UNIVERSIDADE D
COIMBRA

Jessica Maciel de Castro

Methods and Tools for Security Attack Detection in Microservices

Thesis in the context of the Informatics Engineering, Programming Methodologies and Software Engineering, advised by Professor Marco Vieira and Professor Nuno Laranjeiro and presented at Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

March 2026

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Jessica Maciel de Castro

Methods and Tools for Security Attack Detection in Microservices

PhD Thesis submitted to the University of Coimbra
Advised by Professor Marco Vieira and Professor Nuno Laranjeiro.

March 2026

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Jessica Maciel de Castro

Métodos e Ferramentas para Deteção de Ataques de Segurança em Microsserviços

Tese de Doutoramento submetida à Universidade de Coimbra
Orientada pelo Professor Marco Vieira e Professor Nuno Laranjeiro.

Março 2026

The work presented in this thesis was carried out within the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC) in the context of the following projects:

- **AIDA: Adaptive, Intelligent and Distributed Assurance Platform** co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalization COMPETE 2020 (POCI-01-0247-FEDER-045907) and by the Portuguese Foundation for Science and Technology under CMU Portugal Program.
- **NEXUS - Pacto de Inovação Transição Verde e Digital para Transportes, Logística e Mobilidade** financed by the Portuguese Recovery and Resilience Plan (PRR), no.C645112083-00000059 (investment project no. 53)
- **Power - Empowering a digital future** co-financed by the ERDF and COMPETE 2020 (POCI-01-0247-FEDER-070365).

This work has been partially supported by the following grants:

- **Ph.D. grant**: **FCT - Fundação para a Ciência e Tecnologia**, I.P. by project reference and DOI identifier https://doi.org/10.54499/2023.00207.BD.
- **Research grant**: **AIDA - POCI-01-0247-FEDER-045907**; Grant number: DPA-20-620.
- **Research grant**: **POWER - POCI-01-0247-FEDER-070365** Grant number: IT137-22-199.

Acknowledgements

Completing this PhD has been a long journey, and I am deeply grateful to the many people who supported me along the way.

First and foremost, I would like to express my sincere gratitude to my supervisors, Marco and Nuno, for their guidance, support, and encouragement throughout this work. Their ideas and confidence were fundamental to the development of this research and to my growth as a researcher.

I would also like to thank Katerina for her collaboration and for the valuable discussions and contributions that enriched this research.

My gratitude extends to all members of the SSE group, CISUC, University of Coimbra, and especially to my friends at the Laprie Laboratory (SSE G4.5). Thank you for the conversations, domino games, and companionship that made the experience even more meaningful.

I would like to thank all my friends. Special thanks to my friends in João Pessoa, whose friendship, even from a distance, has always been a source of encouragement and joy.

I am deeply grateful to my family for their unconditional support. A very special thank you to my mother, whose love, sacrifices, and constant encouragement have always been a guiding force in my life.

Finally, I want to thank my wife. She embarked on this journey with me, moving from Brazil to Portugal and standing by my side through every challenge. Her understanding and unwavering support gave me the strength to persevere through the most difficult times. This achievement would not have been possible without her.

To all of you, thank you for being part of this journey.

Statement on the use of artificial intelligence tools

This document was reviewed and refined with the assistance of Large Language Models, such as ChatGPT, or similar tools, which helped check grammar, correct typos, and enhance clarity. Any Artificial Intelligence-generated text or content is clearly indicated within the document and is used only to a limited extent. The overall content and ideas remain solely the responsibility of the author.

Abstract

Modern software systems increasingly rely on microservice architectures to achieve continuous availability, scalability, and flexibility. Despite these advantages, microservice applications introduce security challenges that are difficult to address due to their distributed nature, dynamism, and fine granularity. A single application may comprise dozens or even hundreds of services, exposing numerous access points, generating substantial volumes of data, and operating in a highly ephemeral environment in which service instances are continuously created and terminated. Consequently, such applications are exposed to a range of threats, including Denial-of-Service (DoS) attacks, which can exploit their dynamism and elasticity to blend into legitimate traffic, making detection particularly challenging.

Several challenges arise in the design, implementation, and evaluation of runtime attack detection approaches, including the lack of representative, reproducible datasets to support model development and evaluation, the difficulty of extracting meaningful features from heterogeneous, ephemeral monitoring data, and limited comparability due to inconsistent experimental setups across prior work. **This thesis addresses these challenges by advancing the state of the art in runtime security monitoring and analysis of microservice applications through integrated approaches, models, and tools.**

The main contribution is a comprehensive framework to support research on the cybersecurity of microservice applications and to develop methods for detecting cyberattacks. The framework comprises two modules: (i) a data generation module that contains the components necessary to create datasets that reflect the behavior of microservices under attack, and (ii) a model development and evaluation module that supports the design, training, and assessment of different attack detection methods for microservices. In practice, the framework establishes a modular and platform-agnostic structure for workload generation, attack profiling, dataset construction, and the evaluation of detection models.

The frameworks data generation module is instantiated in a testbed built around a microservice application with a fixed architecture and predefined services (i.e., no dynamic service creation or scaling). We build an experimental setup using the TeaStore microservice application, where user workloads are emulated, and diverse attack profiles are injected, including high- and low-volume DoS attacks. This resulted in the release of a publicly available, labeled dataset that serves as the basis for subsequent attack-detection studies.

Building on this dataset and the proposed framework, we explore the applicability of Logic Scoring of Preference (LSP). This multi-criteria decision-making method computes a score from a set of preferences for attack detection in microservice applications. Concretely, we develop a baseline LSP model and systematically compare different techniques for defining its elements (i.e., weights, operators, and means). The model outputs a single score used to determine whether a microservice application is under a DoS attack. The experimental re-

sults show precision, recall, and F1-score values above 80%, indicating that LSP can effectively characterize the application under attack.

To investigate additional methods for detecting attacks in microservice applications, we use our framework to train and evaluate a wide range of supervised and unsupervised Machine Learning (ML) algorithms (RF, XGB, KNN, SVM, CNN, AE, VAE, OCSVM, LOF, and ISOF). The study systematically compares model performance across different DoS attack types and profiles, revealing trade-offs in performance metrics and data requirements. The results provide actionable guidance on selecting appropriate ML techniques depending on application constraints and detection goals. Supervised ML models achieve the best classification performance (particularly XGBoost). Although unsupervised ML and LSP models perform worse, they remain suitable when attack data are unavailable or costly to generate.

Finally, to extend detection capabilities to dynamic environments, we address DoS attack detection in scalable microservice applications. We propose a novel multimodal, hierarchical anomaly detection model that integrates data from multiple monitoring sources and performs fusion at the instance-, service-, and application-levels to detect DoS attacks. Autoencoder models with different architectures are employed at these three hierarchical levels. The multimodal model is designed to handle dynamic conditions (e.g., workload variability and autoscaling) and uses unsupervised learning to detect both high- and low-volume DoS attacks. An extensive experimental evaluation enables a thorough analysis of the impact of different workload, attack, and attack-type profiles on detection performance. The results show that our model outperforms single-modality approaches, achieving reliable detection performance.

Keywords

Microservices, Security, Attack Detection, Anomaly Detection, DoS.

Resumo

Sistemas de software modernos dependem cada vez mais de arquiteturas de microsserviços para fornecer disponibilidade contínua, escalabilidade e flexibilidade. Apesar das vantagens, aplicações de microsserviços introduzem novos desafios de segurança que são difíceis de resolver devido à sua natureza distribuída, dinamicidade e fina granularidade. Uma única aplicação pode possuir dezenas ou centenas de serviços, expondo diversos pontos de acesso, gerando grandes volumes de dados e operando num ambiente altamente efêmero, onde instâncias dos serviços são constantemente criadas e terminadas. Consequentemente, estas aplicações são alvo de várias ameaças, como ataques de Negação de Serviço (DoS), que exploram o dinamismo e elasticidade para se camuflarem entre o tráfego legítimo, dificultando a deteção dos ataques.

