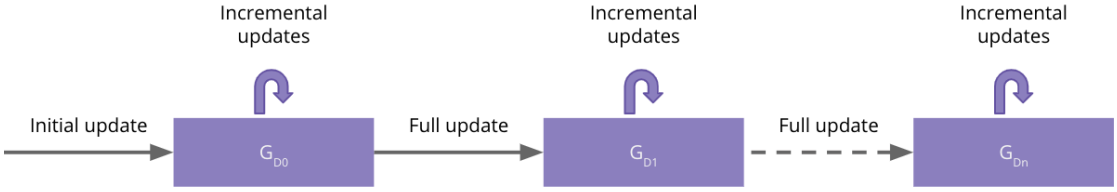


Figure 5.4: Hybrid update strategy for system object dependency graph

Algorithm 1 describes the procedure for constructing the dependency graph according to the pull approach. It takes the system object dependency graph G_D and system object taxonomy G_T on its input. It also takes a stream of raw system component objects RCS to process. First, the algorithm initializes the new version of the dependency graph G'_D with the current state of the graph G_D. Then, for each raw object $ro_i \in RCS$ of type t_i, it extracts an instance of a system component object o_i based on a set of attributes $A(t_i)$ defined in the taxonomy. The extracted object is added to the temporary set of extracted object instances O'. After all object instances are extracted from raw data, the algorithm calculates subsets of objects to delete (O_{delete}), create (O_{create}), and update (O_{update}) in the graph by comparing set of extracted object instances O' to the set of object instances present in the dependency graph O. For instance, a set of objects marked for deletion O_{delete} is designated by subtracting extracted objects from the set of objects present in the current version of the dependency graph, i.e., $O \setminus O'$. For each object marked for deletion, $o_i \in O_{delete}$, all associated links are deleted prior to deleting the corresponding node in G'_D. Both links and nodes are soft-deleted, i.e., the $\tau_{deleted}$ time is set for both elements while element information is preserved in the dependency graph. Further, for each object marked for creation or update, i.e., $o_i \in (O_{create} \cup O_{update})$, the corresponding node is added or updated in the G'_D graph. Then, dependent object types DT are identified from the system object taxonomy. For each dependent type t_j, the corresponding object instances o_j are fetched and checked with the processed object o_i against the connection constraint $C(t_i, t_j)$ obtained from the taxonomy. If the condition for a given pair of object instances is satisfied, the corresponding edge is added to the dependency graph. The algorithm produces an updated dependency graph G'_D on its output.

Algorithm 1: Full update of system object dependency graph

Input: System object dependency graph $G_D = (O, D_O)$,
system object taxonomy $G_T = (T, D_T, A, C)$,
stream of raw system component objects RCS

Output: Updated system object dependency graph $G_D^{'}$

$G_D^{'} \leftarrow G_D$
$O' \leftarrow \emptyset$
for each raw system component object $ro_i \in RCS$ **do**
  $t_i \leftarrow$ type of ro_i in G_T
  $o_i \leftarrow$ object of type t_i with attributes $A(t_i)$ extracted from ro_i
  add node o_i to O'
end for
$O_{delete} \leftarrow O \setminus O'$
$O_{create} \leftarrow O' \setminus O$
$O_{update} \leftarrow O \setminus O_{delete}$
for each $oi \in O_{delete}$ **do**
  soft-delete all o_i links in $G_D^{'}$
  soft-delete o_i node from $G_D^{'}$
end for
for each $o_i \in (O_{create} \cup O_{update})$ **do**
  add or update node o_i in $G_D^{'}$
  $t_i \leftarrow$ type of o_i
  $DT \leftarrow$ adjacents of t_i in G_T
  for $t_j \in DT$ **do**
    for each object o_j of type t_j **do**
      if $C(t_i, t_j)$ satisfied for (o_i, o_j) **then**
        add or update edge (o_i, o_j) in $G_D^{'}$
      end if
    end for
  end for
end for

Algorithm 2 describes the procedure for constructing the graph according to the push approach. Similarly to Algorithm 1, it takes the system object dependency graph G_D and system object taxonomy G_T on its input. It additionally takes a stream of raw system component events RCS_E to process. First, the algorithm initializes the new version of the dependency graph $G_D^{'}$ with the current state of the G_D graph. Then, for each system event $(ro_i, t_E) \in RCS_E$, where ro_i is a raw object of type t_i, it extracts an instance of system component object o_i based on attributes $A(t_i)$ defined in the system object taxonomy. If the object is marked for deletion, i.e., $t_E = delete$, the corresponding links and node o_i are soft-deleted from $G_D^{'}$. If the object is marked for creation or update, i.e., $t_E \in \{create, update\}$, the algorithm adds or updates the node in $G_D^{'}$. Then, it identifies object instances o_j of dependent object types $t_j \in DT$, evaluates object pairs against the connection constraint $C(t_i, t_j)$, and links pairs which satisfy the condition. The algorithm produces an updated dependency graph $G_D^{'}$ on its output.

Algorithm 2: Incremental update of system object dependency graph

Input: System object dependency graph $G_D = (O, D_O)$,
system object taxonomy $G_T = (T, D_T, A, C)$,
stream of raw system component events RCS_E
Output: Updated system object dependency graph $G_D^{'}$

```
G'_D ← G_D
for each system component event (ro_i, t_E) ∈ RCS_E do
  t_i ← type of ro_i in G_T
  o_i ← object of type t_i with attributes A(t_i) extracted from ro_i
  if t_E ∈ {create, update} then
    add or update node o_i in G'_D
    DT ← adjacents of t_i in G_T
    for t_j ∈ DT do
      for each object o_j of type t_j do
        if C(t_i, t_j) satisfied for (o_i, o_j) then
          add or update edge (o_i, o_j) in G'_D
        end if
      end for
    end for
  else if t_E = delete then
    soft-delete all o_i links in G'_D
    soft-delete o_i node from G'_D
  end if
end for
```

5.4 Fault Symptom Ingestion

Algorithms outlined in the previous chapter focus on updating the system object dependency graph with application components such as workload units or services and connecting them with cause-effect orientation. Ultimately, they provide insight into the system topology and describe how system components depend on each other. Fault symptom ingestion, on the other hand, focuses on enriching the dependency graph with the behavioral aspect, i.e., adding information about symptoms occurring in the system. The process is particularly important considering that symptoms occurring at fault time constitute the main input for the inference algorithm. Recall that the primary objective of the inference algorithm is explaining detected faults based on reported symptoms.

Large IT systems generate a significant number of symptoms over time. Symptoms can be temporary or permanent. Temporary symptoms usually indicate periodic system degradation caused by, e.g., a spike in user traffic. They often do not require administrator intervention as they are addressed automatically by auto-scaling and auto-healing mechanisms employed by modern orchestration platforms. In turn, permanent symptoms result from severe faults that cannot be resolved automatically and require remediation by the administrator, e.g., a memory leak. Eventually, permanent failures also get resolved. In general, symptoms are transient events occurring indeterminately for variable periods. They may be frequently activated and deactivated by observability tools. Similar symptom dynamics must be reflected in the RCA model to capture the complete view of situations emerging in the system.

Fault symptom ingestion is a process of enhancing the system object dependency graph with failure symptoms. The process aims to reflect failure propagation throughout the system and highlight affected system regions. Symptoms ingested from observability tools are orderly mapped onto corresponding components in the graph, *symptom sources*, by matching component and symptom attributes according to the system object taxonomy. Like system components, symptoms contain attributes allowing identifying dependent objects, e.g., component name or identifier. A set of attribute-based logical conditions for linking a symptom to the corresponding source component will be referred to as *source mapping*.

Furthermore, fault symptom ingestion is responsible for correcting failure onset for each ingested symptom using changepoint detection techniques introduced in Section 4.3. That is imposed by statistical methods employed in the solution to mine semantic symptom knowledge based on past symptom occurrences. Strict time accuracy is required to infer valid symptom dependencies.

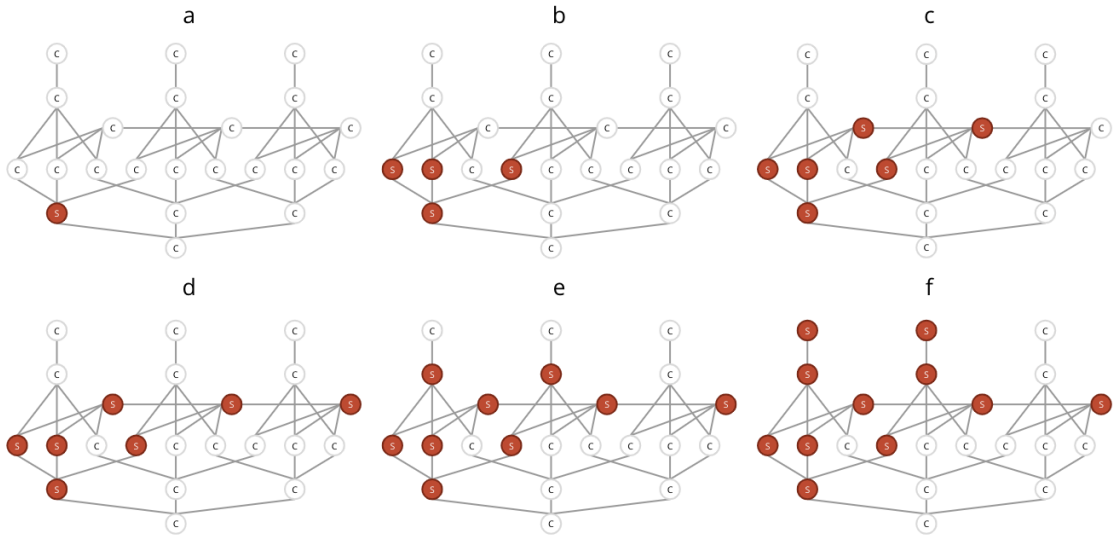


Figure 5.5: Fault symptom ingestion

Figure 5.5 illustrates the process of symptom ingestion using an example with a simplified system object dependency graph. White nodes represent system components, whereas red nodes represent system components combined with reported symptoms. A fault originating at a system component in the low layer (a) propagates vertically and horizontally through the upper layers (b-f). For instance, the root component could be a physical server that was abruptly shut down (a). The server shutdown results in the dysfunction of virtual machines that were placed on the server (b). That, in turn, results in the degradation of two services that aggregated dysfunctional virtual machines (c). The degradation of services, in turn, propagates to the third service (d), which is the client of the first two services. Then, platform agents detect the failure of virtual machines (e) and propagate the failure to higher orchestration modules (f).

Process of Fault Symptom Ingestion

Algorithm 3 describes the process of fault symptom ingestion. It takes system object dependency graph G_D and system object taxonomy G_T on its input. It also takes a stream of raw symptom events RSS to process. Importantly, the dependency graph must be populated with system components prior to symptom ingestion according to the construction process defined in Algorithm 1 and Algorithm 2.

Additionally, for failure onset correction, the algorithm takes the length of the time-series look-behind window w, function for retrieving symptom activation time $\tau_{Activation}$, and function for retrieving symptom tolerance period $\Delta_{Tolerance}$. These parameters are closely related to symptom timestamp correction introduced in Section 4.3.4.

Algorithm 3: Process of fault symptom ingestion

Input: System object dependency graph $G_D = (O = C \cup S, D_O)$,
system object taxonomy $G_T = (T, D_T, A, C)$,
stream of raw symptom events RSS,
function for retrieving symptom activation time $\tau_{Activation}$,
function for retrieving symptom tolerance period $\Delta_{Tolerance}$,
time-series look-behind window length w

Output: System object dependency graph G'_D enhanced with symptom objects,
function for retrieving symptom failure onset time $\tau_{Failure}$

$G'_D \leftarrow G_D$
for each symptom event $rs_i \in RSS$ of type t_i **do**
  $s_i \leftarrow$ symptom object of type t_i with attributes $A(t_i)$ extracted from rs_i
  if $\tau_{Failure}(s_i) = \emptyset$ **then**
    $y \leftarrow$ fragment of s_i symptom time-series designated by $\tau_{Activation}(s_i)$ and w
    if $y = \emptyset$ **then**
      $\tau_{Failure}(s_i) = \tau_{Activation}(s_i)$
    else
      T $\leftarrow CROPS(y)$ {find changepoints using CROPS and PELT methods}
      $\tau_{Reference} \leftarrow \tau_{Activation}(s_i)$
      if $\Delta_{Tolerance}(s_i) \neq \emptyset$ **then**
        $\tau_{Reference} \leftarrow \tau_{Reference} - \Delta_{Tolerance}(s_i)$
      end if
      $\tau_{Failure}(s_i) \leftarrow \arg\min_{\tau \in \mathrm{T}} |\tau_{Reference}(s_i) - \tau|$
    end if
  end if
  if s_i is active **then**
    add or update node s_i in G'_D
    $t_j \leftarrow$ adjacent of t_i in G_T
    $c_i \leftarrow$ source component of type t_j such that $C(t_i, t_j)$ is satisfied for (s_i, c_i)
    if $c_i \neq \emptyset$ **then**
      add or update edge (s_i, c_i) in G'_D
    end if
  else if s_i is inactive **then**
    soft-delete all s_i links in G'_D
    soft-delete s_i node from G'_D
  end if
end for

First, the algorithm initializes the new version of the dependency graph G'_D with the current state of the graph G_D. Then, for each raw object $rs_i \in RSS$ of type t_i transported in an event, it extracts a symptom object s_i based on attributes $A(t_i)$ specified in the system object taxonomy. The extracted symptom object contains normalized information about its status. If the symptom is inactive, the corresponding node s_i and its links are soft-deleted from G'_D, i.e., $\tau_{deleted}$ time is set. If the symptom is active, the algorithm adds or updates the node in G'_D. Then, it finds source component c_i in G'_D for symptom s_i by retrieving component type t_j from the taxonomy and evaluating component objects of type t_j until the matching condition $C(t_i, t_j)$ is satisfied. If the source component is found, the algorithm links symptom s_i to its source c_i. The algorithm produces an updated dependency graph G'_D on its output.

Additionally, the algorithm attempts to correct symptom failure onsets prior to reflecting symptom objects in the dependency graph. If the failure correction was not already applied, the algorithm fetches a time-series fragment within a look-behind window w before the symptom activation time $\tau_{Activation}(s_i)$. If time-series data is unavailable, failure onset is set to the symptom activation time. Otherwise, the CROPS method identifies significant changepoints providing optimal time-series segmentation. Failure onset correction is estimated as a changepoint closest to the reference time point $\tau_{Reference}$. Depending on the availability of tolerance period hint $\Delta_{Tolerance}$, it may be equivalent to symptom emission time or symptom activation time as defined in Equation 4.5 and Equation 4.6, respectively. The algorithm produces the function for retrieving symptom failure onset time $\tau_{Failure}$ in addition to the updated dependency graph.

5.5 Symptom Co-occurrence Map

System object dependency graph and fault symptom ingestion introduced in Section 5.3 and Section 5.4, maintain knowledge required for recognizing system structure and localizing ingested symptoms. They find a particular application in the system correlation framework employed in the inference algorithm. As explained in Section 4.7, they are used in symptom topological distance analysis, in which symptom dependencies are computed by evaluating the distance between symptom sources.

However, according to the framework concept detailed in Section 3.2, topological distance analysis is insufficient for efficient failure diagnosis in the sense that besides taking into account the structural aspect of symptoms occurrence, it is crucial to distinguish symptoms semantically. That is particularly important in the case of parallel faults affecting the same system region. Thus, following the adopted

RCA concept, the proposed solution employs complementary symptom analyses concentrating solely on symptom semantics, namely symptom co-occurrence analysis and symptom time lag analysis, detailed in Section 4.4 and Section 4.5. Both analyses evaluate symptom semantics in the time space.

Similar to symptom topological distance analysis, co-occurrence and time lag analyses rely on system knowledge. Precisely, they are based on semantic symptom dependencies mined from historical symptom data retrieved from the system object dependency graph. In order to discover the dependencies, both correlation methods translate past symptom data into the event space and utilize the Iterative Closest Events (ICE) method for statistical event correlation, detailed in Section 4.4.3. The method finds an optimal event assignment between a pair of event sequences, calculates the time lag between matched event pairs, and estimates the time lag probability distribution. Then, for symptom co-occurrence analysis, the time lag distribution is quantified into a correlation coefficient based on mutual information (Section 4.4.4), whereas in the case of the time lag analysis, the computed probability distribution is used to evaluate time lags of new symptom pairs (Section 4.5).

The ICE method is computationally intensive due to the need to match events between event sequences. Event matching is an optimization problem that searches the space of possible event assignments and, for each assignment, computes the error function comprising costly nearest neighbor computation. Despite applied heuristics and improvements limiting the search space, the method execution time is extended to the boundary that excludes direct method use in real-time symptom analysis. However, since semantic symptom dependencies are not subjected to sudden changes, whereas mild changes require processing a significant number of symptoms, these dependencies do not need to be recalculated for each ingested symptom. Instead, dependencies can be computed offline at longer time intervals and used as a snapshot for real-time symptom analysis.

The proposed solution introduces the symptom co-occurrence map as a dedicated lookup structure for facilitating symptom co-occurrence and time lag analyses. Organized as a graph structure, it allows accessing semantic dependence between pairs of known symptoms at a fixed time. Moreover, the structure maintains additional information required for the symptom time lag analysis, i.e., parameters describing the time lag probability distribution: calculated time lags, mean time lag, and IQR boundaries.

Formally, the co-occurrence map, illustrated in Figure 5.6, is a directed graph $G_S = (ST, D_{ST}, \sigma_S, \mu_S)$, where ST is a finite set of known symptom types and D_{ST} is a set of edges representing discovered semantic symptom dependencies. Additionally, the

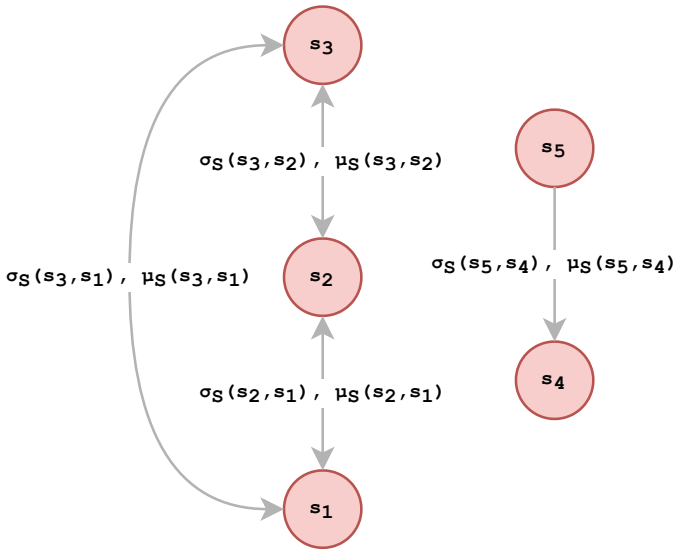


Figure 5.6: Symptom co-occurrence map

co-occurrence map provides functions $\sigma_S(s_i, s_j)$ and $\mu_S(s_i, s_j)$ returning information required for symptom co-occurrence and time lag analyses discussed in Section 4.4 and Section 4.5, respectively. The former function returns the quantified strength of a given semantic symptom dependency, expressed as a standardized coefficient value in range $[0; 1]$. The latter returns parameters of time lag probability distribution, namely distribution mean μ and IQR boundaries $\Theta_{Upper}, \Theta_{Lower}$.

Construction of Symptom Co-occurrence Map

As a statistical method, ICE requires extensive data to return accurate results. The data set must include symptom instances reported in response to system failures that emerged in the past. Depending on the system maturity, historical symptom data may be limited or unavailable. For instance, the system may be newly deployed or has not operated for enough time to produce sufficient symptoms, or the system is resistant to experiencing recurring outages. Moreover, system behavior may be inconsistent in the long term due to the high evolution rate, especially if the system was repeatedly updated or the alert rule configuration was changed many times by the system administrator. Although the ICE method has a tolerance level for handling system changes and abnormalities, the system must remain stable and show deterministic behavior from an overall perspective in order for the method to return valuable results.

Due to the above difficulties, it is recommended that symptom instances for the ICE method are generated in a controlled manner. For this purpose, chaos testing techniques can be used [75]. They are already well established in modern IT projects and yield measurable benefits in terms of the resulting system stability and fault

tolerance [76, 77]. Instead of waiting for a regular failure to occur and putting the user at risk of experiencing failure effects, a proactive approach is taken. The system is repeatedly subjected to arbitrary fault simulations to track its reaction to failures. Consequently, system maintainers can eliminate defects identified during the simulation in advance, create missing alerting rules, and develop necessary procedures for failure remediation.

The chaos testing routine can pair with gathering symptoms required for constructing the symptom co-occurrence map, but for the co-occurrence map to be applicable, the testing must cover a broad spectrum of failure scenarios related to various aspects of system operation. That is dictated by the inference algorithm that must recognize symptom semantics along the fault trajectory path to determine the correct failure root cause and explain fault propagation. Hence, many chaos experiments are necessary to populate the required symptom data. At the same time, these experiments are crucial to achieving system resiliency across components in all system complexity layers.

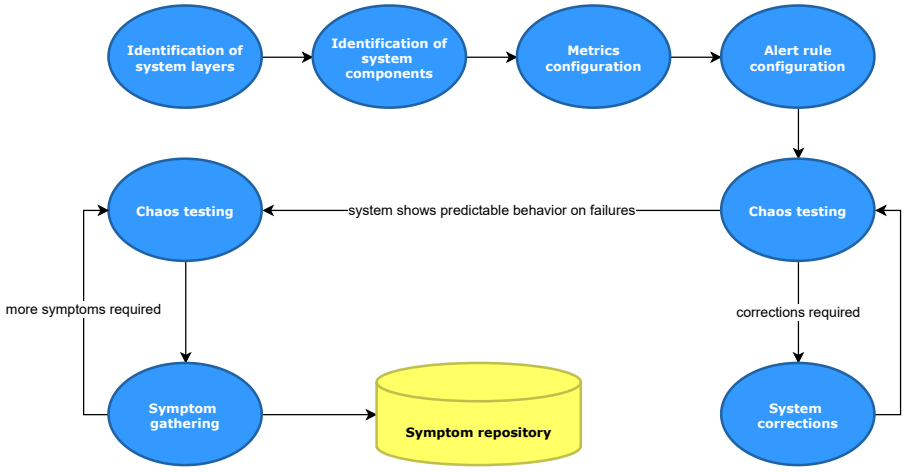


Figure 5.7: Process of gathering symptom sequences prior to constructing the symptom co-occurrence map

Therefore, sequencing symptoms for constructing the symptom co-occurrence map involves several phases of the system life-cycle, including system design, configuration, and testing. The diagram in Figure 5.7 illustrates that process. First, system layers are distinguished, i.e., logical component groups and related technologies, which perform a concrete system function and constitute the indispensable foundation for subsequent layers. Further, critical components at each layer are identified by analyzing system response against known fault vectors. Both process stages can be supported by inspecting the contents of the system object dependency graph. Then, for each component, metrics describing its status and key performance indicators are configured along with corresponding alerting rules warning component abnormalities.

Finally, the system is subjected to fault simulation as part of the chaos testing routine. In the beginning, single fault simulations are performed to ensure that the system gives deterministic responses to failures. Typically, a new system requires many corrections to software and configuration, but its behavior becomes predictable with each subsequent iteration. After the system is behaviorally stable, fault simulations are performed in a loop to produce symptom sequences of adequate length for each failure scenario. Reported symptoms are stored in the dependency graph as a symptom repository for further processing. After collecting a sufficient number of symptoms for each failure scenario, a snapshot of the symptom co-occurrence map can be computed.

Algorithm 4: Construction of symptom co-occurrence map

Input: System object dependency graph $G_D = (O = C \cup S, D_O)$,
minimum mean time lag μ_{min}
Output: Symptom co-occurrence map $G_S = (ST, D_{ST}, \sigma_S, \mu_S)$
for each combination of symptom types (t_A, t_B) in S **do**
assume temporal dependency $A \rightarrow B$
$S_A \leftarrow$ all symptoms of type t_A from S
$S_B \leftarrow$ all symptoms of type t_B from S
$L, t^* \leftarrow ICE(S_A, S_B)$ {find sequence alignment and compute time lags using Iterative Closest Points (ICP) method}
if more than half of $l \in L$ is negative **then**
$L \leftarrow -L$
assume temporal dependency $B \rightarrow A$
end if
$L \leftarrow LOF(L)$ {remove time lag outliers using Local Outlier Factor (LOF) method}
$LC \leftarrow DBSCAN(L)$ {identify time lag clusters using Density-Based Spatial Clustering}
$L \leftarrow$ largest cluster $L^{'} \in LC$
$D \leftarrow KDE(L)$ {estimate time lag probability distribution using Kernel Density Esitimator (KDE)}
$\sigma_S(t_A, t_B) \leftarrow I(D)/H(D)$ {dependency strength computed using Mutual Information correlation coefficient}
$\mu_S(t_A, t_B) \leftarrow (\Theta_{Lower}, \Theta_{Upper}, \mu) \leftarrow IQR(L)$ {compute time lag distribution parameters using interquartile range (IQR) method}
if $\mu < \mu_{min}$ **then**
assume bidirectional temporal dependency $A \leftrightarrow B$
end if
add or update nodes t_A and t_B in G_S
add or update edge between t_A and t_B in G_S based on the assumed temporal dependency direction
end for

Algorithm 4 demonstrates computation of the symptom co-occurrence map based on elements of symptom co-occurrence and time lag analyses detailed in Section 4.4 and Section 4.5. The algorithm takes system object dependency graph G_D containing all past symptom occurrences S ingested during failure simulation. It also takes the constant μ_{min} parameter describing the minimum mean time lag for designating the temporal dependency direction in the graph. For each combination of symptom types (t_A, t_B) among all symptom types present in the sequence of all past symptoms S stored in the dependency graph G_D, subsequences S_A and S_B of considered symptom types are extracted from S, and temporal dependency $A \rightarrow B$ is assumed. Then, using the Iterative Closest Points (ICP) method, optimal alignment of symptom sequences is found in the form of a one-dimensional sequence translation t^*, and time lags L for matched symptom pairs are calculated. If most time lags are negative, the temporal dependency is corrected by inverting signs of all time lags and changing the direction in the assumed temporal dependency. Next, time lag outliers are detected and eliminated using the Local Outlier Factor (LOF) method, and time lag clusters LC are identified among filtered time lags using the Density-based Spatial Clustering (DBSCAN) method. Cluster $L^{'}$ with the most significant number of samples is selected from LC to represent the temporal dependency. Based on the cluster, the time lag probability distribution D is estimated using the Kernel Density Estimator (KDE) method. The distribution is then used to evaluate the dependency strength $\sigma_S(t_A, t_B)$ as the ratio of mutual information I and joint entropy H of symptom co-occurrence. In addition, mandatory parameters $\mu_S(t_A, t_B)$ for the symptom time lag analysis are calculated, including mean time lag μ and time lag boundaries, i.e., Θ_{Lower} and Θ_{Upper}, derived from the interquartile range (IQR) method. If the mean time lag is less than the minimum mean time lag μ_{min}, a bidirectional temporal dependency $A \leftrightarrow B$ is assumed. Finally, nodes representing analyzed symptoms of types t_A and t_B are added to the co-occurrence map G_S together with an edge indicating the dependency direction. The algorithm produces the built symptom co-occurrence map G_S on its output.

5.6 Summary

The proposed solution decomposes the RCA model into four complementary elements listed below to create a holistic and adaptive view of the analyzed system required to implement the conceptualized fault reasoning process.

System object taxonomy maintains meta-information needed to recognize object types from raw system objects and link object instances into a hierarchical structure. It is the only RCA model structure that must be manually acquired.

Further, *system object dependency graph* maintains current and past system structure, i.e., system components and inter-component dependencies constrained by the system object taxonomy.

A significant enhancement related to the dependency graph construction is the process of *fault symptom ingestion*, in which observed symptoms are orderly mapped to graph nodes representing system components affected by a failure. Symptom ingestion enables failure propagation analysis and examining deterministic graph-level symptom dependencies.

Finally, *symptom co-occurrence map* represents the knowledge of symptom semantics. The structure is a graph representing known symptoms and semantic dependencies mined by analyzing symptom co-occurrence based on historical data. Due to computational complexity, the co-occurrence map is generated offline and serves as a lookup for recognizing symptom semantics in the inference algorithm.

Chapter 6

Realization of Inference Algorithm

Given the realizations of the symptom correlation framework and RCA model, this chapter proceeds to consolidate them into a complete fault reasoning process. According to the solution concept, the inference algorithm employs the symptom correlation framework to approximate causal dependencies between observed symptoms based on system knowledge obtained from the RCA model. Consequently, a causality graph representing a unified fault view can be constructed as a basis for discovering fault trajectories. This chapter discusses critical aspects of inference algorithm realization, including integration of symptom correlation methods provided by the symptom correlation framework, consolidation of correlation results into causality graph, estimation of failure causes and effects, processing of potential fault trajectories, and trajectory ordering.

6.1 Realization Overview

Recall formulation of the inference problem presented in the fault diagnosis concept described in Section 3.4. Given a snapshot of the RCA model instance, the objective of the inference algorithm is to evaluate symptoms reported in the analyzed system from distinct operational perspectives, consolidate evaluation results into approximate causal symptom dependencies, and determine the most probable fault trajectory explaining failure manifested by observed symptoms.

In more detail, the inference algorithm takes a snapshot of the RCA model instance at time t as an input. More precisely, it inputs a snapshot of the system object dependency graph G_D^t and the symptom co-occurrence map G_S. These elements provide system information required by adopted symptom correlation analyses. They include current system structure, active symptoms, and mined knowledge of symptom semantics. Additionally, the algorithm takes optional effect symptom s_E as the starting point for the root-cause analysis. The effect symptom s_E can be arbitrarily selected by the system administrator.

In order to comply with the assumed concept of analyzing a holistic view of failure situations, the inference algorithm performs a series of symptom analyses concerning various aspects of system operation. Specifically, it leverages the symptom correlation framework introduced in Section 3.2. The framework defines symptom correlation methods and a strategy for consolidating correlation results, enabling flexible design and extension of the diagnosis process. As such, the algorithm allows adjusting the diagnosis to a given system class and the availability of system knowledge. More importantly, examining emerging failures from distinct operational perspectives ensures the high confidence of approximated causal symptom dependencies.

The proposed symptom correlation framework approximates causal dependencies for single symptom pairs. However, many more symptoms are reported during regular system failures, and thus cause-effect dependency between each symptom pair should be determined. Therefore, a fundamental approach introduced in the realization of the inference algorithm is generalizing framework correlation methods into the matrix space. Consequently, each analysis employed in the framework computes symptom correlation coefficients between all pairs of active symptoms based on the selected diagnosis strategy and produces a square coefficient matrix. Moreover, according to the assumed correlation principles, coefficient values are standardized to range $[0; 1]$ for equal comparison with other correlation methods.

Figure 6.1 summarizes inference algorithm stages. In the first stage, each analysis employed according to the symptom correlation framework takes a snapshot of the

RCA model and computes a matrix of standardized correlation coefficients based on the implemented correlation method. Correlation methods evaluate observed symptoms against structural and semantic system perspectives.

- *Symptom Topological Distance Analysis* concentrates on analyzing topological connections between system components affected by the failure (Section 4.7). The analysis computes a distance coefficient for each symptom pair considering the number of intermediate components and their properties in the system object dependency graph.
- *Symptom Co-occurrence Analysis* examines symptom dependence discovered based on the symptom co-occurrence analysis (Section 4.4). The more consistently sequences of symptoms co-occurred in the past, the higher the correlation coefficient value. Due to computational complexity, the analysis is performed based on pre-computed data maintained in the symptom co-occurrence map.
- *Symptom Time Lag Analysis* reuses information maintained in the symptom co-occurrence map to evaluate if time lags between symptom pairs match estimated time lag probability distributions (Section 4.5). The analysis returns the highest coefficient value for symptom pairs for which the time lag converges to the distribution mean.
- *Symptom Time-series Analysis* investigates the linear correlation between time-series trends associated with processed symptoms (Section 4.6). Here, the correlation coefficient value is equivalent to the employed modified version of the Pearson correlation method.

After performing subsequent analyses, the inference algorithm consolidates their results into an aggregated symptom correlation matrix using the result consolidation strategy (Section 4.8) adjusted to matrix arithmetics. Then, obtained causal dependencies are reflected in the fault view causality graph, which constitutes the primary structure for root cause identification. Nodes in the graph represent active symptoms, while edges reflect approximated causal symptom dependencies. Further, root cause and effect symptoms are designated based on inspecting properties of symptom nodes, and the fault view is searched for candidate fault trajectories connecting them in the graph. Finally, extracted fault trajectories are scored and ranked based on established trajectory scoring and ranking criteria. The most probable fault trajectories are produced at the algorithm output.

In the following sections, individual stages of the inference algorithm are detailed together with algorithms responsible for constructing intermediate analytical structures and implementing the inference process. First, *Symptom Topological Distance*

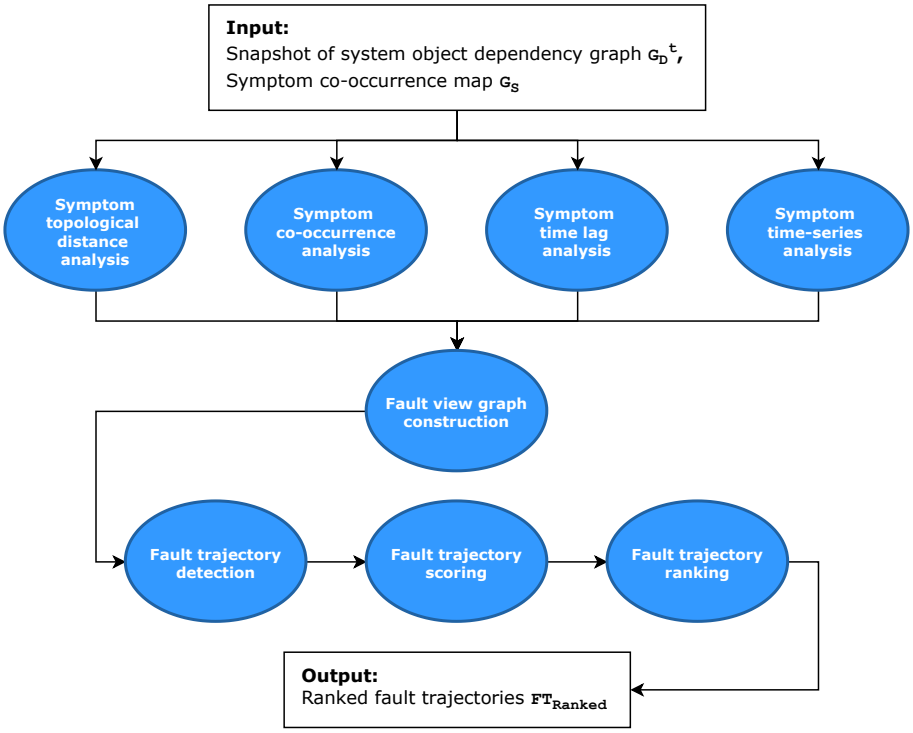


Figure 6.1: Fault reasoning process employed in the inference algorithm with a definition of algorithm input and expected output

Analysis, *Symptom Co-occurrence Analysis*, *Symptom Time Lag Analysis*, and *Symptom Time-series Analysis* are discussed in Section 6.2, Section 6.3, Section 6.4, and Section 6.5, respectively. Next, method for constructing the fault view causality graph is presented in Section 6.6. Further, Section 6.7, Section 6.8, and Section 6.9 present realization of fault trajectory detection, scoring and ranking.

6.2 Symptom Topological Distance Analysis

In the first analysis stage, the inference algorithm utilizes information maintained in the system object dependency graph to assess the distance between system components affected by the failure. The closer symptom sources are localized to each other, the more likely corresponding symptoms are causally related. Likewise, if many intermediate components separate symptom sources, or there is no path connecting them in the graph, one may consider that the symptom dependency is weak or does not exist.

The analysis produces M_{Dist} matrix comprising correlation coefficients based on the topological distance analysis method introduced in Section 4.7. Each matrix cell holds a Cor_{Dist} coefficient value calculated from Equation 4.32 and standardized to range $[0; 1]$.

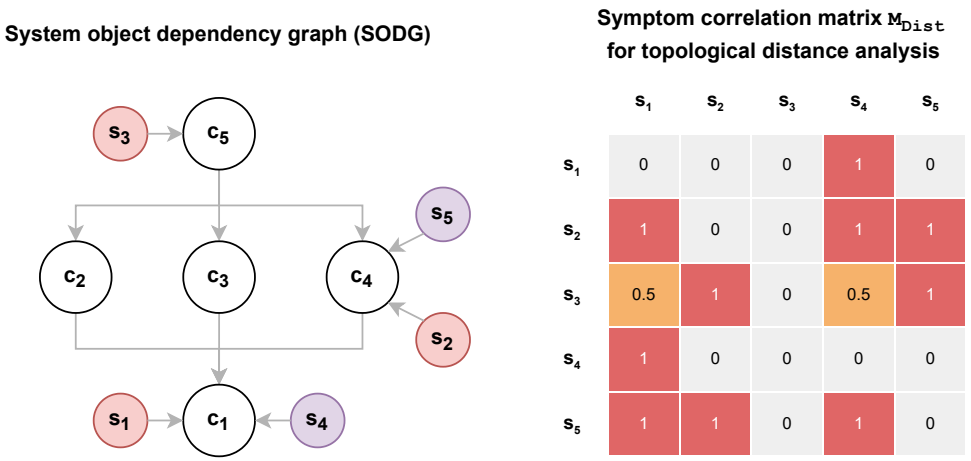


Figure 6.2: Symptom topological distance analysis: relevant fragment of the system object dependency graph with ingested symptoms (on the left side), obtained symptom correlation matrix (on the right side)

Figure 6.2 illustrates an example of symptom topological distance analysis. On the left side, it presents a system fragment with system components $c_{1:5}$ and active symptoms $s_{1:6}$. Symptoms originate from two parallel faults marked in red and purple, respectively. The resulting correlation matrix is shown on the right side. Visibly, the highest coefficient values are held by symptom pairs sharing a source component or pairs with symptom sources located in direct proximity, e.g., symptoms s_1, s_4 share source component c_1, while symptoms s_2, s_1 are located in direct proximity. Hence, both symptom pairs obtained a coefficient value of 1. Conversely, the coefficient value decreases with the increasing distance between source components, e.g., the path connecting components affected by failure manifested by symptoms s_3, s_1 has a length of 2 which translates into a coefficient value of 0.5.

Algorithm 5 describes the construction of the symptom correlation matrix according to topological distance analysis. It takes a snapshot of the system object dependency graph G_D^t on its input. The dependency graph contains a list of active symptoms S^t and a list of system components C^t at time t. For each pair of active symptoms $s_i, s_j \in S^t$, the algorithm extracts source components c_i, c_j by fetching adjacent nodes for both symptoms. Notably, the source component is the only adjacent node for each symptom node. Then, the algorithm finds the shortest path that connects components in the graph. Based on the path length p, it calculates the distance measure using Equation 4.32 and writes it into the correlation matrix. The algorithm returns correlation matrix M_{Dist} on its output.

Algorithm 5: Construction of symptom correlation matrix for symptom topological distance analysis

Input: Snapshot of system object dependency graph
$G_D^t = (O^t = C^t \cup S^t, D_O^t)$
Output: Symptom correlation matrix M_{Dist}
for each permutation of symptoms $s_i, s_j \in S^t$ **do**
$c_i \in C^t \leftarrow$ adjacent of s_i in G_D^t
$c_j \in C^t \leftarrow$ adjacent of s_j in G_D^t
$p \leftarrow$ shortest path between c_i and c_j in G_D^t
if $p = \emptyset$ **then**
$M_{Dist}(i,j) \leftarrow 0$
else if $p = 0$ or $p = 1$ **then**
$M_{Dist}(i,j) \leftarrow 1$
else
$M_{Dist}(i,j) \leftarrow 2^{-(len(p)-1)}$
end if
end for

6.3 Symptom Co-occurrence Analysis

The subsequent symptom correlation analysis examines symptom dependencies in terms of symptom semantics determined based on the event co-occurrence analysis. Symptoms whose sequences co-occurred consistently in the past obtain the highest correlation coefficient value, while symptoms whose sequences did not co-occur consistently get the coefficient value converging to 0.

In order to perform the analysis efficiently, the method uses the symptom co-occurrence map maintained as part of the RCA model. As discussed in Section 5.5, due to complex computations required to discover semantic symptom dependence based on event co-occurrence, a snapshot of dependencies between known symptoms is pre-computed and stored in the co-occurrence map preliminary to the diagnosis. Therefore, symptom co-occurrence analysis is limited to copying pre-computed coefficients from the co-occurrence map into the resulting correlation matrix.

The analysis produces M_{CO} matrix comprising correlation coefficients based on the symptom co-occurrence analysis method introduced in Section 4.4. Each matrix cell holds a Cor_{CO} coefficient value calculated from Equation 4.24 and standardized to range $[0; 1]$. Pre-computed coefficient values are copied from the corresponding edges in the symptom co-occurrence map.

Figure 6.3 illustrates exemplary failure scenario discussed in Section 6.2 with the exact same system component hierarchy and symptom propagation. Relevant information obtained from the symptom co-occurrence map is presented on the right. Nodes

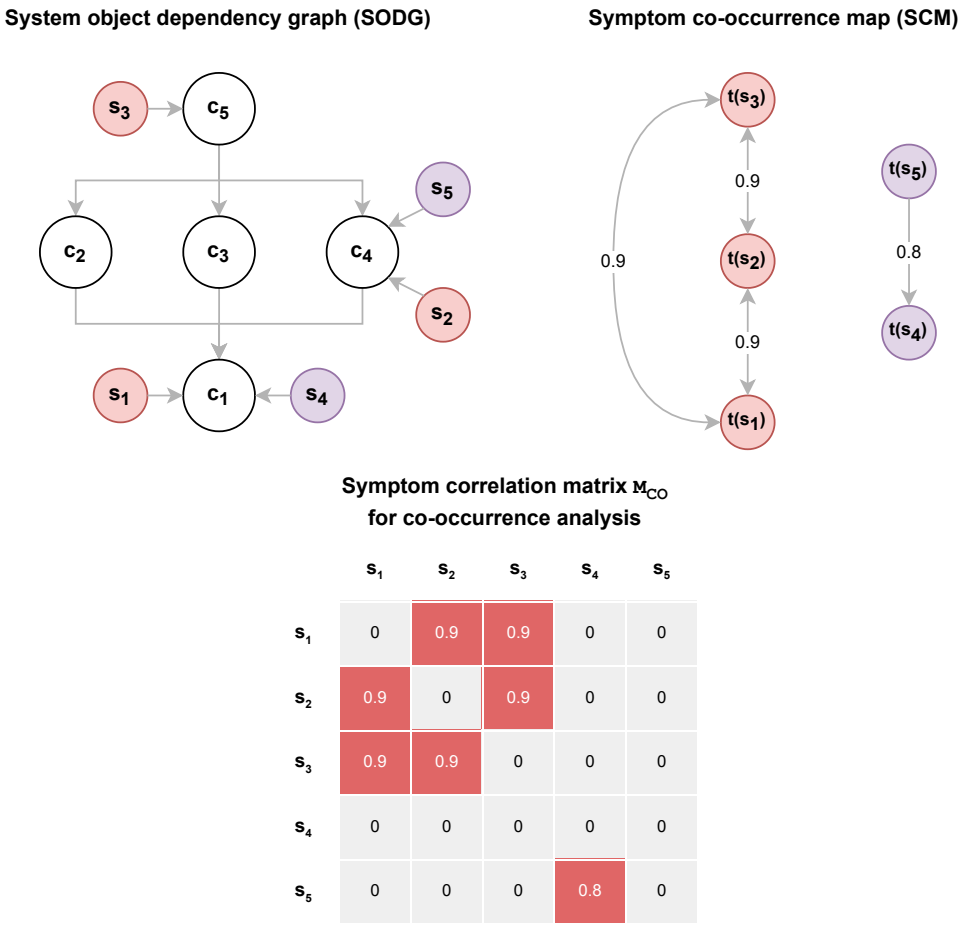


Figure 6.3: Symptom co-occurrence analysis: relevant fragment of the system object dependency graph with ingested symptoms (upper-left), relevant fragment of the symptom co-occurrence map (upper-right), obtained symptom correlation matrix (bottom)

in the graph represent known symptoms, where symbol $t(s)$ denotes the type of symptom s, while edges represent mined semantic symptom dependencies and are labeled with estimated dependency strength. The resulting correlation matrix is shown at the bottom with coefficient values mapped directly from corresponding edges in the co-occurrence map. The matrix clearly illustrates the separation of parallel faults as two distinct coefficient clusters comprising symptoms $s_{1:3}$ and $s_{4:5}$, respectively.

Algorithm 6 describes construction of the symptom correlation matrix. It takes a snapshot of the system object dependency graph G_D^t and symptom co-occurrence map G_S on its input. In particular, as part of the co-occurrence map structure, it takes function σ_S that allows inspecting semantic dependency strength for processed symptom pairs. For each pair of active symptoms $s_i, s_j \in S^t$, the algorithm looks up the dependency strength in the symptom co-occurrence map using the σ_S function and writes the obtained coefficient value into the matrix cell. If a symptom dependency is

not present in the co-occurrence map, the coefficient value is set to 0. The algorithm returns correlation matrix M_{CO} on its output.

Algorithm 6: Construction of symptom correlation matrix for symptom co-occurrence analysis

Input: Snapshot of system object dependency graph
$G_D^t = (O^t = C^t \cup S^t, D_O^t)$,
system co-occurrence map $G_S = (ST, D_{ST}, \sigma_S, \mu_S)$
Output: Symptom correlation matrix M_{CO}
for each permutation of symptoms $s_i, s_j \in S^t$ **do**
$\sigma \leftarrow \sigma_S(s_i, s_j)$
if $\sigma = \emptyset$ **then**
$M_{CO}(i, j) \leftarrow 0$
else
$M_{CO}(i, j) \leftarrow \sigma$
end if
end for

6.4 Symptom Time Lag Analysis

Similar to symptom co-occurrence analysis, time lag analysis operates on the symptom co-occurrence map. In the former case, the co-occurrence map is used as a lookup to obtain semantic symptom dependency strength based on event co-occurrence, while in the latter case, it is used to look up parameters of time lag probability distribution, i.e., distribution mean and IQR boundaries, required to evaluate if time lag between symptoms is valid.

In the time lag analysis, time lags are calculated for each pair of observed symptoms, and each time lag is evaluated against the time lag probability distribution corresponding to symptom types in the processed symptom pair. The coefficient takes high values for time lags centered around the distribution mean and converges to zero as the lag value moves towards IQR boundaries. Intuitively, the more typical a time lag between symptoms, the higher the coefficient value.

The analysis produces M_{TL} matrix comprising correlation coefficients based on symptom time lag analysis method introduced in Section 4.5. Each matrix cell holds a Cor_{TL} coefficient value calculated from Equation 4.27 and standardized to range $[0; 1]$.

Figure 6.4 takes failure scenario and symptom co-occurrence map from example illustrated in Section 6.4. However, contrary to symptom co-occurrence analysis, the symptom co-occurrence map retains parameters of time lag probability distribution instead of the dependency strength. The matrix at the bottom-left depicts time

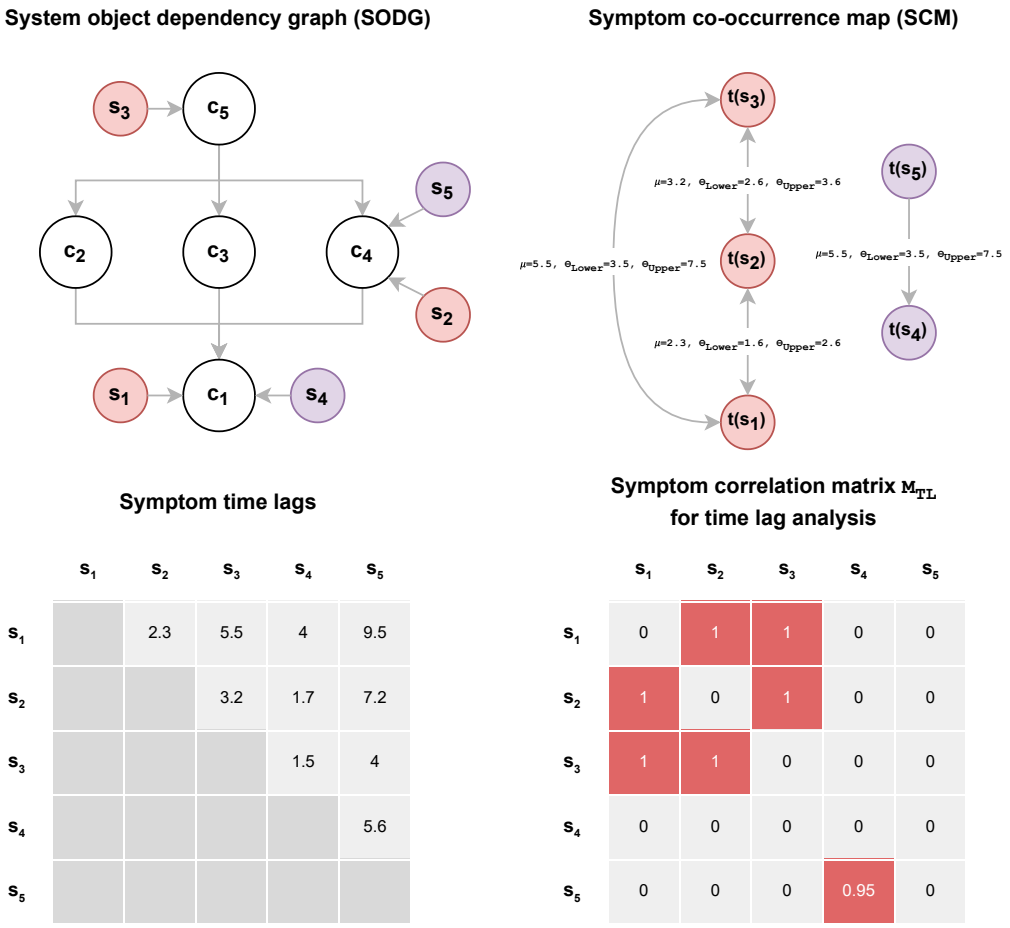


Figure 6.4: Symptom time lag analysis: relevant fragment of the system object dependency graph with ingested symptoms (upper-left), relevant fragment of the symptom co-occurrence map (upper-right), computed symptom time lags (bottom-left), obtained symptom correlation matrix (bottom-right)

lags calculated for observed symptom pairs, whereas the matrix at the bottom-right shows the resulting correlation matrix. Like symptom co-occurrence analysis, time lag analysis correctly separates parallel faults into two distinct coefficient clusters. The coefficient value is the highest for symptom pairs whose time lag matches the time lag distribution mean retrieved from the co-occurrence map.

Algorithm 7 describes construction of the symptom correlation matrix. It takes a snapshot of the system object dependency graph G_D^t on its input. It also takes the symptom co-occurrence map G_S together with the μ_S function required for accessing time lag distribution parameters for requested symptom pairs. In addition, it takes the function $\tau_{Failure}$ for retrieving symptom failure time estimated based on changepoint detection. For each pair of active symptoms $s_i, s_j \in S^t$, the algorithm calculates the time lag by measuring the absolute difference between symptom timestamps corrected with $\tau_{Failure}$ time. Then, using the μ_S function, it looks up

time lag distribution parameters in the symptom co-occurrence map, i.e., mean μ and IQR boundaries $\Theta_{Lower}, \Theta_{Upper}$. If distribution parameters are not available for a given symptom pair, the coefficient value is set to 0. Otherwise, the algorithm computes the correlation coefficient value following Equation 4.27. The algorithm returns correlation matrix M_{TL} on its output.

Algorithm 7: Construction of symptom correlation matrix for symptom time lag analysis

Input: Snapshot of system object dependency graph
$G_D^t = (O^t = C^t \cup S^t, D_O^t)$,
system co-occurrence map $G_S = (ST, D_{ST}, \sigma_S, \mu_S)$,
function for retrieving symptom failure time $\tau_{Failure}$
Output: Symptom correlation matrix M_{TL}
for each permutation of symptoms $s_i, s_j \in S^t$ **do**
$\tau \leftarrow |\tau_{Failure}(s_i) - \tau_{Failure}(s_j)|$ {time lag between symptoms s_i, s_j}
$\mu, \Theta_{Lower}, \Theta_{Upper} \leftarrow \mu_S(s_i, s_j)$
if $\mu = \emptyset$ **then**
$M_{TL}(i,j) \leftarrow 0$
else if $\tau < \mu$ **then**
$M_{TL}(i,j) \leftarrow 1 - \frac{|\mu - \tau|}{|\mu - \Theta_{Lower}|}$
else if $\tau = \mu$ **then**
$M_{TL}(i,j) \leftarrow 1$
else if $\tau > \mu$ **then**
$M_{TL}(i,j) \leftarrow 1 - \frac{|\mu - \tau|}{|\mu - \Theta_{Upper}|}$
else if $\tau > \Theta_{Upper}$ **then**
$M_{TL}(i,j) \leftarrow 0$
end if
end for

6.5 Symptom Time-series Analysis

The last symptom correlation analysis focuses on symptom time-series data. Specifically, it analyzes sudden changes in time-series trends in reaction to failure. The analysis is based on the observation that time-series trends tend to respond simultaneously to anomalous system situations. Typically, trends associated with a subset of reported symptoms increase at the same rate at the moment of failure. Consequently, co-occurrence in time-series trends may suggest correlations between corresponding symptoms. Notably, the analysis is only applicable to symptoms associated with time-series data.

The time-series analysis produces M_{TS} matrix comprising correlation coefficients based on symptom time-series analysis introduced in Section 4.6. Each matrix cell holds a Cor_{TS} coefficient value calculated using a modified version of Pearson

correlation method defined in Equation 4.31. The coefficient value is standardized to range $[0; 1]$.

Algorithm 8 describes construction of the symptom correlation matrix. It takes a snapshot of the system object dependency graph G_D^t on its input. In addition, it takes functions $\tau_{Activation}$ and $\tau_{Failure}$ for retrieving symptom activation and failure times, respectively. For each pair of active symptoms $s_i, s_j \in S^t$, the algorithm determines a time window w for time-series analysis. The window is selected as a shorter one out of windows constrained by failure onsets $\tau_{Failure}(s_i), \tau_{Failure}(s_j)$, estimated in fault symptom ingestion via changepoint detection, and symptom activation times $\tau_{Activation}(s_i), \tau_{Activation}(s_j)$. Then, the algorithm retrieves time-series fragments y_i, y_j designated by failure onsets and time window w, calculates the correlation coefficient value using the Pearson correlation method, and writes the coefficient value into the matrix. The algorithm returns correlation matrix M_{TS} on its output.

Algorithm 8: Construction of symptom correlation matrix for symptom time-series analysis

Input: Snapshot of system object dependency graph
$G_D^t = (O^t = C^t \cup S^t, D_O^t)$,
function for retrieving symptom activation time $\tau_{Activation}$,
function for retrieving symptom failure time $\tau_{Failure}$
Output: Symptom correlation matrix M_{TS}
for each permutation of symptoms $s_i, s_j \in S^t$ **do**
$w = min\{\tau_{Failure}(s_i) - \tau_{Activation}(s_i), \tau_{Failure}(s_j) - \tau_{Activation}(s_j)\}$ {determine time window length for time-series analysis}
$y_i \leftarrow$ fragment of s_i symptom time-series designated by $\tau_{Failure}(s_i)$ and w
$y_j \leftarrow$ fragment of s_j symptom time-series designated by $\tau_{Failure}(s_j)$ and w
$M_{TS}(i, j) \leftarrow PCC(y_i, y_j, w)$ {compute Pearson correlation coefficient}
end for

6.6 Fault View Graph Construction

After computing correlation matrices for implemented symptom correlation analyses, the inference algorithm consolidates them into an aggregated correlation matrix. According to the fault diagnosis concept, aggregated coefficient values derived from analyzing the failure situation from distinct topological and semantical perspectives constitute a basis for identifying causal symptom dependencies.

The aggregated correlation matrix is computed by multiplying coefficient values by configured analysis weights and summing components across correlation matrices. More precisely, the aggregation strategy is a derivative of the strategy defined by the symptom correlation framework in Equation 4.33, which, translated into matrix space, takes the form:

$$M_{Agg}(i,j) = \sum_{k=1}^{n} w_k \cdot M_k(i,j)\,, \tag{6.1}$$

where $M_1, ..., M_n$ are correlation matrices obtained from individual correlation analyses, whereas $w_1, ..., w_n \in [0;1]$ are their weights.

For the considered symptom correlation analyses, the Equation 6.1 can be instantiated to

$$\begin{aligned} M_{Agg}(i,j) = w_{Dist} \cdot M_{Dist}(i,j) + w_{CO} \cdot M_{CO}(i,j) \\ + w_{TL} \cdot M_{TL}(i,j) + w_{TS} \cdot M_{TS}(i,j)\,. \end{aligned} \tag{6.2}$$

Further, considering that discussed correlation analyses return non-zero values even if symptoms are not strongly correlated, whereas the number of symptom dependencies can be significant due to the correlation between each possible symptom pair, the aggregated correlation matrix is reduced to exclude weak dependencies from computations in subsequent algorithm stages. The reduction can be implemented using strict thresholding or a statistical method for outlier detection. In the proposed solution, the former method is used, i.e., all coefficient values falling below a configured threshold are reset to zero.

The reduced aggregated correlation matrix is translated into the fault view causality graph, whose objective is to provide a unified and global insight into active symptoms reported in the system and cause-effect dependencies obtained from the RCA model analysis. The fault view is primarily employed to detect and evaluate fault trajectories in subsequent algorithm stages.

Formally, the fault view is a directed graph $G_{FV} = (S_{FV}, D_{FV}, \sigma_{FV})$ where S_{FV} is a finite set of active symptoms, D_{FV} is a set of directed edges representing consolidated causal symptom dependencies, and σ_{FV} is a function returning aggregated symptom dependency strength for a given symptom pair, i.e., $\sigma_{FV}(s_i, s_j) = M_{Agg}(i,i)$.

Algorithm 9 describes the construction of the fault view graph. It takes a set of correlation matrices $M_1, ..., M_n$ obtained from symptom correlation analyses together

with a set of corresponding analyses weights $w_1, ..., w_n$. In addition, it takes a reduction threshold σ_{min} required for excluding week symptom dependencies from the graph. For each pair of symptoms s_i, s_j in the correlation matrix, the algorithm adds or updates nodes representing both symptoms in the fault view graph. Next, it computes an aggregated correlation coefficient M_{Agg} using the Equation 6.1. If the obtained coefficient value is greater than the configured reduction threshold σ_{min}, the symptom pair is linked in the graph with an edge of weight equal to the calculated coefficient value. The algorithm returns the fault view graph G_{FV} on its output.

Algorithm 9: Construction of fault view causality graph

Input: Symptom correlation matrices $M_1, ..., M_n$,
analysis weights $w_1, ..., w_n$,
dependency reduction threshold σ_{min}
Output: Fault view graph G_{FV}
for each permutation of symptoms s_i, s_j in the correlation matrix **do**
add or update nodes s_i, s_j in G_{FV}
$\sigma \leftarrow \sum_{k=1}^{n} w_k \cdot M_k(s_i, s_j)$
if $\sigma > \sigma_{min}$ **then**
add edge (s_i, s_j) to G_{FV}
$\sigma_{FV}(s_i, s_j) \leftarrow \sigma$
end if
end for

6.7 Fault Trajectory Detection

Given the fault view graph, the inference algorithm designates failure effect and root cause symptoms and determines a set of candidate fault trajectories for further processing, i.e., scoring and ranking. Trajectories constitute an explanation of failure for the system administrator, informing affected system regions and designating vectors of failure propagation across system components.

Fault trajectories originate from effect symptoms and run towards potential root causes as illustrated in Figure 6.5. Root causes are symptoms for which corresponding nodes in the fault view graph have the egress degree equal to zero, i.e.,

$$S_{RC} = \{s \in S_{FV} : outdegree(s) = 0\}, \tag{6.3}$$

where S_{FV} is a set of active symptoms in the fault view graph G_{FV}. In other words, if a symptom does not causally depend on any other symptom, it constitutes a potential root cause. There can be many root cause symptoms in the fault view graph.

Analogously, effect symptoms are designated as the ones for which corresponding nodes in the fault view graph have the ingress degree equal to zero, i.e.,

$$S_E = \{s \in S_{FV} : indegree(s) = 0\}. \tag{6.4}$$

Intuitively, these are symptoms that are not the direct cause of any other symptom. Similar to root cause symptoms, there can be many effect symptoms in the fault view graph. Hence, the fault trajectory detection algorithm must consider paths between many source nodes and many target nodes in the fault view graph.

Formally, fault trajectory is a simple directed path $P_{FT} = (s_E, s_1, s_2..., s_n, s_{RC})$, where $s_E \in S_E$ is an effect symptom, $s_{RC} \in S_{RC}$ is a root cause symptom, whereas $s_1, ..., s_n$ are intermediate trajectory symptoms.

Considering that the fault view has a graph structure, the fault trajectory detection can be realized using the Depth-first Search (DFS) algorithm from graph theory [73]. The algorithm starts at the node corresponding to an effect symptom and traverses as far as possible along each branch before backtracking or reaching the node corresponding to a root cause symptom.

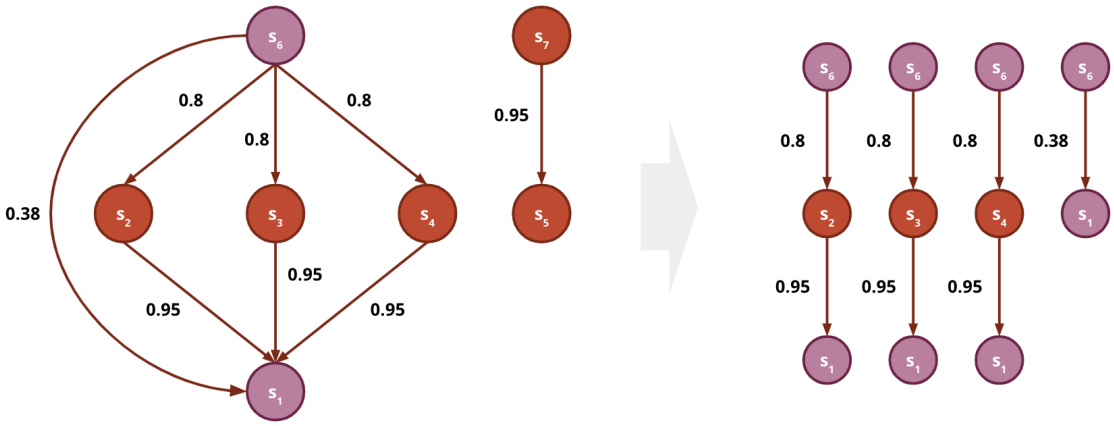


Figure 6.5: Fault trajectory detection: fault view causality graph (on the left side), extracted fault trajectory candidates (on the right side)

Algorithm 10 describes the process of detecting fault trajectories based on the fault view analysis. The algorithm takes the fault view graph G_{FV} and an optional effect symptom s_e on its input. First, it searches the fault view for potential effect symptoms S_E and root cause symptoms S_{RC} using Equation 6.4 and Equation 6.3, respectively. Then, for each pair composed of found effect symptom s_E and root cause symptom s_{RC}, the algorithm finds all simple paths that connect them in the graph using the Depth-first Search (DFS) algorithm. Found paths are appended to

the resulting set of detected fault trajectories FT. The algorithm returns detected fault trajectories FT on its output.

Algorithm 10: Fault trajectory detection

Input: Fault view graph $G_{FV} = (S_{FV}, D_{FV})$,
effect symptom s_E
Output: Fault trajectory candidates FT
$S_E \leftarrow \{s_E\}$ or $\{s \in S_{FV} : indegree(s) = 0\}$
$S_{RC} \leftarrow \{s \in S_{FV} : outdegree(s) = 0\}$
for $s_E \in S_E$ **do**
for $s_{RC} \in S_{RC}$ **do**
$FT^{'} \leftarrow DFS(s_E, s_{RC})$ {find all simple paths between s_E and s_{RC} using Depth-first Search (DFS) algorithm}
append $FT^{'}$ to FT
end for
end for

6.8 Fault Trajectory Scoring

After detecting fault trajectory candidates, the inference algorithm proceeds to evaluate them at the scoring stage. The scoring is based on two criteria derived from the information available in the fault view graph. First, the average dependency strength, denoted as $\sigma_{Strength}$, is measured between symptoms participating in the trajectory. It is calculated by summing approximated causal dependency strengths from trajectory edges and dividing them by the total number of edges. Second, the algorithm measures the total trajectory length, denoted as σ_{Length}, by counting the total number of edges in the trajectory. The strongest and longest fault trajectories obtain the best scores.

Algorithm 11 describes the process of scoring fault trajectories extracted from the fault view graph. The algorithm takes the set of fault trajectory candidates FT on its input. For each candidate trajectory P_{FT}, it calculates $\sigma_{Strength}$ and σ_{Length} scores based on the trajectory structure and dependency strengths. Then, it inserts trajectories annotated with scores into the set of scored fault trajectory candidates FT_{Scored}. The algorithm returns scored fault trajectories FT_{Scored} on its output.

Algorithm 11: Fault trajectory scoring

Input: Fault trajectory candidates FT
Output: Scored fault trajectory candidates FT_{Scored}
for $P_{FT} \in FT$ **do**
$\sigma_{Strength} \leftarrow$ average dependency strength in P_{FT}
$\sigma_{Length} \leftarrow$ total length of P_{FT}
$FT_{Scored} \leftarrow$ append $(FT, \sigma_{Strength}, \sigma_{Length})$
end for

6.9 Fault Trajectory Ranking

In the last algorithm step, scored fault trajectories are ranked by ordering them based on the results obtained at the scoring stage. Importantly, trajectories must be ordered based on both evaluation criteria, i.e., the average dependency strength $\sigma_{Strength}$ and the total trajectory length σ_{Length}. Shorter trajectories are more likely to return a high $\sigma_{Strength}$ score as they comprise fewer path edges holding potentially weak dependencies. At the same time, they show a tendency to omit intermediate symptoms in failure propagation. Therefore, the strongest and longest fault trajectories must be produced on the output of the inference algorithm.

The following procedure is used to enable the double ordering of fault trajectories. First, trajectories are sorted based on the $\sigma_{Strength}$ score using an arbitrary sorting algorithm, e.g., Quick Sort [73]. Then, they are grouped into buckets, each containing trajectories with a similar score but not with a difference greater than $\sigma_{\Delta Strength}$ from the strongest trajectory in the bucket. In the solution, difference of $\sigma_{\Delta Strength} = 0.05$ is assumed for filtering trajectories. Last, trajectories are sorted by the σ_{Length} score independently within each bucket using the same sorting algorithm.

Algorithm 12 describes the process of ranking fault trajectories. The algorithm takes the set of scored fault trajectory candidates FT_{Scored} on its input. In addition, it takes the configured bucket strength score difference $\sigma_{\Delta Strength}$ required for bucketing similar fault trajectories before the secondary ordering. First, the algorithm sorts trajectories in FT_{Scored} by the $\sigma_{Strength}$ score, producing the FT_{Sorted} list. It also initializes the list of ranked fault trajectories FT_{Ranked}. Then, the algorithm proceeds to bucketing trajectories with marginal difference in the $\sigma_{Strength}$ score. It picks the first trajectory from the sorted list FT_{Sorted} as the strongest trajectory in a bucket and uses its strength score $\sigma^{i}_{Strength}$ as a reference for evaluating further trajectories. The algorithm compares strengths of subsequent trajectories until their strength score $\sigma^{j}_{Strength}$ is different from the reference score $\sigma^{i}_{Strength}$ by $\sigma_{\Delta Strength}$. The first trajectory with a score difference greater than $\sigma_{\Delta Strength}$ constitutes the strongest trajectory in the new bucket.

Algorithm 12: Fault trajectory ranking

Input: Scored fault trajectory candidates FT_{Scored},
allowed bucket strength score difference $\sigma_{\Delta Strength}$
Output: Ranked fault trajectories FT_{Ranked}

```
FT_Sorted ← FT_Scored sorted by σ_Strength
FT_Ranked ← initialize list of ranked fault trajectories
i ← 0
while i < len(FT_Sorted) do
    (P^i_FT, σ^i_Strength, σ^i_Length) ← FT_Sorted(i)
    j = i + 1
    while j < len(FT_Sorted) do
        (P^j_FT, σ^j_Strength, σ^j_Length) ← FT_Sorted(j)
        if σ^i_Strength − σ^j_Strength <= σ_ΔStrength then
            break
        end if
        j = j + 1
    end while
    FT_Ranked ← extend with FT_Sorted(i : j) bucket sorted by σ_Length
    i = j
end while
```

The output of the inference algorithm lists fault trajectories constructed from symptoms observed in the system. Trajectories are ordered by likelihood estimated by trajectory length and standardized aggregation of correlation coefficients produced by correlation methods encompassed in the symptom correlation framework.

6.10 Summary

The chapter contributes by proposing the methodology for organizing the normalization and processing of individual correlation methods from the adopted symptom correlation framework. Presented techniques of translating correlation results into matrices, standardization of coefficient values, and weighted result aggregation strategy, combined with matrix arithmetics, solve the problem of consolidating correlation results into a fault view causality graph.

Given the fault view graph, another aspect addressed by the algorithm realization is recognizing failure causes and extracting relevant fault trajectories as a failure explanation for the system administrator. The algorithm realizes these stages by adopting techniques from graph theory. Further, extracted fault trajectories are scored and ranked using the established trajectory scoring and ranking criteria based on trajectory properties. Most probable fault trajectories are produced at the algorithm output.

Chapter 7

Prototype Implementation

Previous chapters outline the RCA solution concept and concretize its realization by means of defining RCA model structures and elaborating inference algorithm stages. However, an experimental evaluation is necessary to prove that the presented solution realization is valid. For that purpose, a deployable and executable solution implementation must be developed. This chapter motivates the technology stack selected for solution implementation and explains implementation details, including solution architecture, object-oriented component decomposition, and used libraries. Moreover, the chapter aims to confirm the feasibility of constructing required RCA model structures using data and interfaces offered by existing application platforms and observability tools.

7.1 Technology for Prototype Implementation

First, the technology stack employed in the solution is motivated. Both selection criteria and selected technologies are discussed. Section 7.1.1 elaborates selection of the application platform. Section 7.1.3 motivates observability integrations. Last, Section 7.1.2 details the selection of graph database technology for RCA model persistence.

7.1.1 Application Platform

In order to create a reference implementation for potential use in production-grade systems, the prototype should follow the trend of application technology dominating modern IT systems. Moreover, it should align with the technology stack employed in the company to strengthen the industrial character of the dissertation. Currently, the company is undergoing a transition from heavy to light virtualization by migrating its workloads to a Kubernetes-based application platform.

The application platform is notably the main determinant of the technology stack. Due to functioning in higher system layers, i.e., orchestration and application layers, it typically depends on the realization of lower system layers, thus imposing the employment of specific technologies.

Kubernetes was selected as the application platform due to its widespread use in the IT industry. It automates application deployment, scaling, and upgrades and, over the years, developed abstractions and policy-driven language that describe each aspect of application operation, including configuration, storage, networking, and security.

Another strong argument favoring Kubernetes is a rich ecosystem of tools, including monitoring systems, databases, message queues, or continuous integration utilities. It is governed by the Cloud Native Computing Foundation (CNCF) [1], ensuring the appropriate quality of hosted projects. Moreover, Kubernetes builds a space for the emergence of new technology. For instance, service mesh, improving inter-service communication management, observability, and security, was created mainly for containerized workloads. Most technologies in this category are tightly integrated with Kubernetes and often provide limited support for deployment on other platforms.

A rich tooling ecosystem, innovative technologies not available on other platforms, and significant savings promised by light virtualization cause an increasing part of

[1] Cloud Native Computing Foundation

the industry to shift towards containerized workloads. Most choose Kubernetes as the most functional and stable container orchestrator.

Additionally, to follow trends of modern cloud-native applications, the Istio service mesh was employed as a platform extension. By injecting an Envoy proxy into each application component, it transparently intercepts ingress and egress traffic enabling enhanced traffic management, observability, and security. Istio Proxies deployed throughout application services form a dedicated service layer controlled by the Istio control plane. Traffic policies can be configured using Custom Resource extensions to the Kubernetes API.

7.1.2 Database Technology

The graphical nature of data representation proposed in the elements of the RCA model imposes the selection of specialized graph database technology, ensuring an efficient and durable layer of graph data persistence and processing.

Intuitively, one of the first criteria for selecting a database technology is its performance and the way it deals with large data sets. In this aspect, one should estimate the size and required processing rate of the primary structure contained in the data model. Expectedly, the system object dependency graph constitutes the most extensive and intensively processed structure in the proposed solution. As such, it should be considered as a reference structure for performance evaluation.

Given Kubernetes as an application platform, one can use the capacity limitations of a single Kubernetes cluster to assess the size of the data model. According to the documentation [2], the platform can accommodate a maximum of 150,000 Pods within 5,000 worker nodes. Notably, Pods are the most numerous object type in the dependency graph. Moreover, the implemented prototype is confined to a single Kubernetes cluster as multi-cluster support poses a difficult implementation challenge, and, at the time of writing, multi-cluster technology is not well established in the Kubernetes ecosystem.

Based on the above information, it can be reasonably assumed that the number of nodes in the dependency graph, at least by the level of graph detail assumed in the prototype, will not exceed one million. Instead, the dominant part of the graph structure will be formed by graph edges. Indeed, assuming that each Pod, on average, is hosted on a Node, supervised by a Deployment, aggregated behind a Service, and configured by a ConfigMap or Secret, which together create at least 4-5

[2]Kubernetes cluster limitations

links, the estimated upper limit of edges will be 5-10 million. Although moderately large, the expected graph scale is not challenging for modern graph databases.

Moreover, given that the limit of concurrent requests to the Kubernetes API, the main source of system objects in the prototype, is low and defaults to 600 requests per second [3], it can be estimated that the frequency of changes applied into the dependency graph will not exceed 10000 transactions per second. However, it must be emphasized that the limit can be much higher for larger deployments with a tuned configuration. Similar to graph scale considerations, most graph databases can handle the estimated processing rate.

Another criterion for database technology selection is the support for graph operations. In particular, the selected technology must provide a convenient interface and native acceleration for graph algorithms used in the proposed solution. For instance, the set of algorithms required in dependency graph construction and failure analysis comprises shortest path searching, simple path traversing, calculating node degrees, and determining node neighborhood. In addition, as a potential future extension, support for analytical features, including PageRank or graph centrality, may be of significant importance.

Further, the selected graph technology must preserve a historical record of system objects and object dependencies required for point-in-time graph snapshotting, detailed in Section 5.3. The literature denotes such a category of dynamic graph structures as temporal graphs [78]. Importantly, graph dynamics imply problems related to parallel processing by many entities [79]. Notably, the problem occurs in constructing the dependency graph as many concurrent processes read component and symptom data from external sources (APIs and event streams) and reflect them in the graph. Parallel graph updates without mechanisms ensuring proper order of change application threaten to violate the graph integrity. In this area, several interesting works emerged that propose graph models and algorithms implementing distributed temporal graph updates with safe parallelism [80, 81, 82].

While there exists a large body of research examining temporal graphs and parallel graph processing, no mature implementation was found that would allow objectively testing them in the prototype. Notably, enterprise-class systems rarely employ immature technologies. Instead, they prefer thoroughly tested technology that provides comprehensive documentation and active community support. Therefore, by employing stable technology, the prototype retains the ability to become a reference solution implementation.

[3] Kubernetes API in-flight request limits

ArangoDB

Therefore, a decision was made to select a database technology that is well-established in the industry, even if it does not satisfy all selection criteria. Missing elements in native database functionality were augmented programmatically.

Conclusively, ArangoDB was selected as the primary database. As a multi-model engine, it allows storing and processing the data in document, key-value, and graph format using a unified interface, thus enabling the prototype to seamlessly store additional helper data structures that do not necessarily have a graph structure. That constitutes a valuable benefit over other database options focusing strictly on the graph model. Moreover, ArangoDB provides native support for standard graph algorithms and offers a graph query language (AQL), enabling filtering graph nodes and edges by attributes and edge patterns.

7.1.3 Observability Integrations

Observability is a crucial and significantly developed toolset area of mature application platforms. It is dictated by the demand for high availability and high performance established by modern enterprise applications. The observability toolset must provide key application performance indicators and ensure insight into their runtime context. Typically, observability agents are localized across several system layers to diversify insight into system operational aspects and extend the spectrum of identified system defects.

Kubernetes platform selected for solution evaluation offers a wide range of observability tools. Some were ported from other platform ecosystems (e.g., Zabbix), while others were explicitly created for Kubernetes (e.g., Prometheus). Most tools considered in this dissertation are supervised by CNCF.

The selection of observability tools for solution prototype implementation was based on three criteria. First, each tool must offer an alerting feature as alerts constitute the foundation for symptom co-occurrence analysis and are the main input for the inference algorithm. Obligatorily, information transported in emitted alerts must enable associating alerts with source components. Second, chosen observability toolset must maximize the variety of analyzed system operation aspects and process telemetry data across several system complexity layers. Third, although not mandatory, open-source tools are preferred over commercial ones. The following sections summarize observability tools selected for integration in the prototype.

Prometheus

Prometheus [4] is a defacto standard monitoring and alerting system for Kubernetes. It is based on the pull approach. Specifically, application and infrastructure elements or relevant metric exporters must expose an HTTP endpoint with labeled metrics and values formatted using appropriate syntax and naming rules. Prometheus collects the metric data by scraping configured endpoints at moderately equal time intervals, called scrape intervals, and persists the data as time-series in its storage for subsequent reading and filtering by the user. In addition, Prometheus provides a rich PromQL query language enabling efficient metric filtering and defining alerting expressions. Prometheus evaluates alerting rules against collected metric data at regular time intervals and activates alarms when corresponding expressions are satisfied. Alert notification processing is carried out by Alertmanager [5], an external component that receives alerts from Prometheus, deduplicates them, and routes them to configured backends.

Built for Kubernetes, Prometheus implements many platform-related integrations. First, it provides containerized exporters for application elements such as databases, message queues, or storage. Moreover, it provides exporters dedicated to Kubernetes. For instance, Node Exporter, deployed as a DaemonSet, exposes a scrape endpoint on each Kubernetes node with metrics related to the operating system and hardware resource consumption. In turn, Kube State Metrics subscribes to the Kubernetes event stream and generates metrics related to the statuses of Kubernetes objects. Further, Prometheus supports automated scrape endpoint discovery via the Kubernetes API. Finally, Prometheus annotates emitted alerts with Kubernetes metadata for alert source localization.

Considering common use in the IT industry, open-source license, tight integration with Kubernetes, availability of metric exporters for a wide range of application and infrastructure elements, and advanced alerting features, Prometheus was selected as the primary monitoring system subjected to integration in the prototype.

Elasticsearch

Complementary to Prometheus, which concentrates on collecting metric data of time-series characteristics, Elastic stack [6] was used to aggregate log data populated by application and platform components. As an integrated solution, it incorporates three components, namely Elasticsearch, Filebeat, and Kibana. Elasticsearch is an open-

[4] Prometheus homepage
[5] Alertmanager homepage
[6] Elastic homepage

source distributed search engine based on the Apache Lucene library. It is responsible for storing, indexing, searching, and analyzing large volumes of document data in real-time. Unlike Prometheus, which uses a pull approach for data gathering, Elasticsearch operates on the push approach, i.e., normalized data is transmitted directly to Elasticsearch by external Filebeat aggregation units. Filebeat instances consume log entries from configured workloads and then orderly execute their processing, including entry parsing, normalization, enrichment, and deduplication, before transporting them to Elasticsearch. Finally, the last stack component, Kibana, provides graphical tooling for exploring and visualizing data indexed in Elasticsearch.

Although designed for bare-metal and virtual machine workloads, Elasticsearch offers adequate integration with Kubernetes. All components are fully containerized, while stack installation is automated using a Helm chart. Importantly, the chart deploys Filebeat instances as a DaemonSet, enabling log collection from application and platform components running on each Kubernetes node. In addition, Elastic provides dedicated Filebeat plugins for Kubernetes that annotate each document transported to Elasticsearch with domain-specific information on the log entry source, e.g., name and namespace of a Pod that generated the log entry. That is especially important for fault symptom ingestion as annotations constitute source mapping for ingested symptoms.

Due to alerting features contained in the commercial part of the Elastic stack, Elastalert [7] was selected as an alternative tool with an open-source license. Elastalert is a framework for alerting anomalies, spikes, or other patterns of interest from data stored in Elasticsearch. It is based on alert rules in YAML format that utilize expressions from the Elasticsearch query language. Elasticsearch is periodically queried for data that, passed through alert rules, allows determining when a match is found. For instance, a rule may count returned results in a time window and check whether the total number of items exceeds the defined threshold. When a match occurs, alerts are activated and notified to configured backends.

Falco

Falco [8] constitutes an interesting supplement to the aforementioned observability tools. Instead of analyzing metrics or logs produced by application and platform components, it focuses on examining the user behavior. Among other things, the tool tracks user interactions with the platform, such as adding or modifying Kubernetes objects, launching containers, or executing commands in the terminal. In response to

[7] Elastalert homepage
[8] Falco homepage

these actions, Falco performs practical validations driving the detection of significant user oversights posing a potential threat to the security and operation of the system. For instance, in response to launching a container, Falco checks whether a sensitive path from the host filesystem is attached to the container or whether the container is overprivileged. Considering that administration, especially changes in system configuration, is the primary root cause of failures in IT systems, suspicious action events are valuable from the RCA perspective.

Like other selected tools, Falco exposes a gRPC API to retrieve alert lists. In addition, it provides a notification system that allows reporting alerts to arbitrary HTTP backends. Due to tight integration with Kubernetes, alert payloads contain the information required to identify alert sources, e.g., the name and namespace of a Pod.

Zabbix

Zabbix [9] was employed to monitor low system layers, mainly network and hardware layers. While it provides the functionality to monitor elements at higher platform and application layers, including monitoring Kubernetes workloads, Prometheus was selected for that purpose due to better platform compatibility. The use of Zabbix was limited to its unique low-level monitoring aspects. In this area, Zabbix automatically detects items such as network interfaces, network devices, processors, disks, or file systems. Then, using appropriate drivers and protocols, it analyses anomalies in their operation, e.g., physical disk damage, incorrect network interface status, increased processor temperature, or CPU fan failure.

Moreover, Zabbix provides an alerting system that allows forwarding alerts to arbitrary HTTP backends. Unfortunately, the alert payload is limited to including the name of a host which triggered an alert. Although not acceptable for application-level alerting, the information is sufficient for low-level alert mapping.

Kiali

Due to many faults propagating at the application service level, dependencies between application services constitute an essential part of the system structure maintained in the system object dependency graph.

Several methods exist for acquiring inter-service communication topology in cloud-native environments. Some are based on low-level mechanisms. For instance, Cilium

[9] Zabbix homepage

[10] utilizes the extended Berkeley Packet Filter (eBPF) [11] technology to inject custom network processing logic into the Linux kernel. As a result, it enables the implementation of network processing and observability policies with maximum efficiency and granularity. At the same time, policy enforcement is entirely transparent to applications. The Hubble [12] component integrates with Cilium by producing network metrics collected based on the eBPF and making them available to Prometheus. Then, based on metrics, it discovers service communication paths, protocols, and performance indicators in L3, L4, and L7 network layers.

Other approaches focus on getting service maps in higher abstraction layers. Kiali [13], selected for integration in the prototype, is a management console for the Istio service mesh. Its primary function is constructing the virtual mesh topology representation. Like Hubble, its interface exposes information describing used protocols and communication health on each service path, allowing the detection of request errors or delays. Kiali functionality is based on processing service metrics aggregated by Prometheus from Istio Proxy instances deployed throughout application services.

At the time of writing, Kiali does not offer an alerting feature. Its main use in the prototype implementation is assembling the service mesh topology.

7.2 Implementation Details

Following the technology stack motivation, solution implementation aspects are discussed in detail. First, a high-level view of prototype architecture is presented concerning the main solution modules. Then, aspects of problem decomposition are presented, including RCA model construction, fault symptom ingestion, and inference algorithm stages. The elaboration focuses mainly on object decomposition according to the Object-Oriented Paradigm (OOP). Complementary, Unified Modeling Language (UML) diagrams are presented to summarize object interfaces and illustrate relationships between object classes. In addition, the section mentions software libraries employed in solution implementation. Used library versions are listed in Table 7.1.

The structure of elaboration is as follows. First, Section 7.2.1 presents solution architecture. Next, Section 7.2.2 describes graph abstractions required for processing graphical data structures in the RCA model. Further, Section 7.2.3, Section 7.2.4, and Section 7.2.5 discuss elements of RCA model construction, including system

[10] Cilium homepage
[11] eBPF homepage
[12] Hubble homepage
[13] Kiali homepage

Table 7.1: Software library versions used in prototype implementation

Library	Version
Numpy	1.21.0
Pandas	1.3.0
SciPy	1.6.3
Scikit Learn	0.24.2
NetworkX	2.5.1
Ruptures	1.1.3
Python-Arango	5.4.0
Flask	1.1.1
Flask RESTX	0.2.0
React	16.13.1
D3.js	5.16.0

object dependency graph, fault symptom ingestion, and symptom co-occurrence analysis, respectively. Then, the implementation of inference algorithm stages is detailed in Section 7.2.6. Last, the Graphical User Interface (GUI) is demonstrated in Section 7.2.7.

7.2.1 Solution Architecture

A high-level view of the solution architecture is demonstrated in Figure 7.1. It illustrates data flow between key solution modules realizing system knowledge gathering and root cause analysis.

The diagram denotes two sources of system knowledge, i.e., *Topology source* and *Symptom source*. They provide data streams transporting information regarding system components and reported symptoms. In the implemented prototype, the application platform, i.e., Kubernetes, constitutes the primary source of information about system components, while integrated observability tools, namely Prometheus, Elastalert, Zabbix, and Falco, provide information about active symptoms.

Raw component data are processed by the *Component reflector* module. It uses dedicated object *Probes* that retrieve data from configured system topology sources via *Upstream proxies*, extract them to internal RCA model representation using object *Extractors*, and add processed objects to the *System object dependency graph*. Similarly, the *Symptom reflector* uses dedicated symptom *Ingestors* that ingest raw symptoms notified by integrated observability tools via *Ingestion API* and extract them using *Extractors*.

Components and symptoms are added to the *System object dependency graph* via *Graph API* backed by the ArangoDB database. In addition to providing CRUD

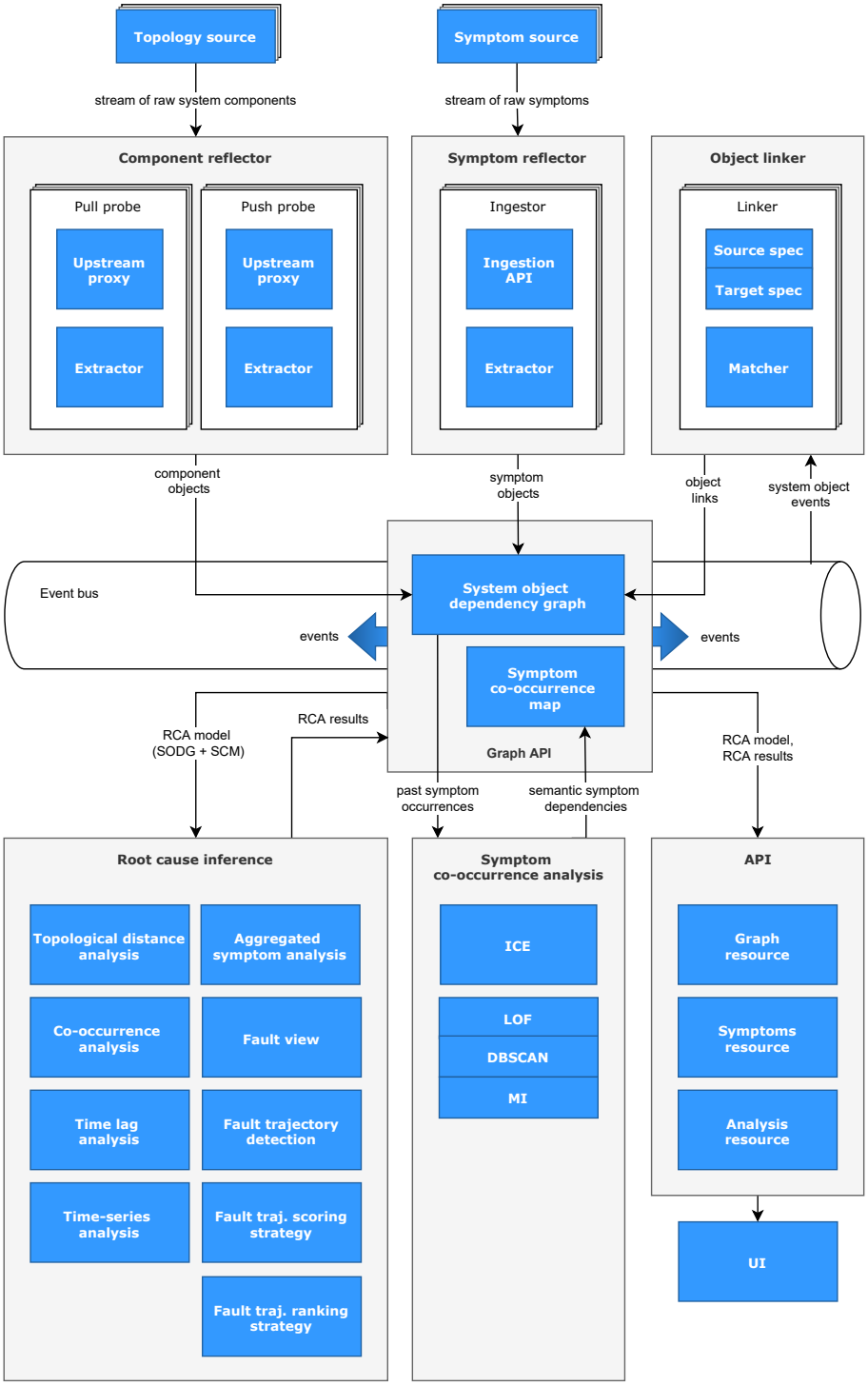


Figure 7.1: Overview of key elements in solution architecture

operations on graph structures employed in the RCA model, the module provides an event system enabling the integration of subsequent solution modules. In particular, adding objects to the dependency graph triggers the emission of events on the *Event bus*. Among others, these events are consumed by the *Object linker* module implementing object *Linkers* responsible for linking system objects into a hierarchy enforced by the adopted system object taxonomy. For each object added to the

dependency graph, the corresponding object *Linker* evaluates a series of matching conditions, enclosed in the *Matcher* implementation, between a given object, identified by the *Source spec*, and object instances on the other side of the considered object relation, identified by the *Target spec*. Discovered connections are persisted in the dependency graph through the *Graph API*.