Há, consequentemente, vários desafios ao longo de toda a linha de pesquisa em deteção de ataques em aplicações de microsserviços em tempo de execução: uma ausência de datasets representativos e reprodutíveis para apoiar o desenvolvimento e avaliação de modelos; é difícil extrair informação relevante devido à heterogeneidade, granularidade, efemeridade e distribuição dos serviços; a comparação com o estado da arte também é limitada pela falta de consistência experimental em trabalhos anteriores. **Esta tese lida com todos esses desafios, avançando o estado da arte de segurança de aplicações microsserviços na monitorização e análise de segurança em tempo de execução, via abordagens, modelos e ferramentas integrados.**

A primeira contribuição principal é a *framework* abrangente para apoiar a investigação sobre a cibersegurança das aplicações de microsserviços e para desenvolver métodos de deteção de ataques cibernéticos. A *framework* contém dois módulos: (i) um módulo de geração de dados que contém os componentes necessários para criar conjuntos de dados que refletem o comportamento de microsserviços quando atacados e (ii) um módulo de desenvolvimento e avaliação de módulos que dão suporte a conceção, treino e avaliação de diferentes métodos para detetar ataques em microsserviços. A *framework* define uma estrutura modular e agnóstica à plataforma para geração de workload, perfis de ataques, construção de conjuntos de dados, e avaliação de modelos de deteção.

O módulo de geração de dados da *framework* é instanciado e nós apresentamos uma *testbed* para uma aplicação de microsserviços estática (i.e., sem criação ou escalonamento dinâmico de serviços). Nós detalhamos a configuração experimental a utilizar TeaStore, um aplicação de microsserviços, onde nós emulamos a geração de dados de usuários e injetamos diferentes perfis de ataque, incluindo ataques de DoS de alto-volume e baixo-volume. Essa campanha resultou na publicação de um conjunto de dados rotulado, o que serve como dados para desenvolver os estudo de deteção de ataque subsequentes.

Construindo a partir desse conjunto de dados e da framework proposta, o estudo seguinte explora a aplicabilidade de *Logic Scoring of Preference* (LSP), um método de tomada de decisão multicritério para computar uma pontuação baseada em um conjunto de preferências, para deteção de ataques em aplicações de mi-

crosserviços. Para isso, desenvolvemos um modelo base e comparamos diferentes técnicas para definir os elementos que compõem um modelo LSP (i.e., pesos, operadores, e médias). A saída do modelo é uma pontuação única que é usada para determinar se a aplicação está sofrendo um ataque DoS. Os resultados do estudo experimental mostraram as taxas de *precision*, *recall*, *e f1-score* com mais de 80%, a indicar que LSP pode caraterizar efetivamente a aplicação sob ataque.

Para pesquisar outros métodos que podem ser utilizados para desenvolver modelos para detetar ataques em aplicações de microsserviços, nós usamos nossa *framework* para treinar e avaliar uma ampla gama de algoritmos de aprendizagem de máquina supervisionados e não-supervisionados (RF, XGB, KNN, SVM, CNN, AE, VAE, OCSVM, LOF, and ISOF). O estudo compara performance de modelos através de diferentes tipos e perfis de ataque DoS, revelando compensações entre as métricas de performance e os requisitos de dados. Os resultados fornecem orientações práticas sobre a seleção de técnicas de aprendizagem de máquina apropriadas, dependendo das restrições da aplicação e dos objetivos da deteção. Os resultados indicam que modelos supervionados tem melhor performance, especialmente com o algortimo XGBoost. Mesmo que algorithmos não supervisionados e modelos de LSP tenham pior performance, eles pode ser utilizados quando dados de ataque não estão disponíveis ou são custosos para gerar.

Finalmente, para extender a capacidade de deteção para ambientes dinamicos, nós lidamos com deteção de ataques DoS em aplicações de microsserviços escalonáveis. Nós propomos um novo modelo de deteção de anomalia hierárquico e multimodal que integra dados de múltiplas fontes de monitoramento e realiza a fusão de dados nos níveis de instância, serviço e aplicação. Modelos de autoencoders com diferentes arquiteturas são utilizadas para esses três níveis de hierarquia. O modelo multimodal é projetado para lidar com condições dinâmicas (ex., variação de carga de usuário e autoescalonamento), e usa aprendizado não-supervisionados para detetar ataques DoS de alto volume e baixo volume. Uma avaliação experimental extensiva suporta uma análise minuciosa do impacto na performance de diferentes perfis de carga de usuário, e perfis e tipos de ataques. Os resultados mostram que nosso modelo supera modelos com uma única modalidade, apresentando uma performance de deteção confiável.

Palavras-Chave

Microsserviços, Segurança, Deteção de Ataques, Deteção de Anomalias, DoS.

The contributions of this thesis resulted in the following publications in international peer-reviewed conferences and journals:

- Castro, J., Goseva-Popstojanova, K., Laranjeiro, N., & Vieira, M. (2026). Detecting DoS Attacks in Scalable Microservices using Multimodal Anomaly Detection. Submitted to *IEEE International Conference on Web Services (ICWS)*.
- Castro, J., Laranjeiro, N., Goseva-Popstojanova, K., & Vieira, M. (2025). Developing Attack Detection Models for Microservice Applications: A Comprehensive Framework and Its Illustration and Validation on DoS Attacks. *IEEE Transactions on Dependable and Secure Computing*. DOI: 10.1109/TDSC.2025.3590197.
- Castro, J., Laranjeiro, N., & Vieira, M. (2023, July). Exploring Logic Scoring of Preference for DoS Attack Detection in Microservice Applications. In *2023 IEEE International Conference on Web Services* (pp. 573-584). DOI: 10.1109/ICWS60048.2023.00076.
- Castro, J., Laranjeiro, N., & Vieira, M. (2023, October). Generating Realistic Attack Data for Microservices: Framework and Case Study. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing* (pp. 60-69). DOI: 10.1145/3615366.3615377

Two additional publications with preliminary results were published during this Ph.D. work:

- Castro, J., Laranjeiro, N., & Vieira, M. (2022, November). Detecting DoS Attacks in Microservice Applications: Approach and Case Study. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing* (pp. 73-78). DOI: 10.1145/3569902.3569916.
- Castro, J., Laranjeiro, N., & Vieira, M. (2023, June). Techniques and Tools for Runtime Security Monitoring and Analysis of Microservices. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)* (pp. 191-193). DOI: 10.1109/DSN-S58398.2023.00052

Contents

Acronyms

AE Autoencoder.

AitM Adversary-in-the-Middle.

API Application Programming Interface.

APM Application Performance Management.

CNN Convolutional Neural Network.

CSV Comma-Separated Values.

DDoS Distributed Denial of Service.

DoS Denial of Service.

EDoS Economic Denial of Sustainability.

ELOOCV Experiment-wise Leave-One-Out Cross-Validation.

FN False Negatives.

FP False Positives.

GAI Generative AI.

gRPC Remote Procedure Call.

HPA Horizontal Pod Autoscaling.

ISOF Isolation Forest.

JSON JavaScript Object Notation.

KNN K-Nearest Neighbors.

LOF Local Outlier Factor.

LSP Logic Scoring of Preference.

MCDM Multi-Criteria Decision-Making.

ML Machine Learning.

OCSVM One-Class SVM.

OECD Organisation for Economic Co-operation and Development.

REST Representational State Transfer.

RF Random Forest.

SOA Service-Oriented Architecture.

SOAP Simple Object Access Protocol.

SVM Support Vector Machines.

TN True Negatives.

TP True Positives.

VAE Variational Autoencoder.

VM Virtual Machine.

VMS Virtual Machines.

WAM Weighted Arithmetic Means.

WHM Weighted Harmonic Means.

WPM Weighted Power Means.

XGB XGBoost.

List of Figures

List of Tables

Chapter 1

Introduction

Modern society is built upon a foundation of complex software systems. These systems deliver essential public services, support the management of global financial markets, process and store sensitive personal information, and coordinate critical business workflows. Properties like reliability, availability, and security of this digital infrastructure have direct and profound economic and societal consequences. In this context, system downtime or data breaches are no longer minor inconveniences; they can translate directly into significant revenue loss, erosion of public trust, irreversible reputational damage, and, especially in the case of critical services, lead to severe and direct harm to public safety [Chen et al., 2022; OECD, 2019; Security and Institute, 2024].

To meet the accelerating demands for massive scale, continuous deployment, and high resilience, the software industry has undergone a fundamental architectural shift. The rigid, slow-to-change monolithic architectures of the past proved incapable of supporting the agility required for modern digital services [Auer et al., 2021]. In response, organizations have widely embraced cloud-native technologies, container-based services, and orchestration platforms (e.g., Kubernetes [Cloud Native Computing Foundation, 2022]). Above all, they have adopted microservices as the new architectural standard for building and managing complex, large-scale applications [Chen et al., 2025; Wang et al., 2021].

The microservices paradigm has become widespread in software development due to its ability to manage and reduce the complexity of developing and maintaining computer systems [Wang et al., 2020]. This architectural style empowers developers to construct highly scalable, flexible, and manageable systems [Yu et al., 2019]. Instead of a single, tightly-coupled application, a microservice application is composed of several small, independent services. Each service runs in its own process, implements a distinct business function, and communicates with others through lightweight, network-exposed mechanisms, such as Representational State Transfer (REST) Application Programming Interface (API) [Neumann et al., 2018]. Services can be implemented using different technologies and can be deployed, updated, and deleted individually [Chen et al., 2022].

While offering numerous advantages, the microservices architectural style has altered the threat environment. The same properties that make microservices ap-

pealing involve inherent trade-offs, introducing novel and complex security challenges that monolithic systems did not face. For example, the fine service granularity and distributed nature mean that a single application may expose tens or even hundreds of network endpoints, compared to the single, well-defined perimeter of a monolith. This dramatically increases the attack surface available to malicious actors [Haindl et al., 2024]. Furthermore, the application's functionality is provided through the interplay of numerous inter-service dependencies and communications, which creates new risks, including the challenge of securing all service-to-service communication and the possibility of cascading failures, where the malfunction of a single small service can propagate to affect the entire application [Ding et al., 2024].

Beyond this expanded attack surface, microservices operate in a highly dynamic and ephemeral runtime environment. Service instances (e.g., containers) are short-lived, frequently rescheduled, and have rapidly changing network identities. This dynamism makes traditional monitoring, logging, and attack analysis exceptionally difficult [Chen et al., 2022]. Moreover, the freedom to implement each service using different technologies can lead to a fragmented and inconsistent security defense. Heterogeneous tech stacks broaden the range of potential vulnerabilities and make it more difficult to maintain a uniform, robust defense across all services [Flora and Antunes, 2024].

Among the many security threats affecting modern distributed systems, attacks such as Denial of Service (DoS) [Pacheco et al., 2024], API abuse, and other application-layer exploitation pose significant risks. In microservice applications, these threats can be particularly disruptive due to the architecture's distributed, highly dynamic nature. Microservices rely on network-exposed communication and frequently adjust capacity through mechanisms such as autoscaling, factors that make it even more challenging to distinguish malicious activity from normal operational fluctuations.

1.1 Problem Statement

The microservice architecture introduces operational dynamics that create significant ambiguity for attack detectors. In this context, DoS attacks are particularly harmful because their effects often resemble those produced by legitimate workload increases. For instance, a sudden, legitimate peak in client load (a "flash crowd") triggers similar application-level symptoms to those seen during a DoS attack, such as high CPU usage, increased request rates, or elevated internal communication traffic [Caculo et al., 2020]. This ambiguity is exacerbated by autoscaling, a key mechanism used by microservice applications to handle fluctuating client load. While in a monolithic system a DoS attack produces obvious overload effects (e.g., high CPU or memory usage), in a microservice application, autoscaling mechanisms can mask the attack's impact from end users by provisioning more resources, all while the attack continues to succeed in its goal.

Attackers can exploit this behavior to induce Economic Denial of Sustainability (EDoS) [Chamberlain et al., 2025] where the system is flooded with malicious

traffic that mimics legitimate load. The system, behaving as designed, scales up new service instances in response, incurring substantial and unexpected financial costs for the organization without improving or maintaining service quality for genuine users [Bremler-Barr et al., 2024]. This legitimate scaling behavior introduces rapid changes in resource consumption and service composition, which attackers can leverage to conceal malicious activity. As a result, distinguishing attacks from legitimate workload fluctuations becomes significantly more difficult in such highly dynamic environments, further confounding detection models that are not designed to operate under elastic conditions. Addressing these operational complexities demands a new generation of runtime security monitoring and analysis techniques and tools. However, the realization of such solutions is hindered by four fundamental challenges that span across every stage of the research pipeline.

First, there is a critical lack of representative and reproducible datasets [Cao et al., 2022; Flora et al., 2023; Odusami et al., 2020]. This challenge arises even before any model can be trained or evaluated. Existing benchmarks and public datasets mostly target monolithic systems or low-level network traffic [Alshaibi et al., 2022; Mvula et al., 2023; Sharafaldin et al., 2019] and fail to capture the complex, dynamic interactions, layered architecture, and scaling behavior associated with microservices. Researchers are often forced to build their own testbeds and synthetic data generators [Ficco et al., 2025; Flora and Antunes, 2024; Jacob et al., 2022], which differ widely in configuration and assumptions. As a result, studies cannot be fairly compared, and progress in the field is fragmented. What is missing is a structured framework of techniques and tools that defines how workloads (i.e., genuine work to be carried out by the application), attacks, and monitoring data should be produced, collected, and shared to foster reproducibility.

Second, once data is generated, a new challenge emerges: understanding and extracting value from the data itself. A running microservice application generates a large volume of heterogeneous data, including system resource, container, and service state information, as well as inter-service communication patterns, from hundreds of distributed, ephemeral instances of services [Grohmann et al., 2019; Waseem et al., 2021]. Determining which features are actually informative for detecting DoS behavior among these heterogeneous data remains a demanding task. The high dimensionality and temporal variability complicate feature engineering: selecting too few attributes risks missing important signals, while selecting too many introduces noise and redundancy.

The third challenge concerns the complexity of developing attack detection models [Chen et al., 2022]. The effectiveness of any model hinges upon the quality of its measurable inputs. This immediately raises critical questions about feature engineering and use: developers must decide whether to rely on raw features, engineered indicators (e.g., rates of change, statistical aggregations), or composite features derived from monitoring data across multiple, interconnected services. Furthermore, the lack of consensus leads to the exploration of diverse detection methodologies, which vary significantly in terms of how they utilize features and their resulting performance characteristics.