System object dependency graph enriched with symptoms collected over a sufficient period provides data needed to compute semantic symptom dependencies based on the co-occurrence analysis. In this matter, the *Symptom co-occurrence* module retrieves past symptom occurrences from the dependency graph and performs costly dependency calculation using the ICE method encapsulated in the *ICE* submodule. Discovered symptom dependencies are stored in the *Symptom co-occurrence map* managed by the *Graph API*. In the prototype, the co-occurrence analysis is triggered manually.

The *Root cause inference* module implements subsequent stages of the inference algorithm. It takes symptom co-occurrence map on its input. In addition, it takes the current snapshot of the dependency graph obtained by filtering object timestamps. First, the algorithm performs a series of symptom correlation analyses encapsulated in *Topological distance anlaysis*, *Co-occurrence analysis*, *Time lag analysis*, and *Time-series analysis* submodules. Then, it consolidates correlation matrices produced by individual analyses into an aggregated correlation matrix using *Aggregated symptom correlation* submodule and converts it into a *Fault view* causality graph. The *Fault view* graph is an in-memory structure used for the analysis of a particular failure instance. Hence, it is not persisted in the database. Finally, the algorithm extracts, evaluates, and ranks fault trajectories in *Fault trajectory detection*, *Fault trajectory scoring*, and *Fault trajectory ranking* submodules. Ranked fault trajectories are persisted in the database as root cause analysis results.

Finally, the prototype exposes system knowledge and root cause analysis results via REST API. It is decomposed into distinct API resources, i.e., *Graph resource*, *Symptoms resource*, and *Analysis resource* allowing the retrieval of dependency graph structure, active symptoms, and inferred fault trajectories. The API is utilized by Graphical User Interface (GUI), created to visualize RCA model structures and inference results.

7.2.2 Graph API

As presented in Section 5.3 and Section 5.5, RCA model structures, mainly system object dependency graph and symptom co-occurrence map, are graph-oriented. In

addition, graph algorithms are used in elements of the inference algorithm as detailed in Section 6.2. Therefore, the prototype implementation must provide appropriate abstractions and toolset regarding graph data access, processing, and persistence. These features, enclosed in the *Graph API* module, will be discussed in reference to the system object dependency graph.

According to Section 7.1.2 which describes the criteria for selecting graph technology, the graph database must provide an interface that allows managing graph objects, i.e., nodes and edges. Moreover, it must be equipped with basic support for graph algorithms employed in the solution. Further, it must guarantee insight into historical graph states by maintaining a registry of objects that existed in the past. Finally, graph operations must be concurrently safe to enable parallel graph transformation by independent processes.

The selected ArangoDB database engine satisfies some of the above requirements. Specifically, it provides an access interface and query language for managing and searching graph objects. It also natively supports a minimum set of graph algorithms. However, it does not provide the ability to generate graph snapshots and lacks mechanisms for safe concurrency. Hence, these two elements were handled programmatically at the prototype software level.

Figure 7.2 decomposes the dependency graph access interface. For clarity, it will be discussed bottom-up.

Database engine integration was abstracted using the adapter design pattern, which, in essence, allows the integration of distinct but domain-related interfaces using a unified interface. In other words, it allows abstracting interfaces of many different databases in a unified interface that can be seamlessly utilized by high-level modules. As such, the applied graph database abstraction follows the open-closed principle, enabling database technology replacement without modifying the existing code. In the *Graph API*, the abstract *GraphDriver* class defines a graph adapter interface exposing methods for performing CRUD operations on graph nodes and edges. The interface of the adapter was implemented for the ArangoDB database and encapsulated in *ArangoDBDriver* class. Its implementation uses the ArangoDB client library for Python to perform database operations.

Further, the *GraphDriver* interface is contained in the *Graph* class, which constitutes the primary interface for managing graph structures in the RCA model. The *Graph* class enriches graph operations with time retrospection and concurrency safety mechanisms that are not natively supported by the selected database technol. Consequently, database selection criteria were reduced while additional processing aspects became orthogonal to the database technology.

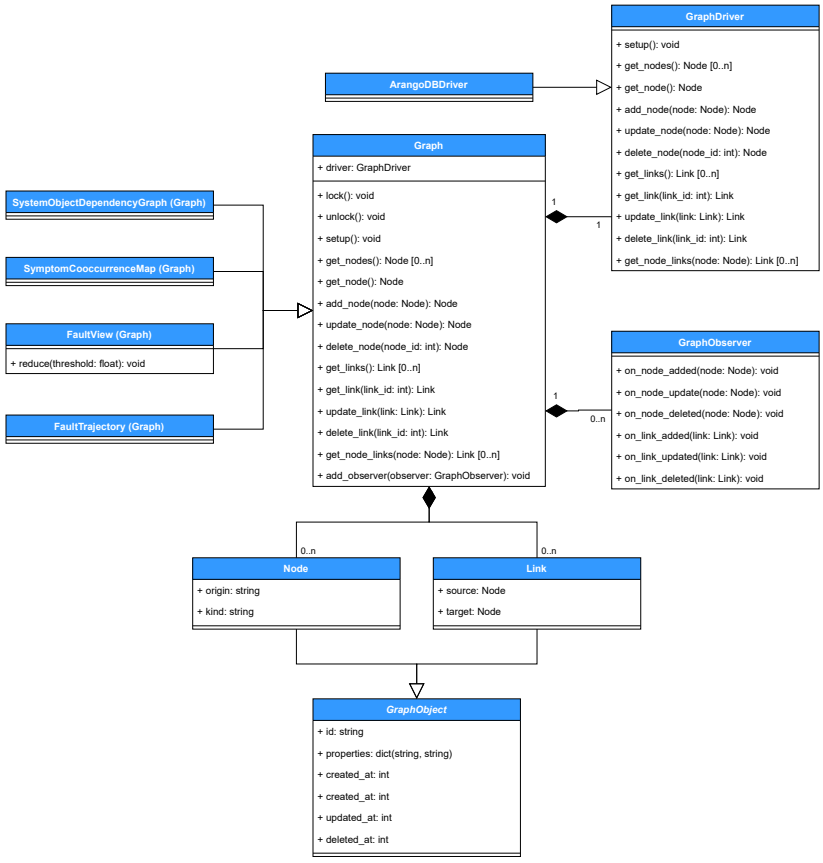


Figure 7.2: Object-oriented decomposition of graph abstraction used for modeling the system object dependency graph and other graphical data structures in the RCA model

Graph retrospection was implemented by annotating graph objects with timestamps, i.e., creation time, update time, and deletion time, set by *Graph* methods on each object creation, update, or deletion, respectively. In the case of the system object dependency graph, creation and update timestamps are obtained directly from processed object data to reflect the most up-to-date system information. Moreover, whenever an object is deleted in the system, its internal counterpart in the dependency graph gets soft-deleted, i.e., a deletion timestamp is set to denote object deletion while object information is preserved. Then, by adequately filtering objects based on timestamps, as defined in Equation 5.4, one can produce a graph snapshot reflecting system structure at any point in the past.

The described timestamp solution enables generating dependency graph snapshots at requested time points. However, the approach is inefficient as it requires scanning all active and soft-deleted objects. More precisely, if graph history is long, whereas the graph itself has extensive structure, the operation may quickly reach the scalability limits. A better approach, considered in the prototype but omitted due to implementational challenges, would be to use event sourcing techniques, i.e., represent

graph history as a sequence of change events. When applied one after another, events allow restoring graph state at any point in time. Then, only event deltas must be applied to generate graph states for subsequent time points. In addition, intermediate graph snapshots could be constructed at regular time intervals, e.g., once a day, to facilitate the snapshotting process in the inference algorithm. Given a time point, one would find the closest snapshot and apply a small subset of events between snapshot generation time and the requested snapshotting time. Similar approaches to temporal graph processing were studied in the literature [81].

Graph processing concurrency was secured using a multi-processing lock mechanism provided by Python standard library [14]. It enforces only a single process entering the graph modification block. Parallel processes retrieve and process data from configured knowledge sources, and when an operation on graph objects is required, attempt locking the graph. The lock blocks subsequent locking requests issued by other processes until all updates performed by the current process are completed.

Like graph snapshotting, the implemented concurrency safety mechanism fulfills its function but is ineffective on a large scale. Blocking the entire graph structure at each node linking may be a potential bottleneck in systems with a high evolution rate. A significant future improvement may be locking only parts of the graph subjected to modification, e.g., node pairs and corresponding edges that connect them. Moreover, there exist solutions addressing the concurrency problem without locking, e.g., by employing the actor model [80].

Finally, due to many entities depending on the graph data, the observer design pattern was applied to enable arbitrary actions in reaction to graph changes, e.g., linking new nodes added to the graph. In the observer pattern, an object, named subject, maintains a list of its dependents, called observers, and notifies them of state changes by calling their methods. Implementationally, the *Graph API* module provides the abstract *GraphObserver* interface that defines methods triggered in response to the creation, modification, and deletion of graph objects. In order to subscribe to graph events, dependent entities must implement the interface and register it using the *add_observer* method exposed by the *Graph*. Further, the *Graph* maintains a list of registered observers. Each CRUD operation performed on graph objects iterates over the observers and triggers corresponding observer methods. For instance, in response to deleting a node from the graph, the *on_node_delete* method is executed.

[14] Python multi-processing library

7.2.3 System Object Dependency Graph

The construction of the system object dependency graph was decomposed into abstractions reflected in *Probes*, *Ingestors*, and *Linkers*, illustrated in Figure 7.3, Figure 7.4, and Figure 7.5, respectively. They enable graph extension with component and symptom objects derived from arbitrary sources of system knowledge. Such integration flexibility ensures insight into essential system complexity layers and allows the analysis of symptoms warning a broad spectrum of system operation aspects.

Probes

Probes, decomposed in Figure 7.3, are responsible for processing raw information about system objects of a given type and normalizing them to an internal representation in accordance with the adopted data model. There are two types of *Probes* derived from pull and push approaches for constructing the dependency graph, detailed in Section 5.3. The former, *PullProbes*, are based on periodic fetching of all object instances via system management API and synchronizing them with the current state of the dependency graph as defined in Algorithm 1. *PushProbes*, in turn, subscribe to the system event stream and reflect received change notifications in corresponding object instances as discussed in Algorithm 2.

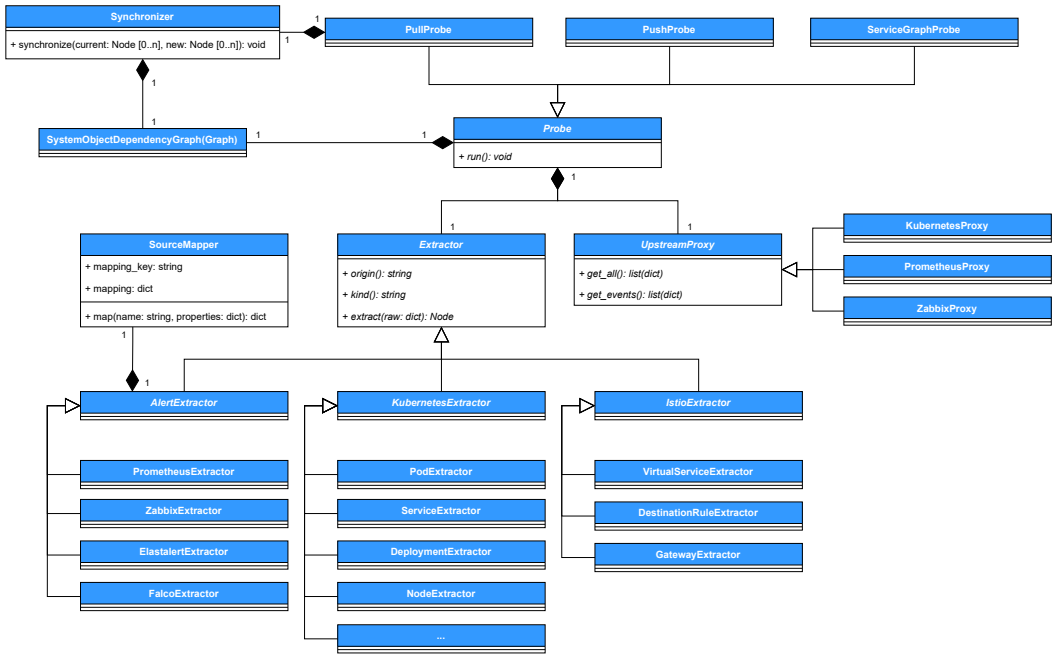


Figure 7.3: Object-oriented decomposition of *Probes* in the process of constructing the system object dependency graph

Each *Probe* comprises two elements, namely *UpstreamProxy* and *Extractor*. The former constitutes abstraction around access to system object data. It provides a unified interface for retrieving object information from the system management API and subscribing to the system event stream. The latter encapsulates the extraction of required object attributes from raw object data based on the adopted system object taxonomy.

After launching, *Probes* begin acquiring system object data through *Upstream proxies*. Then, object attributes are extracted from raw object data using *Extractors*. Last, normalized objects are added to the dependency graph as nodes via the *Graph API*. As a side effect, the graph emits events that trigger further object processing by dependent solution modules, e.g., linking.

In order to gather system objects important from the RCA perspective, a series of *Probes* was implemented for Kubernetes and Istio as origins of components in system layers constituting the application runtime environment. Kubernetes realizes an orchestration layer for containerized workloads. As a source of knowledge, it describes objects representing workloads, application services, application configuration, and persistence volumes. In turn, Istio, as a service mesh functioning in the application layer, complements the information related to application services.

Both Kubernetes and Istio *Probes* are based on the API exposed by the Kube API Server. Kubernetes API is a resource-based programmatic interface provided via HTTP protocol. It supports CRUD operations on native Kubernetes objects (e.g., Pods or Deployments) and Custom Resource extensions (e.g., VirtualServices in Istio). In addition, the API offers an efficient resource watch mechanism for tracking object changes via notifications. Both the REST interface and the watch mechanism were employed in the implementation of *PullProbes* and *PushProbes*, respectively.

Ingestors

Similar to *Probes*, *Ingestors*, decomposed in Figure 7.4, are solution units responsible for processing raw system objects and reflecting them in the system object dependency graph. However, unlike *Probes*, they focus on processing information related to symptoms. Moreover, instead of obtaining symptom information by utilizing management APIs or system event streams, they ingest raw symptoms notified by observability tools. Typically, observability tools support event-based alert notifications that can be forwarded to arbitrary backends. Implemented prototype leverages that mechanism as an integration point, enabling a more responsive gathering of symptom data for root cause analysis. Each *Ingestor* exposes a dedicated *Ingestion API* for ingesting alerts from a selected observability tool. Further, API endpoints, i.e.,

HTTP methods, query paths, expected payloads, and request validation, are adapted to the notification mechanism offered by the observability tool. After receiving a symptom, *Ingestor* processes it in analogy to *Probe*, i.e., normalizes raw object using *Extractor* and reflects its internal representation in the dependency graph.

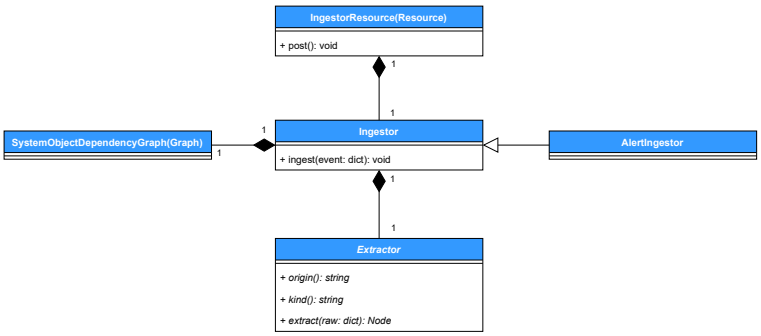


Figure 7.4: Object-oriented decomposition of *Ingestors* in the process of constructing the system object dependency graph

Linkers

Probes and *Ingestors* receive raw system objects, normalize them into internal object representation, and add or update corresponding nodes in the dependency graph. They concentrate solely on graph nodes corresponding to objects of a given type and do not attempt to link them with objects of other types. The rationale for such limitation of concern is twofold. First, operations on graph nodes are atomic and can be performed by many processes in parallel without disturbing the integrity of the graph structure. Second, according to the adopted Kubernetes object taxonomy, some objects create dependencies with other object instances of more than one object type. Given that each object type is processed by a different process, contrary to node operations, linking nodes in the graph is not concurrently safe and requires proper process coordination.

Linkers, decomposed in Figure 7.5, are units responsible for connecting objects in the dependency graph in reaction to graph events, e.g., node creation or modification. They process pairs of object instances of given types and evaluate whether an object pair should be linked in the graph based on the matching criterion. Each *Linker* comprises a *Matcher* and two specifications of linked objects, i.e., *Source spec* and *Target spec*. The *Matcher* encompasses the logic of evaluating whether a given object pair should be linked in the dependency graph. The evaluation is performed based on object attributes extracted by *Probes*. Further, *Source spec* and *Target spec* are specifications of source and target nodes in linked object dependencies. A specification describes the object type and an optional list of its attributes. Specifications are

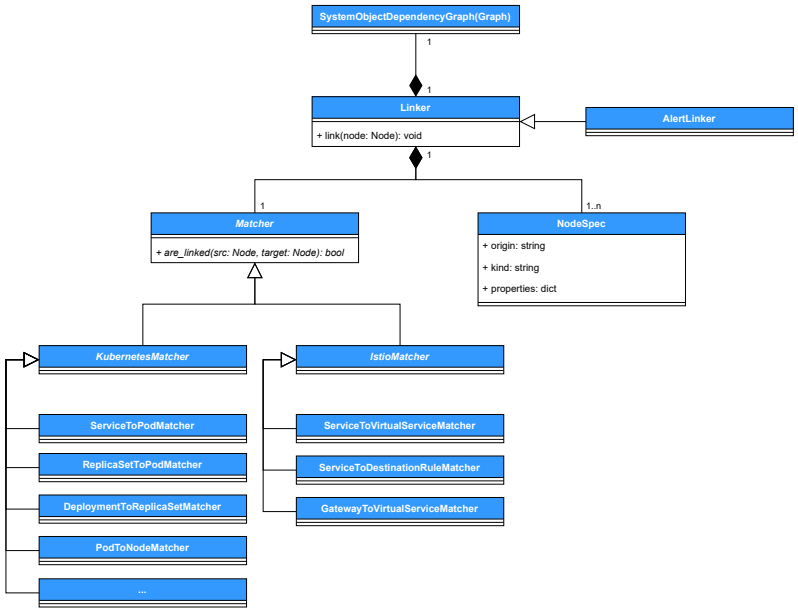


Figure 7.5: Object-oriented decomposition of *Linkers* in the process of constructing the system object dependency graph

used to fetch relevant graph nodes on both sides of the considered connection and form candidate links for evaluation by the *Matcher*.

Service Mapping

As pointed out in Section 7.1.3, communication paths between application services are essential from the RCA perspective and thus must be reflected in the system object dependency graph. In the prototype, application services are represented by means of Service objects obtained from the Kubernetes API. Connections between services are, in turn, fetched from the Istio observability tool - Kiali. However, Kiali provides an assembled service topology that comprises all application services from a given namespace. Therefore, a specialized *ServiceGraphProbe* was implemented. Unlike standard *Probe*, which concentrates on processing system objects sequentially without linking, *ServiceGraphProbe* fetches a complete service map from Kiali, finds corresponding Service objects in the dependency graph, and links them according to the map structure.

7.2.4 Fault Symptom Ingestion

From the implementation perspective, the key element of the fault system ingestion process is mapping symptoms to source components, i.e., components affected by failure manifested in reported symptoms. The exact information on symptom location

in system structure is essential for the inference algorithm, mainly due to structural symptom dependencies that can be inferred in the topological distance analysis.

According to Section 5.4, mapping symptoms to source components is realized using the source mapping, i.e., a set of attribute-based logical expressions read from the system object taxonomy that defines conditions for linking symptoms in the dependency graph. For that reason, as discussed in Section 7.1.3, emphasis was put on selecting observability integrations that provide alerting features capable of emitting symptoms that transport a minimum of information required for locating symptom sources.

Listing 1 presents an exemplary alert payload obtained by Prometheus *Probe* from Prometheus API. Precisely, the example presents the payload of *IstioRequestDurationHigh* symptom emitted in the context of application service managed by Istio in response to detected latency in communication with other services. Visibly, the payload transports information enabling source component localization. In this case, the source component is the Kubernetes Service object identified uniquely by name and namespace described by *destination_service_name* and *destination_service_namespace* attributes, respectively.

```
{
  "status":"firing",
  "labels":{
    "alertname":"IstioRequestDurationHigh",
    "destination_service_name":"cartservice",
    "destination_service_namespace":"hipster",
    "severity":"warning"
  },
  "annotations":{
    "description":"Istio request duration exceeds 500ms in last 1 minute.",
    "for":"1",
    "query":"sum(rate(istio_request_duration_ms_sum[1m])) by
    ↪  (destination_service_name,
    ↪  destination_service_namespace)/sum(rate(istio_request_duration_ms_count[1m]))
    ↪  by (destination_service_name, destination_service_namespace)",
    "summary":"Istio request duration is higher than usual."
  },
  "activeAt":"2020-01-10T12:40:05.339884758Z"
}
```

Listing 1: Examplary payload for *IstioRequestDurationHigh* symptom obtained from Prometheus

Combined with a set of rules from the system object taxonomy, a subset of which is presented in Listing 2, Prometheus *AlertExtractor* recognizes the rule for *IstioRequestDurationHigh* symptom based on regular expression. Then, it discovers the origin and kind of the source component based on *origin* and *kind* attributes specified

in the rule. In the example, parameter values are static and set to *kubernetes* and *service*, respectively, as all Istio symptoms are service-oriented. Last, based on the *properties* parameter, the *Extractor* maps labels extracted from Prometheus payload to object attributes in internal object representation. Here, Service name and namespace are retrieved from *destination_service_name* and *destination_service_namespace* attributes. As a result, the *Extractor* produces a source mapping, presented in Listing 3, that constitutes a query for searching source component in the dependency graph. Due to the separation of concerns, object linking is performed by *Linker* module in further graph processing.

```
prometheus:
  mappings:
    - name: Istio*
      source_mapping:
        origin: kubernetes
        kind: service
        properties:
          name: destination_service_name
          namespace: destination_service_namespace
```

Listing 2: Source mapping rules for Istio symptoms

```
{
  "source_mapping":{
    "kind":"service",
    "origin":"kubernetes",
    "properties":{
      "name":"cartservice",
      "namespace":"hipster"
    }
  }
}
```

Listing 3: Source mapping obtained based on rules presented in Listing 2 for Prometheus symptom described in Listing 1

7.2.5 Symptom Co-occurrence Analysis

Symptom co-occurrence analysis is the primary method for mining knowledge about symptom semantics. It translates symptoms into the event space, groups event instances into sequences, and examines the consistency of sequence co-occurrence in time. The more consistently events co-occur across sequences, the stronger the inferred symptom dependency.

In this aspect of solution realization, described in Section 4.4.3, a modified version of the Iterative Closest Events (ICE) method was proposed. ICE comprises several

statistical methods that complement each other to estimate the semantic symptom dependency. First, the Iterative Closest Points (ICP) algorithm is used to solve the optimization problem of finding event sequence assignments. Here, algorithm implementation from the Python ICP library [15] was used. Further, after aligning event sequences, time lags are calculated for each matched event pair, and multi-stage time lag filtering is performed to maximize the accuracy of the resulting time lag distribution. Outliers are rejected using the Local Outlier Factor method [16], while the remaining time lags are clustered using the Density-Based Spatial Clustering method [17]. Implementations of both methods were adopted from the Scikit-learn library. Finally, the largest time lag cluster is considered representative of the temporal dependency, and time lag distribution is estimated using the Kernel Density Estimator [18] from the SciPy library.

Given the time lag probability distribution produced by the ICE method, the co-occurrence method computes the correlation coefficient based on mutual information. Correlation coefficient was implemented using the standard Python *math* library and elements of Numpy [19]. In addition, SciPy was used to calculate entropy integrals [20] from Equation 4.22.

7.2.6 Inference Algorithm

Like other solution elements, the inference algorithm benefits from the OOP paradigm. By abstracting stages of failure analysis and providing reusable algorithm building blocks, it enables composing diagnosis strategy adequate for particular system class and system use-cases. That is consistent with the system correlation framework integrated into the inference algorithm, detailed in Section 6.1.

Recall the main stages of the inference algorithm. First, the algorithm performs symptom correlation analysis using implemented correlation methods, i.e., topological distance analysis, co-occurrence analysis, time lag analysis, and time-series analysis. Then, it consolidates obtained coefficient values into an aggregated symptom correlation matrix. The matrix is translated into the fault view graph. Based on its inspection, potential effect and root cause symptoms are discovered along with fault trajectories connecting them in the graph. Last, trajectories are scored and

[15] Python Iterative Closest Points
[16] Scikit-learn - Local Outlier Factor
[17] Scikit-learn - Density-Based Spatial Clustering
[18] SciPy - Gaussian Kernel Density Estimator
[19] Numpy homepage
[20] SciPy - Integral computation

ranked according to adopted scoring and ranking criteria. The strongest trajectories are returned at the algorithm output.

Following the separation of concerns principle, described algorithm stages were encapsulated in distinct classes, each fulfilling single algorithm responsibility. Their logical organization is illustrated in Figure 7.6 and Figure 7.7.

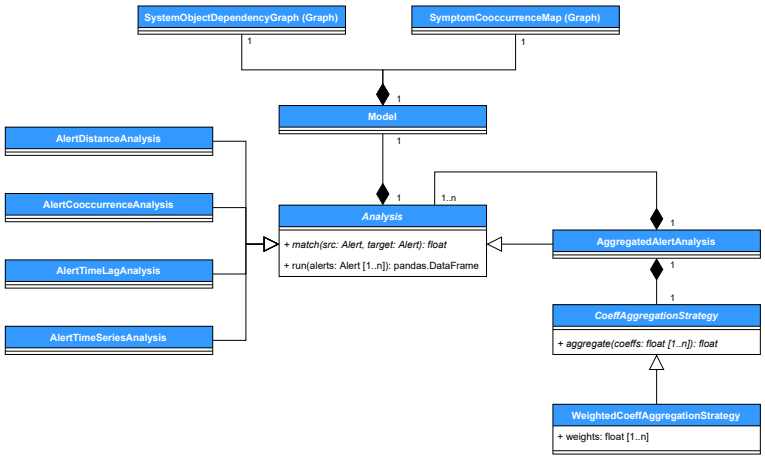


Figure 7.6: Object-oriented decomposition of symptom correlation analyses

Figure 7.6 decomposes the problem of symptom correlation analysis. The *Analysis* is an abstract class constituting the skeleton for concrete symptom correlation analyses. It comprises an instance of the RCA model, encapsulated in the *Model* class, as input for symptom correlation. The model, in turn, composes system object dependency graph and symptom co-occurrence map structures reflected in *SystemObjectDependencyGraph* and *SymptomCooccurrenceMap* derived from the common *Graph* base. In addition, the *Analysis* exposes two methods. The former, *match* method, is an abstract method returning the coefficient value for a given symptom pair. It is overridden in derivative classes based on the concrete symptom correlation strategy. The latter, *run* method, matches each symptom pair and produces a correlation matrix converted into a *DataFrame* from the Pandas package. *DataFrame* [21] is a two-dimensional data structure with labeled indexes that allows for efficient matrix transformations integrated with numeric libraries, e.g., Numpy.

Four concrete symptom correlation analyses were implemented in the prototype, enclosed in *AlertDistanceAnalysis*, *AlertCooccurrenceAnalysis*, *AlertTimeLagAnalysis*, and *AlertTimeSeriesAnalysis*. Each analysis overrides the *match* method with the calculation of correlation coefficient Cor_{Dist}, Cor_{CO}, Cor_{TL}, or Cor_{TS}, defined by Equation 4.32, Equation 4.24, Equation 4.27, and Equation 4.31, respectively. Cal-

[21] Pandas - DataFrame

culations are limited to the standard Python *math* library except for the topological distance analysis, where graph algorithms from the NetworkX library were used to support finding the shortest paths [22] between symptom sources in the dependency graph.

The *AggregatedAlertAnalysis* is a particular type of analysis responsible for consolidating correlation results into an aggregated symptom correlation matrix. The analysis composes an arbitrary number of analyses implementing the *Analysis* interface. In addition, the analysis employs a coefficient aggregation strategy abstracted in the *CoeffAggregationStrategy* class. In its *match* method, the *AggregatedAlertAnalysis* iterates through configured analyses, obtains correlation coefficients for a given symptom pair using their *match* methods, and computes the aggregated correlation coefficient using the provided aggregation strategy. The *WeightedCoeffAggregationStrategy* was implemented to support the weighted coefficient aggregation introduced in Equation 6.1.

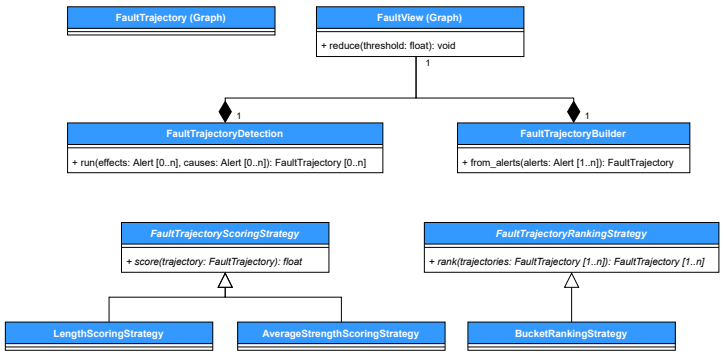


Figure 7.7: Object-oriented decomposition of inference algorithm stages

Further, Figure 7.7 decomposes the root cause analysis. Fault view, which is the main structure used in the inference, was modeled as *FaultView* class inheriting from the common *Graph* base. It extends the standard graph interface with the *reduce* method that removes edges with dependency strengths lower than a given threshold value. The graph is constructed from the aggregated correlation matrix produced at the output of the *run* method in the *AggregatedAlertAnalysis*.

Fault trajectory detection is realized by the *FaultTrajectoryDetection* class. It associates *FaultView* and *FaultTrajectoryBuilder* through composition. The latter is a helper class supporting the fabrication of fault trajectory candidates. The trajectory detection is based on graph algorithms from the NetworkX package. First,

[22] NetworkX - Shortest path searching

graph nodes are evaluated against degrees [23] to detect potential effect and root cause symptoms. Then, candidate fault trajectories are designated by searching for paths [24] connecting found symptoms in the fault view graph. Last, the detection uses the builder to instantiate trajectory paths into a set of *FaultTrajectory* objects for further processing. As a path, *FaultTrajectory* is a subclass of the *Graph*.

Given fault trajectory candidates, the inference algorithm scores them according to adopted scoring criteria. The stage is abstracted in the *FaultTrajectoryScoringStrategy* class by means of the abstract *score* method responsible for calculating a score for a given trajectory candidate. In the prototype, two concrete scoring strategies were implemented, namely *LengthScoringStrategy* and *AverageStrengthScoringStrategy*. They follow the scoring criteria explained in Algorithm 11. The former computes the score by counting trajectory edges, whereas the latter averages the dependency strength.

The *FaultTrajectoryRankingStrategy* class abstracts the last algorithm stage, i.e., fault trajectory ranking. Analogously to *FaultTrajectoryScoringStrategy*, it exposes a single abstract *rank* method, which aims to order a given set of trajectory candidates according to the ranking criteria. The *BucketRankingStrategy* implements the double-sorting approach detailed in Algorithm 12.

7.2.7 Graphical User Interface

Graphical User Interface (GUI) was designed and implemented to improve the inspection of RCA model structures and facilitate the analysis of results obtained in failure diagnoses. The interface comprises several features, including:

- visualizing system object dependency graph,
- filtering the dependency graph by object types and attributes,
- exploring symptoms collected in the process of fault symptom ingestion,
- highlighting system regions affected by a failure,
- introspecting the dependency graph in time,
- triggering root cause analysis for selected effect symptoms,
- visualizing fault trajectories produced in fault analysis.

[23] NetworkX - Counting nodes, edges, and neighbors
[24] NetworkX - Simple path searching

The component was implemented using JavaScript programming language and React library. Moreover, the D3.js library was used to produce dynamic, interactive visualizations of graph data. Further, the component was backend by REST API implemented using the Flask framework with the Flask RESTX extension.

Due to their size, screenshots illustrating exemplary GUI use cases were moved to the Appendix A.

7.3 Summary

This chapter presented a reference implementation of the proposed root cause analysis solution for cloud-native applications, created to evaluate the dissertation contribution. This prototype complies with the proposed realization of the symptom correlation framework (Chapter 4), RCA model (Chapter 5), and inference algorithm (Chapter 6) in accordance with the root cause analysis concept (Chapter 3).

Prototype implementation confirmed the feasibility of building an RCA solution in accordance with the solution concept. More precisely, using existing cloud-native technology, it is possible to implement a solution capable of providing a holistic view and analysis of cloud-native application structure and behavior. In particular, the prototype accomplishes the main challenge of automating the RCA model construction. Notably, the resulting dynamic model structure is capable of effectively adapting to the high frequency of changes occurring in the cloud environment.

Chapter 8

Evaluation

The proposed root cause analysis solution calls for a sound evaluation that confronts expectations formulated in the context of thesis statement, solution concept, and solution realization. Thus, in the scope of functional solution evaluation, proposed mechanisms related to the RCA model should be verified, particularly the construction of RCA model structures, including the system object dependency graph and symptom co-occurrence map. Additionally, the evaluation should confirm the correctness of methods employed in the inference algorithm, including symptom correlation analyses, fault view construction, fault trajectory extraction, and fault trajectory ranking. Furthermore, the essential evaluation goal is to prove that the solution can be successfully applied to real-life application scenarios and that it provides the expected level of automation for system knowledge mining and the process of fault inference.

8.1 Evaluation Overview

This chapter evaluates the functionality of the elaborated fault diagnosis process against faults emerging in the assumed category of cloud systems.

The evaluation revolves around two solution elements. The first element subjected to evaluation is RCA model construction. Referring to Section 5.1 the model constitutes mandatory input for root cause analysis providing the context needed to recognize system object dependencies, localize active symptoms, and support the analysis with mined knowledge of symptom semantics. Key model structures participating in the inference process and thus subjected to evaluation are system object dependency graph (Section 5.3) and symptom co-occurrence map (Section 5.5). The former maintains system object dependencies discovered by retrieving raw system information and processing it according to the system object taxonomy (Section 5.2). The latter describes semantic dependencies between known symptoms obtained through statistical analysis of past symptom occurrences. The primary evaluation goal is to confirm that procedures employed in RCA model construction are well integrated and produce the expected holistic system view required in root cause analysis. As part of the evaluation, RCA model structures acquired based on synthetic and real system data are validated, and semantic compliance of discovered object and symptom dependencies is discussed in the context of the considered system.

The second solution element subjected to functional evaluation is the inference algorithm detailed in Section 6.1. The algorithm is responsible for discovering fault trajectories from observed symptoms. As part of the algorithm, symptoms are correlated using established symptom correlation methods (Section 6.2, Section 6.3, Section 6.4, Section 6.5) based on system knowledge stored in the constructed RCA model. Then, results from individual correlation methods are consolidated into a fault view causality graph (Section 6.6). Last, given the fault view graph, fault trajectories are extracted (Section 6.7), scored (Section 11), and ranked (Section 12) based on the adopted scoring and ranking strategies. The evaluation discusses results produced by symptom correlation methods, examines the contribution of individual methods to the fault view structure, and verifies fault trajectories formed at the algorithm output.

8.2 Functional Evaluation on Synthetic Data

First, functional evaluation based on synthetic data is conducted to confirm theoretical assumptions and methods developed in the solution. The approach enables overcoming issues imposed by the true environment, such as non-deterministic system operation, high evolution rate, or limited system resources. Moreover, statistical methods employed in the solution require a large volume of historical data to produce accurate results. Synthetic generation facilitates obtaining the data, which normally would be challenging and time-consuming to collect.

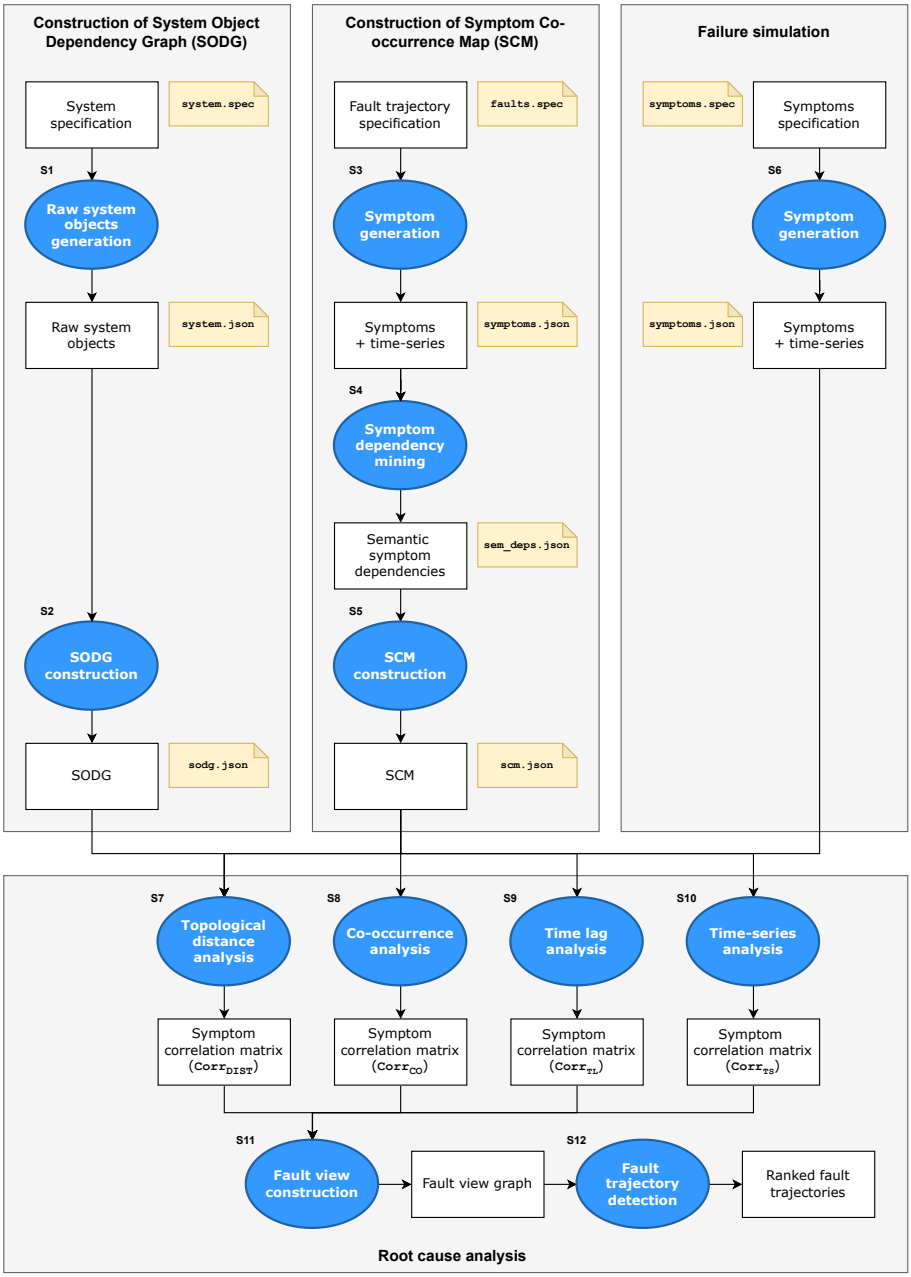


Figure 8.1: Concept of the solution evaluation process based on synthetic data

Figure 8.1 illustrates the concept of solution evaluation process. The diagram presents subsequent process stages (*S1* to *S12*) along with the input and output of each stage. The evaluation process begins with three pipelines, each producing a partial input to the root cause analysis performed in stages *S7* to *S12*. The objective of the first pipeline, denoted as *S1* $\rightarrow$ *S2*, is building the system object dependency graph. Based on a high-level system specification, raw system objects are generated (*S1*) and provided on the input of the solution module computing the dependency graph structure (*S2*). Further, in the second pipeline, denoted as *S3* $\rightarrow$ *S4* $\rightarrow$ *S5*, symptom co-occurrence map is generated. First, symptom instances are populated by simulating fault trajectories based on a high-level specification of causal symptom occurrences (*S3*). Then, event co-occurrence analysis is used to discover semantic symptom dependencies from generated symptom data (*S4*). Last, semantic dependencies are provided to the solution module, which computes the structure of the symptom co-occurrence map (*S5*). Finally, the third pipeline, denoted as *S6*, generates symptom instances based on a high-level specification to simulate new fault instances for testing the root cause analysis.

Subsequently, the inference algorithm is evaluated given the RCA model initialized from the outputs of the first two pipelines and faults manifested by symptoms produced in the third pipeline. First, symptom correlation is performed by four symptom correlation methods (*S7* to *S10*). Second, the fault view causality graph is created (*S11*) by aggregating correlation results produced by individual correlation methods. Last, fault trajectories are extracted from the fault view graph and ranked according to the trajectory scoring strategies (*S12*).

8.2.1 Application Scenario

The scheme of Kubernetes application used in functional solution evaluation is illustrated in Figure 8.2. The scheme comprises two Kubernetes object kinds, namely Pods and Services. The former represents application workload units, whereas the latter abstracts the exposure of workloads as network services. According to the scheme, each application service is implemented as a Kubernetes Service object. Additionally, each Service aggregates the configured number of Pod workload units to simulate application topology for scale and high availability.

The considered application has a tree structure typical for applications built based on the microservices architecture. It consists of the root *svc-d* service and two service dependency layers composed of services named *svc-0-x* and *svc-0-x-y*, where x and y are numerical service indices. The *svc-d* service constitutes an application entry point, whereas services *svc-0-x* implement high-level application features by

aggregating functionality from low-level services *svc-0-x-y*. Whenever a user performs a request to the *svc-d* service, the service performs a sequence of nested requests to high-level services, which in turn perform nested requests to low-level services. It is assumed that requests are performed sequentially. An exemplary request chain is: *svc-d* → *svc-0-0* → *svc-0-x-y*.

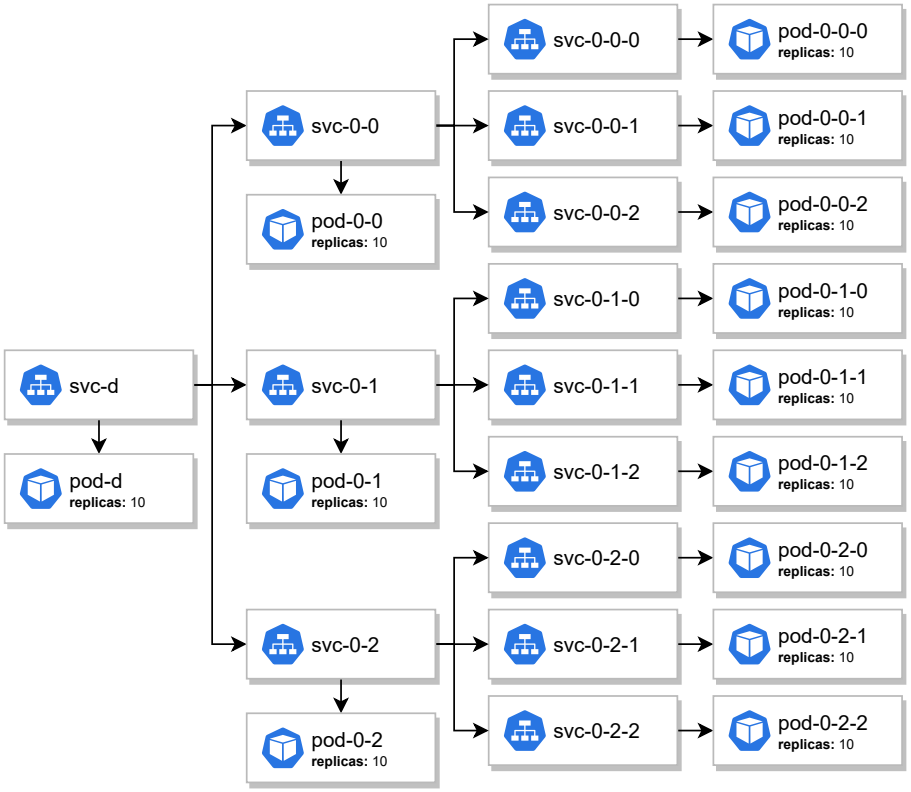


Figure 8.2: Synthetic application scenario

8.2.2 Construction of RCA Model

According to the first evaluation pipeline, raw Kubernetes objects were generated (stage *S1* in Figure 8.1) based on the application scheme depicted in Figure 8.2 and provided to the solution module responsible for building the system object dependency graph (stage *S2*). Figure 8.3 presents the resulting structure.