Several methodologies can be employed to address this challenge. Supervised learning methods typically deliver strong predictive performance, but they require large volumes of labeled data [Zhou, 2018]. In microservice environments, however, obtaining accurate and timely labels is labor-intensive, and trained models often fail to generalize beyond the specific workloads and attack scenarios observed during training. On the other hand, unsupervised anomaly detection approaches learn normal system behavior and flag deviations, which makes them attractive for detecting previously unseen attacks; however, their sensitivity to legitimate workload fluctuations, such as rapid changes in traffic volume, resource consumption, or service composition, often results in unstable performance and elevated false-positive rates [Chamberlain et al., 2025; Soldani and Brogi, 2022]. Finally, Multi-Criteria Decision-Making (MCDM) methods, such as Logic Scoring of Preference (LSP) [Dujmovic, 2018], offer explicit interpretability by encoding expert knowledge through structured aggregation mechanisms, but their effectiveness depends on well-calibrated attributes and parameter configurations, which may not transfer well across different applications or deployment contexts.

The fourth and last challenge is that even when models are developed, their evaluation and comparison lack consistency. Assessing the performance of a detection model in microservice applications is inherently difficult: different studies rely on different metrics, use heterogeneous workloads, or evaluate models under dissimilar conditions. Moreover, microservice applications exhibit substantial variability across services, deployments, and scaling configurations, meaning that a model performing well for one application or workload profile may behave very differently for another. Evaluating a model under both static and elastic conditions further complicates comparisons, as autoscaling can significantly change the behavior of both benign and malicious traffic. As a result, the field lacks a coherent understanding of which techniques are most effective under specific conditions, highlighting the urgent need for standardized evaluation frameworks.

The central problem is not only "how to detect DoS attacks in microservices," but to address the entire methodological and technical pipeline. Therefore, research is needed to develop novel solutions for securing microservice applications at runtime. Specifically, it is necessary to identify which monitoring features are most relevant for detecting DoS attacks and how detection models should analyze these features. Furthermore, detection approaches must be systematically developed, evaluated, and compared to determine which methods are best suited for identifying attacks in microservice environments.

1.2 Contributions

This thesis advances the state of the art in security monitoring and analysis for microservice applications by **proposing a comprehensive framework and systematically evaluating multiple attack-detection approaches**. These approaches range from LSP-based MCDM techniques to Machine Learning (ML)-based methods. In addition, it introduces a hierarchical, multimodal anomaly detection

model tailored to the elastic and distributed nature of microservice environments, with particular emphasis on detecting DoS attacks.

In summary, the key contributions of this thesis are:

1. **A comprehensive framework for microservice attack data generation, model development, and evaluation.** This contribution introduces and implements a modular framework composed of two main modules: Data Generation and Model Development and Evaluation. The framework structures the generation of realistic attack datasets and the systematic development and evaluation of detection models for microservice applications. It supports reproducible experimentation and serves as the methodological foundation for all subsequent contributions. This work resulted in the publication: *"Developing Attack Detection Models for Microservice Applications: A Comprehensive Framework and Its Illustration and Validation on DoS Attacks"* [Castro et al., 2025].

2. **An LSP-based detection approach designed to microservice applications.** The LSP method is, for the first time, employed as a tool to build attack detection models. To this end, we propose a new approach to applying the method to create a *Tree Structure* that can be used to build attack-detection models for microservice applications, accounting for their specificities, such as the varying number of services within a single microservice application and the possibility of replicas. To assess applicability, we define a base model and compare several methods for defining the individual score-aggregation elements (weights, operators, and the mean). This contribution led to the publication: *'Exploring logic scoring of preference for DoS attack detection in microservice applications'* [Castro et al., 2023a].

3. **Development, evaluation and comparison of several supervised and unsupervised ML-based algorithms for the detection of DoS attacks on microservice applications.** This empirical study provides the community with actionable recommendations and a clear understanding of the performance trade-offs (e.g., accuracy, data requirements, interpretability, and tuning effort) associated with each approach, guiding practitioners and researchers in selecting the appropriate method for their specific context. This contribution led to the publication: *'Developing Attack Detection Models for Microservice Applications: A Comprehensive Framework and Its Illustration and Validation on DoS Attacks'* [Castro et al., 2025].

4. **A hierarchical, multimodal anomaly detection model for scalable microservices.** A three-level hierarchical model that fuses four monitoring data modalities (system resource features, network traffic, service/instance state, and service interactions) across instance, service, and application levels. It is specifically designed to handle autoscaling and to successfully detect both high- and low-volume DoS attacks under dynamic workload conditions. This contribution led to the submission: *Detecting DoS Attacks in Scalable Microservices using Multimodal Anomaly Detection* to ICWS 2026.

5. **Publicly available experimental artifacts to support reproducibility.** All datasets, scripts, workload profiles, attack profiles, and model implementations developed in the experimental studies are made publicly available. These artifacts provide a reusable foundation for future research, enabling independent validation, extension, and comparison of detection techniques for microservice security. This contribution led to publication: *'Generating Realistic Attack Data for Microservices: Framework and Case Study'* [Castro et al., 2023b], and submission: *Detecting DoS Attacks in Scalable Microservices using Multimodal Anomaly Detection* to ICWS 2026.

1.3 Outline of the Thesis

This thesis is structured as a logical progression through interconnected studies. The remainder of this thesis is structured as follows.

Chapter 2 provides a comprehensive review of the literature. It reviews key concepts related to microservice applications, security, and relevant detection methods, including ML and MCDM, such as the LSP.

Chapter 3 presents the first contribution: a comprehensive, structured framework for generating realistic attack datasets, and developing and evaluating detection models. This framework provides the methodological foundation for the subsequent chapters.

Chapter 4 details the instantiation of the framework's Data Generation module. It describes the experimental testbed for a static (non-scaling) microservice application and the process of generating a labeled dataset of DoS attacks.

Chapter 5 addresses the Model Development and Evaluation module of the framework. It analyzes the applicability of the LSP method as a new approach for DoS detection in microservice applications. It explains the rationale for selecting LSP, the design of attribute criteria and aggregation structures, and the evaluation of models under different operators and weighting methods. For this, it uses the dataset from Chapter 4.

Chapter 6 also addresses the Model Development and Evaluation module of the framework. It develops and evaluates a range of supervised and unsupervised ML models using the same dataset from Chapter 4. It concludes with a comparative analysis of the performance, strengths, and weaknesses of supervised ML, unsupervised ML, and LSP-based models.

Chapter 7 presents the last study. Motivated by the limitations of existing models in handling elasticity, this chapter proposes a novel, hierarchical, and multimodal anomaly detection model. This model is validated on a new, more complex dataset generated from a scalable microservice environment.

Chapter 8 concludes the thesis by summarizing the key findings from all chapters and outlining directions for future work.

Chapter 2

Background and Related Work

This chapter presents a comprehensive review of the state of the art in securing microservice-based applications. It consolidates the fundamental concepts underlying microservice architectures, analyzes the challenges associated with attack detection in distributed and dynamic environments, and surveys existing detection techniques relevant to this context. Representative studies are discussed, including work on DoS detection, anomaly detection, supervised and unsupervised learning approaches, and multimodal models.

The chapter begins by introducing the principles of microservice architecture in Section 2.1, including deployment models, scalability mechanisms, benchmarking applications, monitoring infrastructures, and workload modeling. In Section 2.2, it then examines security in microservice environments, outlining key security principles, challenges, and common attack types, with particular emphasis on DoS attacks. Next, Section 2.3 reviews methodological foundations for attack detection in microservices, covering classification performance metrics, the Logic Scoring of Preference (LSP) method, supervised and unsupervised Machine Learning (ML) techniques, and multimodal approaches. Section 2.4 concludes with a summary of the main insights.