The obtained dependency graph includes expected Kubernetes object kinds. Its structure correctly maps service dependencies illustrated in Figure 8.2. Additionally, each Service is linked to the correct number of Pod workload units. Due to space constraints, workload units were folded into single nodes with the exact number of Pod objects given in parentheses.

In the second evaluation pipeline, symptom data were generated (stage *S3* in Figure 8.1) for testing the extraction of semantic symptom knowledge and construction of symptom co-occurrence map. Symptom sequence configuration provided to the

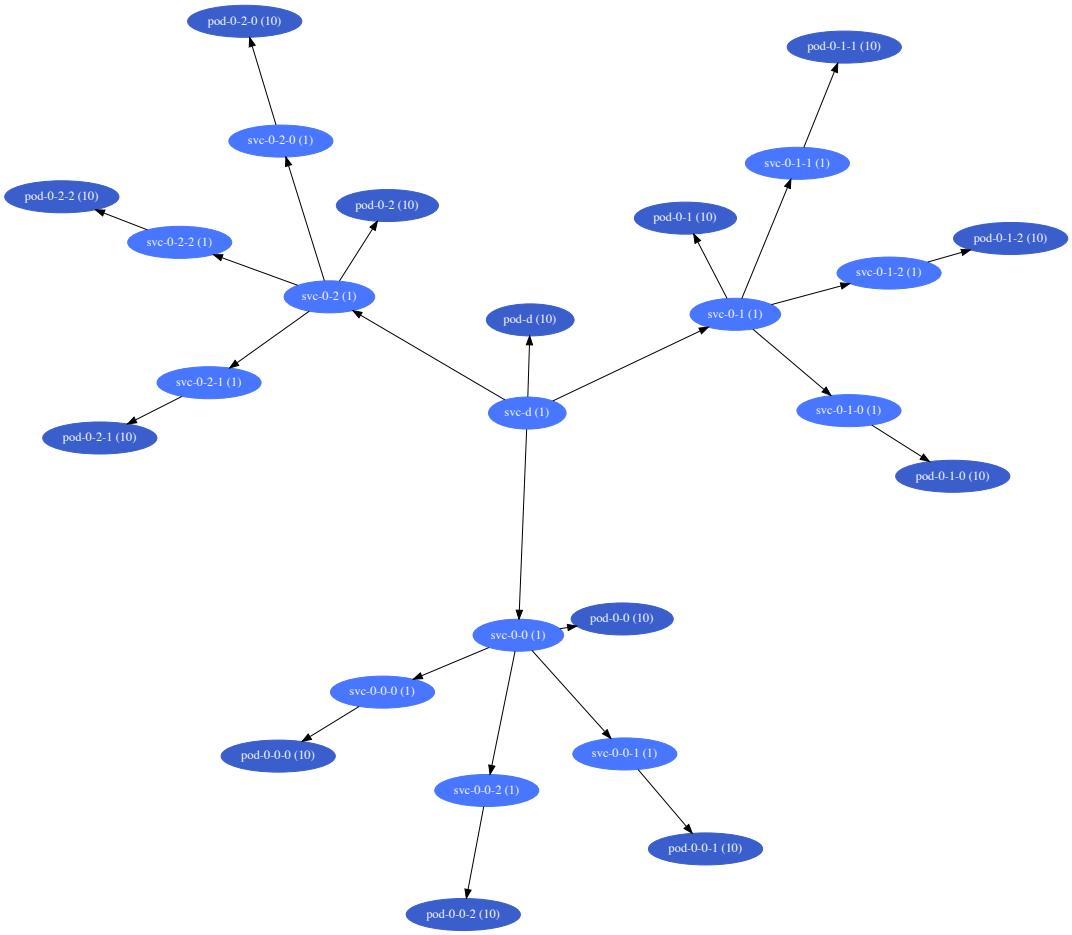


Figure 8.3: System object dependency graph obtained by processing system objects generated synthetically based on reference application scenario illustrated in Figure 8.2

symptom generator is detailed in Table 8.1. The table contains the specification of five fault trajectories representing distinct application failures. Each row corresponds to a single trajectory and contains the number of trajectory repetitions along with the specification of symptom occurrences. The specification of symptom occurrences is split into nested rows corresponding to subsequent symptoms participating in the trajectory. Each symptom is described with its name, source component, offset, and duration. The offset imposes the causal order in the trajectory by indicating symptom time lag in relation to the previous symptom beginning. Further, due to space constraints, numerical indexes in component names were replaced with letters x and y. Both letters correspond to indexes ranging from 0 to 2. For instance, in the case of the first fault trajectory the pattern *svc-0-x-y* expands to the following component names: *svc-0-0-0*, *svc-0-0-1*, *svc-0-0-2*, *svc-0-1-0*, *svc-0-1-1*, *svc-0-1-2*, *svc-0-2-0*, *svc-0-2-1*, *svc-0-2-2*. Each permutation of component indexes was considered, i.e., for each permutation, e.g., $(x = 0; y = 1)$, concrete numeric values were substituted into name patterns. For the assumed range of index

values, the number of permutations is 9, which means that each fault trajectory was simulated for 9 component permutations, thus covering all possible bottom-up symptom propagation paths in the application topology.

Table 8.1: Fault trajectory specification for synthetic symptom data generation

ID	Repetitions	Symptom Occurrences			
		Symptom	**Source component**	**Offset**	**Duration**
1	300	high-cpu	pod-0-x-y	0	120
		high-latency	svc-0-x-y	5	120
		high-latency	svc-0-x	5	120
		high-latency	svc-d	7	120
2	300	app-err-1	pod-0-x-y	0	120
		high-err	svc-0-x-y	5	120
		high-err	svc-0-x	5	120
		high-err	svc-d	7	120
3	300	app-timeout	pod-0-x-y	0	120
		high-latency	svc-0-x-y	5	120
		high-latency	svc-0-x	5	120
		high-latency	svc-d	7	120
4	100	app-err-2	pod-0-0-0	0	120
		app-err-3	pod-0-0-0	20	120
		app-err-4	pod-0-0-0	50	120
5	100	app-err-2	pod-0-0-0	0	120
		app-err-3	pod-0-0-0	20	120
		app-err-5	pod-0-0-0	10	120

Trajectories *1-3* represent situations typical for applications based on microservices architecture where a fault in a low-level component propagates throughout service layers until reaching a user-facing layer. In the first trajectory, the fault is CPU overload in the *pod-0-x-y* component causing increased request latency in services *svc-0-x-y*, *svc-0-x* and *svc-d*. The second trajectory is triggered by an application error in the *pod-0-x-y* component and generates a cascade of request errors across related services. Further, the third trajectory starts with an application timeout in the *pod-0-x-y* component, which, similar to the first trajectory, causes high latency errors in application services. Trajectories *4-5* describe symptom propagation in the boundary of a single component, i.e., *pod-0-0-0*. They represent a series of application errors, e.g., collected from application logs. Both trajectories share root cause symptom {*app-err-2 on pod-0-0-0*}.

Symptoms were generated by recreating fault trajectories according to the number of repetitions and the specification of symptoms occurrence, resulting in 123300 symptom instances. A randomness factor was introduced in symptom generation to make them similar to symptoms occurring in a true system. For each time lag

calculated from the offset given in the symptom specification, random noise was added according to the normal distribution with a mean equal to the symptom offset and a standard deviation of 0.5. Additionally, instances of fault trajectories were placed in time slots randomly distributed over the configured period of one year. Specifically, total fault duration was determined for each fault trajectory specification by adding durations of individual symptoms. Then, the configured period was divided into slots of length equal to the estimated fault duration, and slots were randomly chosen according to the number of fault trajectory repetitions. Finally, symptom generation was performed within randomly selected slots designating beginnings and ends of subsequent fault instances.

Table 8.2: Time-series specification for synthetic symptom data generation

Symptom	**Baseline**	**Jump**	**Noise**
high-cpu	2	4	0.05
high-latency	200	300	0.5

Since the developed solution comprises elements related to time-series analysis, i.e., symptom timestamp correction based on changepoint detection (Section 4.3) or symptom time-series correlation (Section 4.6), time-series data were generated in addition to symptom instances. Table 8.2 outlines generator parameters for symptoms associated with time-series data. Baseline defines the value level when the application functions under normal conditions, jump describes the anomalous value level during application failure, whereas noise describes standard deviation for random value spread according to normal distribution. Time-series data were obtained by generating a number of time-series data points with initial values equal to the baseline. Then, values in the time range corresponding to failure duration were replaced with the jump value. Last, random noise was applied to time-series values.

Generated symptoms were provided to the solution module responsible for constructing the symptom co-occurrence map. Based on the event co-occurrence method, the module computes semantic symptom dependencies (stage *S4* in Figure 8.1) and processes them to build the graph structure representing mined symptom knowledge (stage *S5*).

Due to the high complexity of the resulting co-occurrence map, it is presented in the form of the adjacency matrix shown in Figure 8.4. Columns and rows represent symptom pairs subjected to correlation, whereas matrix cells hold estimated correlation strengths. Red cells indicate strong symptom dependency with a correlation coefficient close to 1, while white cells indicate a weak dependency with a correlation coefficient converging to 0.

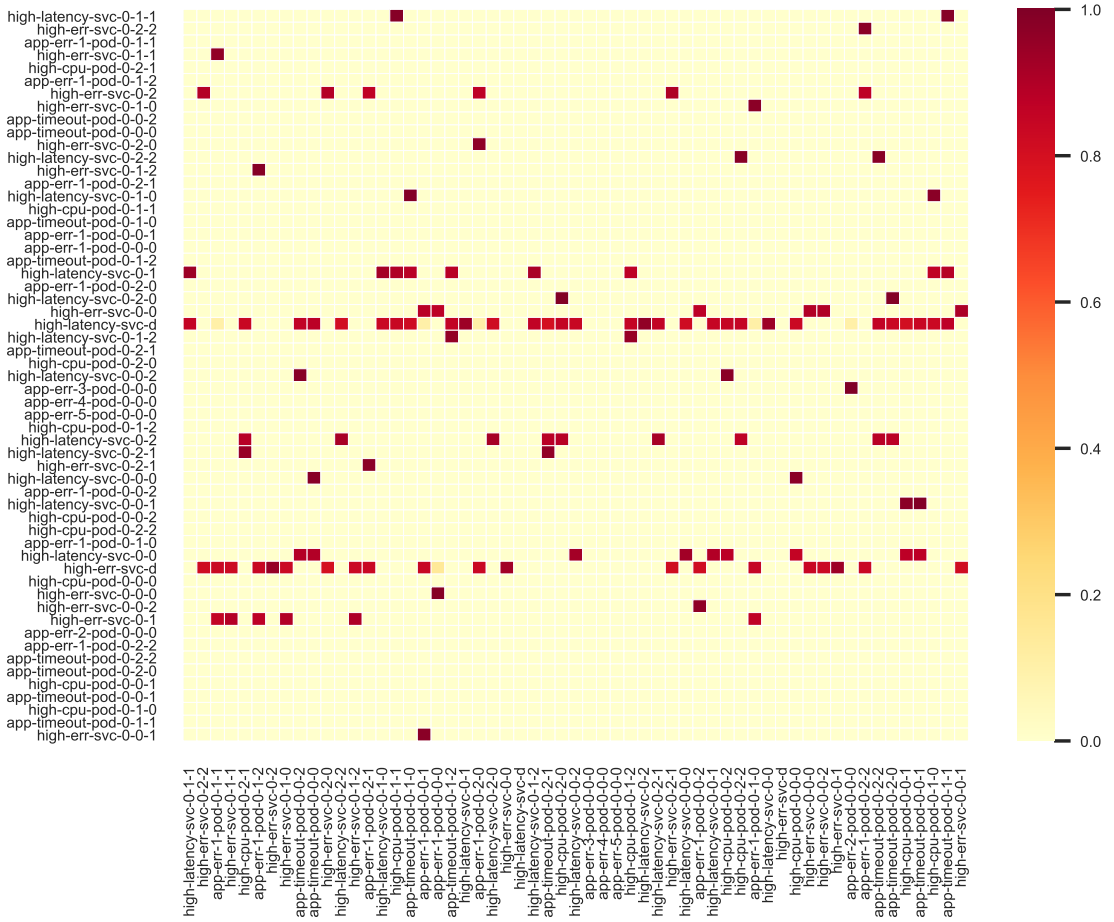


Figure 8.4: Adjacency matrix for symptom co-occurrence map computed from symptoms generated synthetically based on fault trajectory specification detailed in Table 8.1

For the visualization purpose, the two most significant fragments were extracted from the symptom co-occurrence map. In order to extract the fragments, the number of graph edges was reduced by filtering out weak dependencies with values falling below the value threshold of 0.9. Then, the map was searched for the largest subgraphs constituting weak components in the graph structure. Resulting subgraphs are presented in Figure 8.5 and Figure 8.6.

Subgraph illustrated in Figure 8.5 reflects fault trajectories *1* and *3* specified in Table 8.1. The structure correctly reflects the cause-effect sequence of symptom occurrences. Significant strength values for discovered dependencies confirm that the employed symptom co-occurrence method detailed in Section 4.4 is capable of successfully aligning shifted symptom sequences and computing correlation coefficient based on estimated time lag probability distribution. Notably, the acquired co-occurrence map exposes valuable insight into causal symptom dependencies. For instance, it reveals that symptoms warning of high request latency on application services, e.g., {*high-latency on svc-0-0-0*}, may be triggered by high CPU load or application timeout

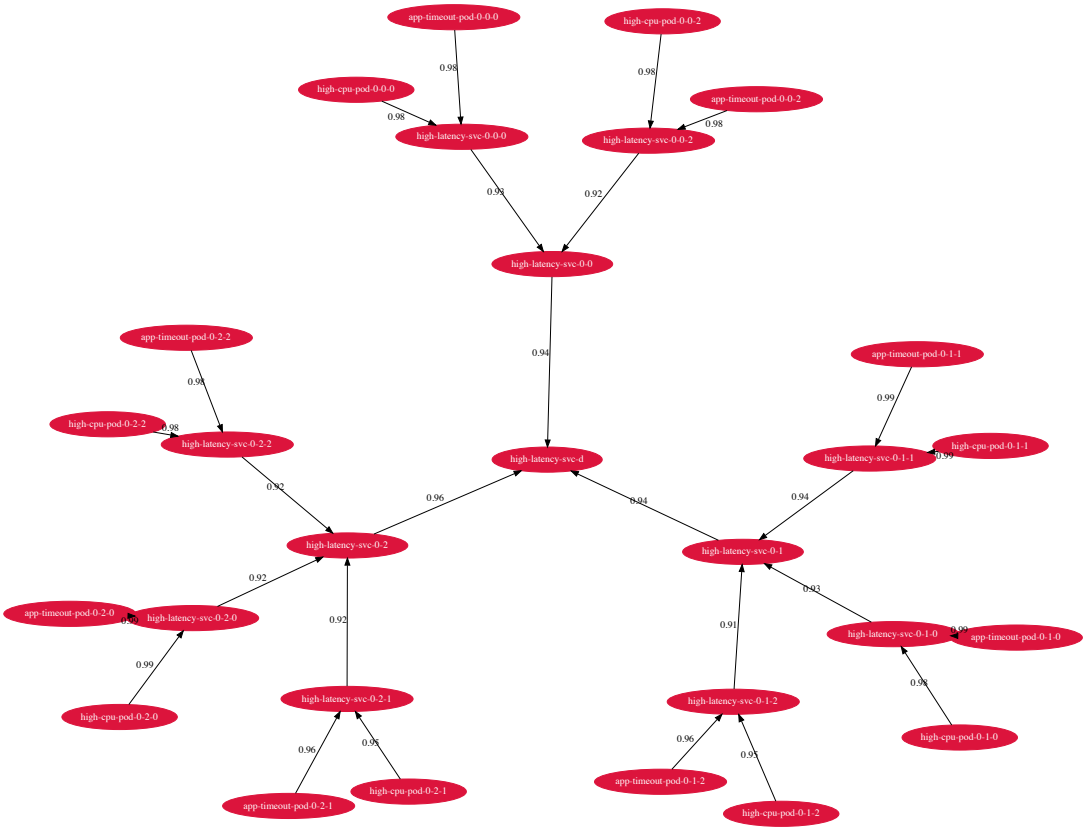


Figure 8.5: Subgraph of reduced symptom co-occurrence map reflecting fault trajectories *1* and *3*

manifested in symptoms {*high-cpu on pod-0-0-0*} and {*app-timeout on pod-0-0-0*}, respectively.

Subgraph illustrated in Figure 8.6 reflects fault trajectory number *2*. Similar to the previous example, the structure accurately replicates the cause-effect sequence of symptoms in the trajectory, where application errors cause request errors in subsequent application services.

Discussed stages of constructing elements of the RCA model constitute a learning stage at which knowledge of system structure and behavior is acquired and semantic symptom dependencies are discovered. The objective of the following evaluation stage is to examine whether obtained model structures can be effectively utilized in the inference algorithm to determine the causes of new failure instances. In a sense, fault diagnosis inverts the RCA model, i.e., in the first stage, simulated faults train the RCA model with symptom semantics, while in the second stage, the trained model is used to infer the causes of new failures. The following sections validate the inference algorithm and the constructed RCA model against failure scenarios covering single and parallel faults.

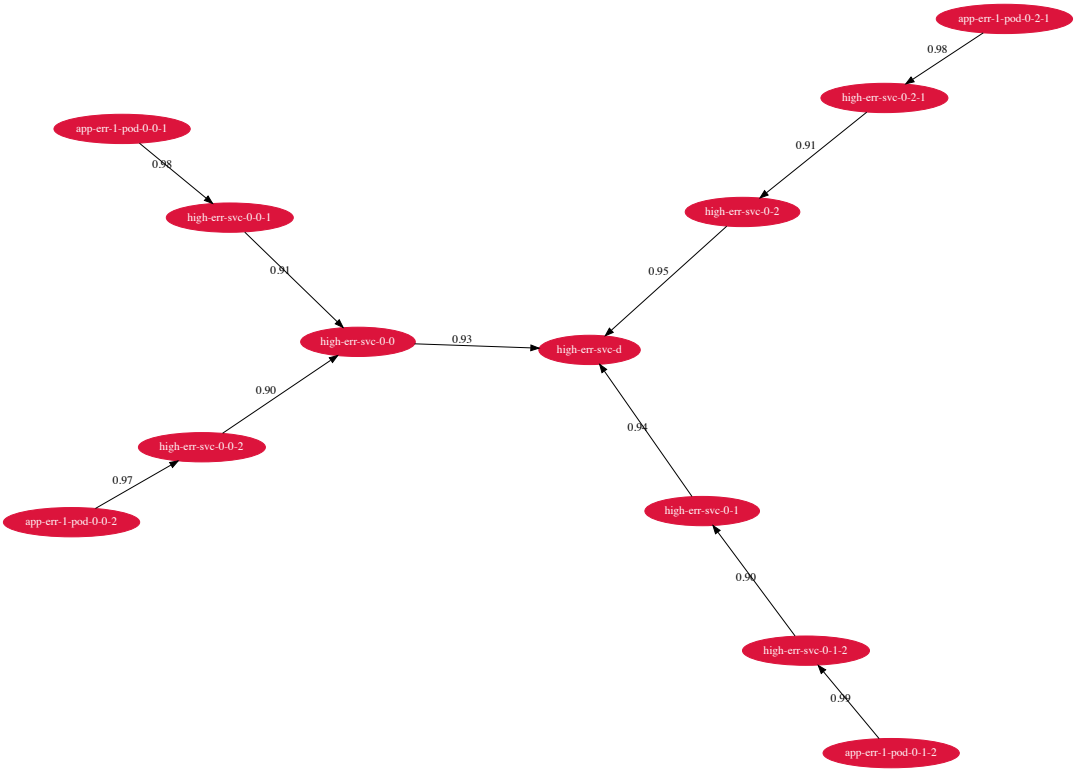


Figure 8.6: Subgraph of reduced symptom co-occurrence map reflecting fault trajectory number *2*

8.2.3 Diagnosing Single Faults

The first scenario simulates a situation in which a single failure occurs within a subset of application components. Its objective is to validate basic root cause analysis functionality and introduce the reader to the RCA evaluation process and discussion of results.

The timeline plot for the simulated scenario is shown in Figure 8.7. In the plot, the horizontal axis describes the time in seconds, while red bars show timelines of individual symptom occurrences. Each bar marks the beginning and end of a given symptom. Moreover, bars describe basic symptom information, including name, source component, offset, and total duration. The offset indicates time lag in relation to the previous symptom beginning.

Considered failure is reproduction of fault trajectory number *1* from the fault trajectory specification (Table 8.1). It reflects a situation where a fault in a low-level component manifests throughout subsequent service layers until reaching the user-facing layer. The considered failure consists of four symptoms emitted by adjacent application components. Starting from the bottom, CPU overload simulated on the *pod-0-0-0* component results in emitting the *high-cpu* symptom, causing a chained

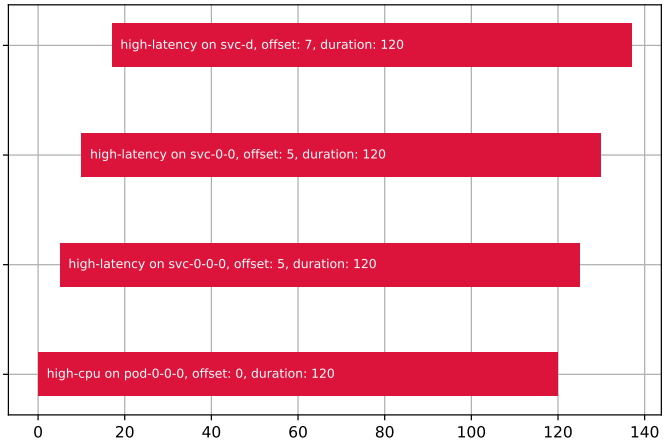


Figure 8.7: Single fault timeline reproducing fault trajectory number *1* from the fault trajectory specification in Table 8.1

latency increase on dependent application services. Subsequently, *high-latency* symptoms arise on services *svc-0-0-0* and *svc-0-0* until reaching the *svc-d* service. Each symptom lasts for approximately 120 seconds. Time lags between symptoms range from 5 to 7 seconds.

Paragraphs below discuss root cause analysis results obtained in simulated failure. First, symptom correlation matrices are presented to point the contribution of individual symptom correlation methods to the overall analysis (stages *S7* to *S10* in Figure 8.1). Next, the consolidation of correlation methods into aggregated symptom correlation matrix is examined with reference to the generated fault view graph structure (stage *S11*). Last, scored and ranked fault trajectories extracted from the fault view graph are summarized and validated (stage *S12*).

Symptom correlation results are presented in the form of correlation matrices produced by analyses employed in the symptom correlation framework (Section 4.2). Matrix cells contain correlation coefficient values in the range $[0; 1]$, where 0 indicates a lack of correlation while 1 indicates the strongest correlation. Coefficient values describe two properties of symptom dependency, i.e., dependency existence and strength. A value greater than 0 confirms dependency existence and quantifies the dependency strength, while a value close to 0 indicates no relationship. Further, the dependency direction, interpreted as *symptom X is caused by symptom Y*, is included in the distribution of symptoms across matrix rows and columns. Matrix rows designate dependency start, whereas matrix columns designate dependency end. Additionally, based on the known specification of simulated failure scenarios, symptoms in matrix rows and columns are grouped by symptom type and arranged according to the valid cause-effect sequence. Symptom order implies forming correlation coefficient clusters that visually help the reader understand symptom dependency

patterns. Importantly, the symptom order does not impact the implementation of the fault diagnosis process where symptom ordering is still arbitral.

Symptom Topological Distance Analysis

Correlation matrix for symptom topological distance analysis (stage *S7* in Figure 8.1) is presented in Figure 8.8. The analysis was introduced in Section 6.2. It calculates coefficient values by searching the system object dependency graph for shortest paths connecting components affected by a failure, i.e., source components. As the path length increases, the coefficient value is lowered geometrically. Thus, it takes the highest value for symptom pairs emitted on the same application component or source components located in direct proximity.

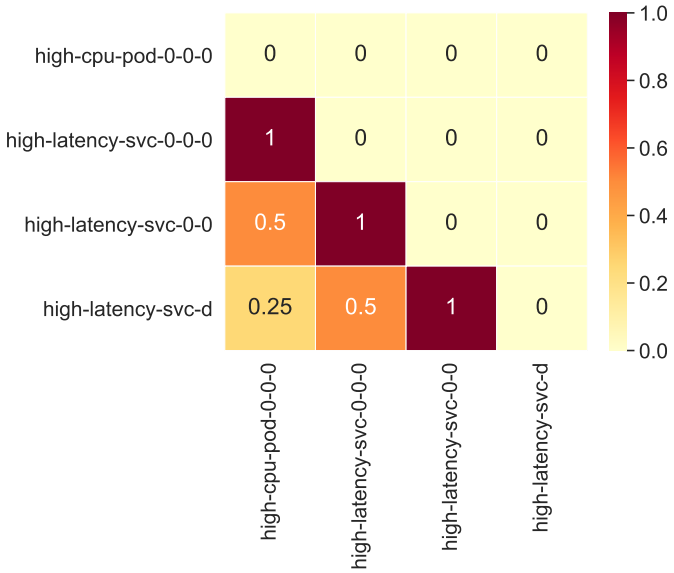


Figure 8.8: Correlation matrix produced in symptom topological distance analysis as part of diagnosis scenario involving single fault

In the considered failure scenario, the correlation matrix indicates the strongest correlation for symptoms with source components that are directly connected in the dependency graph (Figure 8.3). For instance, the coefficient takes the value of 1 for dependency $\{$*high-latency on svc-0-0-0*$\}$ $\rightarrow$ $\{$*high-cpu on pod-0-0-0*$\}$. Further, the coefficient value is weakened for symptoms with source components localized two or three edges apart. For instance, dependency $\{$*high-latency on svc-d*$\}$ $\rightarrow$ $\{$*high-cpu on pod-0-0-0*$\}$, generates coefficient value of 0.25 due to source components located three edges apart.

Symptom Co-occurrence Analysis

Correlation matrix for symptom co-occurrence analysis (stage *S8* in Figure 8.1) is presented in Figure 8.9. The analysis was introduced in Section 6.3. In the analysis, sequences of historical symptom occurrences are matched in the event space to mine semantic symptom dependence. The more consistently instances of two event types co-occurred in the past, the higher the coefficient value. Due to the computational expensiveness of the above method, symptom dependencies are precomputed and stored in the symptom co-occurrence map. During the analysis, dependency strengths are copied from relevant graph edges directly into the correlation matrix.

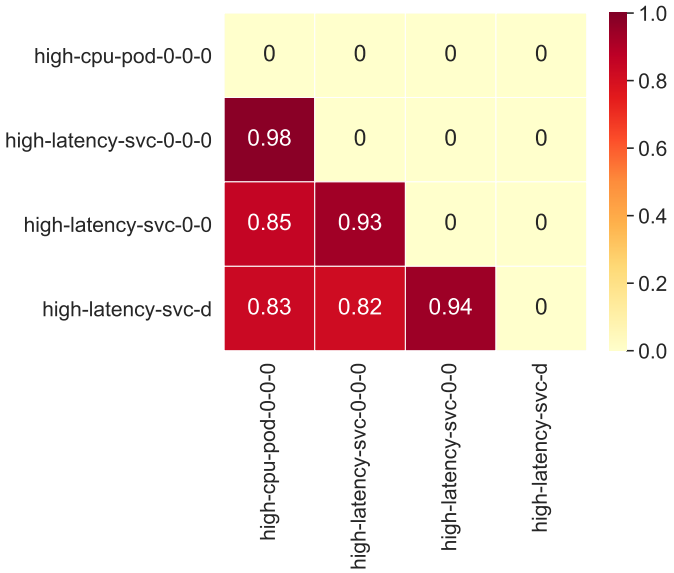


Figure 8.9: Correlation matrix produced in symptom co-occurrence analysis as part of diagnosis scenario involving single fault

The correlation matrix shows strong correlation between all emitted symptoms. Obtained coefficient values are highest for symptoms matching the exact cause-effect order in the simulated scenario, for instance, {*high-latency on svc-0-0-0*} → {*high-cpu on pod-0-0-0*}. According to the symptom co-occurrence map (Figure 8.5), these symptom pairs showed statistically strongest co-occurrence in the past resulting in coefficient values ranging from 0.93 to 0.98. Further, coefficient values are slightly weakened for symptom pairs connected via intermediate symptoms in the cause-effect sequence. Coefficient values for these pairs are still high due to symptom correlation transitivity, i.e., {*svc-d* → *svc-0-0* → *svc-0-0-0*} implies {*svc-d* → *svc-0-0-0*}. Marginal decrease in dependency strength for transitive dependencies results from time lag noise introduced by subsequent symptoms in the trajectory, i.e., the longer the transitive dependency, the stronger the accumulated noise amplified by partial time lag divergence generated by individual symptom pairs.

Symptom Time Lag Analysis

Correlation matrix for symptom time-lag analysis (stage *S9* in Figure 8.1) is presented in Figure 8.10. The analysis was introduced in Section 6.4. In the analysis, time lags calculated for new symptom pairs are compared to corresponding time lag probability distributions retrieved from the symptom co-occurrence map. The more convergent a new time lag is to the distribution mean, the higher the correlation coefficient value.

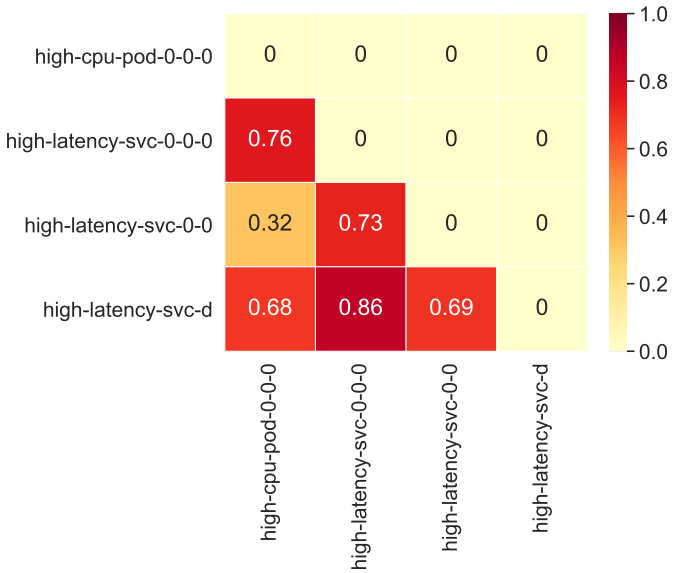


Figure 8.10: Correlation matrix produced in symptom time lag analysis as part of diagnosis scenario involving single fault

The correlation matrix shows a correlation between all symptoms emitted in the simulated fault. Similar to the symptom co-occurrence analysis, the number of discovered dependencies and high coefficient values result from correlation transitivity. However, due to coefficient values calculated based on time lags of new symptom pairs rather than a precomputed dependency strength copied directly from the co-occurrence map and supported by a large volume of historical data, the analysis is affected by temporal noise, which visibly contributes to coefficient value decrease and its fluctuation between 0.32 and 0.76.

Symptom Time-series Analysis

Correlation matrix for symptom time-lag analysis (stage *S10* in Figure 8.1) is presented in Figure 8.11. The analysis was introduced in Section 6.5. In the analysis, Pearson coefficient is calculated for symptom pairs associated with time-series data. The coefficient value converges to 1 if the linear relationship between time-series values is strongest and converges to 0 if there is no linear relationship.

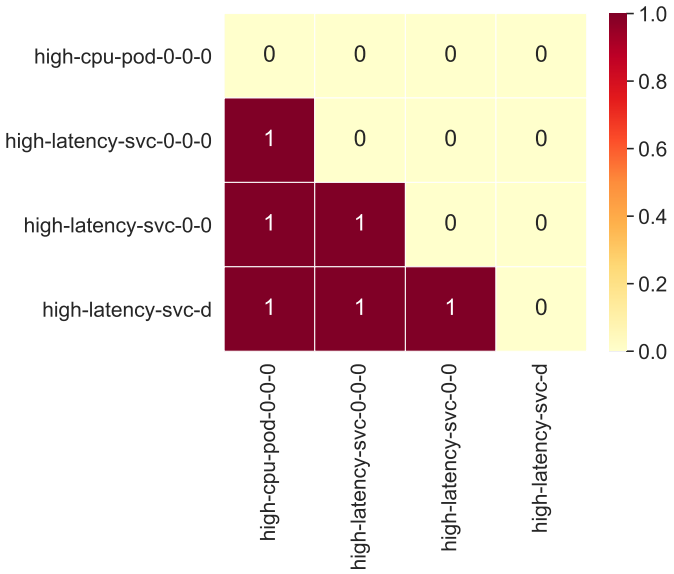


Figure 8.11: Correlation matrix produced in symptom time-series analysis as part of diagnosis scenario involving single fault

According to the time-series specification in Table 8.2, all symptoms participating in the considered failure scenario are associated with time-series data. The coefficient value for each symptom pair is approximately equal to 1, indicating a strong correlation. Similar to other statistical analyses, the number of correlations and high coefficient values result from correlation transitivity, which, analogously to temporal event co-occurrence, translates into time-series trends showing simultaneous reaction to simulated faults.

Construction of Fault View Graph

Further, proceeding to the next stage, correlation matrices from performed symptom correlation analyzes are consolidated into aggregated correlation matrix and translated into the fault view causality graph (stage *S11* in Figure 8.1). The aggregated correlation matrix is computed using the weighted coefficient aggregation strategy detailed in Section 6.6. If obtaining correlation information for a symptom pair using a given correlation method is not feasible, the consolidated coefficient value is decreased proportionally to the method weight. After computation, the aggregated matrix comprises coefficient values representing approximated causal symptom dependencies.

Due to the extensive parameter space, a comprehensive weight study was not considered in this dissertation. Weights of individual symptom correlation analyzes were selected arbitrarily and tuned through observations in preliminary experiments. Selected weights are presented in Table 8.3. The highest weight of 0.5 was assigned

Table 8.3: Weights selected for correlation coefficient aggregation strategy

Symptom analysis	Weight
Topological distance analysis	0.5
Co-occurrence analysis	0.25
Time lag analysis	0.25
Time-series analysis	0

to the topological distance analysis, which constitutes the foundation of the overall analysis by providing deterministic information on the causal dependency between identified symptom sources. Further, as pointed out in Section 4.5, co-occurrence and time lag analyses complement each other in terms of providing accurate information on symptom semantics. Hence, they were assigned equal weights of 0.25. Last, the contribution of time-series analysis was excluded from experiments because, based on prior observations, it showed ineffective in determining causal symptom dependencies and, in many cases, introduced dependency bias, negatively affecting the accuracy of resulting correlations in the aggregated correlation matrix. Nevertheless, results from time-series analysis are discussed to indicate potential method application in other failure scenarios.

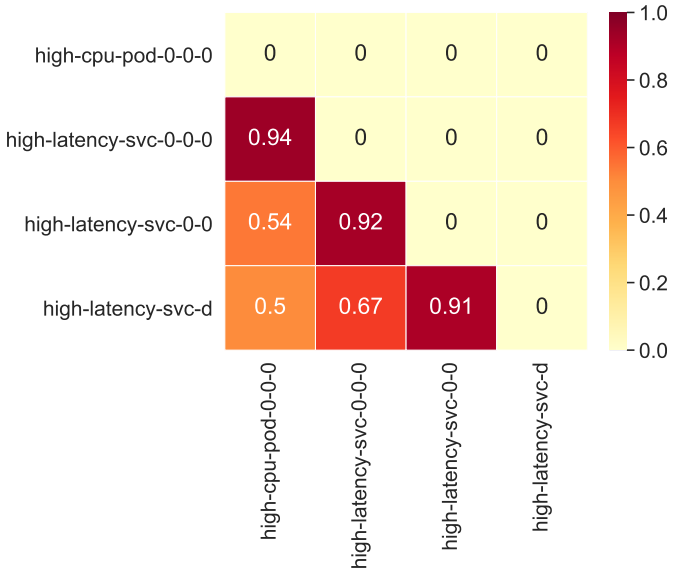


Figure 8.12: Aggregated symptom correlation matrix produced in single fault diagnosis based on coefficient aggregation strategy with weights specified in Table 8.3

Consolidation of presented correlation matrices into aggregated symptom correlation matrix is demonstrated in Figure 8.12. The consolidation reveals the correct fault trajectory and weakens symptom dependencies resulting from correlation transitivity. That is visible in the fault view graph illustrated in Figure 8.13. The correct cause-effect sequence of symptoms, i.e., $\{$*high-latency on svc-d*$\} \rightarrow \{$*high-latency on svc-0-0*$\} \rightarrow \{$*high-latency on svc-0-0-0*$\} \rightarrow \{$*high-cpu on pod-0-0-0*$\}$, is denoted

by strongest dependencies with coefficient values ranging between 0.91 and 0.94. Conversely, invalid causal dependencies such as {*high-latency on svc-d*} → {*high-cpu on pod-0-0-0*}, are significantly weakened. They hold low coefficient values ranging between 0.50 and 0.67, disqualifying derivtive fault trajectories from achieving high scores at the trajectory scoring stage.

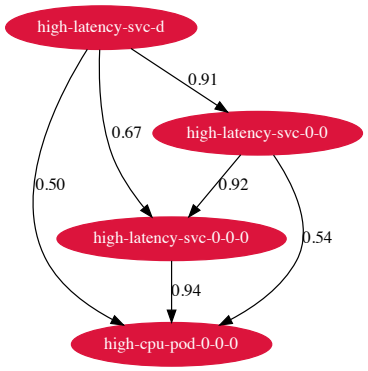


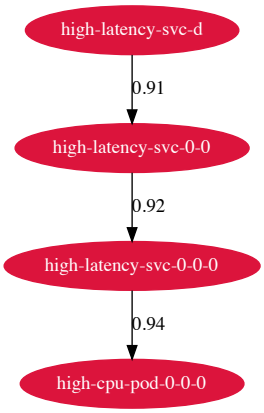
Figure 8.13: Fault view causality graph build from aggregated symptom correlation matrix depicted in Figure 8.12

Fault Trajectory Detection

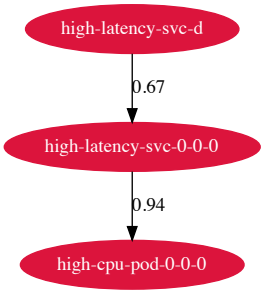
Finally, Figure 8.14 shows the four best fault trajectories obtained at the output of the inference algorithm. The trajectory detection (stage *S12* in Figure 8.1) is based on ranking trajectory paths extracted from the fault view graph considering two criteria: average dependency strength and total trajectory length. Intuitively, fault trajectory detection searches for the strongest and longest paths in the fault view graph. In addition, trajectory paths are constrained by root cause and effect symptoms identified by evaluating degrees of symptom nodes in the fault view graph.

As expected, the highest scored trajectory correctly reflects the cause-effect sequence of symptoms imposed by the simulated failure. All relevant symptoms are present and follow correct causal order resulting in average dependency strength of $\sigma_{strength} = 0.92$ and trajectory length of $\sigma_{length} = 3$. Contrary, subsequent fault trajectories skip intermediate symptoms due to correlation transitivity and achieve a lower σ_{length} score. Moreover, all trajectories maintain correct root cause and effect symptoms thanks to the adopted identification method that inspects the degrees of nodes in the fault view graph.

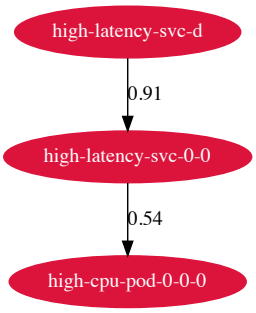
Results obtained in the presented failure scenario show that the proposed synergy of symptom correlation methods allows recreating correct fault trajectories based on symptoms derived from single failures.



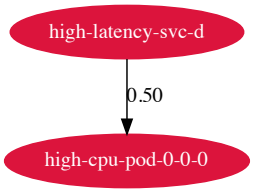
(a) $\sigma_{strength} = 0.92$, $\sigma_{length} = 3$



(b) $\sigma_{strength} = 0.80$, $\sigma_{length} = 2$



(c) $\sigma_{strength} = 0.73$, $\sigma_{length} = 2$



(d) $\sigma_{strength} = 0.50$, $\sigma_{length} = 1$

Figure 8.14: Ranked fault trajectories obtained in single fault diagnosis scenario

8.2.4 Diagnosing Semantically-independent Parallel Faults

The second scenario simulates two parallel faults occurring within the same subset of application components. It recreates fault trajectories *1* and *2* from the fault trajectory specification (Table 8.1). The scenario goal is to check if the inference algorithm can distinguish symptoms originating from independent faults and correctly separates them as distinct fault trajectories. Considered faults are semantically independent, i.e., symptoms across faults are pairwise independent and are manifested in separate root causes and propagation.

The timeline plot for the simulated scenario is shown in Figure 8.15. Failures are marked in two shades of red. The first fault is triggered by an application error in *pod-0-0-0* component reported as *app-err-1* symptom, causing request errors on related application services reported as *high-err* on *svc-0-0-0*, *svc-0-0*, and *svc-d*, subsequently. The second fault is equal to the one considered in the first scenario, i.e., CPU overload on the *pod-0-0-0* component causing high request latency on

subsequent application services. Time lags between symptoms are consistent with those described in the fault trajectory specification and range from 5 to 7 seconds. Each symptom lasts for 120 seconds.

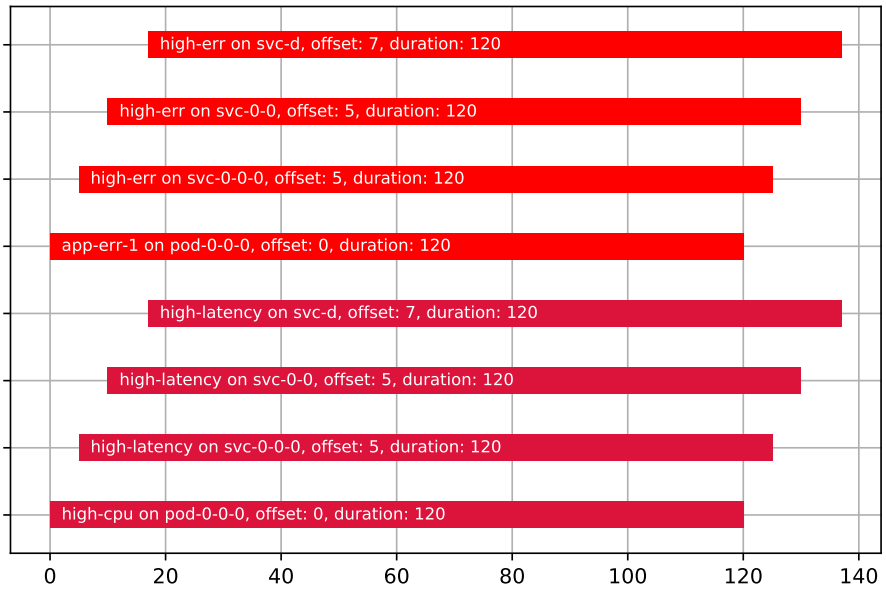


Figure 8.15: Simulated fault timeline

Similar to the previous experiment, the following paragraphs describe symptom correlation results produced by individual symptom correlation methods, discuss the structure of the obtained fault view graph, and validate fault trajectories returned at the algorithm output.

Symptom Topological Distance Analysis

Correlation matrix for symptom topological distance analysis is presented in Figure 8.16. As expected, the correlation coefficient takes the value of 1 for symptoms reported on source components that are directly connected, and it decreases geometrically as the distance between symptom sources increases, taking values ranging between 0.25 and 0.5. Due to the bottom-up nature of fault propagation, causal dependencies overlap with the hierarchy of considered system objects in the dependency graph. Hence, the highest coefficient values partially designate correct fault trajectories. For instance, $\{$*high-latency on svc-d*$\} \rightarrow \{$*high-latency on svc-0-0*$\}$, $\{$*high-latency on svc-0-0*$\} \rightarrow \{$*high-latency on svc-0-0-0*$\}$ and $\{$*high-latency on svc-0-0-0*$\} \rightarrow \{$*high-cpu on pod-0-0-0*$\}$, obtain the highest coefficient value of 1 and thus form the correct fault trajectory: $\{$*high-latency on svc-d*$\} \rightarrow \{$*high-latency on svc-0-0*$\} \rightarrow \{$*high-latency on svc-0-0-0*$\} \rightarrow \{$*high-cpu on pod-0-0-0*$\}$.