2.1 Microservice Architecture

The evolution of software systems toward large-scale, highly resilient, and continuously delivered platforms has fundamentally driven the adoption of microservice architecture as a dominant paradigm in modern software engineering [Gannon et al., 2017; Velepucha and Flores, 2023]. This architectural style provides the operational and structural context for the runtime security and attack detection solutions investigated in this thesis.

2.1.1 Fundamentals of Microservices

At its core, microservice architecture is grounded in the Service-Oriented Architecture (SOA) principle of composing complex systems from small, loosely coupled services, each encapsulating a single business capability [Wang et al., 2021]. A microservice-based application is therefore composed of multiple autonomous services that can be developed, tested, deployed, and maintained independently. Each service may employ different programming languages, frameworks, and data storage technologies, enabling teams to select the most suitable tools for specific functional requirements [Fowler and Lewis, 2014]. Microservices typically execute in isolated processes and communicate via lightweight, network-exposed mechanisms, most commonly RESTful Application Programming Interface (API) [Neumann et al., 2018], but also gRPC, message-oriented middleware (e.g., RabbitMQ), or WebSocket [Çiftçi and Çiloğlugil, 2025; Gannon et al., 2017].

In contrast, monolithic architectures represent the traditional approach to software development, in which presentation logic, business rules, and data access are tightly coupled and deployed as a single unit. Microservices, by design, emphasize loose coupling, independent deployability, and bounded contexts. As illustrated in Figure 2.1, functionality that is co-located within a single monolithic deployment is decomposed into distinct services in a microservice architecture. This decomposition enables independent evolution of services: modifying or redeploying one service does not require redeployment of the entire system, thereby reducing operational risk while significantly increasing deployment velocity and overall organizational agility [Fowler and Lewis, 2014].

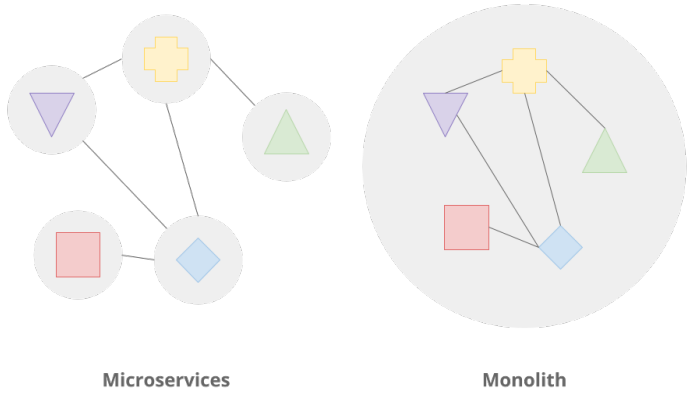


Figure 2.1: Comparison between Microservices and Monolith architecture.

Microservices exhibit several advantageous properties that have driven their widespread adoption in cloud-native systems [Mateus-Coelho et al., 2021; Taibi et al., 2018; Velepucha and Flores, 2023]:

- **Scalability:** services can be scaled independently based on their individual resource demands, enabling fine-grained horizontal scaling and avoiding the overprovisioning typically required in monolithic systems.
- **Simplified Maintenance:** as each service is relatively small and focused on a single responsibility, its codebase is easier to understand, test, and maintain throughout the entire software development lifecycle.

- **Technology Heterogeneity:** each service may use the programming language, framework, or database that best fits its functionality, promoting architectural flexibility.
- **Failure Isolation:** service boundaries limit the blast radius of failures: a malfunctioning service is less likely to compromise the entire application, especially when supported by resilience patterns such as retries, circuit breakers or timeouts.
- **Continuous Deployment:** the ability to deploy and roll back services independently supports rapid iterative development and DevOps practices.
- **Independent Feature Evolution:** new features or fixes can be introduced in a single service without modifying or redeploying unrelated components.

2.1.2 Characteristics, Deployment and Scalability

Microservice architectures rely on a set of operational and infrastructural characteristics that shape how services are built, deployed, interact, scale, and evolve at runtime. These characteristics stem primarily from the architecture's distributed nature. Microservices are commonly packaged as containers using technologies such as Docker, which provide lightweight isolation, reproducibility of runtime environments, and simplified dependency management [Gannon et al., 2017]. Containerization enables consistent deployment across heterogeneous hosts, accelerates development workflows, and improves resource isolation. Moreover, multiple container instances can be launched on a single physical server or Virtual Machine (VM) with minimal startup overhead, reducing deployment costs and increasing elasticity [Liu et al., 2020].

Deploying a microservice application typically involves building container images, managing them in registries (e.g., Docker Hub [Docker, Inc., 2025]), and orchestrating their execution using platforms such as Kubernetes [Cloud Native Computing Foundation, 2022]. Orchestration platforms automate deployment, scheduling, scaling, and recovery of service instances. The Kubernetes platform, in particular, provides the following key features:

- **Pod and Container Management:** a service instance (also known as a replica) runs as pods that may be redeployed, restarted, or rescheduled across worker nodes automatically.
- **Declarative Configuration:** the desired system state (e.g., number of replicas, resource limits, service endpoints) is defined declaratively, and Kubernetes continuously reconciles the actual state.
- **Self-Healing:** failed containers are restarted automatically, and unhealthy pods are replaced without human intervention.
- **Horizontal Pod Autoscaling (HPA):** services scale dynamically based on workload metrics such as CPU, memory, or custom application indicators.

These orchestration capabilities are tightly coupled with the inherent scalability of microservice architectures. Microservice architectures are inherently scalable, enabling efficient resource allocation and dynamic adaptation to fluctuating workloads. To achieve this elasticity, two primary scaling strategies are commonly employed [Blinowski et al., 2022]:

- **Horizontal scaling:** involves adding or removing service instances to accommodate changing demand. This is the dominant approach in microservice systems due to its flexibility and compatibility with container orchestration platforms. When traffic increases, additional instances are activated; when demand decreases, instances are terminated to conserve resources.
- **Vertical scaling:** involves increasing the computational resources (e.g., CPU, memory, storage) allocated to a single service instance. Although less common in microservices, vertical scaling may be necessary for compute-intensive services or those with strict performance requirements.

Beyond these strategies, microservice systems commonly employ **load balancing**, which distributes incoming traffic across multiple instances to prevent overloading a single component and to ensure efficient resource utilization and low latency, as well as **autoscaling**, which dynamically adjusts the number of instances in response to workload fluctuations, maintaining performance during peak demand while optimizing resource usage during periods of low activity.

2.1.3 Workloads for Microservice Applications

The term workload broadly refers to the set of all inputs and service requests received by a computing system over a specific period of time [Calzarossa et al., 2016]. A workload profile describes how requests arrive at a system and how they evolve, capturing characteristics such as request rate, concurrency level, and temporal variability. In microservice applications, workload profiles must capture not only external request patterns but also internal service-to-service interactions, which may involve request exits, service chaining, and asynchronous communication [Ueda et al., 2016].

Realism in workload profiling is particularly critical for microservice systems because small changes in external demand can trigger disproportionately large variations in internal system activity. For example, a single user request may result in multiple downstream service invocations, each with distinct resource requirements and communication overheads. Inter-service communication, therefore, plays a central role in shaping workload profiles in microservice architectures. Consequently, realistic workload models must account not only for external request patterns but also for communication intensity, dependency structures, and service interaction graphs [Ueda et al., 2016].

When analyzing workload profiles in microservice environments, it is important to note that workload dynamics directly influence service scaling behavior. Depending on the workload's intensity and temporal evolution, services may scale

up or down dynamically. However, due to complex service dependencies and non-linear interactions, system performance is often difficult to reproduce deterministically across experiments. The same workload profile may lead to different runtime behaviors depending on scheduling, resource contention, and autonomous orchestration decisions.

Workload profiles can be classified according to their temporal characteristics. These workload intensity types are common to both monolithic and microservice architectures; however, their impact can differ significantly depending on how applications are structured and deployed. Following [Ding and Huang, 2021], three primary workload intensity types are distinguished:

- **Gentle workloads:** these are characterized by a relatively steady and constant request arrival rate over time. Gentle workloads are commonly used to evaluate baseline system behavior under stable operating conditions and to assess steady-state resource utilization.
- **Periodic workloads:** periodic workloads exhibit recurring patterns in request intensity, often driven by time-dependent factors such as daily usage cycles, business hours, or scheduled batch operations. These workloads are useful for studying system behavior under predictable load variations and for evaluating scaling and resource management strategies. Although periodic workloads may involve rising and declining demand, such changes typically occur gradually and reflect normal application usage patterns.
- **Bursty workloads:** bursty workloads, including flash-crowd scenarios, are characterized by sudden and short-lived spikes in request rates. These workloads are particularly relevant for stress testing, as they can expose transient performance bottlenecks, overload conditions, and cascading failures that may not manifest under steady or periodic loads.

Different workload types stress microservice-based systems in distinct ways, primarily due to the distributed execution of services and the dynamic nature of scaling mechanisms. In scalable microservice applications, workload fluctuations may trigger frequent scale-up and scale-down events, affecting resource allocation, service placement, and inter-service communication patterns. Runtime environments are also affected differently by workload characteristics. Under steady workloads, runtimes may reach stable operating states, whereas periodic and bursty workloads can induce frequent changes in execution paths, memory usage, and scheduling behavior.

A substantial body of research on microservice scalability and resource management relies on real-world traces and publicly available datasets to construct workload intensity profiles. These traces naturally exhibit variability, burstiness, and long-tailed distributions that are representative of production environments.

Wikipedia access traces are among the most frequently used datasets for modeling workload intensity in microservice research. These traces capture real user access patterns over extended periods and reflect both periodic behavior and sudden popularity spikes. For example, [Ding and Huang, 2021] traces Wikipedia

back to 20072008 as the primary source of workloads for evaluating autoscaling decisions under time-varying demand. Similarly, [Zhang et al., 2022] relies on Wikipedia traces to model delay-sensitive service workloads and study scaling responsiveness. In [Shafi et al., 2024] and [Xie et al., 2024], Wikipedia traces are combined with other datasets to construct heterogeneous workloads, enabling the evaluation of scaling mechanisms under mixed traffic conditions.

The WorldCup dataset, also known as the FIFA World Cup 1998 trace, records HTTP requests to the official World Cup website during the tournament. This dataset is well known for its highly bursty behavior, driven by match schedules and sudden surges in user interest. [Shafi et al., 2024] and [Abdullah et al., 2020] incorporate WorldCup traces alongside other workloads to expose autoscaling systems to extreme spikes and rapid load changes. Likewise, [Bali et al., 2024] uses WorldCup traces in its evaluation to assess proactive scaling strategies during sudden traffic surges. Finally, [Xiao and Hu, 2022] uses multiple workload patterns derived from this dataset (e.g., slowly rising, rapidly rising, and smoothly fluctuating) to validate a deep reinforcement learning-based horizontal autoscaler for Kubernetes-deployed microservices.

The NASA traces dataset originate from web servers at NASA research centers and represent scientific and institutional access patterns from 1995. Compared to WorldCup traces, NASA datasets typically exhibit more regular access behavior with moderate variability. [Shafi et al., 2024], [Bali et al., 2024], and [Abdullah et al., 2020] include NASA traces in their workload construction, often in combination with other datasets. Additional classic datasets include the ClarkNet and Calgary traces (from 1995 and 1990, respectively), which capture access patterns to an Internet service provider and a university website. These datasets typically show moderate request rates with daily cycles and are frequently used in workload characterization studies. In [Shafi et al., 2024], ClarkNet and Calgary traces are used alongside Wikipedia, WorldCup, and NASA datasets to construct composite workloads.

It is important to note that many of the aforementioned datasets originate from the 1990s and early 2000s. While they remain valuable for studying workload variability and burstiness, they may not fully reflect the behavior of modern cloud-native applications, which exhibit different usage patterns, interaction styles, and traffic characteristics. In addition to trace-driven workloads, several studies rely on synthetic workload generation and benchmark-based profiles to define workload intensity. These approaches are commonly used when real-world data are unavailable, when controlled experimentation is required, or when specific workload patterns must be reproduced.

Abdullah et al. [Abdullah et al., 2020] presented four synthetic workloads with distinct characteristics, including different burst patterns and peak timings. Xie et al. [Xie et al., 2024] extended this approach by defining five workload profiles based on the workloads proposed by Abdullah et al.. Synthetic workloads allow precise control over arrival rates, burst durations, and intensity levels, facilitating systematic evaluation of scaling mechanisms and validation of resource management strategies under various stress conditions.

Beyond workload intensity and request arrival patterns, realistic workload generation for microservice-based applications requires explicit modeling of user behavior. In modern distributed systems, particularly in web-based applications, system load is not solely determined by the number of requests, but also by how users interact with the application, the actions they perform, and the sequence in which they perform them.

Empirical studies have shown that user behavior strongly influences the structure of the workload, its temporal dynamics, and resource-demand patterns [Calzarossa et al., 2016]. In microservice architectures, where a single user action may trigger multiple service invocations across different services, the impact of user behavior on system load is further amplified [Ueda et al., 2016].

Real user behavior is inherently heterogeneous and difficult to model. Users exhibit diverse navigation paths, session lengths, and interaction frequencies, and their actions are often influenced by external factors such as time of day, social trends, or application-specific events. Web workloads are typically session-based and heavy-tailed, with a large number of short sessions and a small number of very long sessions [Janevski and Goseva-Popstojanova, 2013].

In contrast, many experimental evaluations rely on synthetic user models, which represent users through predefined roles or behavioral profiles. These roles differ not only in the number of requests they generate but also in the structure, length, and dependency of their sessions. For example, a buyer session may involve a complex sequence of dependent actions (e.g., browsing, adding items to a cart, completing a purchase). In contrast, a visitor session may consist of only a few independent page views. Such distinctions are important because they lead to different patterns of service invocation and resource utilization in microservice-based systems [Avritzer et al., 2020].

In web application workloads, user modeling is commonly expressed in terms of roles or behavioral profiles. Typical examples include [Avritzer et al., 2020; Janevski and Goseva-Popstojanova, 2013]:

- **Browsers or visitors**, who primarily navigate pages and consume content.
- **Buyers or customers**, who follow transaction-oriented paths such as searching for products, adding items to a cart, and completing purchases.
- **Order users**, who mainly access the system to check completed orders or delivery status.

Many benchmark microservice applications already provide predefined user profiles and workload scripts. For example, TeaStore provides Browse and Buy request profiles [Von Kistowski et al., 2018], and Sock Shop offers scripts that simulate user traffic across the full functionality of the application [Microservices-demo, 2018]. These built-in profiles facilitate realistic workload generation and reproducible experimentation.

2.1.4 Benchmarking Applications

Several representative benchmark applications have been developed to support research on microservice architectures. These benchmarks aim to reflect realistic application structures, technology stacks, and deployment environments, while remaining easy to use, instrument, and integrate with monitoring and load generation tools [Detti et al., 2023; Fischer et al., 2024; von Kistowski et al., 2025].

Sock Shop is a microservice-based demonstration application for online sock retail [Microservices-demo, 2018]. It is designed to showcase and test microservice and cloud-native technologies and is implemented using heterogeneous technologies, including Go kit and Node.js. The application comprises eight microservices: frontend, orders, payment, user (database), catalogue (database), cart, shipping, and queue-master. Sock Shop has been widely used in research on intrusion detection, software aging, and other domains.