However, the correlation view gets significantly distorted by symptom co-occurrence within the same subset of application components. Strong clusters of coefficient values manifest that in upper-right and bottom-left quadrants, for instance, {*high-cpu on pod-0-0-0*} → {*app-err-1 on pod-0-0-0*} takes the value of 1 due to symptoms reported on the same application component, i.e., *pod-0-0-0*. Notably, symptoms *high-cpu* and *app-err-1* are not semantically related in the analyzed scenario.

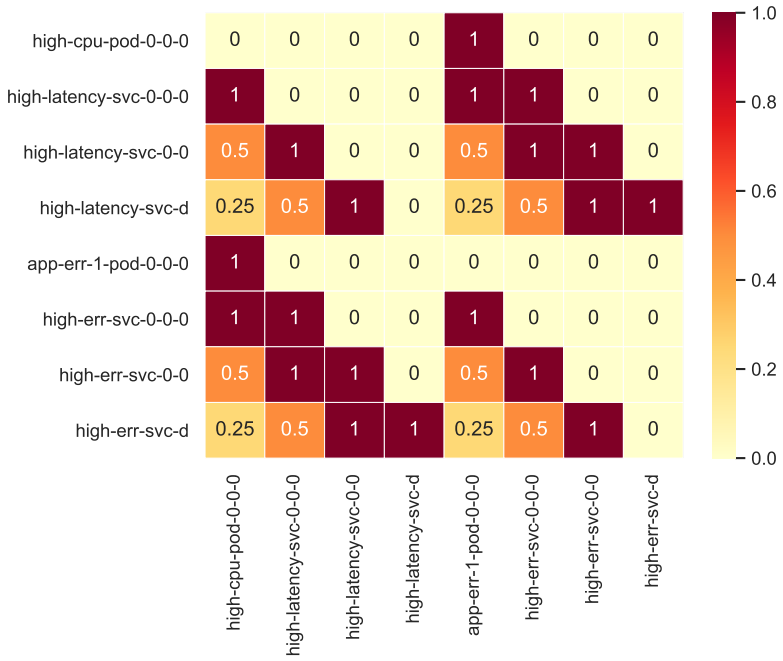


Figure 8.16: Correlation matrix produced in symptom topological distance analysis as part of diagnosis scenario involving parallel semantically-independent faults

Due to considered parallel faults propagating within the same subset of application components, both fault trajectories are reflected identically in the output matrix. Specifically, the upper-left matrix quadrant contains coefficient values for the first failure, while the lower-right quadrant contains values for the second failure. Coefficient clusters localized in these quadrants designate correct fault trajectories.

Symptom Co-occurrencce Analysis

Correlation matrix for symptom co-occurrence analysis is illustrated in Figure 8.17. The matrix shows a strong correlation between all symptom pairs within each failure. Obtained coefficient values are highest and range between 0.90 and 0.99 for symptoms matching the exact cause-effect order in the simulated failure scenario, for instance, {*high-latency on svc-0-0-0*} → {*high-cpu on pod-0-0-0*}. Further, coefficient values are slightly weakened for symptom pairs distant by intermediate symptoms in the cause-effect sequence, taking values in the range between 0.83 and 0.87.

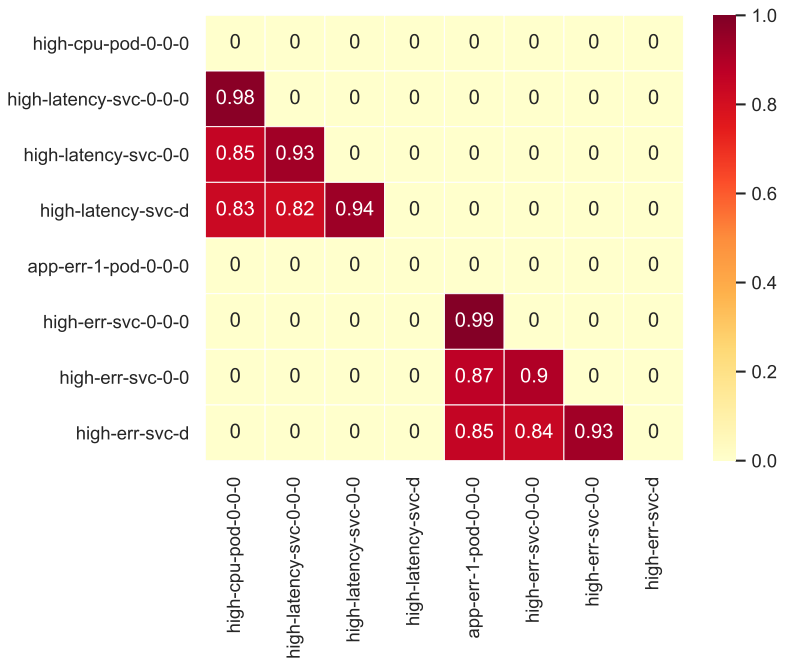


Figure 8.17: Correlation matrix produced in symptom co-occurrence analysis as part of diagnosis scenario involving parallel semantically-independent faults

Apparently, simulated faults are semantically separated, forming distinct coefficient clusters as opposed to the previous analysis. That emphasizes the importance of the co-occurrence method in similar scenarios. While the topological distance analysis could not separate symptoms from parallel faults as they affect the same application region, the co-occurrence analysis allows recognizing the semantic difference between failures based on knowledge of past symptom occurrence.

On the other hand, coefficient values are high for all symptom pairs within a failure due to the correlation transitivity. As a consequence, coefficient values for symptom dependencies that deviate from the causal sequence simulated in the scenario, e.g., $\{$*high-latency on svc-d*$\} \rightarrow \{$*high-cpu on pod-0-0-0*$\}$, differ only by 0.16 from valid causal symptom dependencies. Under certain conditions, temporal noise imposed by the correlation transitivity could be minimized and thus contribute to the equalization of coefficient values. That, in turn, would distort the confidence in determining the existence and direction of symptom dependencies based on the co-occurrence method. Visibly, symptom correlation based on the topological distance can compensate for this effect by providing deterministic dependency information based on the application structure.

Symptom Time Lag Analysis

Correlation matrix for symptom time lag analysis is presented in Figure 8.18. The matrix shows a correlation between symptoms emitted within each failure in the simulated scenario, with coefficient values fluctuating between 0.23 and 0.80. A wide range of coefficient values is the effect of the randomness factor introduced by calculations conducted on time lags of active symptom pairs. The highest coefficient values are assigned to symptom dependencies showing direct semantic dependence. Time lags calculated for such pairs are likely to converge to the time lag distribution mean. Contrary, many intermediate symptoms maximize the chance of accumulating significant temporal noise and shifting the time lag towards outlier boundaries.

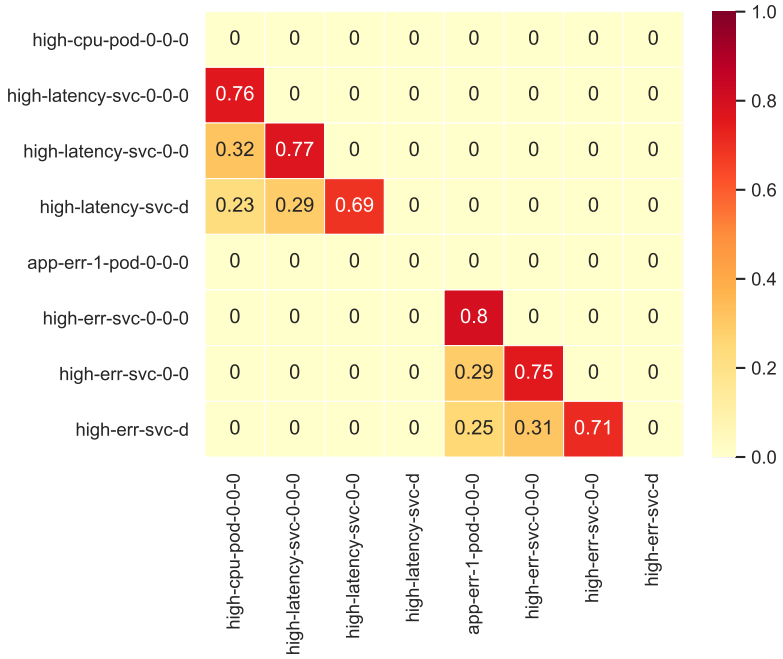


Figure 8.18: Correlation matrix produced in symptom time lag analysis as part of diagnosis scenario involving parallel semantically-independent faults

Similar to the co-occurrence analysis, faults are semantically separated, appearing as distinct coefficient clusters. That is a consequence of both co-occurrence and time lag analysis being based on symptom dependencies stored in the symptom co-occurrence map. In the former case, precomputed symptom dependency strength is copied from graph edges directly into the correlation matrix. In the latter case, coefficient values are calculated by evaluating time lags of new symptom pairs against corresponding time lag probability distributions. Thus, regardless of the failure scenario, correlation matrices for both analyzes contain equally shaped coefficient clusters but result in different coefficient values.

The evaluation scenario confirms that time lag analysis is complementary to the co-occurrence analysis as it allows weakening symptom dependencies resulting from the correlation transitivity.

Symptom Time-Series Analysis

Correlation matrix for symptom time-series analysis is shown in Figure 8.19. According to the time-series specification in Table 8.2, in the considered scenario, time-series data were available for *high-cpu* and *high-latency* symptoms participating in one of two simulated faults. Thus, the analysis forms a single coefficient cluster with all values approximately equal to 1, indicating a strong pairwise correlation between time-series. Like other statistical analyses, discovered symptom dependencies are influenced by correlation transitivity.

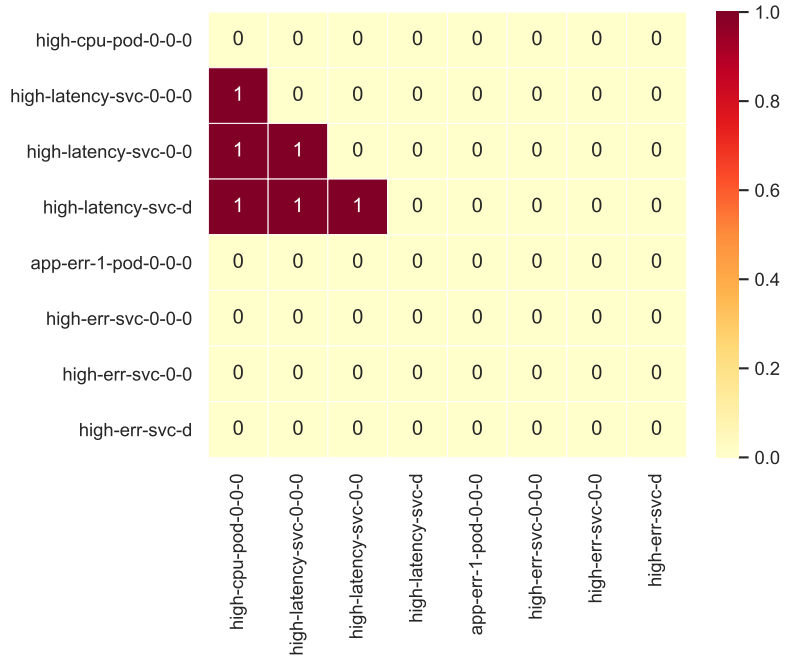


Figure 8.19: Correlation matrix produced in time-series analysis as part of diagnosis scenario involving parallel semantically-independent faults

Construction of Fault View Graph

Consolidation of correlation results into aggregated symptom correlation matrix is demonstrated in Figure 8.20. It is based on the weighted coefficient aggregation strategy with weights specified in Table 8.3. The consolidation exhibits highest coefficient values for symptom dependencies forming correct fault trajectories. The first fault trajectory, i.e., {*high-latency on svc-d*} → {*high-latency on svc-0-0*} → {*high-latency svc-0-0-0*} → {*high-cpu on pod-0-0-0*}, is designated by coefficient values ranging between 0.91 and 0.94, while the second trjectory, i.e., {*high-err on*

svc-d} → {*high-err on svc-0-0*} → {*high-err on svc-0-0-0*} → {*app-err-1 on pod-0-0-0*}, is designated by coefficient values ranging between 0.79 and 0.95. Remaining dependencies resulting from correlation transitivity or symptom co-occurrence within the same application components are appropriately weakened and hold values ranging between 0.39 to 0.65.

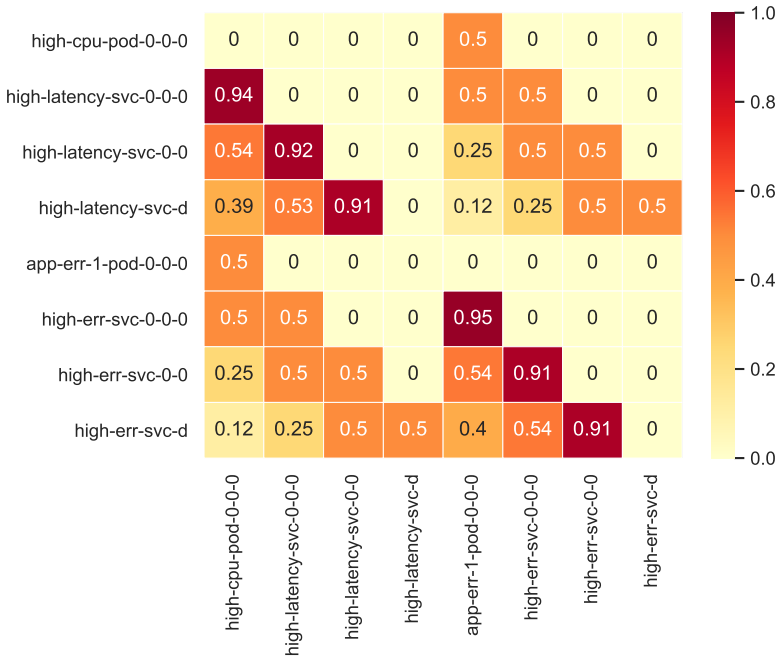


Figure 8.20: Aggregated symptom correlation matrix produced in parallel semantically-dependent faults diagnosis based on coefficient aggregation strategy with weights specified in Table 8.3

The obtained fault view graph illustrated in Figure 8.21 shows a complex structure with far more connections than causal dependencies between active symptoms. Nevertheless, the graph structure allows successfully isolating paths representing correct fault trajectories thanks to adopted trajectory extraction and evaluation methods.

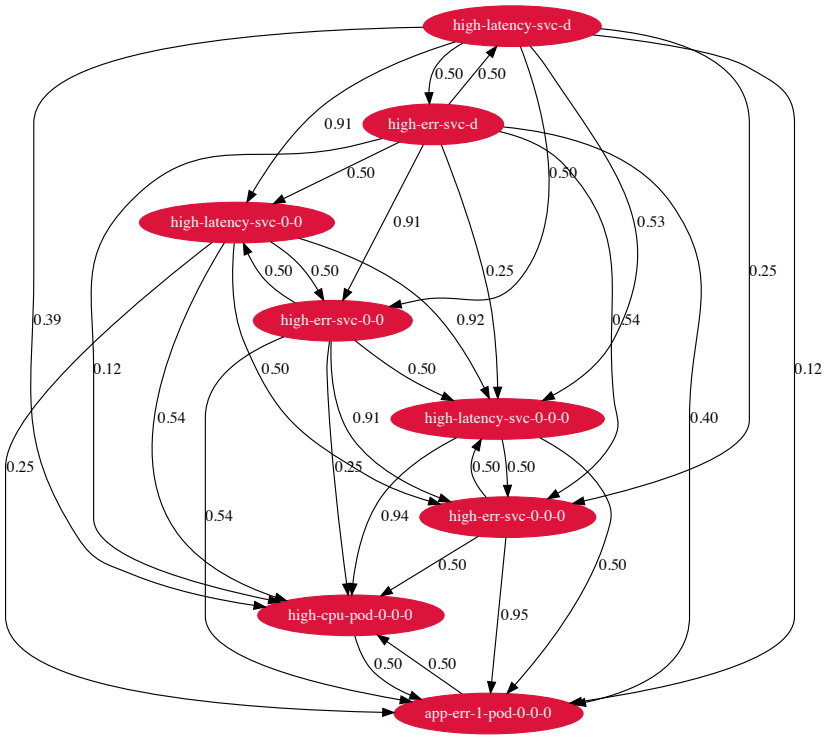
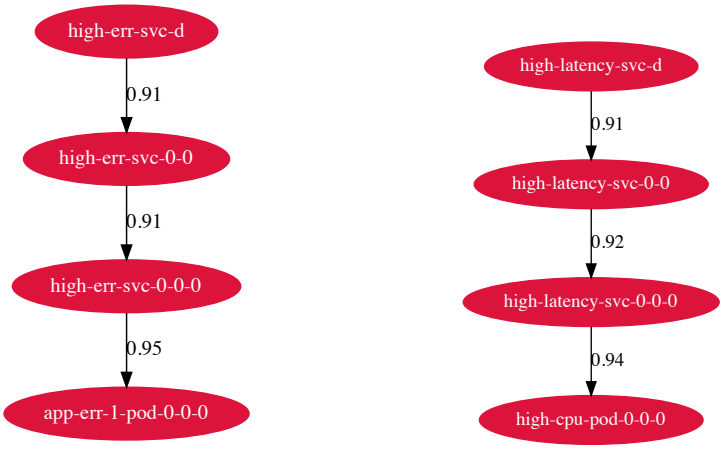


Figure 8.21: Fault view causality graph build from aggregated symptom correlation matrix depicted in Figure 8.20

Fault Trajectory Detection

Two best-scored fault trajectories obtained at the output of the inference algorithm are presented in Figure 8.22. The resulting trajectories accurately reflect simulated faults. All relevant symptoms are present, semantically separated, and follow the correct causal order. Both trajectories are scored equally, with the average dependency strength $\sigma_{strength} = 0.92$ and trajectory length $\sigma_{length} = 3$.

The simulated evaluation scenario confirms the effectiveness of the proposed inference algorithm in isolating parallel semantically-independent faults. Among correlation methods used in the inference, topological distance and co-occurrence analyses made the most significant contribution. Notably, the synergy of correlation results obtained from these methods ensures the expected outcome. Independently, methods suffer correlation disturbance implied by correlation transitivity and symptom co-occurrence within the same application region, making it impossible to assess the valid cause-effect sequence of symptoms in the trajectory.



(a) $\sigma_{strength} = 0.92$, $\sigma_{length} = 3$ **(b)** $\sigma_{strength} = 0.92$, $\sigma_{length} = 3$

Figure 8.22: Ranked fault trajectories obtained in parallel semantically-independent faults diagnosis scenario

8.2.5 Diagnosing Semantically-dependent Parallel Faults

The third scenario simulates two parallel faults emerging within the same subset of application components. Similar to the second scenario, the goal is to check whether the inference algorithm is able to correctly distinguish symptoms activated as part of independent faults and isolate them as distinct fault trajectories. However, unlike the previous scenario, where failures were semantically independent, failures considered in the third scenario are semantically related. Specifically, some symptoms, particularly root cause symptoms, are identical in both failures.

The timeline plot for the simulated failure scenario is shown in Figure 8.23. As before, failures are marked in two shades of red. They simulate fault trajectories number *4* and *5* from the fault trajectory specification detailed in Table 8.1, i.e., propagation of application logs within single application components. In the scenario, symptoms originating from both faults propagate within two application components, i.e., *pod-0-0-0-A* and *pod-0-0-0-B*. According to the presented fragment of the system object dependency graph illustrated in Figure 8.24, these components are not directly connected but implement similar behavior as they are aggregated under the same application service *svc-0-0-0*. Each failure is triggered by an application error, manifested as *app-err-2*, which then causes another error reported as *app-err-3*. Then, the first fault effects in *app-err-4* error, whereas the second fault effects in *app-err-5* error. Moreover, failures are shifted by an interval of 5 seconds. The time lag between symptoms is consistent with the fault trajectory specification and,

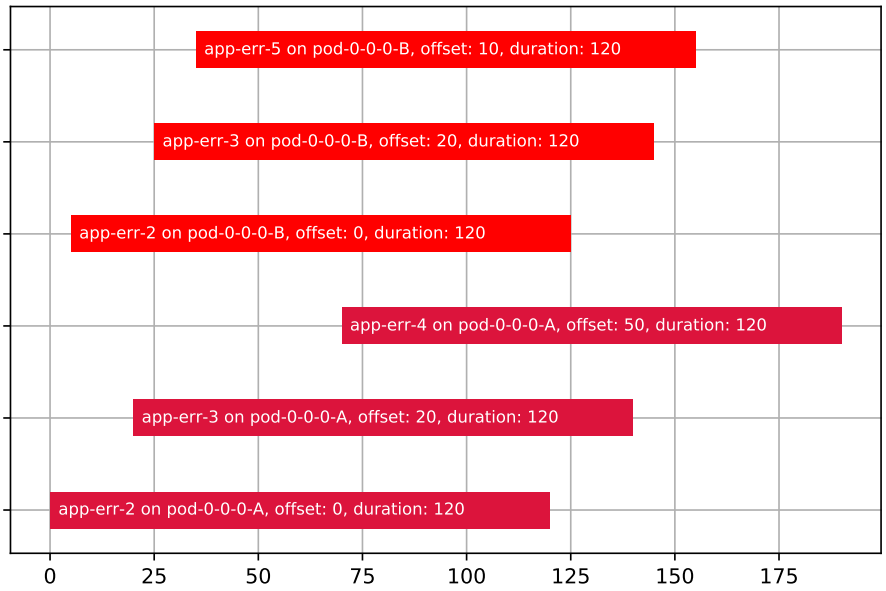


Figure 8.23: Simulated fault timeline

depending on the symptom pair, ranges from 10 to 50 seconds. The duration of each symptom is 120 seconds.

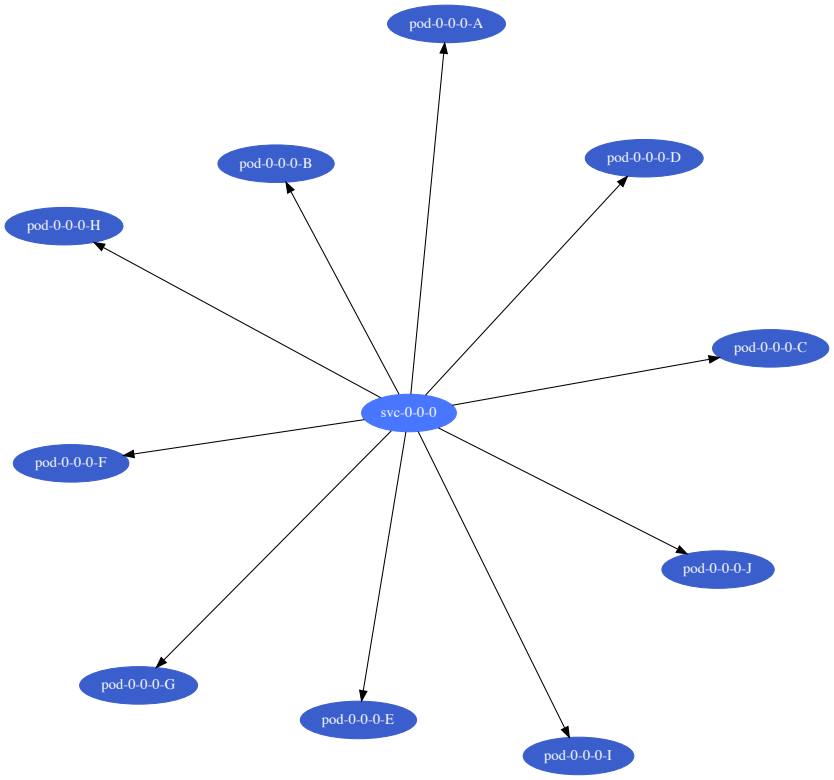


Figure 8.24: Fragment of system object dependency graph illustrating single application service with unfolded Pod instances

Symptom Topological Distance Analysis

Correlation matrix for symptom topological distance analysis is presented in Figure 8.25. The matrix comprises two coefficient value clusters in upper-left and lower-right quadrants, correctly separating simulated faults. However, because symptoms reported in both faults co-occur within the same subset of application components, strong structural correlations were identified for all symptom pairs in each cluster. Moreover, for the same reason, coefficient values for both failures are symmetrical to the matrix diagonal, implying a bidirectional symptom dependency. That disturbs the fault view, making it impossible to determine a valid causal symptom order.

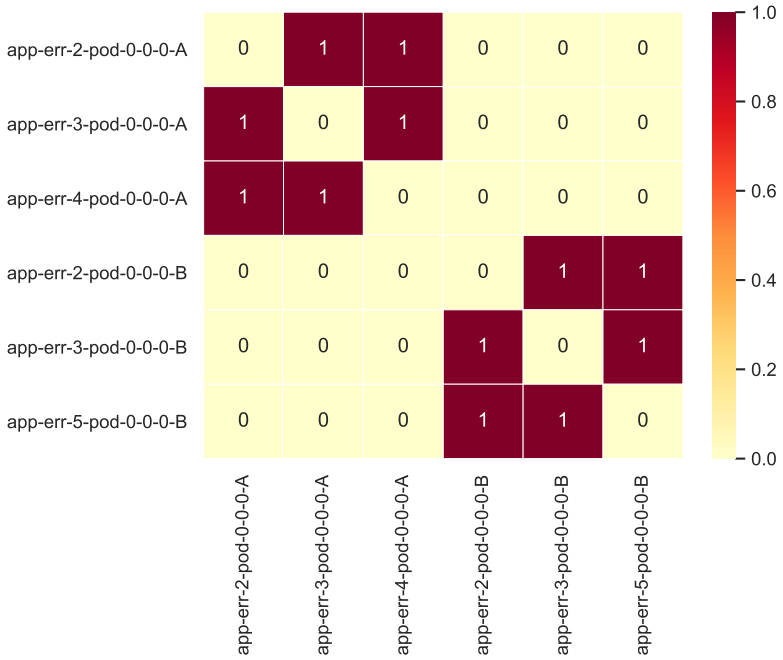


Figure 8.25: Correlation matrix produced in symptom topological distance analysis as part of diagnosis scenario involving parallel semantically-dependent faults

Symptom Co-occurrence Analysis

Correlation matrix for symptom co-occurrence analysis is presented in Figure 8.26. The matrix shows semantic filtering of symptom dependencies which, in contrast to the topological distance analysis, maintains correct symptom dependency direction, i.e., there are no bidirectional dependencies, appearing as coefficient clusters symmetrical to the matrix diagonal. Clusters of coefficient values in upper-left and lower-right matrix quadrants hold valid symptom dependencies, while coefficients in the upper-right and lower-left quadrants result from the semantic symptom interference across two failures. The co-occurrence analysis cannot technically differentiate symptoms across parallel faults because symptoms are pairwise semantically correlated. For instance, {*app-err-3 on pod-0-0-0-A*} is strongly correlated with both {*app-err-2*

on pod-0-0-0-A} and {*app-err-2 on pod-0-0-0-B*} due to consistent co-occurrence of symptoms *app-err-3* and *app-err-2* in the past.

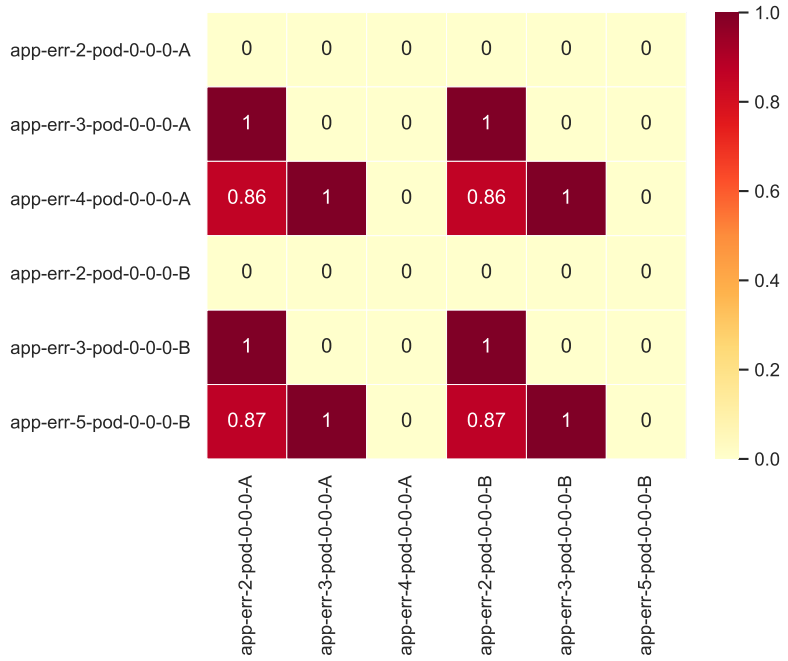


Figure 8.26: Correlation matrix produced in symptom co-occurrence analysis as part of diagnosis scenario involving parallel semantically-dependent faults

Symptom Time Lag Analysis

Correlation matrix for symptom time lag analysis is presented in Figure 8.27. The matrix exhibits coefficient clusters that correctly differentiate symptoms occurring within *pod-0-0-0-A* and *pod-0-0-0-B* components. Valid causal symptom dependencies take coefficient values ranging from 0.74 to 0.77.

Time lag analysis plays a crucial role in diagnosing semantically dependent faults. Complementary to the symptom co-occurrence analysis, it evaluates time lags for observed symptom pairs against corresponding time lag probability distributions stored in the symptom co-occurrence map. Since failures subjected to analysis are shifted by 5 seconds, time lags between symptoms across failures get significantly extended and, consequently, considered outliers. That, in turn, translates into correlation coefficient values converging to zero. As a consequence, symptom dependencies resulting from semantical failure interference are excluded.

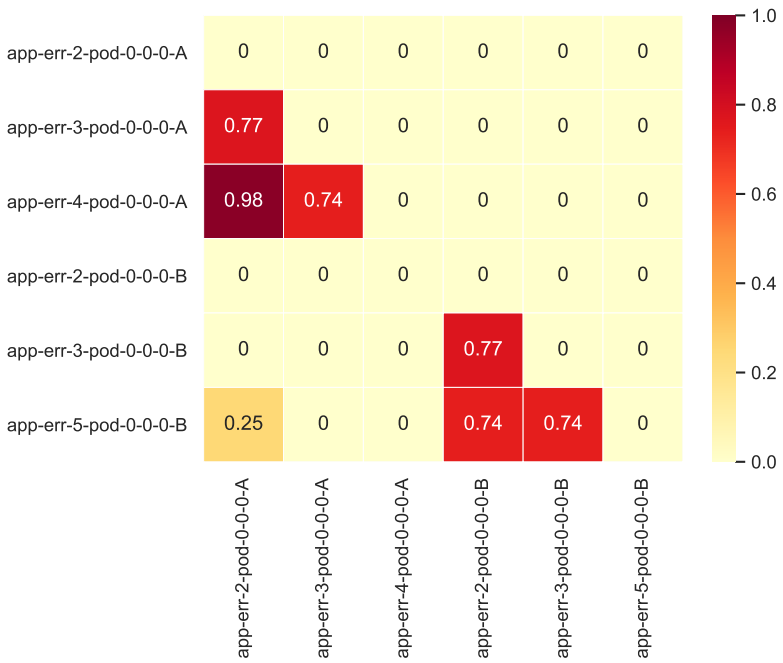


Figure 8.27: Correlation matrix produced in symptom time lag analysis as part of diagnosis scenario involving parallel semantically-dependent faults

Construction of Fault View Graph

Consolidation of correlation matrices into aggregated symptom correlation matrix is demonstrated in Figure 8.28 . It is based on the weighted coefficient aggregation strategy with weights specified in Table 8.3. The consolidation exhibits correct fault trajectories in the fault view graph, depicted in Figure 8.29. The highest coefficient values are assigned to dependencies forming correct fault trajectories. The first fault trajectory, i.e., $\{$*app-err-2 on pod-0-0-0-A*$\} \rightarrow \{$*app-err-3 on pod-0-0-0-A*$\} \rightarrow \{$*app-err-4 on pod-0-0-0-A*$\}$, is designated by coefficient values ranging between 0.94 and 0.96, while the second fault trajectory, i.e., $\{$*app-err-2 on pod-0-0-0-B*$\} \rightarrow \{$*app-err-3 on pod-0-0-0-B*$\} \rightarrow \{$*app-err-5 on pod-0-0-0-B*$\}$, is designated by coefficient values ranging between 0.90 and 0.96. Remaining dependencies resulting from semantic failure interference or symptom co-occurrence within the same application region are appropriately weakened and hold values ranging between 0.22 and 0.50.

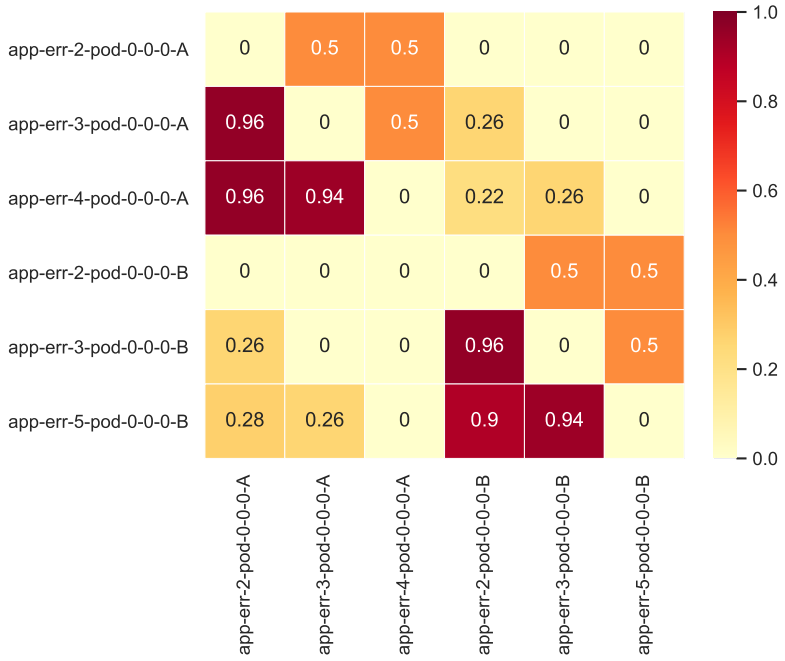


Figure 8.28: Aggregated symptom correlation matrix produced in parallel semantically-independent faults diagnosis based on coefficient aggregation strategy with weights specified in Table 8.3

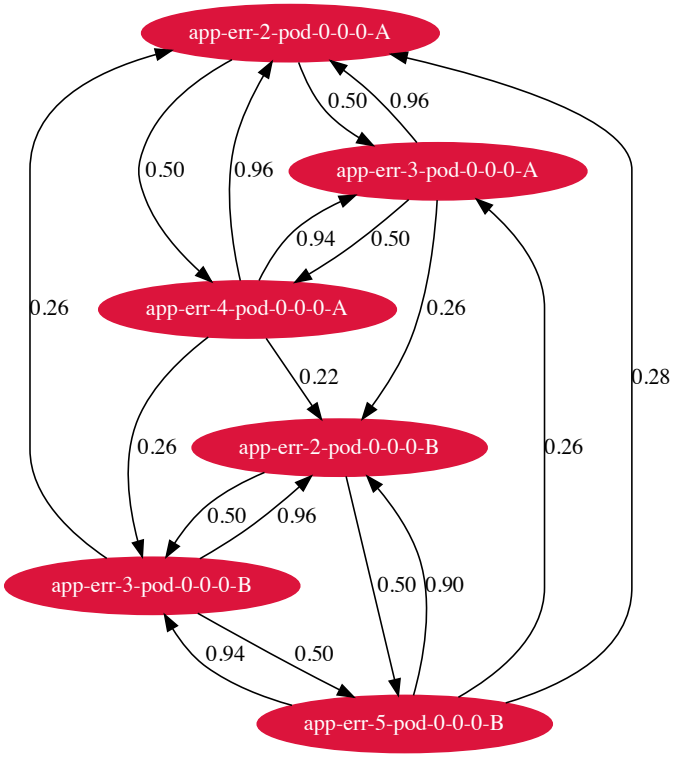
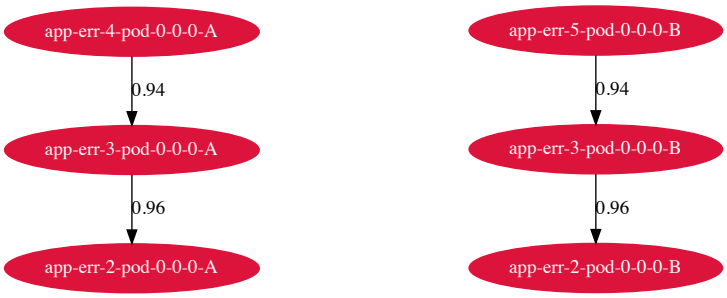


Figure 8.29: Fault view causality graph build from aggregated symptom correlation matrix depicted in Figure 8.28

Fault Trajectory Detection

Best-scored fault trajectories obtained at the output of the inference algorithm are presented in Figure 8.30. The first two trajectories correctly reflect simulated faults, including relevant symptoms and maintaining their causal order, whereas subsequent trajectories, although they retain correct root cause and effect symptoms, miss intermediate symptom *app-err-3*. Due to the correlation transitivity, the third trajectory indicates the best average strength score of all trajectories with $\sigma_{strength} = 0.96$. However, thanks to bucketing fault trajectories by similar strength as explained in Section 6.9, marginal score difference with respect to other trajectories was disregarded, and the trajectory length became the decisive criterion. As a result, correct fault trajectories with average dependency strength $\sigma_{strength} = 0.95$ and trajectory length $\sigma_{length} = 2$ presented greater likelihood.



(a) $\sigma_{strength} = 0.95$, $\sigma_{length} = 2$ **(b)** $\sigma_{strength} = 0.95$, $\sigma_{length} = 2$

(c) $\sigma_{strength} = 0.96$, $\sigma_{length} = 1$ **(d)** $\sigma_{strength} = 0.90$, $\sigma_{length} = 1$

Figure 8.30: Ranked fault trajectories obtained in parallel semantically-dependent faults diagnosis scenario

Conclusively, the last scenario confirms the solution effectiveness in determining trajectories of parallel semantically-dependent faults. Differentiating symptoms across failures was implemented mainly by topological distance and time lag analyzes. In addition, bucketing trajectories by similar strength scores took a significant part in ranking them in the final algorithm stage.

8.3 Functional Evaluation on Live System Data

The first phase of the functional evaluation confirmed solution correctness for system structure and failure scenarios obtained from generated synthetic data. The use of synthetic data was crucial to adequately validate assumptions made in the dissertation and initially test the solution against specific faults that would be difficult to reproduce and examine in a real system. Furthermore, synthetic data facilitated the analysis due to the employed statistical methods requiring a large volume of system telemetry data to produce accurate results.

The synthetic data approach confirmed key thesis assumptions and justified further experimentation. However, although the synthetic data were generated as accurately as possible to represent a live system, taking into account elements of randomness, the synthetic data does not establish sufficient grounds for stating that the solution would operate successfully in real use cases.

Therefore, following the first experimental phase, the second phase tests the developed RCA solution against an application implementing real business use cases, deployed within a live cloud infrastructure, and integrated with concrete platform services. The phase goal is to check whether incorporating system operation aspects omitted in the first evaluation phase affects the quality of root cause analysis and whether adopted statistical methods have sufficient tolerance for the non-deterministic system behavior. Moreover, the phase studies the synergy of available platform services, mainly observability.., and orchestration tools, in constructing a holistic system view and examines the impact of telemetry sampling and outlier issues on diagnosis results.

8.3.1 Test Environment Setup

In order to ensure the appropriate quality of solution evaluation, a test environment was built with the use of hardware and software of enterprise quality. The hardware layer was realized by four Cisco UCS servers belonging to the Cisco Unified Computing System (UCS) family - a server platform for data centers often used in production-grade systems. Each server had the following hardware parameters: Intel Xeon E5-2680 CPU 2.70GHz with 2 sockets, 8 processor cores per socket, 32 logical processors; 256 GB of memory; LSI Logic MRSASRoMB-4i Serial SCSI 560 GB disk. In addition, all servers were connected to the 1 Gbps network.

For the evaluation purpose, 16 virtual machines (VMs) were provisioned within the allocated physical servers, i.e., four virtual machines per physical server. KVM

was used as a hypervisor along with libvirt for virtualization management. Virtual machines were installed with the Linux Centos 7 operating system and contextualized using Kickstart. Each virtual machine had the following hardware specification: CPU passthrough with 32 virtual sockets; 56 GB of memory; 125 GB of disk space. All virtual machines were connected to the physical network via a virtual bridge.

Within the provisioned pool of virtual machines, a Kubernetes 1.17 cluster was created, consisting of 2 master nodes and 14 worker nodes. Nodes were named with subsequent numbers, i.e., *node1*, *node2*, ..., *node16* where *node1* and *node2* fulfilled the master role while the remaining ones acted as worker nodes. The cluster installation was performed using the Kubespray installation utility, which, based on the cluster specification in YAML format, automated the installation on target host machines by orderly executing specialized Ansible roles.

The logical organization of worker nodes is summarized in Figure 8.31. It presents the placement of test component workloads across designated worker groups and worker assignments to virtual machines and bare-metal servers. Complementary, Table 8.4 and Table 8.5 specify versions and resource limits of individual platform and application components installed in the test environment.

Worker nodes were split into two functional groups designated by Kubernetes labels. First 6 worker nodes, i.e., *node3*, ..., *node8*, were labeled with *exp-control*, while remaining 8 worker nodes, i.e., *node9*, ..., *node16*, were labeled with *exp-subject*. The former group hosted workloads responsible for experiment execution and observation, i.e., ensuring orchestration of test application workloads, triggering and supervising fault simulations, and collecting telemetry data required for root cause analysis (monitoring and logging data). The latter group was dedicated to hosting application workloads subjected to fault simulation. Primarily, the functional differentiation of worker nodes was implemented to prevent the impact of simulated faults on experiment control and the efficiency of experiment observation.

Following the Kubernetes cluster installation, the environment was extended with observability add-ons to collect monitoring and logging data related to the Kubernetes cluster status and its workloads. A basic Prometheus stack was installed as a centralized monitoring system, comprising Prometheus Server, Node Exporter, Kube State Metrics, Alertmanager, and Grafana. Prometheus acts as a metric collector pulling metric data from configured scrape endpoints at uniform time intervals. Node Exporter running as a DaemonSet, i.e., a distinct Pod on each Kubernetes node, denoted as *NE* in Figure 8.31, exposes a scrape endpoint with metrics related to the operating system and hardware resource consumption on a given node. Kube State Metrics provides another Prometheus exporter that listens to the Kubernetes

Table 8.4: Component versions used in test environment

Component	Version
Kubespray	2.12.10
Ansible	2.7.16.
Kube API	1.17.5
Kube scheduler	
Kube controller manager	
Core DNS	1.6.0
NFS provisioner	3.1.0
Istio	1.11.4
Kiali	1.38.0
Prometheus Operator	0.44.0
Prometheus Server	2.22.1
Alertmanager	0.21.0
Node Exporter	1.0.1
Kube State Metrics	1.9.7
Grafana	7.2.1
Elasticsearch	7.15.0
Filebeat	
Kibana	
Chaos Mesh	1.0.3
Percona MongoDB Operator	1.9.0
Percona MongoDB Server	4.4.6
ArangoDB	3.6.0

API Server and generates metrics related to the status of Kubernetes objects (Pods, Deployments). Alertmanager receives alerts detected by the Prometheus Server based on alert rule expressions, deduplicates them, and pushes notifications to defined observers. Finally, Grafana constitutes the monitoring dashboard, which visualizes metric data collected by Prometheus. The component stack was installed using Prometheus Operator and Helm.