TrainTicket [Zhou et al., 2018] is a microservice-based railway ticketing system consisting of 24 business-logic services (41 services in total). Users can search for tickets, book seats, make payments, receive notifications, and modify bookings. The application has been widely used in microservice research, particularly in fault-injection and failure-analysis works, as it provides built-in failure scenarios.

TeaStore is a Microservice-based application for benchmarking, performance modeling, and research on resource management [Von Kistowski et al., 2018]. It consists of five core services (WebUI, Auth, Image-Provider, Persistence, and Recommender), a Registry service, and a database. Each service can be replicated, added, or removed at runtime. The WebUI service provides the frontend interface and orchestrates interactions with the Image-Provider, Auth, Persistence, and Recommender services. The Image-Provider supplies images in different resolutions, Auth handles authentication and session management, Persistence manages database access and caching, and Recommender generates product recommendations based on purchase history. The Registry maintains service discovery information, including IP addresses and ports.

As illustrated in Figure 2.2, during startup, the Image-Provider retrieves images from Persistence, and the Recommender trains its model using order history. At runtime, WebUI coordinates user requests, while all service instances periodically send heartbeats to the Registry. In this thesis, TeaStore is adopted as the primary experimental application due to its realistic architecture, extensibility, and widespread use in microservice research.

DeathStarBench [Gan et al., 2019] is an open-source microservice benchmark suite comprising several large-scale, end-to-end applications, including a social network (36 microservices), media service (38 microservices), e-commerce platform (41 microservices), banking system (34 microservices), and IoT coordination applications for UAV swarms. These applications are implemented in multiple programming languages (e.g., Java, Python, C++, Node.js) and represent complex, real-world service interactions.

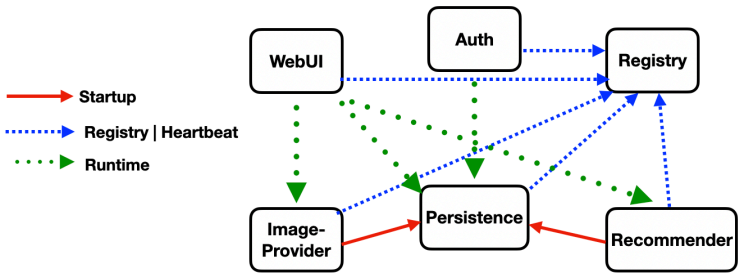


Figure 2.2: Architecture of TeaStore (adapted from [Von Kistowski et al., 2018]).

2.1.5 Monitoring Microservices

Monitoring is a fundamental requirement for ensuring the robustness, performance, and security of microservice-based systems. It serves multiple purposes, including performance analysis, fault diagnosis, root-cause analysis, and security monitoring. From a performance perspective, monitoring enables continuous observation of system attributes such as response time, service availability, and resource utilization. These metrics support the identification of bottlenecks and capacity limitations, enabling proactive optimization. Monitoring is also essential for diagnosing failures by collecting logs and traces that capture execution events and inter-service interactions. Crucially, monitoring is central to attack and anomaly detection, which is the primary focus of this thesis. By learning normal system behavior and detecting deviations, such as abnormal CPU usage or request patterns, monitoring data can be used to identify security threats and system malfunctions [Faseeha et al., 2025].

Microservice monitoring is commonly structured around three data types: metrics, logs, and traces. Metrics capture quantitative performance and resource utilization data, and logs record textual event information, including errors and transactions. Traces provide detailed end-to-end request paths across services, supporting dependency analysis and root-cause diagnosis. In this thesis, monitored metrics are referred to as features when used as input to ML models, and as attributes when used in LSP models.

Monitoring can be performed at multiple levels: (i) network level, observing inter-service communication; (ii) system level, monitoring the underlying infrastructure (physical or virtual machines); and (iii) service level, focusing on application-layer metrics such as response time and CPU usage. Service-level monitoring is particularly relevant for detecting application-layer Denial of Service (DoS) attacks.

There are several monitoring tools, also known as Application Performance Management (APM). Researchers and industry practitioners frequently implement these tools. They are widely used and already offer solutions for monitoring microservice applications. Prometheus is an open-source monitoring system known for its efficient time-series data storage and retrieval. One of its core features is PromQL, a powerful query language that has become a de facto standard for querying metrics in microservice platforms. Grafana (grafana.com) complements Prometheus by offering rich visualization and dashboard capabilities, enabling

teams to create interactive charts and alerts for real-time monitoring. Dynatrace (dynatrace.com), on the other hand, is a commercial APM solution that offers advanced features such as AI-driven anomaly detection, distributed tracing, and automated root cause analysis, making it well-suited for large-scale enterprise environments.

While these tools provide powerful data collection and visualization capabilities, they lack native support for attack injection, controlled workload generation, and systematic model evaluation, limiting their suitability for experimental security research. An advantage is that APM tools can be easily integrated into orchestrators and can connect to existing services that collect and provide monitoring data, such as cAdvisor and kube-state-metrics. Another example that can be integrated is tracing data, such as Jaeger or Zipkin. And also data collected using service meshes.

Service meshes introduce an infrastructure layer that transparently manages inter-service communication. They consist of a data plane and a control plane. The data plane handles communication between services and usually includes a set of network proxies, known as sidecar proxies, deployed alongside each service that intercept and manage traffic between services. The control plane is responsible for configuring and managing the proxies in the data plane, handling service discovery, load balancing, traffic routing, authentication, and observability.

Istio is a service mesh designed for microservice environments, which extends Kubernetes to establish a programmable, application-aware network. Therefore, it can handle the traffic routing, security, and observability, and is very flexible and configurable. **Linkerd** is another well-known service mesh for Kubernetes clusters that adds security, observability, and reliability without the complexity of other meshes. **Consul** is a service networking platform from HashiCorp that provides a service mesh as one of its features. Consul is a service networking platform from HashiCorp that includes a service mesh. Unlike previous ones made for Kubernetes, this one can work both inside and outside Kubernetes and includes service discovery, health checks, configuration, and a mesh. The last one, **Kuma**, is a service mesh built on top of Envoy, created by Kong. Therefore, it supports both Kubernetes and traditional Virtual Machines (VMS). It was designed to be simpler than Istio but more flexible than Linkerd.

2.2 Security in Microservices

Securing microservice-based applications differs fundamentally from securing monolithic applications. The security perimeter of a microservice application is considerably more complex, as illustrated in Figure 2.3. While in a monolithic system the entire application is protected as a single unit, in a microservice architecture each service constitutes an independent security boundary and potential attack surface [Yarygina and Bagge, 2018]. Consequently, microservices inherit the challenges of securing distributed systems, while introducing additional risks due to fine-grained service decomposition, extensive inter-service communication, and dynamic service lifecycles [Yarygina and Bagge, 2018]. This redefinition

of the security perimeter fundamentally alters threat models and defense strategies, requiring security mechanisms to operate across multiple layers and components rather than at a single centralized boundary.