As a logging solution, the EFK stack was installed consisting of three components, namely Elasticsearch, Filebeat, and Kibana. Elasticsearch is a document database with a full-text search engine based on Lucene. Similar to Prometheus Server, it acts as a centralized data aggregator. However, unlike Prometheus, which utilizes the pull approach to scrape metric data from configured endpoints, Elasticsearch employs the push approach, thus ingesting log data transmitted by external entities (e.g., Filebeat or Fluentd). In the developed test environment, Elasticsearch was provisioned as a cluster comprising 3 master nodes, 3 ingest nodes, and 5 data nodes. Master nodes are responsible for cluster-wide operations such as managing indexes, tracking cluster topology, and deciding Shard allocation. Ingest nodes execute pre-processing data pipelines. Data nodes store indexed documents and

Table 8.5: Component resource limits used in test environment

Component	CPU [vCPU]		Memory [GB]	
	Requests	**Limits**	**Requests**	**Limits**
Kube API	Limits unset			
Kube scheduler	Limits unset			
Kube controller manager	Limits unset			
Core DNS	Limits unset			
NFS provisioner	0.1	0.5	0.1	0.3
Istiod	1	2	1	1
Istio Ingress Gateway	1	2	1	1
Istio Proxy	0.5	2	0.1	1
Kiali	1	1	0.5	0.5
Prometheus Operator	0.1	1	0.1	0.5
Prometheus Server	4	4	8	8
Alertmanager	1	1	1	1
Node Exporter	1	2	1	2
Kube State Metrics	1	2	1	2
Grafana	1	1	1	1
Elasticsearch Client Node	8	16	16	16
Elasticsearch Data Node	16	32	32	32
Elasticsearch Master Node	4	8	8	8
Filebeat	4	8	4	8
Kibana	1	2	1	2
Elastalert	1	1	1	1
Chaos Mesh	1	2	1	1
Percona MongoDB Operator	0.1	1	0.1	0.5
Percona MongoDB Mongos	8	8	8	8
Percona MongoDB Config Servers	4	4	4	4
Percona MongoDB Shard	8	8	8	8
Solution prototype (core)	1	2	1	2
ArangoDB	1	2	1	2

handle data-related operations, including CRUD, search, and aggregations. Further, Filebeat is responsible for consuming log data populated by containers running within Pods, parsing and enhancing log entries, and transporting them directly to Elasticsearch. Similar to Node Exporter in the Prometheus stack, Filebeat runs as a DaemonSet, denoted as *FB* in Figure 8.31, to forward log data from containers placed on each Kubernetes node. Finally, Kibana constitutes a GUI for viewing and searching documents indexed in Elasticsearch.

In addition, to make the test environment more similar to a modern application runtime environment, the Kubernetes cluster was extended with the Istio service mesh. Istio builds a dedicated infrastructure layer that facilitates communication between services deployed as part of large microservices applications. It injects an Envoy proxy

container into each application Pod to intercept ingress and egress traffic transparently. As a result, it enables flexible traffic control, rich observability, and enhanced security. Additionally, Istio provides an advanced control plane encapsulated in the Istiod component that, based on high-level Custom Resource manifests describing the desired traffic scheme, produces an Envoy-compatible configuration and distributes it to registered proxies. Istiod, together with the Istio Ingress Gateway, which provides L7 service mesh load balancing, were installed within the *exp-control* group of worker nodes. In turn, Istio Proxy instances launched alongside application containers were scheduled within the *exp-subject* group of worker nodes. The entire Istio deployment was installed using the istioctl installation utility.

Chaos Mesh was employed to simulate faults in the test application. By design, the tool is a framework for chaos engineering in Kubernetes, providing Custom Resources for defining faults related to hardware stress, network anomaly, and Kubernetes workload issues. Chaos Mesh consists of two components, i.e., Chaos Controller Manager and Chaos Daemon. The former reconciles Custom Resource objects by orchestrating chaos experiments across Chaos Daemon instances deployed on Kubernetes nodes. The latter injects requested faults on target nodes and workloads using low-level utilities such as stress-ng or system network rules. Chaos Controller Manager was installed on the *exp-control* group of worker nodes, whereas Chaos Daemon was installed as a DaemonSet, denoted as *CD* in Figure 8.31.

Finally, the solution prototype, detailed in Chapter 7, was installed as an evaluation subject, implementing elements of the solution concept proposed in this dissertation. The prototype comprises a core component responsible for constructing RCA model structures based on information retrieved from platform components. It persists structures of RCA model in multi-model ArangoDB database. In addition, the core component performs a complete root cause analysis in reaction to ingested symptoms. Moreover, the project provides a React-based UI component that allows visualizing RCA model structures and inspecting results returned by the inference algorithm. The project was installed in the *exp-control* group of worker nodes.

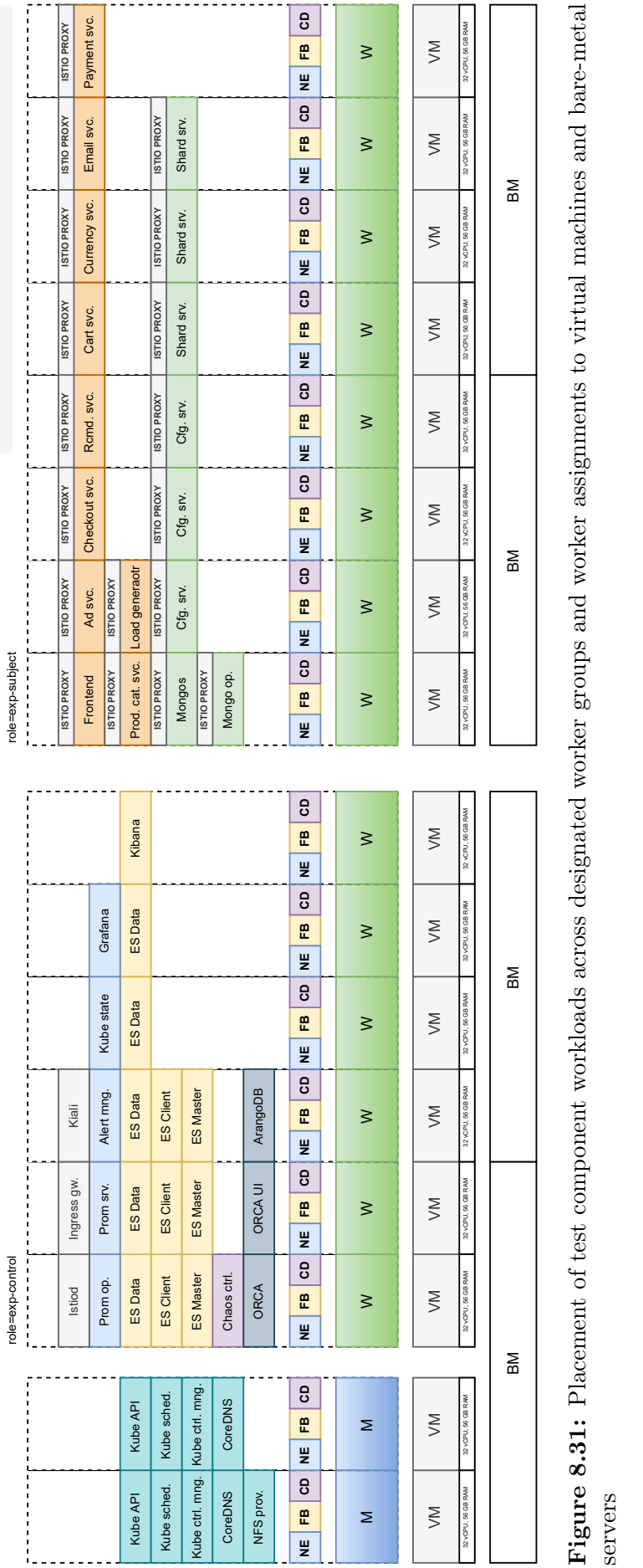


Figure 8.31: Placement of test component workloads across designated worker groups and worker assignments to virtual machines and bare-metal servers

8.3.2 Application Scenario

Test application considered during the second experimental phase is presented in Figure 8.32. The application is microservices demo named *Online Boutique* [1]. It implements an e-commerce web interface where users can browse products, add them to the cart, and finalize the order, including payment and delivery options. The application is built of 10 microservices, namely *frontend*, *ad*, *checkout*, *recommendation*, *cart*, *currency*, *email*, *payment*, *product catalog* and *shipping*. The *frontend* service constitutes application entry point exposing a web interface to the end-user. Underneath, it communicates with other services to aggregate information or perform operations on user demand. The roles of individual application services are outlined in Table 8.6.

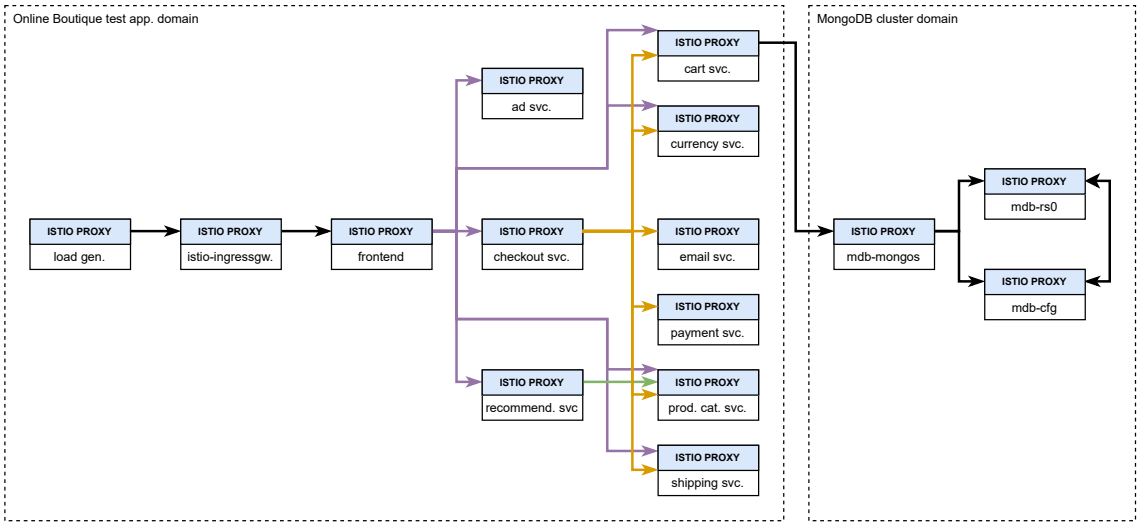


Figure 8.32: Reference test application architecture for functional solution evaluation on live system data

Microservices of the *Online Boutique* application are implemented using polyglot programming languages, including Go, Python, Java, and C#. Internally, they communicate using the gRPC protocol except for the *frontend* service, which exposes an HTTP interface for seamless web browser access. In addition, each application component is fully containerized and prepared for deployment within Kubernetes and Istio service mesh.

Additionally, the application utilizes the MongoDB document database as a cart store. Initially, the cart was implemented to use Redis as a database backend. However, due to the extensive use of MongoDB in company projects, the application was purposely modified and integrated with the latter database. Consequently, the RCA solution

[1] Online Boutique microservices application demo

Table 8.6: Overview of *Online Boutique* microservices

Service	Description
frontend	Exposes an HTTP server to serve the website.
cart	Stores and retrieves product items from the shopping cart maintained in the MongoDB database.
productcatalog	Provides the list of products from a JSON file and ability to search products and get individual product details.
currency	Converts one money amount to another currency. Uses real values fetched from European Central Bank.
payment	Charges the given credit card (mock) with the given amount.
shipping	Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock).
email	Sends order confirmation email to a user (mock).
checkout	Retrieves user cart, prepares order and orchestrates the payment, shipping and the email notification.
recommendation	Recommends other products based on what is given in the cart.
ad	Provides text ads based on given context words.

was evaluated against faults comprising the MongoDB domain, strengthening the industrial character of the dissertation.

A sharded MongoDB cluster was deployed from the enterprise MongoDB distribution by Percona using the Percona MongoDB Operator for Kubernetes. Sharded cluster consists of three major components depicted in Figure 8.33. Shards maintain subsets of user data fragmented by sharding key ranges. Config Servers are responsible for maintaining the Shard registry and cluster settings. Mongos act as query routers implementing an interface between user applications and the sharded cluster. Typically, client applications connect to the MongoDB cluster through a Mongos instance. Mongos reads the sharding scheme from the Config Servers and then distributes user queries to Shards containing relevant subsets of data based on the sharding key included in the query.

Both Config Servers and shards are Replica Sets. Each Replica Set comprises a group of members, i.e., MongoDB Servers maintaining the same data set for data redundancy and high availability. A designated member fulfills the role of the primary node responsible for processing write and read queries. Other members act as secondaries that replicate data from the primary member. If the primary is unavailable, an eligible secondary holds an election to elect a new primary.

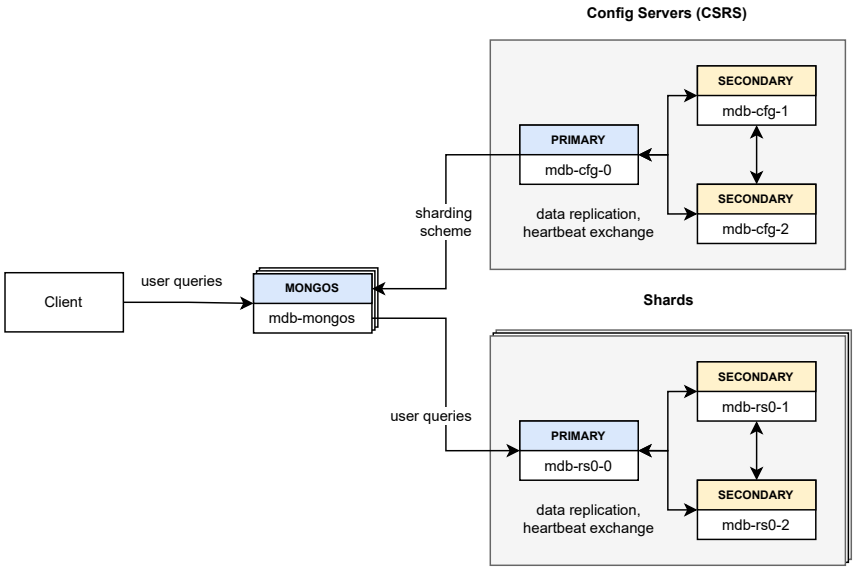


Figure 8.33: Overview of MongoDB cluster topology

Correct application setup and formation of the service mesh are confirmed in Figure 8.34. The diagram presents the application graph obtained in Kiali - Istio dashboard visualizing service mesh based on service metrics collected from Istio Proxies. Triangle symbols depict application services, while rounded squares represent application workloads. Graph edges reflect discovered service communication paths, where green paths indicate healthy gRPC or HTTP traffic while blue paths indicate regular TCP transmission. Compared to the reference diagram illustrated in Figure 8.32, service names in the dashboard are purposely extended with the *service* suffix.

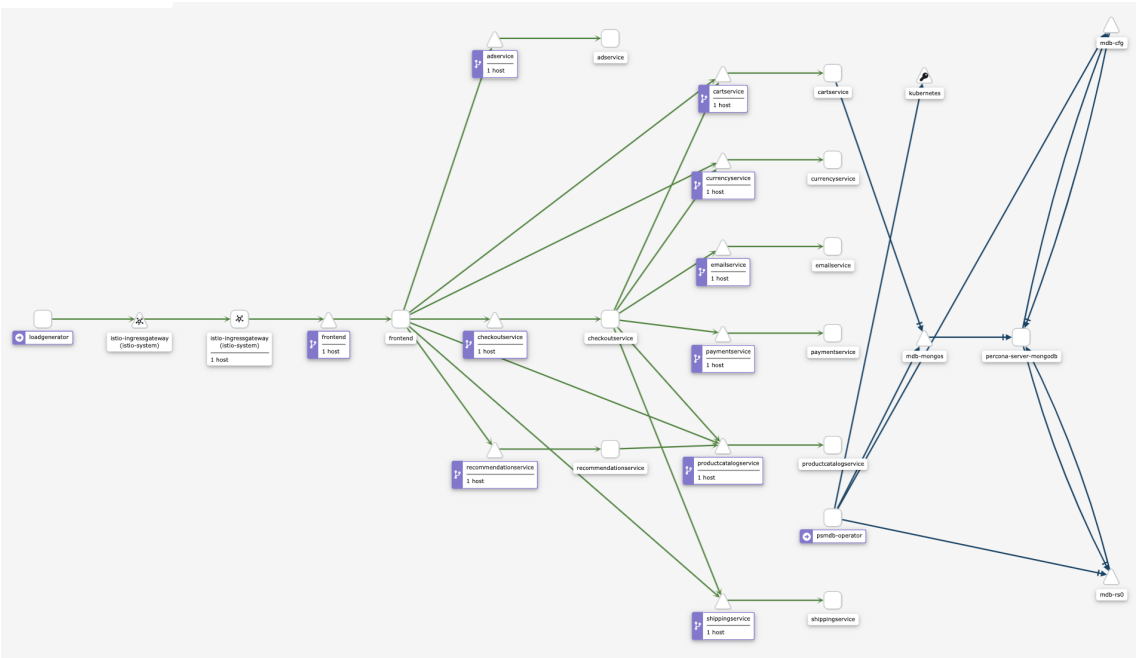


Figure 8.34: Visualization of test application service mesh in Kiali dashboard

8.3.3 Construction of RCA Model

As described in Section 6.1, the root cause inference process is based on the knowledge of system structure and behavior represented in the form of two RCA model structures, i.e., system object dependency graph and symptom co-occurrence map. The former, introduced in Section 5.3, reflects system objects and inter-object dependencies important from the RCA perspective. Its structure is built continuously in real-time based on reading system information from the system management plane and evaluating it against the defined system taxonomy. The latter, described in Section 5.5, maintains the knowledge of semantic symptom dependencies discovered by performing symptom co-occurrence analysis on historical symptom data. Due to the large volume of data needed to train the employed statistical algorithms and the high computational cost of the co-occurrence method, the structure of the symptom co-occurrence map is built in the offline phase.

As subsequent test application components were deployed in the Kubernetes cluster, mechanisms employed in the solution prototype automatically and in real-time produced the structure of the system object dependency graph based on data accumulated from orchestration and observability endpoints available in the provisioned cloud platform. Obtained dependency graph structure encompasses *545* Kubernetes objects of *13* kinds and *901* dependencies in total, where *88* objects and *161* dependencies constitute part of the test application. For visualization purpose, graph structure was filtered by Kubernetes namespace designated for test application and divided into fragments corresponding to service, workload, and configuration planes depicted in Figure 8.35, Figure 8.36, and Figure 8.37, respectively.

Figure 8.35 illustrates a portion of the system object dependency graph reflecting the service plane, where graph nodes represent application services in terms of Kubernetes Service objects, whereas graph edges represent communication paths between them. As described in Section 7.1.3, inter-service dependencies are determined based on service communication metrics gathered by the Prometheus Server from Istio Proxy instances installed alongside application components. The resulting structure of the dependency graph contains a complete set of test application services, including services belonging to the MongoDB cluster. It also accurately maps service communication paths that match reference paths presented in Figure 8.32. In addition, individual service communication layers are visible, i.e., application entry point, the *frontend* service, has direct connections with *ad*, *checkout*, *recommendation*, *currency*, *shipping*, *cart*, and *productcatalog* services, constituting the first service layer. Then, the *checkout* service has connections with *currency*, *shipping*, *cart*, *payment*, *email*, and *productcatalog* services, where *payment* and *email* services constitute

the second service nesting layer. Finally, the *cart* service communicates with the MongoDB cluster via the Mongos service (*mdb-mongos*). Communication within the MongoDB cluster is correctly mapped. It originates at the query router, i.e., Mongos (*mdb-mongos*), which reads the sharding scheme from Config Servers (*mdb-cfg*), and delegates user queries to appropriate Shard, in this case, *mdb-rs0*. The self-loop within the *mdb-rs0* service corresponds to the internal Shard communication and heartbeats exchanged between Shard members.

The *frontend-external* service is a service of type *LoadBalancer*, i.e., it enables communication with the test application from outside the Kubernetes cluster. Since external communication was not used in the experiment, the relevant communication metrics were not processed by Istio, and consequently, the service was not linked to other services in the graph. That is a correct result because if communication does not occur, it should be excluded from the graph structure.

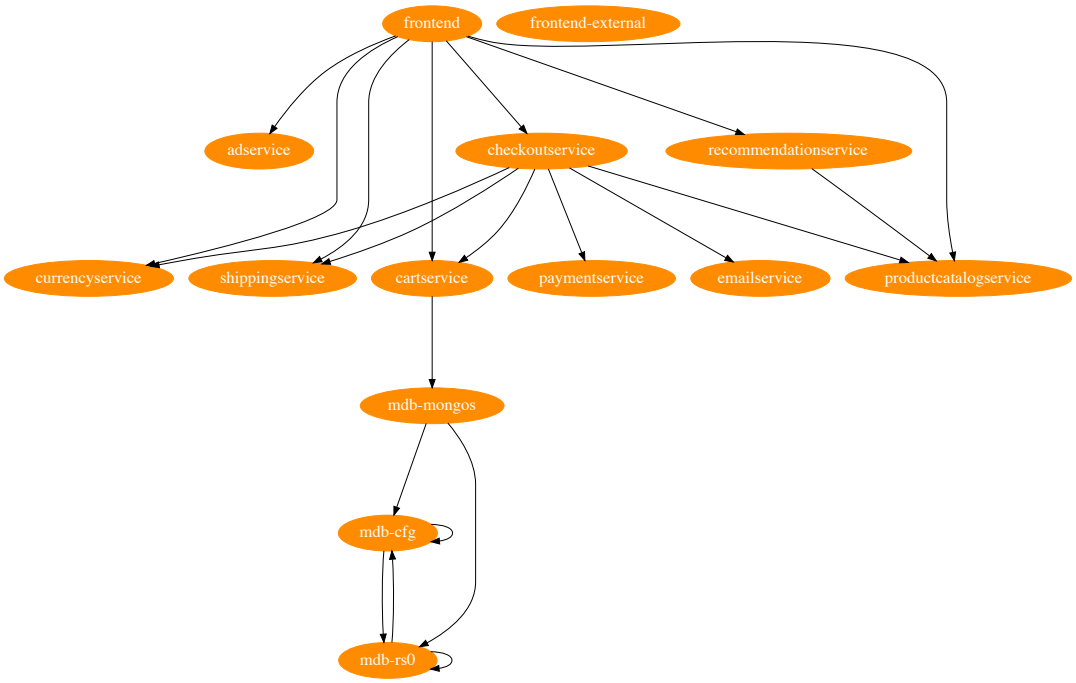


Figure 8.35: Subgraph of system object dependency graph reflecting Kubernetes objects from the service plane. Services are marked in orange.

Further, the workload plane, i.e., Kubernetes objects associated with processing units and entities that supervise them, is illustrated in Figure 8.36. In the diagram, Kubernetes Pods, ReplicaSets, Deployments, and StatefulSets are marked in colors blue, purple, green, and yellow, respectively. Recall that Pods constitute principal workload processors in Kubernetes. ReplicaSets are responsible for stable maintenance of the desired number of Pods. Deployments manage seamless upgrades of ReplicaSets. Finally, StatefulSets manage stateful applications and provide guarantees about the

ordering and identity of their Pods. As discussed in Section 7.2, workload objects are read using Kubernetes API endpoints and the resource watch mechanism.

In the obtained dependency graph structure, workloads for individual application services are correctly mapped and take form of distinct subgraphs. For instance, the *cartservice-v1* Deployment along with associated ReplicaSet and Pod correspond to the *cart* service, whereas the StatefulSet *mdb-cfg* along with associated Pods *mdb-cfg-0*, *mdb-cfg-1* and *mdb-cfg-2*, correspond to the *mdb-cfg* service.

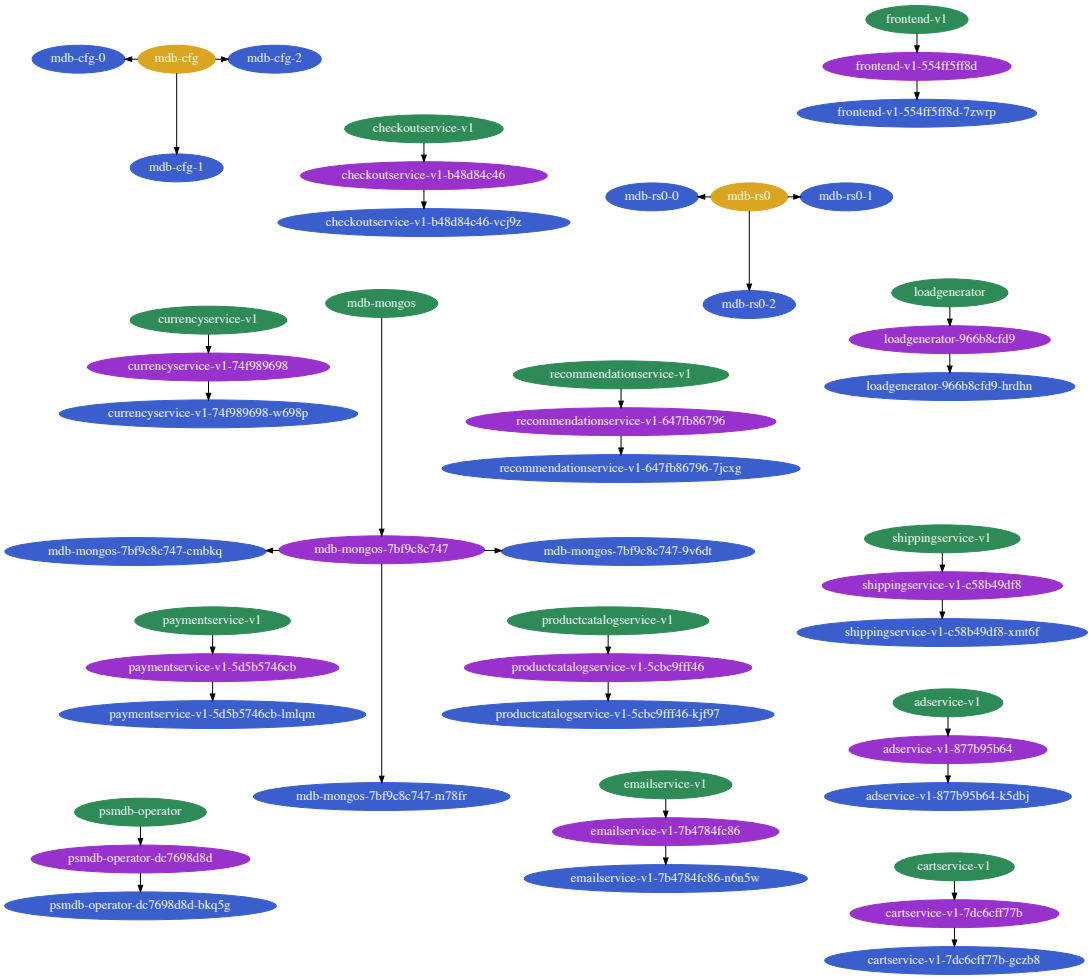


Figure 8.36: Subgraph of system object dependency graph reflecting Kubernetes objects from the workload plane. Pods, ReplicaSets, Deployments, and StatefulSets are marked in colors blue, purple, green, and yellow, respectively.

Last, Figure 8.37 illustrates configuration plane for the test application. Its structure comprises Kubernetes Pods, marked in blue, and associated ConfigMaps and Secrets, marked in gray and pink. Both ConfigMaps and Secrets store configuration data in key-value format. Similar to the workload plane, configuration objects are fetched automatically using Kubernetes API and the resource watch mechanism.

Dependency graph structure related to the configuration plane is significantly more complicated than previously discussed service and workload planes, mainly due to the number of discovered dependencies. That is caused by the fact that configuration objects are shared across many Pods. For instance, the configuration of the mongod process running within Config Servers and Shard workloads is stored in two ConfigMaps, namely *mdb-cfg-mongod* and *mdb-rs0-mongod*, each shared by all Pods of the corresponding workload. Similarly, the *mdb-secrets* Secret containing usernames and passwords of MongoDB cluster users is shared by all Pods belonging to the Mongos workload, i.e., *mdb-mongos*.

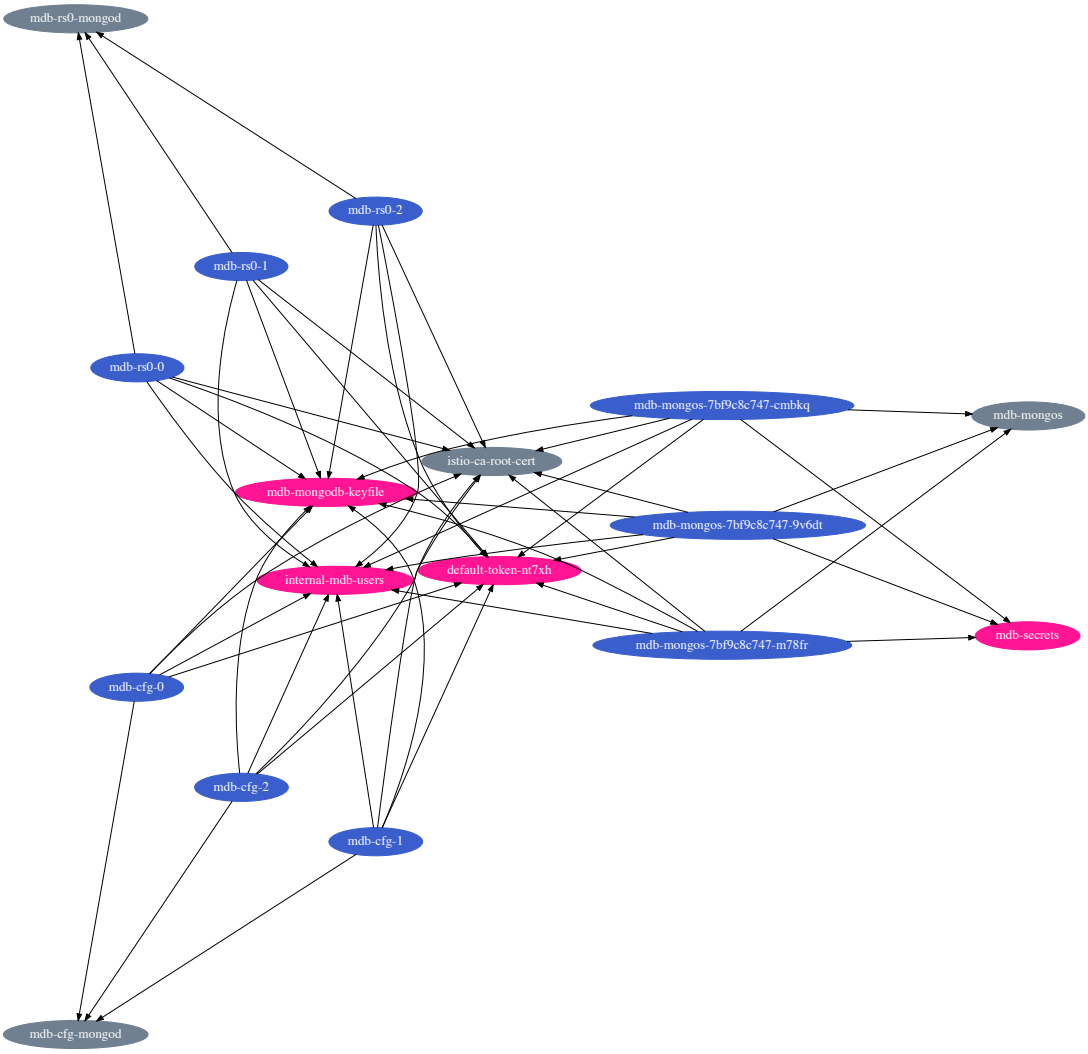


Figure 8.37: Subgraph of system object dependency graph reflecting Kubernetes objects from the configuration plane. Pods, Secrets, and ConfigMaps are marked in colors blue, gray, and pink, respectively.

The second RCA model structure produced by the implemented solution is the symptom co-occurrence map, describing semantic dependencies between known symptoms. Contrary to the system object dependency graph, which is constructed

in real-time, the co-occurrence map is constructed offline before any failure diagnosis is performed. That is due to the latter structure relying on symptom co-occurrence, which is computationally intensive and requires many symptom instances to be processed by the employed statistical algorithms.

In order to facilitate the acquisition of historical symptom data needed to train statistical models, the test application was exposed to continuous and automated fault simulation. The simulation was carried out using elements of chaos testing. Specifically, the Chaos Mesh tool was used to simulate specific faults in regular time intervals within a limited subset of system objects.

Table 8.7: Configuration of chaos scenarios for fault simulation

Action	Affected Pods	Mode	Schedule	Duration
Pod Failure	mdb-rs0-x	Random One	0 * * * *	5 min.
CPU Stress	mdb-rs0-x	All	0 * * * *	5 min.

Table 8.7 presents configuration of simulated chaos scenarios. In the table, *Action* maps to a specific chaos action supported by the Chaos Mesh tool. Two types of actions were used in performed experiments, i.e., *Pod Failure* and *CPU Stress*. *Pod Failure* injects a fault into a set of Pods, making them unavailable for a given period, whereas *CPU Stress* emulates CPU overload in a Pod using the stress-ng Linux kernel utility [2]. Further, *Affected Pods* and *Mode* columns designate a set of Pods subjected to chaos testing. The former determines an initial set of Pods, while the latter determines the strategy of selecting Pods from the initial set. In conducted experiments, the initial set consisted of all Pods belonging to the *mdb-rs0* Shard, while the *Mode* was set to *Random One* or *All*. The *Schedule* column describes the failure simulation frequency, expressed in the CRON format. Finally, *Duration* defines the duration of a single failure simulation.

The *Pod Failure* scenario is depicted in Figure 8.38. It simulates fault in a random Pod within the *mdb-rs0* Shard causing reporting of the *KubePodCrashLooping* symptom. Due to heartbeat exchange within Replica Set associated with the Shard, a failure in a Replica Set member is immediately detected by other members, resulting in the emission of *MongoDBHeartbeatFailedToHostInReplicaSet* symptoms. Moreover, cluster members observe each other and exchange information about the current cluster view using the gossip protocol. Member failure propagates to remaining cluster nodes, resulting in *MongoDBHostFailedInReplicaSet* symptoms on Shard members, Config Servers, and Mongos instances. Then, a symptom is reported on the Pod hidden behind the *cart* service due to the MongoDB client not being able to complete cart management queries on the configured majority of Shard

[2]Documentation of stress-ng Linux kernel utility

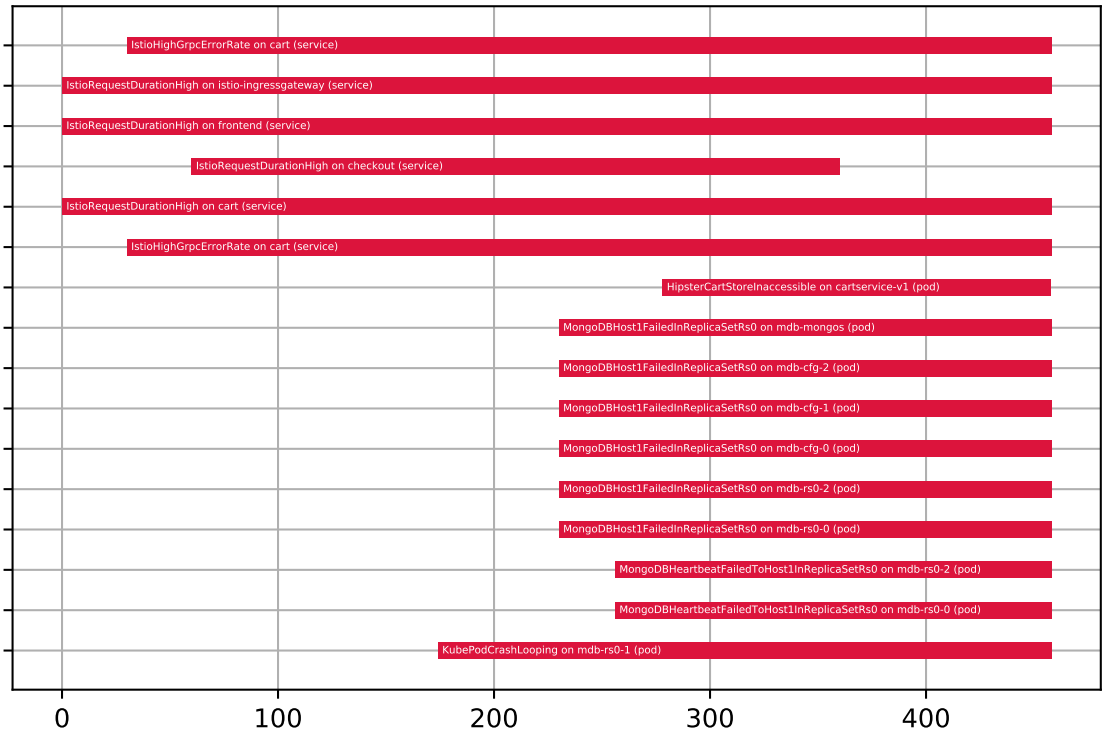


Figure 8.38: Fault timeline for *Pod Failure* scenario defined in Table 8.7

members. The write concern configured in the client enforces query confirmation by all three members, while only two members are available at the failure time. That results in the emission of the *CartStoreInaccessible* symptom. Failure in the *cart* service Pod, in turn, causes cascading propagation of symptoms related to gRPC and HTTP request errors, namely *IstioHighGrpcErrorRate* and *IstioHigh5xxErrorRate*, in subsequent application services, i.e., *cart* service and services having a direct or transient relationship with the *cart* service. In addition, due to MongoDB client timeouts imposed by the lack of Shard majority, application services produce latency symptoms, i.e., *IstioRequestDurationHigh*, on the path between *frontend* and *cart* services.

Notably, mentioned symptoms are not reported in causal order but, as pointed out in Section 4.1, in the order enforced by test environment condition, telemetry data sampling, and alert rule configuration. For instance, Istio symptoms constituting failure effect occur before the *KubePodCrashLooping* symptom reflecting failure root cause. Given that eliminating these factors cannot be assumed, the developed scenario is valid and strengthens testing the solution under practical conditions.

The *CPU Stress* scenario is visualized in Figure 8.39. The scenario simulates a high CPU load on all Shard members, which is usual for temporary database overload. A pool of 100 workers was used to occupy 95% of the overall processor time assigned to the database workload. As a result, processor time assigned to affected Pods was fully

saturated for most of the test, effectively blocking or delaying application queries originating from the *cart* service. Excessive CPU usage in Shard members, exceeding the defined resource limit in Pod definition, causes CPU throttling and translates into a series of *CPUThrottlingHigh* symptoms. Subsequently, the delay in database query execution significantly extends the duration of gRPC and HTTP requests, which reflects in *IstioRequestDurationHigh* symptoms reported on application services on the path between *frontend* and *cart* services.

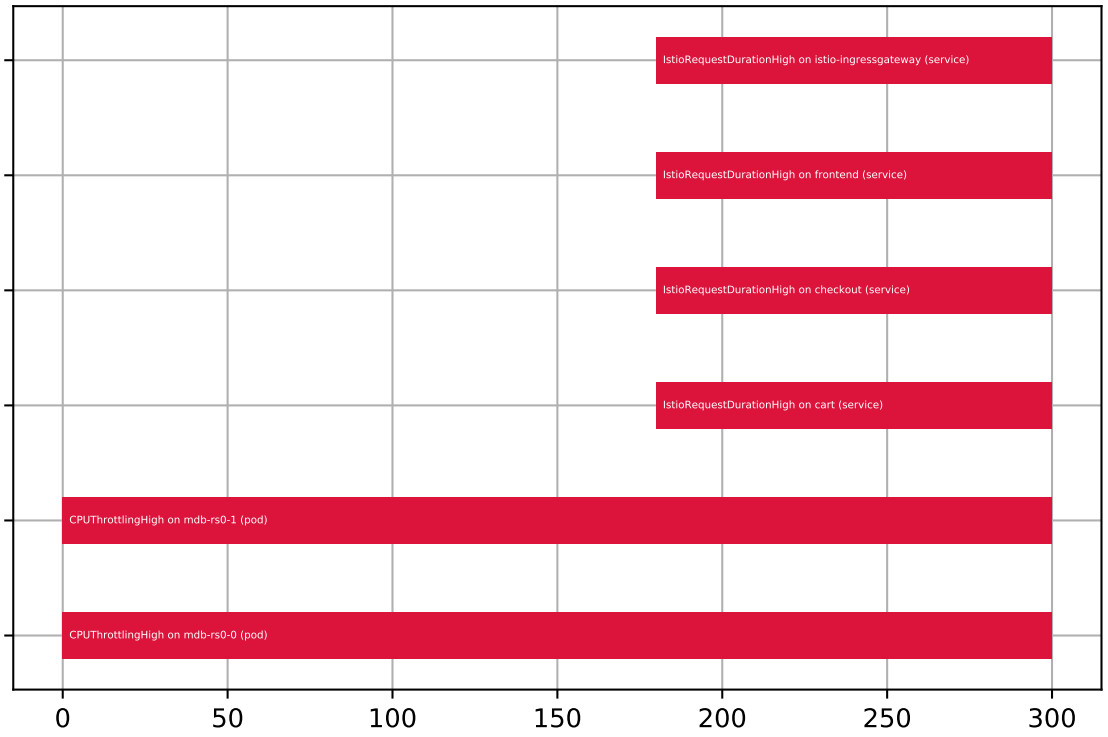


Figure 8.39: Fault timeline for *CPU Stress* scenario defined in Table 8.7

Contrary to the *Pod Failure* scenario, symptoms in the *CPU Stress* scenario occur in causal order, i.e., Istio symptoms constituting failure effect are signaled after CPU overload symptoms constituting failure root cause. However, the order of Istio symptoms emitted across application services cannot be determined due to the simultaneous moment of symptom reporting. Similar to the previous scenario, the limitation results from telemetry sampling by the adopted monitoring system.

Defined chaos scenarios were repeated 100 times each at hourly intervals. Longer intervals were necessary to stabilize the test environment after fault simulation, silence all symptoms, and revert baseline metric characteristics in observability tools. Considering the time intervals, it took a minimum of 4 days to achieve the required number of chaos tests for each scenario.

Symptom timeline depicted in Figure 8.40 comprises symptom instances emitted in test environment in response to *Pod Failure* simulation specified in Table 8.7. The

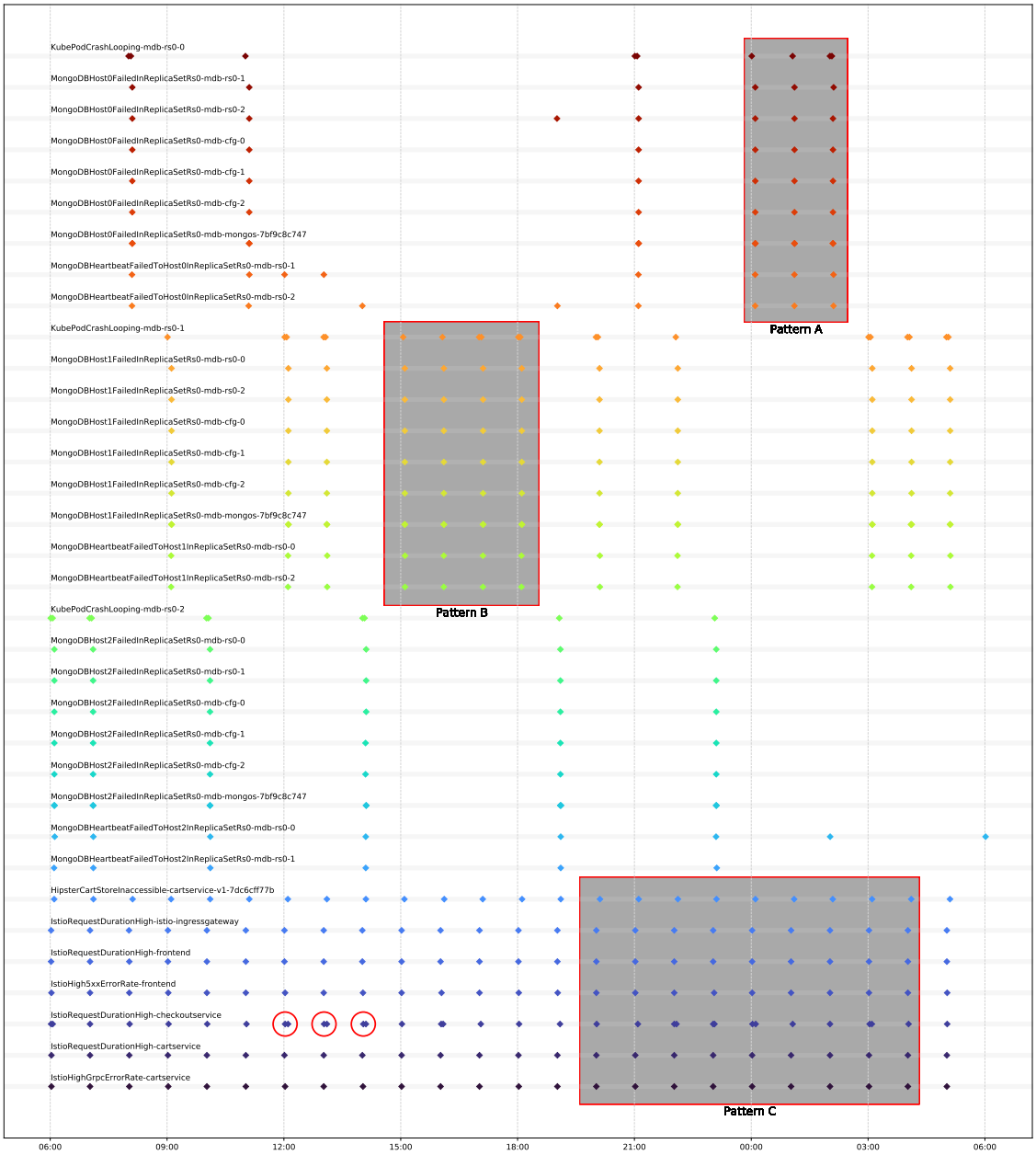


Figure 8.40: Timeline of symptoms reported by the test application in response to *Pod Failure* scenario described in Table 8.7: *Pattern A* and *Pattern B* denote groupings of MongoDB symptoms emitted in response to failure simulation on *mdb-rs0-0* and *mdb-rs0-1* members; *Pattern C* denotes grouping of Istio symptoms emitted in each failure instance; red circles denote duplicated Istio symptoms constituting outliers in the co-occurrence analysis

horizontal axis represents time, while the vertical axis comprises subsequent series of symptom instances for individual symptom types differentiated by application components. Due to the large timeline size, symptom time-space was limited to one day, translating into 24 out of 100 experiment instances. Observation of repetitive symptom co-occurrence patterns supersedes the temporal resolution of the diagram.

Visibly, instances of each symptom type repeat consistently, forming symptom patterns corresponding to fault simulations on three distinct Shard members, i.e., *mdb-rs0-0*, *mdb-rs0-1*, and *mdb-rs0-2*. These patterns manifest in the form of characteristic vertical groupings. For instance, *Pod Failure* simulation on *mdb-rs0-0* member, signaled by the *KubePodCrashLooping* symptom triggers the *MongoDBHost0FailedInReplicaSetRs0* symptom on all other cluster nodes and the *MongoDBHeartbeatFailedToHost0InReplicaSetRs0* symptom on remaining Shard members, denoted as *Pattern A* in Figure 8.40, whereas the simulation of *Pod Failure* on *mdb-rs0-1* member, signaled by the *KubePodCrashLooping* symptom triggers *MongoDBHost1FailedInReplicaSetRs0* and *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* symptoms, denoted as *Pattern B* in Figure 8.40. In this scenario, symptoms instances co-occur consistently across fault simulations on a specific Shard member and never across fault simulations on distinct members. As discussed in Section 5.5, similar symptom patterns enable mining semantic symptom dependence for constructing the symptom co-occurrence map.

In addition, instances of some symptom types occur at each experiment instance regardless of which cluster node is subjected to fault simulation. For instance, *Pod Failure* simulation on arbitrary Shard member triggers a series of service-level symptoms, denoted as *Pattern C* in Figure 8.40, i.e., *IstioRequestDurationHigh*, *IstioHigh5xxErrorRate*, and *IstioHighGrpcErrorRate* on the request path including *istio-ingressgateway*, *frontend*, *cart*, and *checkout* services.

Further, the symptom timeline incorporates outliers appearing as symptom instances reported unexpectedly due to malfunction in system operation or failure of alert rule evaluation. For instance, the 7th, 8th, and 9th occurrences of the *IstioRequestDurationHigh* symptom on the *checkout* service, denoted as red circles in Figure 8.40, are duplicated. This phenomenon strengthens practical solution examination. Due to anomalies emerging regularly in IT systems, tolerance of used statistical methods must be tested against similar defects.

Despite minor outliers found in collected data, the above results confirm the thesis assumption that IT systems designed and configured for failure resilience show a deterministic reaction to faults allowing the discovery of semantic symptom dependencies.

Subsequently, symptoms obtained in fault simulations were provided to the solution module responsible for constructing the symptom co-occurrence map. The module computes symptom correlation rules based on the event co-occurrence method and uses them for building the graph structure representing semantic symptom knowledge. Due to the high complexity of the resulting co-occurrence map, it is

presented in the form of an adjacency matrix shown in Figure 8.41. Columns and rows include symptom pairs subjected to correlation, whereas matrix cells hold estimated correlation strengths for corresponding symptom pairs.

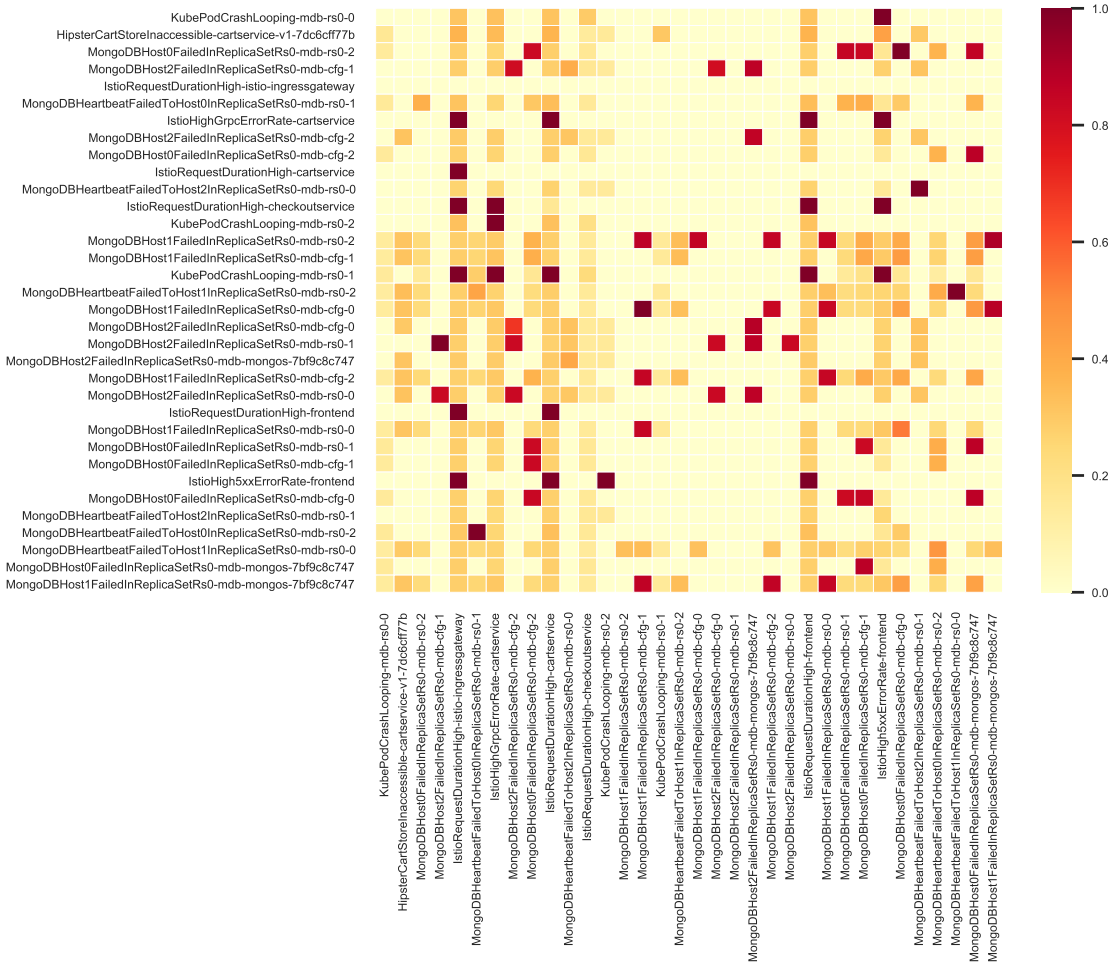


Figure 8.41: Adjacency matrix for symptom co-occurrence map computed from symptoms collected from observability tools installed in test environment

For the visualization purpose, the most significant fragments were extracted from the symptom co-occurrence map. First, the number of graph edges was reduced by filtering out weak dependencies whose values fall below the value threshold of 0.6. Then, the map was searched for the largest subgraphs constituting weak components in the graph structure.

Subgraph illustrated in Figure 8.42 shows resulting semantic symptom dependencies computed based on the history of MongoDB symptoms that co-occurred in response to *Pod Failure* simulation on *mdb-rs0-0* Shard member. In this scenario, symptoms were emitted by MongoDB cluster nodes except for the one that crashed. Specifically, instances of *MongoDBHost0FailedInReplicaSetRs0* symptom signaling failure of *mdb-rs0-0* member were reported on remaining Shard members, i.e., *mdb-rs0-1* and *mdb-rs0-2*, on all Config Servers, i.e., *mdb-cfg-0*, *mdb-cfg-1* and *mdb-cfg-2*, and on

Mongos instances. *MongoDBHost0FailedInReplicaSetRs0* symptoms emitted in the conducted experiment showed consistent co-occurrence, depicted in Figure 8.40, resulting in coefficients values ranging from 0.83 to 1.

Symptom dependencies produced in response to the failure of *mdb-rs0-1* and *mdb-rs0-2* Shard members produced analogous subgraphs to the one illustrated in Figure 8.42, obtaining high coefficient values ranging from 0.85 to 1 and from 0.68 to 1, respectively.

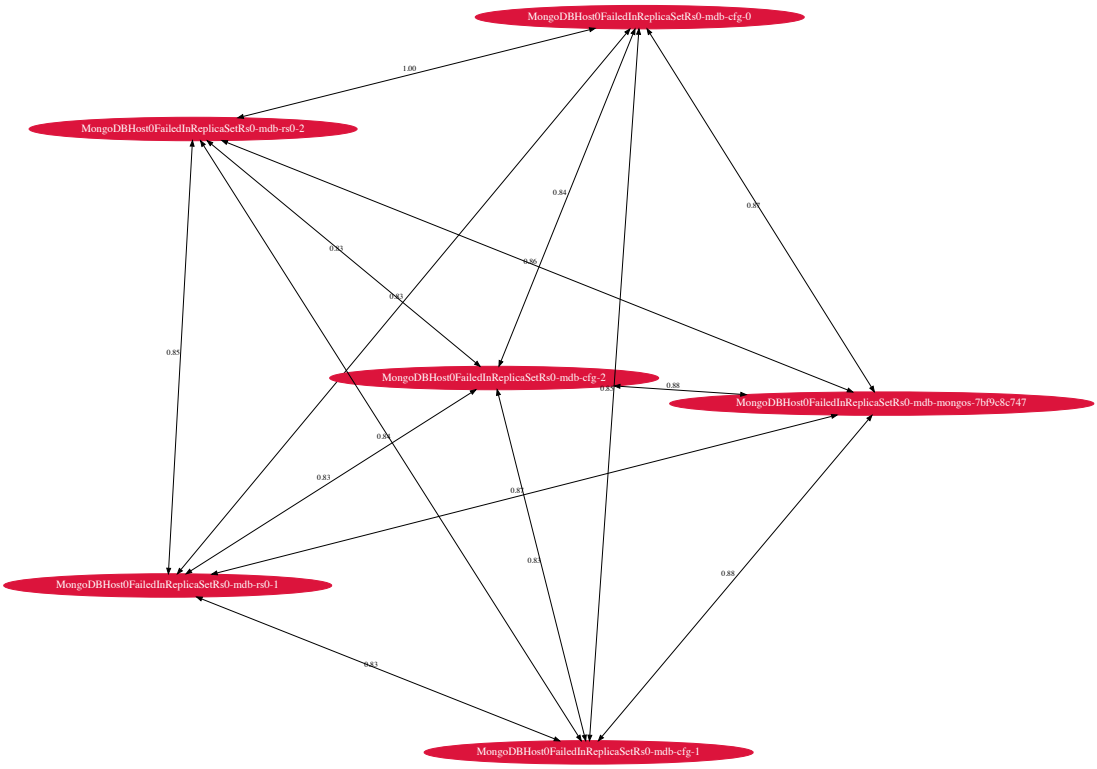


Figure 8.42: Subgraph of reduced symptom co-occurrence map reflecting symptoms related to MongoDB host failure

Subgraphs illustrated in Figure 8.43 show MongoDB symptoms related to heartbeat exchange between Shard members as part of the replication protocol. In failures affecting the *mdb-rs0-0* member, remaining members reported the *MongoDBHeartbeatFailedToHost0InReplicaSetRs0* symptom due to missing heartbeats from the *mdb-rs0-0* member. Similarly, in reaction to failures of *mdb-rs0-1* and *mdb-rs0-2* members, symptoms *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* and *MongoDBHeartbeatFailedToHost2InReplicaSetRs0* were observed, respectively. Prompt emission of symptoms on neighbor members resulted in consistent symptom co-occurrence. Consequently, pairs of heartbeat symptoms emitted in response to the failure of a particular Shard member gained the highest correlation coefficient value

of 1. At the same time, symptoms observed across faults simulated on distinct Shard members showed a lack of semantic dependence.

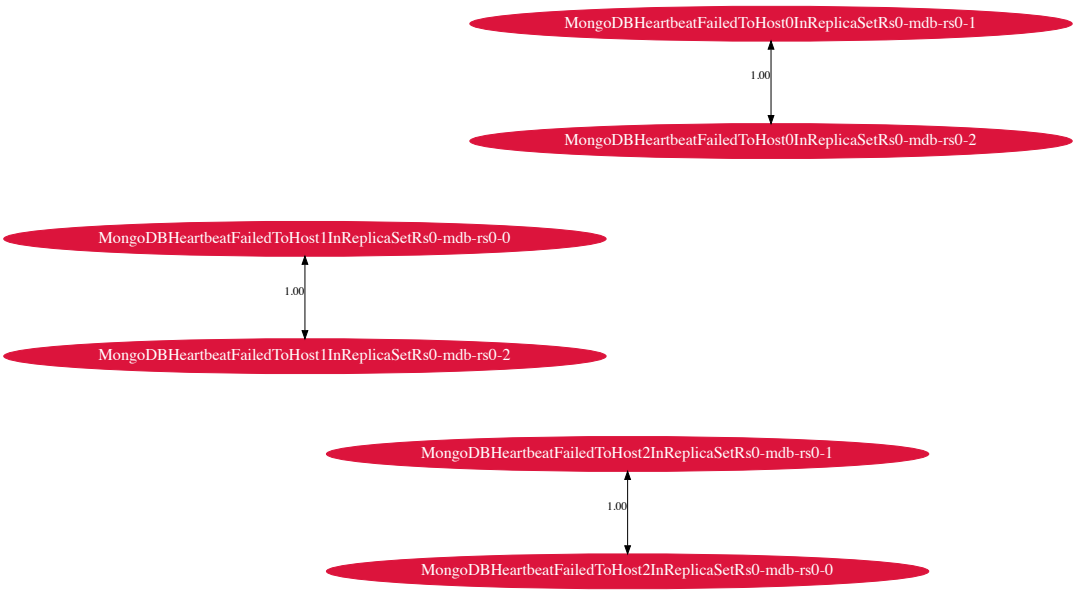


Figure 8.43: Subgraph of reduced symptom co-occurrence map reflecting symptoms related to MongoDB heartbeat failure

Finally, subgraph illustrated in Figure 8.44 exhibits semantic dependencies discovered between Istio symptoms related to communication defects emerging in application services. Symptoms were emitted at each iteration of the *Pod Failure* experiment regardless of the Shard member subjected to failure simulation. Due to the high responsiveness of Istio metrics, the impact of database failure on application service performance was immediately detected, producing Istio symptoms on relevant application services with time lag converging to zero. That, in turn, contributed to the consistent symptom co-occurrence reflected in maximal correlation coefficient values of 1.

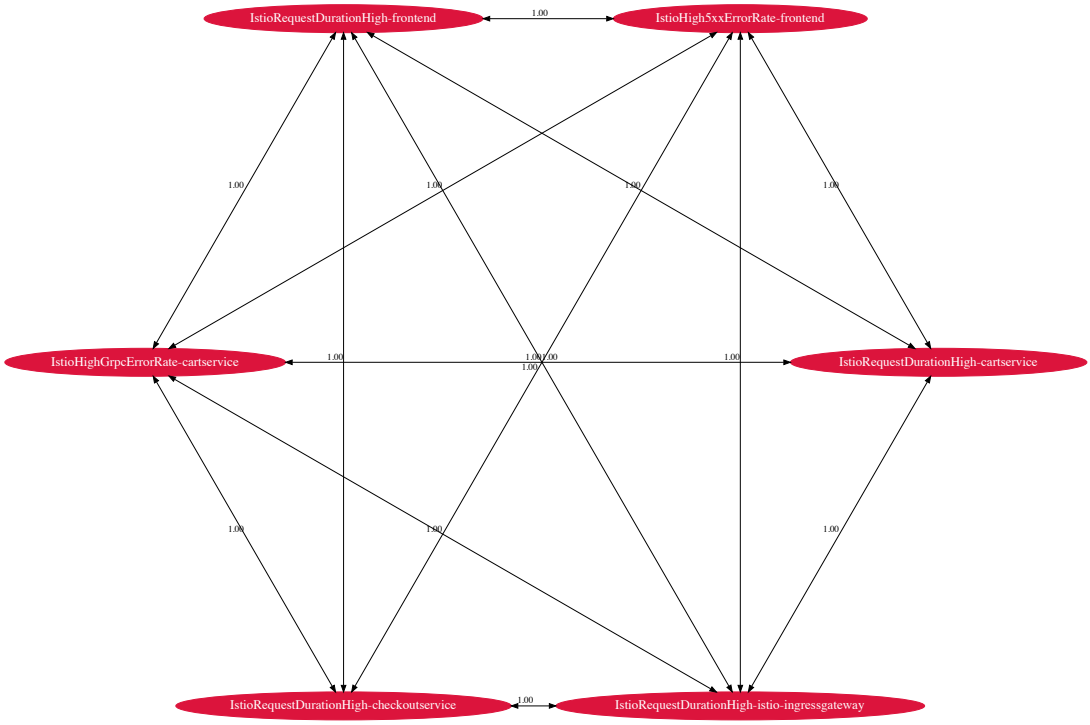


Figure 8.44: Subgraph of reduced symptom co-occurrence map reflecting symptoms related to Istio service-to-service communication

8.3.4 Fault Diagnosis

After the knowledge of system structure and behavior, expressed through the system object dependency graph and symptom co-occurrence map was acquired, evaluation proceeded to the experimental phase comprising the diagnosis of new fault instances.

In order to determine the cause of failure, the solution performs four symptom correlation analyses, each centering around a different system operational aspect enclosed in the RCA model. Each analysis produces a matrix of standardized correlation coefficients computed for subsequent symptom pairs. Matrices are consolidated into an aggregated symptom correlation matrix using the weighted coefficient aggregation strategy. Next, the aggregated symptom correlation matrix is translated into a fault view causality graph used to identify the potential root cause and affect symptoms, extract candidate fault trajectories, and rank trajectories according to adopted scoring criteria. Finally, ranked fault trajectories are provided at the algorithm output.

Symptom Topological Distance Analysis

Correlation matrix for symptom topological distance analysis detailed in Section 6.2 is presented in Figure 8.45. The matrix exposes several strong coefficients clusters with values of 1. The first cluster is build of *MongoDBHost1FailedInReplicaSetRs0* and *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* symptoms emitted on MongoDB cluster Pods. Symptom dependencies form a full graph. That is imposed by MongoDB inter-service communication illustrated in Figure 8.35, i.e., MongoDB cluster components communicate with each other to handle replication and gossip protocols. In addition, cluster coefficients are symmetrical to the matrix diagonal due to the communication occurring in both directions.

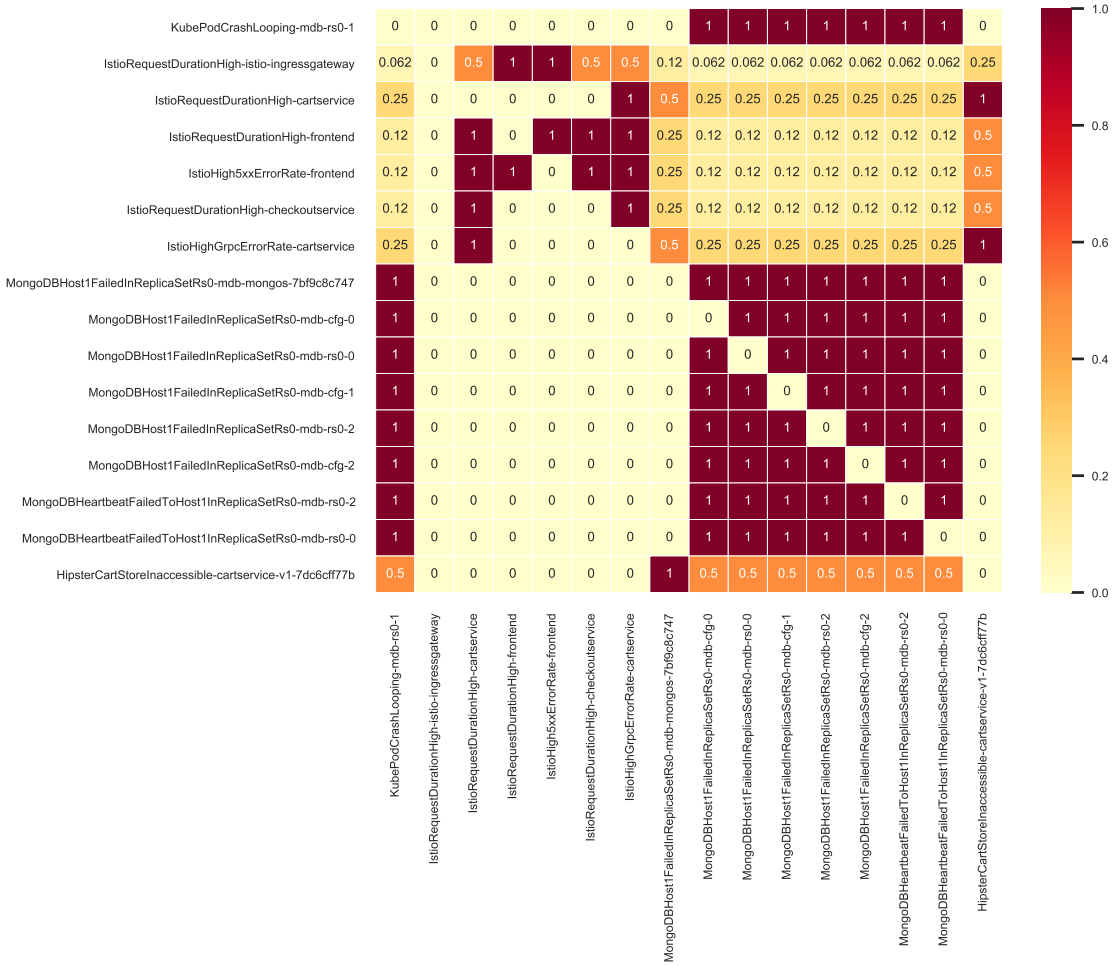


Figure 8.45: Correlation matrix produced in symptom topological distance analysis

The second coefficient cluster comprises a series of dependencies between *KubePodCrashLooping* symptom reported on *mdb-rs0-1* Shard member and all MongoDB symptoms, i.e., *MongoDBHost1FailedInReplicaSetRs0* and *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* reported on MongoDB cluster Pods. Similar to dependencies between MongoDB symptoms discussed in the previous paragraph, the correlation is

imposed by inter-service communication between MongoDB cluster components. Importantly, the *KubePodCrashLooping* symptom derived from the Kubernetes domain constitutes the root cause of simulated failure. Strong symptom dependence with MongoDB symptoms forms a crucial connection between Kubernetes and MongoDB domains.

Another coefficient cluster visible in the obtained correlation matrix comprises symptoms originating from the Istio domain. The cluster reflects communication paths and, at the same time, failure propagation throughout test application services. By analyzing the occurrence of *IstioRequestDurationHigh* symptom, one may observe causal service dependencies $\{frontend \rightarrow checkout\}$ and $\{checkout \rightarrow cart\}$, which correspond to the path $\{frontend \rightarrow checkout \rightarrow cart\ \}$ in reference service map depicted in Figure 8.32. In addition, pairs of Istio symptoms were reported on some application services, signaling high error rate and request latency, for instance, *IstioRequestDurationHigh* and *IstioHighGrpcErrorRate* emitted on the *cart* service. As expected, these symptom pairs, sharing the same source component, obtained the coefficient value of 1.

Further, the dependency between {*CartStoreInaccessible on cartservice*} and {*MongoDBHost1FailedInReplicaSetRs0 on mdb-mongos*} constitutes significant association in the matrix as it bridges symptoms across Istio and MongoDB domains. It is essential to emphasize that in topological distance analysis, dependency detection is orthogonal to associated telemetry data and relies solely on deterministic inspection of structural system dependencies in the dependency graph. Thus, although distinct domains of symptom sources may negatively affect correlation based on statistical methods, they do not affect the structural analysis allowing the inclusion of meaningful dependencies in the inference process. That observation highlights the efficiency of topological distance analysis in root cause identification.

Lastly, coefficient values for dependencies between Istio symptoms emitted on application services and MongoDB symptoms emitted on MongoDB cluster Pods are weakened according to the distance between source components in the dependency graph. Coefficient values range from 0.062 to 0.5 depending on the shortest path length. For instance, dependency between {*IstioRequestDurationHigh on cartservice*} and {*MongoDBHost1FailedInReplicaSetRs0 on mdb-cfg-0*} has coefficient value of 0.25 due to source components distant by three edges apart.

Symptom Co-occurrence Analysis

The correlation matrix for symptom co-occurrence analysis introduced in Section 6.3 is illustrated in Figure 8.46. The matrix reflects semantic symptom dependencies copied from the symptom co-occurrence map, depicted in Figure 8.41, for symptoms reported in the simulated failure scenario. Notably, obtained matrix separates strong semantic dependencies from weak ones. Strong dependencies are reflected in coefficient values ranging from 0.84 to 1, whereas weak dependencies are reflected in coefficient values ranging from 0 to 0.43.

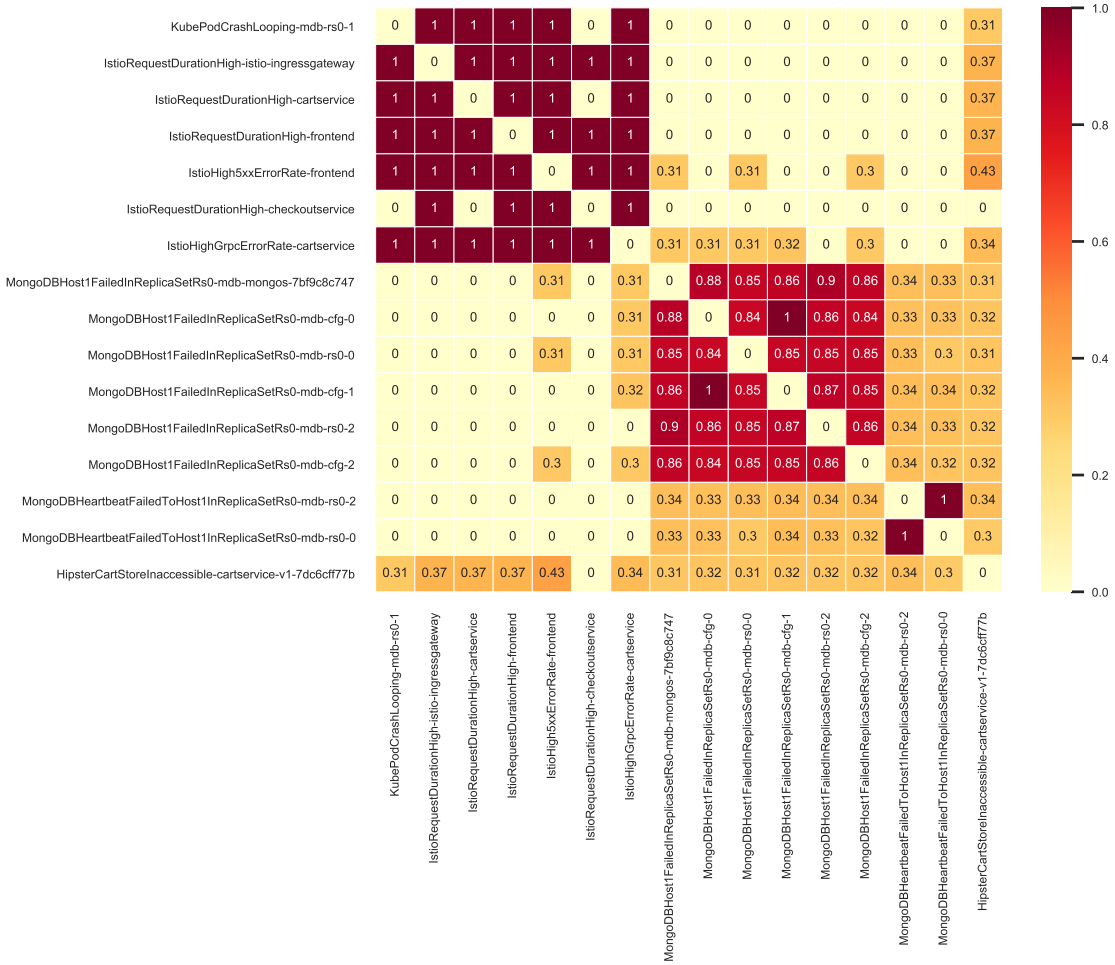


Figure 8.46: Correlation matrix produced in symptom co-occurrence analysis

The strongest semantic dependence appears between Istio symptoms and concern communication between test application services. Symptoms such as *IstioRequestDurationHigh*, *IstioHighGrpcErrorRate*, and *IstioHigh5xxErrorRate* co-occurred consistently in the past resulting in coefficient value of 1 for most application service pairs. That is especially visible in the subset of Istio symptoms in the symptom co-occurrence map presented in Figure 8.44. The consistent co-occurrence of Istio symptoms results from the high responsiveness of service metrics collected by Istio

Proxies, i.e., Istio Proxy samples service metrics with a high frequency, allowing timely recognition of service quality degradation.

According to the subgraph of symptom co-occurrence map presented in Figure 8.42, *MongoDBHost1FailedInReplicaSetRs0* symptoms reported on MongoDB cluster Pods showed equally strong co-occurrence in the past resulting in coefficient values ranging between 0.84 and 1. Further, the subgraph presented in Figure 8.43 exhibits another strong semantic dependency reaching a coefficient value of 1 is visible for *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* symptoms reported across Shard members. Contrary, *MongoDBHost1FailedInReplicaSetRs0* and *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* symptoms did not co-occur consistently due to system distortion. Hence, symptom dependencies of that kind, although exist, obtained low coefficient values ranging from 0.3 to 0.34.

Symptom Time Lag Analysis

In terms of coefficient value distribution, the correlation matrix for symptom time lag analysis, shown in Figure 8.47, coincides with the correlation matrix generated in the symptom co-occurrence analysis. Two major coefficient clusters are exposed in the matrix. The former comprises dependencies between Istio symptoms emitted on test application services. The latter correlates MongoDB symptoms reported on MongoDB cluster Pods. Time lags between symptoms obtained in the experiment were well fitted to time lag probability distributions obtained from the symptom co-occurrence map. That translated into high coefficient values for symptom pairs that showed strong co-occurrence in the past but also for pairs that co-occurred with mild or severe distortion. Most symptom dependencies take values above 0.5. The strongest symptom dependencies take values ranging from 0.82 to 1.

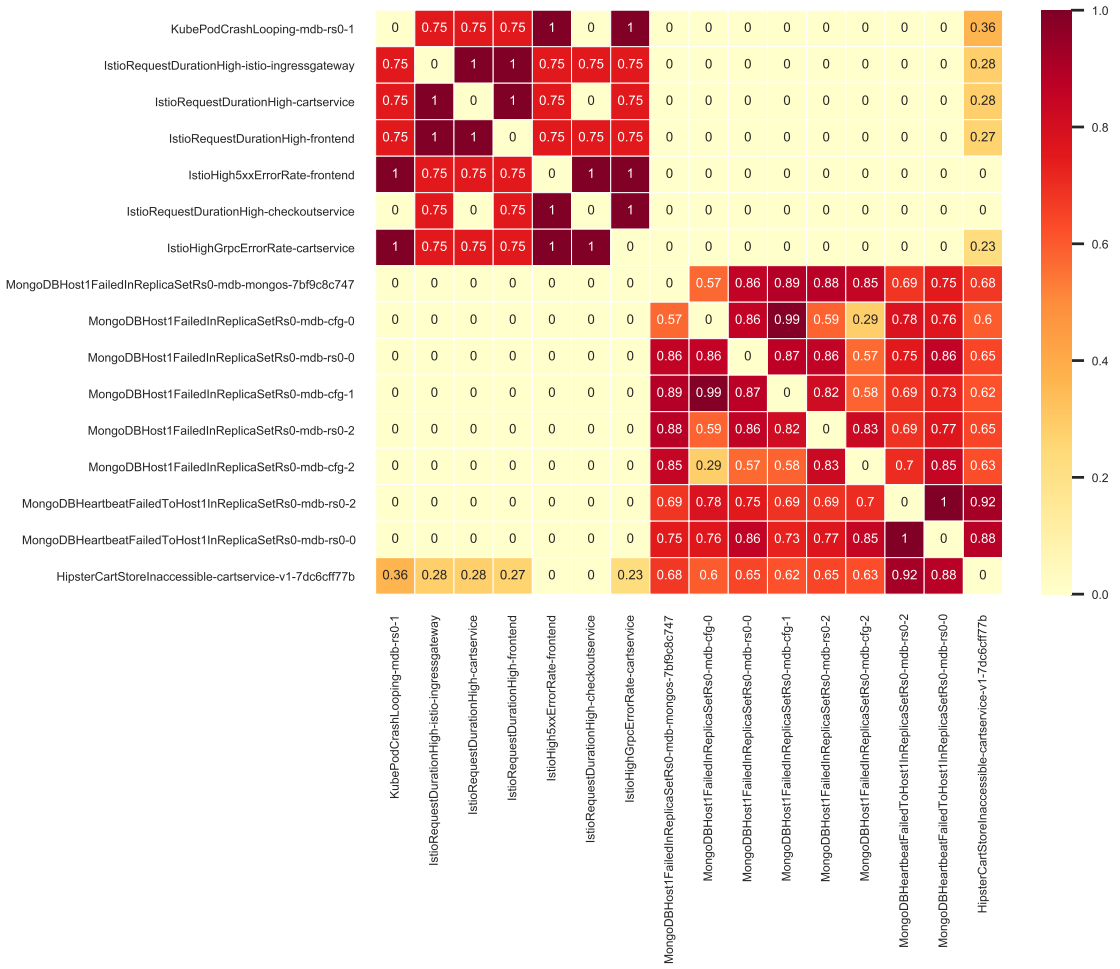


Figure 8.47: Correlation matrix produced in symptom time lag analysis

Construction of Fault View Graph

Consolidation of correlation matrices into aggregated symptom correlation matrix is demonstrated in Figure 8.48. It is based on the weighted coefficient aggregation strategy with weights specified in Table 8.3. The consolidation reveals correct fault trajectory and weakens correlations resulting from correlation transitivity. That is visible in the fault view graph illustrated in Figure 8.49.

By analyzing the matrix rows-wise, for each symptom, one can evaluate causal dependencies with other symptoms to identify the strongest candidates for inclusion in the resulting fault trajectory. Moreover, one can recognize the contributions of individual symptom correlation methods to the aggregated symptom correlation matrix.

For instance, considering the row corresponding to the {*IstioRequestDurationHigh on frontend*}, one may infer causal dependencies with {*IstioRequestDurationHigh on cartservice*}, {*IstioHighGrpcErrorRate on cartservice*}, {*HighIstioHigh5xxErrorRate on*

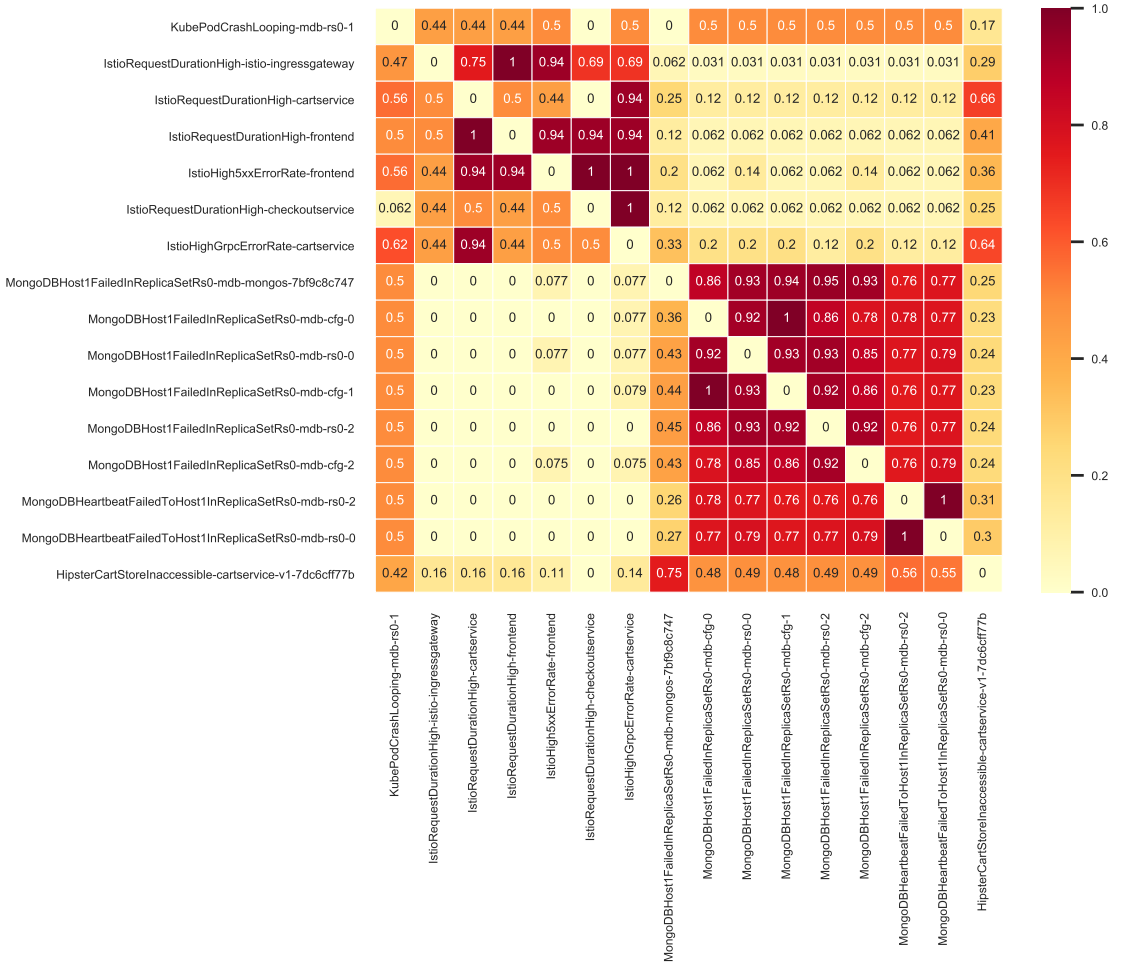


Figure 8.48: Correlation matrix aggregated from all symptom correlation analyzes based on coefficient weights specified in Table 8.3

frontend}, and {*IstioRequestDurationHigh on checkoutservice*}. These dependencies hold the highest aggregated coefficient values ranging between 0.94 and 1. They designate correct trajectory directions orginating from the considered symptom and exclude connections to semantically or topologically unrelated ones. Another example is the row corresponding to the {*CartStoreInaccessible on cartservice*}. In this case, only one symptom shows a significant correlation, i.e., {*MongoDBHost1FailedInReplicaSetRs0 on mdb-mongos*}. Here, the dependency holds the coefficient value of 0.75.

Each matrix row suggests possible directions for shaping fault trajectories originating from a given symptom. Depending on the results returned by individual symptom correlation methods, the aggregation of coefficient values may reinforce or weaken symptom dependencies. As a result, there can be zero, one, or many strong dependencies in a given matrix row. Single dependency indicates a clear causal relationship with another symptom, whereas many strong dependencies imply alternative fault trajectories, which must be compared in subsequent algorithm stages according to the adopted trajectory scoring criteria. A special case is the lack of strong symptom

dependency in a matrix row. It suggests that the symptom is a potential root cause of analyzed failure.

The inference algorithm automates a similar diagnostic process in the sense that it analyzes the fault view graph obtained from aggregated symptom correlation matrix and, using graph algorithms and scoring criteria, extracts and ranks potential fault trajectories. High correlation coefficient values drive trajectory selection and limit the search scope, whereas degrees of symptom nodes in the graph designate potential failure causes and effects.

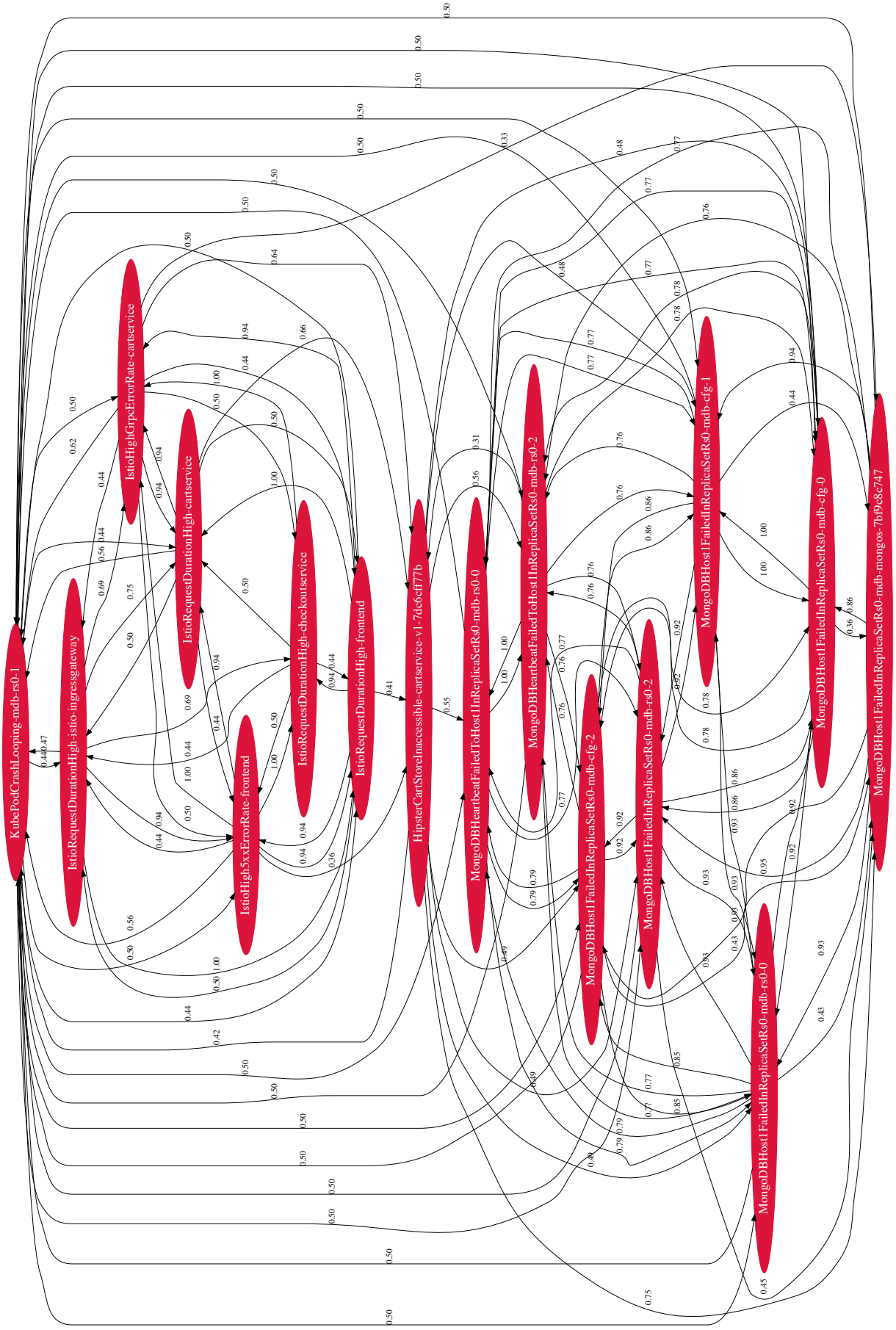


Figure 8.49: Fault view causality graph built from aggregated symptom correlation matrix depicted in Figure 8.48

Fault Trajectory Detection

Finally, Figure 8.50 shows the best fault trajectory obtained at the output of the inference algorithm. As expected, the highest scored trajectory correctly reflects the cause-effect sequence of symptoms from the simulated failure. All relevant symptoms are present and follow the correct causal order resulting in average dependency strength of $\sigma_{strength} = 0.95$ and trajectory length of $\sigma_{length} = 15$. The trajectory maintains correct root cause and effect symptoms, i.e., {*KubePodCrashLooping on mdb-rs0-1*} and {*IstioRequestDurationHigh on istio-ingressgateway*}.

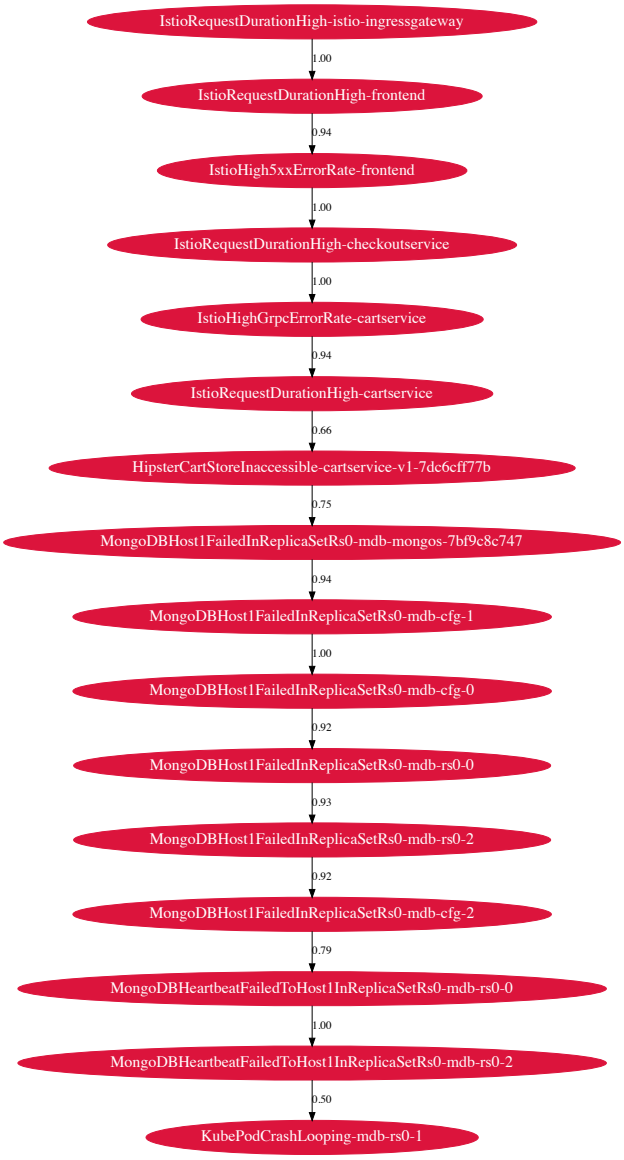


Figure 8.50: Best fault trajectory discovered by the inference algorithm

Trajectory begins with Istio symptoms related to high request latency, ordered by application service occurrence on the request path, i.e., *IstioRequestDurationHigh* on

ingressgateway, *frontend*, *checkoutservice*, and *cartservice*. They are accompanied by symptoms signaling request errors, i.e., *IstioHigh5xxErrorRate* on *frontend* and *IstioHighGrpcErrorRate* on *cartservice*. Dependencies between Istio symptoms show high aggregated correlation coefficient values ranging from 0.94 to 1.

Next, the trajectory comprises the *CartStoreInaccessible* symptom emitted on the *cart* service. That is a critical trajectory point as it constitutes a bridge between Istio symptoms and symptoms derived from the MongoDB domain. Weak symptom dependency with a coefficient value of 0.66 is influenced by low correlation scores obtained in symptom co-occurrence and time lag analyses, returning values of 0.37 and 0.28, respectively. Dependency is strengthened mainly by the topological distance analysis, which returned a correlation coefficient value of 1. It is worth noting that even though the aggregated coefficient value weakens average dependency strength, which is the primary criterion in the adopted trajectory scoring strategy, the mean change is marginal and, at the same time, including the symptom creates a connection with symptom propagation comprising strong symptom dependencies. The subsequent trajectory part compensates for the mean weakening.

The *CartStoreInaccessible* symptom emitted on the *cart* service connects the first trajectory part to the propagation of MongoDB symptoms holding strong dependencies with values ranging between 0.79 and 1. The inter-communication between MongoDB services outlines the order of cause-effect dependencies. The trajectory begins with {*MongoDBHost1FailedInReplicaSetRs0 on mdb-mongos*} as the *cart* service connects to the database through Mongos as a query router. Then, *MongoDBHost1FailedInReplicaSetRs0* symptoms emerge on MongoDB Pods grouped under *mdb-rs0* and *mdb-cfg* services. That is dictated by the fact that Mongos reads the sharding scheme from the Config Servers *mdb-cfg* and distributes user queries to the *mdb-rs0* Shard. In addition, all cluster members communicate with each other bi-directionally as part of the replication and gossip protocols. According to the topological distance analysis, MongoDB symptoms form a full graph similar to their source components. Consequently, coefficient values equalize, producing a value of 1 for each symptom dependency. As such, the topological distance analysis does not contribute to dependency filtering and designating dependency directions. Symptom dependencies are imposed strictly by symptom co-occurrence and time lag analyses.

In this case, even manually, it is challenging to infer the correct MongoDB symptom ordering. They co-occurred simultaneously, signaling the same MongoDB cluster defect. In addition, symptom sources were interdependent in a complicated way, i.e., MongoDB services formed a full graph. Therefore, considering the above difficulties, the inference algorithm produced satisfying results by grouping related MongoDB

symptoms and locating them appropriately in the fault trajectory. In fact, the ordering of MongoDB symptoms, except for the one emitted on the *mdb-mongos* service, does not make a significant contribution in explaining the root cause of failure.

Last, the *MongoDBHeartbeatFailedToHost1InReplicaSetRs0* symptom connects trajectory to the root cause symptom, i.e., *KubePodCrashLooping* on *mdb-rs0-1*. Again, the weak correlation coefficient value of 0.5 results from low coefficient values obtained in symptom co-occurrence and time lag analyses, returning the value of 0. The aggregated coefficient value is based solely on the strong correlation discovered in the topological distance analysis, which returned the coefficient value of 1. That provides evidence for the interchangeability of adopted symptom correlation methods. In situations where symptom dependency is valid, but some methods are infeasible, other methods compensate for the dependency strength. The non-deterministic system operation, which disturbed statistical algorithms employed in symptom co-occurrence and time lag analyses, was compensated by the topological distance analysis relying on deterministic information about system topology. Consequently, coefficient aggregation returned a non-zero coefficient value, and the dependency was included in the fault view graph for root cause analysis.

Moreover, including the week *KubePodCrashLooping* symptom dependency in the trajectory is the effect of inferring the symptom as the potential failure root cause. Mandatory connection of fault trajectories with potential root causes is another mechanism implemented in the inference algorithm that complements symptom correlation analysis.

Conclusively, the functional evaluation conducted on a live system returned satisfactory results, confirming the correctness of assumptions adopted in the dissertation and validating the solution against detecting root causes of defects in cloud-native applications. Despite the advanced application deployment and complex failure scenario, the synergy of symptom correlation analyses employed in the solution, combined with knowledge of system structure and symptom semantics enclosed in the RCA model, enabled determining the root cause and trajectory of a new failure instance.

8.4 Summary

This chapter presented the functional solution evaluation to verify if the proposed root cause analysis approach can be successfully applied to real-life failure scenarios simulated in cloud-native applications.

The functional evaluation confirms that the proposed RCA solution correctly identifies failure root causes and fault trajectories based on the holistic system view enclosed in constructed RCA model. The solution produced correct results for each failure scenario, including single and parallel faults, both semantically-dependent and semantically-independent. In addition, it was observed that different symptom correlation methods fulfill a significant part in each failure scenario category, proving the correctness of the adopted symptom diagnosis concept as the consolidation of several symptom correlation methods.

Moreover, the evaluation demonstrates that the solution meets the assumed RCA system dimensions and shows properties that were not achieved by existing works.

Chapter 9

Conclusion

The contribution of previous chapters covered a complete root cause analysis solution for cloud-native applications. This chapter summarizes the contribution, verifies the thesis statement, and discusses the potential directions of future work.

9.1 Thesis Verification

The formulated thesis statement expects the proposition of the root cause analysis solution, which can be used for localizing causes of defects emerging in cloud-native applications. The solution is supposed to enable the analysis of failures in the holistic context of system structure and behavior, taking into account symptom semantics and facing the high frequency of changes imposed by modern cloud systems. To effectively solve the stated problem, this dissertation addresses several important aspects.

First of all, the proposed solution resolves the issue of efficiently gathering and representing knowledge about system structure and semantics in alignment with system variability. More precisely, the novel concept presented in the dissertation introduces the RCA model comprising data structures and corresponding construction algorithms that enable updating the model incrementally with subsequent system changes. Notably, the emphasized automation and adaptability of model construction show to be crucial in ensuring the appropriate efficiency and accuracy of root cause inference.

According to concept realization, current and past system structure, i.e., system components and inter-component dependencies, is recorded in the system object dependency graph and enriched with observed symptoms in the process of fault symptom ingestion. Dependencies between components and symptoms are, in turn, designated based on attribute matching conditions constrained by the system object taxonomy structure. The resulting dependency graph proves to be a solid foundation for discovering deterministic symptom correlations reported on adjacent system components and a beneficial supplement to statistical-based correlation methods.

Further, symptom semantics are represented in the symptom co-occurrence map consisting of symptoms and mined semantic symptom dependencies discovered using the statistical method of event co-occurrence analysis. The decision to approximate semantic dependencies with temporal analysis produced satisfactory results, allowing the discovery of dependencies that could not be found using regular correlation methods employed in existing works. Moreover, the analysis of historical symptom data in the adopted co-occurrence method significantly reduces false positives resulting from plain correlations.

Another point achieved in solution realization is the capability of identifying causal dependencies between observed symptoms in root cause identification. For that purpose, the elaborated inference algorithm employs the symptom correlation framework. Its concept derives from the rule that correlation does not imply causality. Based on

that rule, the dissertation introduces an approach to symptom dependency mining by assuming that a causal relationship can be effectively approximated by consolidating several independent correlation methods. In this matter, the framework defines four symptom correlation analyses, namely topological distance analysis, co-occurrence analysis, time lag analysis, and time-series analysis, each considering a different subset of information about system structure and emerging symptoms.

Finally, the solution successfully extracts fault trajectories and identifies exact failure causes. The proposed inference algorithm concept leverages the symptom correlation framework to consolidate elaborated symptom correlation analyses. First, algorithm realization performs subsequent analyses to produce matrices of correlation coefficients from observed symptom pairs. Then, matrices are combined into an aggregated symptom correlation matrix, and a causality graph is constructed as a unified fault view. The causality graph comprises strong causal dependencies obtained using the approximation method and constitutes the main structure for performing root cause inference.

In addition, the dissertation discusses mechanisms required to extract failure root cause and fault trajectory. Precisely, degrees of symptom nodes in the graph determine potential failure causes and effects. Fault trajectories are, in turn, extracted by searching paths between designated nodes and ordered by proposed trajectory scoring and ranking criteria.

The thorough evaluation performed on the solution prototype led to several important conclusions. First, an in-depth solution verification based on synthetic data confirmed that the solution can correctly identify fault trajectories and root causes of single and parallel faults. Notably, the evaluation proved that in the case of parallel faults, root causes of both semantically-dependent and semantically-independent faults can be correctly determined. In addition, it was observed that different symptom correlation methods fulfill the critical part in each failure scenario. More precisely, symptom co-occurrence analysis took the leading part in semantically-independent failure scenarios, symptom time lag analysis guided semantically-dependent scenarios, while symptom topological distance analysis reinforced symptom dependencies at the level of system structure. That proves the effectiveness of the adopted concept of symptom diagnosis as the consolidation of symptom correlation methods.

Further, the functional evaluation conducted within a live system testbed showed that the proposed solution has a sufficient tolerance level for the non-deterministic aspects of system operation. Specifically, in a complex fault simulation involving a test microservice application integrated with a clustered database, the solution explained failure propagated from the database workload into the user-facing application layer

and indicated the correct root cause. The generated fault trajectory encompassed symptoms based on diverse telemetry data reported across several cloud complexity layers. Moreover, a significant contribution of the evaluation was confirming the feasibility of automating RCA model construction through integrating observability plane and cloud platform APIs as knowledge sources.

Importantly, the presented solution concept and realization meet all RCA system requirements established for cloud-native applications, including fault analysis across multiple cloud layers, holistic insight into system operation, recognition of symptom semantics, failure explainability, isolating parallel faults, and application-agnostic analysis.

The combination of formulated conclusions clearly illustrates that the contribution of this dissertation was not only sufficient for the successful verification of the thesis statement but also introduces a significant improvement to the current state-of-the-art in the domain of root cause analysis.

9.2 Future Work

It was proved that the proposed root cause analysis solution could be successfully used in the dynamic environment of cloud-native applications. However, several aspects could be improved in the scope of the future work:

- Symptom topological distance analysis proposed in Section 4.7 could be enriched by interpreting the properties of nodes along the path connecting symptom sources. Combined with graph metrics such as centrality or PageRank, the analysis could produce results that fully leverage system knowledge of inter-component dependencies.
- Consolidation of correlation methods in the realization of the inference algorithm could be improved by determining the contributions of individual methods intelligently, e.g., based on data availability or the context of a given failure. Currently, the consolidation is implemented using a weighted aggregation strategy with method weights configured manually, as detailed in Section 4.8.
- The fault view causality graph tends to grow in size in scenarios involving extensive failures. The current simplified approach, described in Section 6.6, assumes the graph can be reduced by filtering out weak dependencies below the configured threshold. However, that approach leads to either too mild or too sensitive filtering. The graph reduction could be solved in a more sophisticated way. For instance, the threshold could be selected using a statistical method

such as IQR, or a heuristic could be elaborated to designate causality subgraph holding symptoms relevant to the diagnosis.

- Elements from Bayesian theory [83] could be used to improve the quality of the symptom co-occurrence map. Specifically, algorithms related to Bayesian networks enable obtaining stronger causal properties of the graph structure, directing symptom dependencies, and redefining dependencies resulting from transitivity.

Appendix A

Prototype Evaluation: Graphical User Interface

This appendix presents screenshots illustrating exemplary GUI use cases as a continuation of the prototype description in Section 7.2.7.

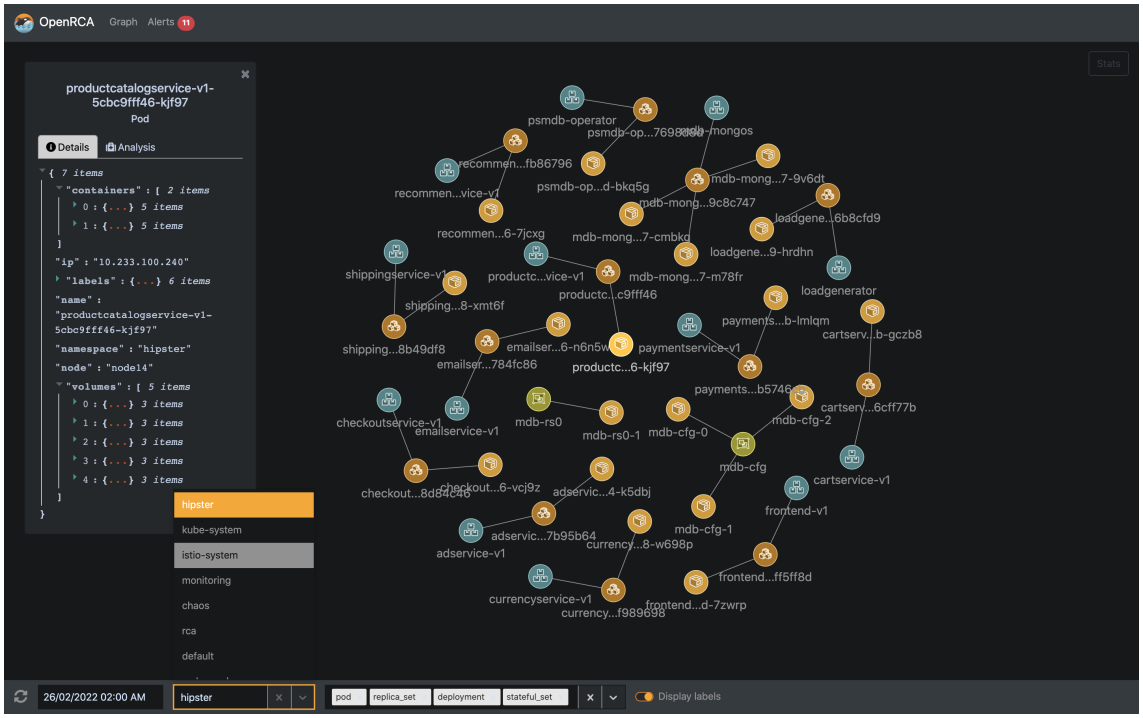


Figure A.1: Visualization of the system object dependency graph. The interface allows narrowing the analysis down to a single application by filtering the dependency graph by application namespace. In addition, the user can select an arbitrary subset of object types to be included in the graph.

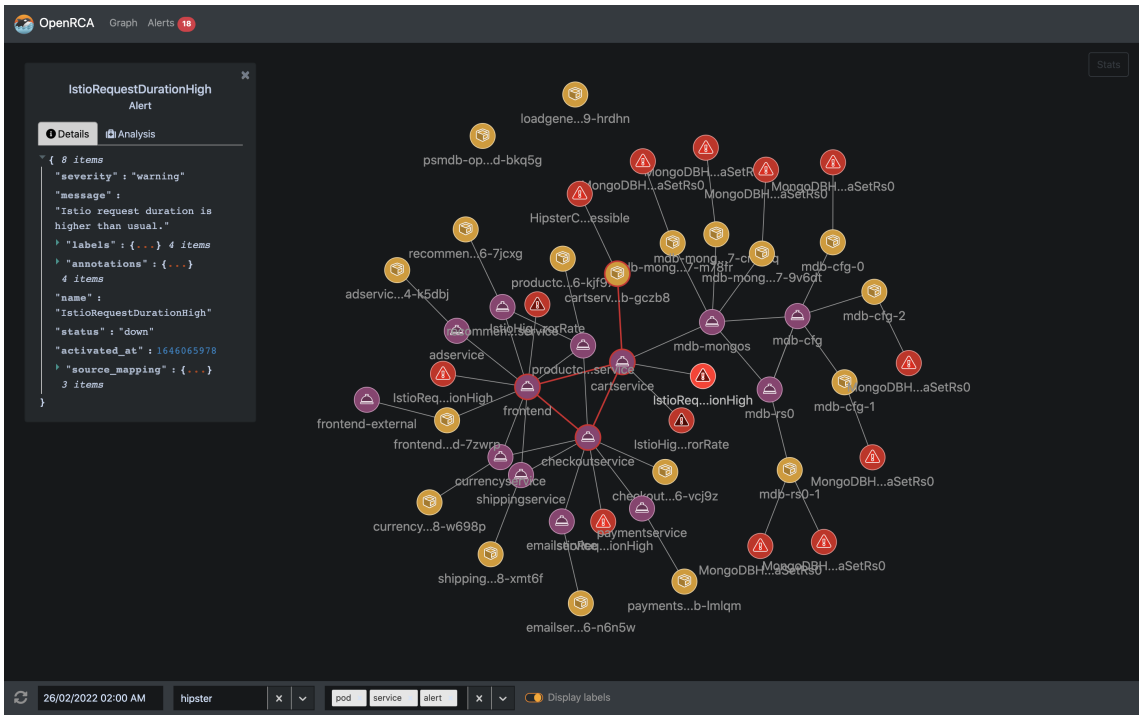


Figure A.2: The visualization exposes symptoms ingested into the dependency graph as part of the fault symptom ingestion process. Symptom nodes are marked in red. In addition, edges between source components are highlighted to indicate system regions affected by a failure.

Figure A.3: Illustration of dependency graph snapshotting. The feature can be controlled by a time selector located in the lower-left corner. On the left side, one can see the past state of the dependency graph at 5:00 AM, whereas on the right side, one can see the current graph state with active failure symptoms.

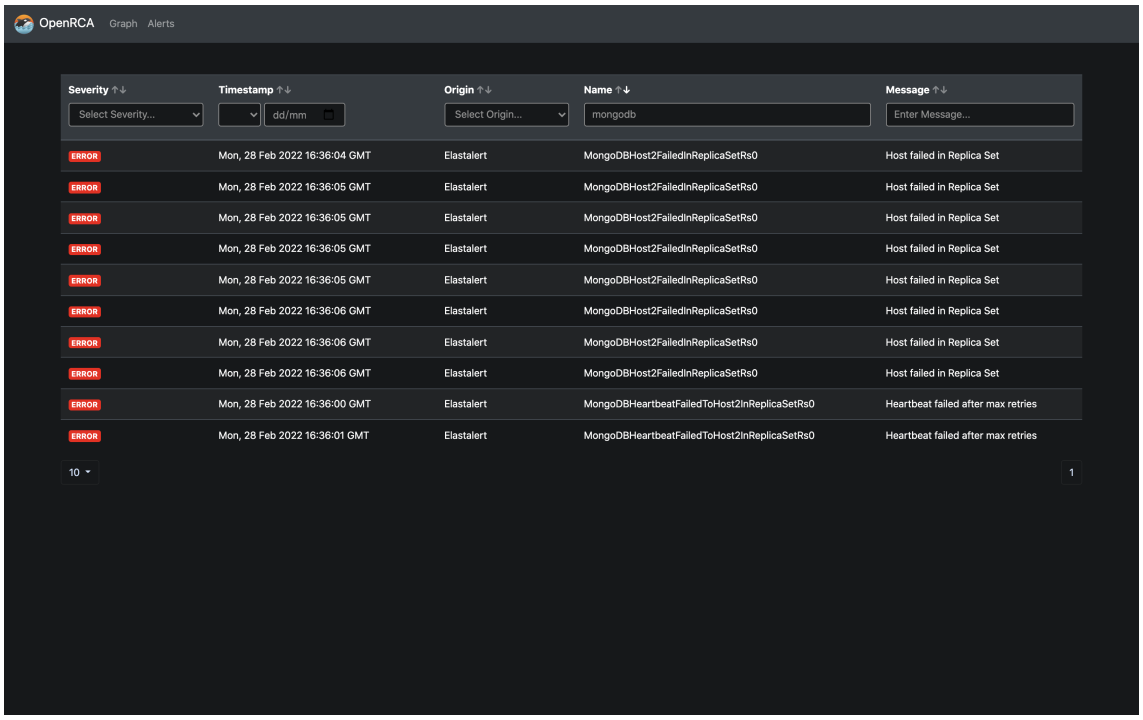


Figure A.4: The alerting panel summarizes active symptoms and allows filtering them by name, message, or severity.

List of Figures

List of Tables

List of Algorithms

Listings

Bibliography

[1] Nane Kratzke and Peter-Christian Quint. "Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study". In: *Journal of Systems and Software* 126 (Jan. 2017), pp. 1–16. DOI: 10.1016/j.jss.2017.01.001.

[2] Christoph Fehling et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8.

[3] Dmitry Namiot and Manfred sneps-sneppe. "On Micro-services Architecture". In: *Interenational Journal of Open Information Technologies* 2 (Sept. 1, 2014), pp. 24–27.

[4] Yale Yu, Haydn Silveira, and Max Sundaram. "A microservice based reference architecture model in the context of enterprise architecture". In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). Xi'an, China: IEEE, Oct. 2016, pp. 1856–1860. ISBN: 978-1-4673-9613-4. DOI: 10.1109/IMCEC.2016.7867539. URL: http://ieeexplore.ieee.org/document/7867539/ (visited on 04/22/2022).

[5] Vindeep Singh and Sateesh K Peddoju. "Container-based microservice architecture for cloud applications". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017 International Conference on Computing, Communication and Automation (ICCCA). Greater Noida: IEEE, May 2017, pp. 847–852. ISBN: 978-1-5090-6471-7. DOI: 10.1109/CCAA.2017.8229914. URL: http://ieeexplore.ieee.org/document/8229914/ (visited on 04/22/2022).

[6] WangHanzhang et al. "GRANO: interactive graph-based root cause analysis for cloud-native distributed data platform". In: *Proceedings of the VLDB Endowment* (Aug. 1, 2019). Publisher: VLDB Endowment PUB4722. URL:

https://dl-1acm-1org-100000dzv0053.wbg2.bg.agh.edu.pl/doi/abs/10.14778/3352063.3352105 (visited on 04/05/2020).

[7] Jörg Thalheim et al. "Sieve: Actionable Insights from Monitored Metrics in Microservices". In: *arXiv:1709.06686 [cs]* (Sept. 19, 2017). arXiv: 1709.06686. URL: http://arxiv.org/abs/1709.06686 (visited on 04/21/2022).

[8] Rami Sellami, Sami Bhiri, and Bruno Defude. "Supporting multi data stores applications in cloud environments". In: *IEEE Transactions on Services Computing* (2015), pp. 1–1. ISSN: 1939-1374. DOI: 10.1109/TSC.2015.2441703. URL: http://ieeexplore.ieee.org/document/7118237/ (visited on 04/22/2022).

[9] Bartosz Zurkowski and Krzysztof Zielinski. "Towards Self-Organizing Cloud Polyglot Database Systems". In: *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Umea, Sweden: IEEE, June 2019, pp. 82–87. ISBN: 978-1-72812-731-6. DOI: 10.1109/SASO.2019.00019. URL: https://ieeexplore.ieee.org/document/8780526/ (visited on 04/22/2022).

[10] Aniruddh M et al. "Comparison of Containerization and Virtualization in Cloud Architectures". In: *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. 2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). Bangalore, India: IEEE, July 9, 2021, pp. 1–5. ISBN: 978-1-66542-849-1. DOI: 10.1109/CONECCT52877.2021.9622668. URL: https://ieeexplore.ieee.org/document/9622668/ (visited on 04/22/2022).

[11] Maciej Gawel and Krzysztof Zielinski. "Analysis and Evaluation of Kubernetes Based NFV Management and Orchestration". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). Milan, Italy: IEEE, July 2019, pp. 511–513. ISBN: 978-1-72812-705-7. DOI: 10.1109/CLOUD.2019.00094. URL: https://ieeexplore.ieee.org/document/8814580/ (visited on 04/22/2022).

[12] Leila Abdollahi Vayghan et al. "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). San Francisco, CA, USA: IEEE, July 2018, pp. 970–973. ISBN: 978-1-5386-7235-8. DOI: 10.1109/CLOUD.2018.00148. URL: https://ieeexplore.ieee.org/document/8457916/ (visited on 04/22/2022).

[13] Jirayus Sithiyopasakul et al. "Automated Resource Management System based on Kubernetes Technology". In: *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. 2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON). Chiang Mai, Thailand: IEEE, May 19, 2021, pp. 1146–1149. ISBN: 978-1-66540-382-5. DOI: 10.1109/ECTI-CON51831.2021.9454911. URL: https://ieeexplore.ieee.org/document/9454911/ (visited on 04/22/2022).

[14] Mathias Meyer. "Continuous Integration and Its Tools". In: *IEEE Software* 31.3 (May 2014), pp. 14–16. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2014.58. URL: https://ieeexplore.ieee.org/document/6802994/ (visited on 04/22/2022).

[15] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53 (Apr. 1, 2010), pp. 50–58. DOI: 10.1145/1721654.1721672.

[16] Dhanya R Mathews et al. "Insights into Multi-Layered Fault Propagation and Analysis in a Cloud Stack". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). Chicago, IL, USA: IEEE, Sept. 2021, pp. 714–716. ISBN: 978-1-66540-060-2. DOI: 10.1109/CLOUD53861.2021.00092. URL: https://ieeexplore.ieee.org/document/9582267/ (visited on 04/22/2022).

[17] Marc Solé et al. "Survey on Models and Techniques for Root-Cause Analysis". In: *arXiv:1701.08546 [cs]* (July 3, 2017). arXiv: 1701.08546. URL: http://arxiv.org/abs/1701.08546 (visited on 09/21/2020).

[18] Soila P. Kavulya et al. "Failure Diagnosis of Complex Systems". In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by Katinka Wolter et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 239–261. ISBN: 978-3-642-29031-2 978-3-642-29032-9. DOI: 10.1007/978-3-642-29032-9_12. URL: http://link.springer.com/10.1007/978-3-642-29032-9_12 (visited on 09/22/2020).

[19] Ma łgorzata Steinder and Adarshpal S. Sethi. "A survey of fault localization techniques in computer networks". In: *Science of Computer Programming*. Topics in System Administration 53.2 (Nov. 1, 2004), pp. 165–194. ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.01.010. URL: http://www.sciencedirect.com/science/article/pii/S0167642304000772 (visited on 09/22/2020).

[20] A. Hanemann. "A hybrid rule-based/case-based reasoning approach for service fault diagnosis". In: *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*. 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06). Vienna, Austria: IEEE, 2006, 5 pp. ISBN: 978-0-7695-2466-5. DOI: 10.1109/AINA.2006.29. URL: http://ieeexplore.ieee.org/document/1620468/ (visited on 04/16/2022).

[21] Roxane Mody. "Automating Root-Cause Analysis: EMC Ionix Codebook Correlation Technology vs. Rules-based Analysis". In: (), p. 12.

[22] Zhengong Cai et al. "A Real-Time Trace-Level Root-Cause Diagnosis System in Alibaba Datacenters". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 142692–142702. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2944456.

[23] Li Wu et al. "MicroRCA: Root Cause Localization of Performance Issues in Microservices". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. Budapest, Hungary: IEEE, Apr. 2020, pp. 1–9. ISBN: 978-1-72814-973-8. DOI: 10.1109/NOMS47738.2020.9110353. URL: https://ieeexplore.ieee.org/document/9110353/ (visited on 04/13/2021).

[24] Jian-Guang Lou et al. "Mining dependency in distributed systems through unstructured logs analysis". In: *Operating Systems Review* 44 (Mar. 12, 2010), pp. 91–96. DOI: 10.1145/1740390.1740411.

[25] A Aghasaryan et al. "Modeling Fault Propagation in Telecommunications Networks for Diagnosis Purposes". In: (), p. 8.

[26] Jinjin Lin, Pengfei Chen, and Zibin Zheng. "Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments". In: *Service-Oriented Computing*. International Conference on Service-Oriented Computing. Springer, Cham, Nov. 12, 2018, pp. 3–20. DOI: 10.1007/978-3-030-03596-9_1. URL: https://link-1springer-1com-1000048zv005b.wbg2.bg.agh.edu.pl/chapter/10.1007/978-3-030-03596-9_1 (visited on 04/05/2020).

[27] Zijie Guan, Jinjin Lin, and Pengfei Chen. "On Anomaly Detection and Root Cause Analysis of Microservice Systems". In: *Service-Oriented Computing – ICSOC 2018 Workshops*. International Conference on Service-Oriented Computing. Springer, Cham, Nov. 12, 2018, pp. 465–469. DOI: 10.1007/978-3-030-17642-6_45. URL: https://link-1springer-1com-1000048zv005b.wbg2.bg.agh.edu.pl/chapter/10.1007/978-3-030-17642-6_45 (visited on 04/05/2020).

[28] Álvaro Brandón et al. "Graph-based root cause analysis for service-oriented and microservice architectures". In: *Journal of Systems and Software* 159 (Jan. 1, 2020), p. 110432. ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.110432. URL: http://www.sciencedirect.com/science/article/pii/S0164121219302067 (visited on 04/05/2020).

[29] Mazda Marvasti et al. "An Anomaly Event Correlation Engine: Identifying Root Causes, Bottlenecks, and Black Swans in IT Environments". In: *VMware Technical Journal* 2 (June 11, 2013), pp. 35–45.

[30] Hanzhang Wang et al. "Groot: An Event-graph-based Approach for Root Cause Analysis in Industrial Settings". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, Australia: IEEE, Nov. 2021, pp. 419–429. ISBN: 978-1-66540-337-5. DOI: 10.1109/ASE51524.2021.9678708. URL: https://ieeexplore.ieee.org/document/9678708/ (visited on 04/19/2022).

[31] Armen Aghasaryan, Makram Bouzid, and Dimitre Kostadinov. "Stimulus-based sandbox for learning resource dependencies in virtualized distributed applications". In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN). Paris: IEEE, Mar. 2017, pp. 238–245. ISBN: 978-1-5090-3672-1. DOI: 10.1109/ICIN.2017.7899419. URL: http://ieeexplore.ieee.org/document/7899419/ (visited on 07/01/2020).

[32] Juan Qiu et al. "A Causality Mining and Knowledge Graph Based Method of Root Cause Diagnosis for Performance Anomaly in Cloud Applications". In: *Applied Sciences* 10 (Mar. 22, 2020), p. 2166. DOI: 10.3390/app10062166.

[33] Di Wu et al. "Detecting Leaders from Correlated Time Series". In: Apr. 1, 2010, pp. 352–367. DOI: 10.1007/978-3-642-12026-8_28.

[34] Chen Luo et al. "Correlating events with time series for incident diagnosis". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*. the 20th ACM SIGKDD international conference. New York, New York, USA: ACM Press, 2014, pp. 1583–1592. ISBN: 978-1-4503-2956-9. DOI: 10.1145/2623330.2623374. URL: http://dl.acm.org/citation.cfm?doid=2623330.2623374 (visited on 10/25/2020).

[35] Marc-Andre Zöller, Marcus Baum, and Marco Huber. "Framework for Mining Event Correlations and Time Lags in Large Event Sequences". In: July 1, 2017. DOI: 10.1109/INDIN.2017.8104876.

[36] Marc-André Zöller. "Identification of Correlations and Root Causes in Event Sequences". In: (), p. 92.

[37] Marco Huber, Marc-Andre Zöller, and Marcus Baum. "Linear programming based time lag identification in event sequences". In: *Automatica* 98 (Sept. 17, 2018), pp. 14–19. DOI: 10.1016/j.automatica.2018.08.025.

[38] Hamid Reza Motahari-Nezhad et al. *Event correlation for process discovery from web service interaction logs*. 2010.

[39] Yi Cai et al. "Event Relationship Analysis for Temporal Event Search". In: Lecture Notes in Computer Science. Apr. 1, 2013. ISBN: 978-3-642-37449-4. DOI: 10.1007/978-3-642-37450-0_13.

[40] Chunqiu Zeng et al. "Mining temporal lag from fluctuating events for correlation and root cause analysis". In: *Proceedings of the 10th International Conference on Network and Service Management, CNSM 2014* (Jan. 16, 2015), pp. 19–27. DOI: 10.1109/CNSM.2014.7014137.

[41] Yunyue Zhu, Dennis Shasha, and Yunyue Zhu Dennis Shasha. "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time". In: *In VLDB*. 2002, pp. 358–369.

[42] Ting Wang et al. "Spatio-temporal patterns in network events". In: Proceedings of the 6th International Conference on Emerging Networking Experiments and Technologies, Co-NEXT'10. Jan. 1, 2010, p. 3. DOI: 10.1145/1921168.1921172.

[43] Mustafa Demetgul. "Fault diagnosis on production systems with support vector machine and decision trees algorithms". In: *The International Journal of Advanced Manufacturing Technology* 67 (Aug. 1, 2012). DOI: 10.1007/s00170-012-4639-5.

[44] Olumuyiwa Ibidunmoye, Francisco Hernandez-Rodriguez, and Erik Elmroth. "Performance Anomaly Detection and Bottleneck Identification". In: *ACM Computing Surveys* 48 (June 1, 2015). DOI: 10.1145/2791120.

[45] Samaneh Aminikhanghahi and Diane J. Cook. "A Survey of Methods for Time Series Change Point Detection". In: *Knowledge and information systems* 51.2 (May 2017), pp. 339–367. ISSN: 0219-1377. DOI: 10.1007/s10115-016-0987-z. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5464762/ (visited on 02/03/2021).

[46] Gerrit J. J. van den Burg and Christopher K. I. Williams. "An Evaluation of Change Point Detection Algorithms". In: *arXiv:2003.06222 [cs, stat]* (May 25, 2020). arXiv: 2003.06222. URL: http://arxiv.org/abs/2003.06222 (visited on 02/03/2021).

[47] Charles Truong, Laurent Oudre, and Nicolas Vayatis. "Selective review of offline change point detection methods". In: *Signal Processing* 167 (Feb. 2020), p. 107299. ISSN: 01651684. DOI: 10.1016/j.sigpro.2019.107299. URL: https://linkinghub.elsevier.com/retrieve/pii/S0165168419303494 (visited on 03/25/2021).

[48] A. J. Scott and M. Knott. "A Cluster Analysis Method for Grouping Means in the Analysis of Variance". In: *Biometrics* 30.3 (Sept. 1974), p. 507. ISSN: 0006341X. DOI: 10.2307/2529204. URL: https://www.jstor.org/stable/2529204?origin=crossref (visited on 08/12/2021).

[49] Christian Rohrbeck. "Detection of changes in variance using binary segmentation and optimal partitioning". In: (), p. 6.

[50] Brad Jackson et al. "An Algorithm for Optimal Partitioning of Data on an Interval". In: *IEEE Signal Processing Letters* 12.2 (Feb. 2005), pp. 105–108. ISSN: 1070-9908. DOI: 10.1109/LSP.2001.838216. arXiv: math/0309285. URL: http://arxiv.org/abs/math/0309285 (visited on 08/12/2021).

[51] R. Killick, P. Fearnhead, and I. A. Eckley. "Optimal detection of changepoints with a linear computational cost". In: *Journal of the American Statistical Association* 107.500 (Dec. 2012), pp. 1590–1598. ISSN: 0162-1459, 1537-274X. DOI: 10.1080/01621459.2012.737745. arXiv: 1101.1438. URL: http://arxiv.org/abs/1101.1438 (visited on 02/03/2021).

[52] Kaylea Haynes, Idris A. Eckley, and Paul Fearnhead. "Computationally Efficient Changepoint Detection for a Range of Penalties". In: *Journal of Computational and Graphical Statistics* 26.1 (Jan. 2, 2017), pp. 134–143. ISSN: 1061-8600, 1537-2715. DOI: 10.1080/10618600.2015.1116445. URL: https://www.tandfonline.com/doi/full/10.1080/10618600.2015.1116445 (visited on 08/07/2021).

[53] Alexandre Lung-Yut-Fong, Céline Lévy-Leduc, and Olivier Cappé. "Homogeneity and change-point detection tests for multivariate data using rank statistics". In: *arXiv:1107.1971 [math, stat]* (Feb. 9, 2012). arXiv: 1107.1971. URL: http://arxiv.org/abs/1107.1971 (visited on 08/07/2021).

[54] Marek Psiuk and Krzysztof Zielinski. "Goal-driven adaptive monitoring of SOA systems". In: *Journal of Systems and Software* 110 (Dec. 2015), pp. 101–121. ISSN: 01641212. DOI: 10.1016/j.jss.2015.08.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215001764 (visited on 09/05/2021).

[55] H.-J. Kettschau, S. Bruck, and P. Schefczik. "LUCAS - an expert system for intelligent fault management and alarm correlation". In: *NOMS 2002. IEEE/IFIP Network Operations and Management Symposium. ' Management Solutions for the New Communications World'(Cat. No.02CH37327)*. NOMS 2002IEEE/IFIP Network Operations and Management Symposium. Florence, Italy: IEEE, 2002, pp. 903–905. ISBN: 978-0-7803-7382-2. DOI: 10.1109/NOMS.2002.1015639. URL: http://ieeexplore.ieee.org/document/1015639/ (visited on 08/12/2021).

[56] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. "Discovering Frequent Episodes in Sequences". In: (), p. 6.

[57] Tao Li et al. "An integrated framework on mining logs files for computing system management". In: Jan. 1, 2005, pp. 776–781. DOI: 10.1145/1081870.1081972.

[58] Sheng Ma and Joseph Hellerstein. "Mining mutually dependent patterns". In: Feb. 1, 2001, pp. 409–416. ISBN: 978-0-7695-1119-1. DOI: 10.1109/ICDM.2001.989546.

[59] Chunqiu Zeng et al. "Mining temporal lag from fluctuating events for correlation and root cause analysis". In: *Proceedings of the 10th International Conference on Network and Service Management, CNSM 2014* (Jan. 16, 2015), pp. 19–27. DOI: 10.1109/CNSM.2014.7014137.

[60] Maria L. Rizzo and Gábor J. Székely. "Energy distance". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 8.1 (Jan. 2016), pp. 27–38. ISSN: 19395108. DOI: 10.1002/wics.1375. URL: http://doi.wiley.com/10.1002/wics.1375 (visited on 06/13/2021).

[61] Paul Besl and H.D. McKay. "A method for registration of 3-D shapes. IEEE Trans Pattern Anal Mach Intell". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 14 (Mar. 1, 1992), pp. 239–256. DOI: 10.1109/34.121791.

[62] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/358669.358692. URL: https://dl.acm.org/doi/10.1145/358669.358692 (visited on 05/22/2022).

[63] Markus M Breunig et al. "LOF: Identifying Density-Based Local Outliers". In: (), p. 12.

[64] Huan Wang et al. "A density-based clustering structure mining algorithm for data streams". In: *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining Algorithms, Systems, Programming Models and Applications - BigMine '12*. the 1st International Workshop. Beijing, China: ACM Press, 2012, pp. 69–76. ISBN: 978-1-4503-1547-0. DOI: 10.1145/2351316.2351326. URL: http://dl.acm.org/citation.cfm?doid=2351316.2351326 (visited on 02/05/2022).

[65] Erich Schubert et al. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". In: *ACM Transactions on Database Systems* 42.3 (Aug. 24, 2017), pp. 1–21. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/3068335. URL: https://dl.acm.org/doi/10.1145/3068335 (visited on 02/05/2022).

[66] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. New York: Routledge, Oct. 25, 2017. 176 pp. ISBN: 978-1-315-14091-9. DOI: 10.1201/9781315140919.

[67] Thomas M Cover and Joy A Thomas. "ELEMENTS OF INFORMATION THEORY". In: (), p. 774.

[68] Gerlof Bouma. "Normalized (Pointwise) Mutual Information in Collocation Extraction". In: (), p. 11.

[69] "VII. Note on regression and inheritance in the case of two parents". In: *Proceedings of the Royal Society of London* 58.347 (Dec. 31, 1895), pp. 240–242. ISSN: 0370-1662, 2053-9126. DOI: 10.1098/rspl.1895.0041. URL: https://royalsocietypublishing.org/doi/10.1098/rspl.1895.0041 (visited on 08/12/2021).

[70] Agustin Garcia Asuero, Ana Sayago, and Gustavo González. "The Correlation Coefficient: An Overview". In: *Critical Reviews in Analytical Chemistry - CRIT REV ANAL CHEM* 36 (Jan. 1, 2006), pp. 41–59. DOI: 10.1080/10408340500526766.

[71] U Kang et al. "Centralities in Large Networks: Algorithms and Observations". In: *Proceedings of the 2011 SIAM International Conference on Data Mining*. Proceedings of the 2011 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, Apr. 28, 2011, pp. 119–130. ISBN: 978-0-89871-992-5 978-1-61197-281-8. DOI: 10.1137/1.9781611972818.11. URL: https://epubs.siam.org/doi/10.1137/1.9781611972818.11 (visited on 05/25/2022).

[72] Lawrence Page et al. "The PageRank Citation Ranking : Bringing Order to the Web". In: *undefined* (1999). URL: https://www.semanticscholar.org/paper/The-PageRank-Citation-Ranking-%3A-Bringing-Order-to-Page-Brin/eb82d3035849cd23578096462ba419b53198a556 (visited on 05/25/2022).

[73] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001. ISBN: 0262032937.

[74] Robert Nickerson et al. "Taxonomy Development In Information Systems: Developing A Taxonomy Of Mobile Applications". In: *HAL, Working Papers* (2009).

[75] Ali Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (May 2016), pp. 35–41. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2016.60. URL: https://ieeexplore.ieee.org/document/7436642/ (visited on 02/05/2022).

[76] Haley Tucker et al. "The Business Case for Chaos Engineering". In: *IEEE Cloud Computing* 5.3 (May 2018), pp. 45–54. ISSN: 2325-6095. DOI: 10.1109/MCC.2018.032591616. URL: https://ieeexplore.ieee.org/document/8383672/ (visited on 02/05/2022).

[77] Kennedy A. Torkura et al. "CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure". In: *IEEE Access* 8 (2020), pp. 123044–123060. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3007338. URL: https://ieeexplore.ieee.org/document/9133399/ (visited on 02/05/2022).

[78] Petter Holme and Jari Saramäki. "Temporal Networks". In: *Physics Reports* 519.3 (Oct. 2012), pp. 97–125. ISSN: 03701573. DOI: 10.1016/j.physrep.2012.03.001. arXiv: 1108.1780. URL: http://arxiv.org/abs/1108.1780 (visited on 03/02/2022).

[79] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: (), p. 11.

[80] Benjamin Steer, Felix Cuadrado, and Richard Clegg. "Raphtory: Streaming analysis of distributed temporal graphs". In: *Future Generation Computer Systems* 102 (Jan. 1, 2020), pp. 453–464. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.08.022. URL: http://www.sciencedirect.com/science/article/pii/S0167739X19301621 (visited on 04/05/2020).

[81] Raymond Cheng et al. "Kineograph: taking the pulse of a fast-changing and connected world". In: *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*. the 7th ACM european conference. Bern, Switzerland: ACM Press, 2012, p. 85. ISBN: 978-1-4503-1223-3. DOI: 10.1145/

2168836.2168846. URL: http://dl.acm.org/citation.cfm?doid=2168836.2168846 (visited on 03/02/2022).

[82] Joseph E Gonzalez, Yucheng Low, and Haijie Gu. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: (), p. 14.

[83] Irad Ben-Gal. "Bayesian Networks". In: *Encyclopedia of Statistics in Quality and Reliability*. Journal Abbreviation: Encyclopedia of Statistics in Quality and Reliability. Mar. 15, 2008. ISBN: 978-0-470-06157-2. DOI: 10.1002/9780470061572.eqr089.