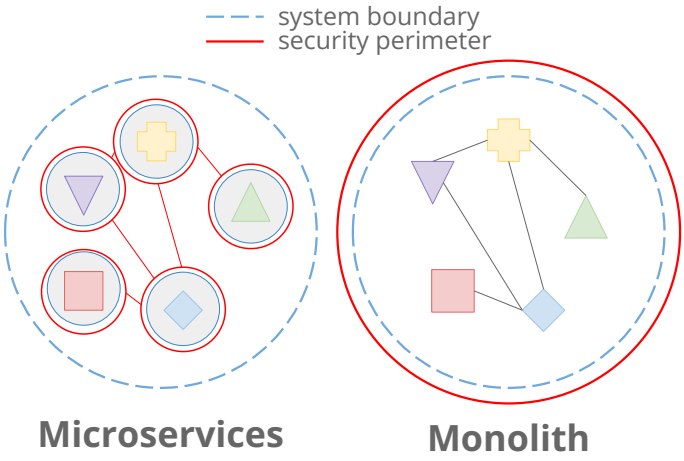


Figure 2.3: Microservices redefine the notion of perimeter security (adapted from [Yarygina and Bagge, 2018]).

2.2.1 Security Principles

Computer security, also referred to as cybersecurity, is concerned with protecting computer systems, software, networks, and data from unauthorized access, attacks, damage, and disruption. Several definitions of cybersecurity have been proposed in the literature, including:

> *"The body of technologies, practices, and processes that are made to help in protecting the networks, devices, data, and programs from any form of attacks, unauthorized access, or damages"* ([Alhayani et al., 2021]);

> *"Cyber Security involves reducing the risk of a malicious attack on software, computers, and networks. This includes tools used to detect break-ins, stop viruses, block malicious access, enforce authentication, enable encrypted communications, and on and on."* ([Amoroso, 2006]);

> *"Prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and nonrepudiation."* ([Force, 2020]).

In the early era of computing, security mechanisms primarily focused on physical protection to restrict access to authorized individuals. In contemporary systems, particularly with the widespread adoption of web and distributed applications, the attack surface has expanded significantly. Nevertheless, the fundamental security objectives remain unchanged [White et al., 2017].

The three fundamental objectives of computer security are:

- **Confidentiality:** ensures that sensitive information is accessible only to authorized individuals. This applies to both data stored in systems and data transmitted between systems.
- **Integrity:** guarantees that third parties can not change the data that they are not authorized to have access to. So, data in a computer system, or data transferred between computer systems, can only be modified or deleted by authorized individuals.
- **Availability:** ensures that systems and services remain accessible and operational whenever authorized users require access.

Threats and vulnerabilities can compromise security objectives. A threat is any circumstance or event with the potential to negatively impact an organization, individual, or system, including unauthorized access, data disclosure, modification, destruction, or denial-of-service attacks. A vulnerability is a weakness in a system that a threat source, such as software defects, misconfigurations, outdated components, or human error, can exploit.

2.2.2 Challenges in Securing Microservices

While microservice architectures offer substantial benefits in terms of scalability, modularity, and deployment agility, their distributed and dynamic nature introduces significant challenges for detecting attacks and anomalies. Unlike monolithic systems, where a single runtime environment can be monitored using traditional intrusion detection mechanisms, microservices consist of numerous loosely coupled components that communicate over network channels and are frequently orchestrated using container platforms and service meshes. This complexity fundamentally impacts detection efficacy.

One of the foremost challenges arises from the **heterogeneous and distributed nature of microservices** [Dias and Siriwardena, 2020]. Individual services may generate different types of telemetry (logs, metrics, traces), operate on distinct data schemas, and be implemented using diverse programming languages and frameworks. This **heterogeneity complicates the construction of unified detection models** and increases the overhead required to correlate events across services. In addition, microservices often scale dynamically and are updated frequently, making static baselines of normal behavior obsolete in short time frames.

A related challenge is the **high dimensionality and volume of data** generated by microservice applications. Modern systems may produce massive streams of logs, traces, and performance metrics from dozens or even hundreds of services [Grohmann et al., 2019; Waseem et al., 2021]. This not only creates storage and processing constraints but also obscures anomalous patterns within large volumes of benign activity. Detection systems must therefore **distinguish legitimate variations caused by scaling events, deployments, or workload changes from**

actual security incidents, a task made more difficult by the scarcity of labeled attack data in microservice environments.

Another significant challenge concerns the **analysis of inter-service communication patterns** [Wang et al., 2023]. Attacks against microservices often manifest as **subtle deviations in request sequences, timing relationships, or service dependency structures** rather than as explicit signatures in network packets. For example, an attack may propagate across multiple services in a way that appears normal when each service is observed in isolation but is anomalous in the context of the overall call graph. Detecting such distributed anomalies requires **models that capture both structural and temporal dependencies among services**.

A further challenge concerns **limitations in monitoring and observability** [Haindl et al., 2024]. Microservice environments frequently rely on sidecar proxies, service meshes, and distributed tracing systems to collect telemetry. While these technologies enhance visibility into service interactions, they also introduce additional layers of abstraction that **may obscure low-level indicators of compromise**. Furthermore, achieving consistent instrumentation across large, heterogeneous deployments is operationally challenging, often resulting in **coverage gaps and detection blind spots**. The instrumentation and telemetry collection process itself can also introduce non-negligible performance overhead.

Finally, **performance and resource constraints** pose significant challenges. Real-time detection systems must process high-throughput streams of events and generate timely alerts without degrading application performance. This is particularly challenging when employing **computationally intensive techniques such as deep learning or graph-based models**, which are increasingly proposed in the literature. Balancing detection accuracy, latency, and resource consumption is therefore critical in production environments.

2.2.3 Attacks Against Microservices

A **cyber-attack** (or simply attack) is any malicious attempt to gain unauthorized access to system services, resources, or information to collect, disrupt, deny, degrade, or destroy system assets. Attackers typically exploit vulnerabilities in software, configurations, dependencies, or communication channels to compromise applications. Due to their distributed, network-intensive, and highly dynamic nature, microservice-based systems introduce new attack surfaces and amplify traditional security threats.

Several studies have highlighted that microservice architectures are particularly exposed to attacks targeting service-to-service communication, APIs, container platforms, and orchestration layers [Haindl et al., 2024; Jayalath et al., 2024; Yu et al., 2019]. The most common attacks against microservice applications can be grouped into the following categories:

- **DoS attacks:** aim to exhaust system resources, rendering services unavailable to legitimate users. In microservice architectures, large numbers of

loosely coupled services communicate over the network, making them especially vulnerable to DoS and Distributed Denial of Service (DDoS) attacks. An attacker may flood a single service or a critical API endpoint with excessive requests, leading to cascading failures across dependent services. The amplification effect caused by synchronous service calls can significantly increase the overall impact of such attacks.

- **API abuse and injection attacks:** microservices rely heavily on RESTful and Remote Procedure Call (gRPC) APIs for communication. Inadequate input validation, weak authentication, and improper authorization controls can lead to classic injection attacks such as SQL injection, NoSQL injection, and command injection. In addition, attackers may exploit poorly secured APIs to perform parameter tampering, mass assignment, or unauthorized function invocation.
- **Session hijacking and Adversary-in-the-Middle attacks:** occur when attackers insert themselves into private communication channels between multiple endpoints. By doing so, they can intercept, modify, or replay sensitive information while legitimate parties believe they are communicating over a secure channel.
- **Supply chain attacks:** microservices often depend on numerous third-party libraries, frameworks, and container images. Attackers can exploit vulnerabilities in these dependencies or inject malicious code into the software supply chain before deployment, compromising applications at scale.
- **Container and Orchestration attacks:** most microservices are deployed using containers and managed by orchestration platforms such as Kubernetes. Vulnerabilities in container images, insecure runtime configurations, or compromised orchestration components can allow attackers to escape container isolation, access host resources, or gain control over the entire cluster.

DoS Attack Types and Tools

This thesis focuses on the security principle of availability, and therefore concentrates specifically on DoS attacks. There are different types of DoS attacks. DoS attacks can be broadly classified into low-volume and high-volume attacks, each exhibiting distinct characteristics.

- Low-volume DoS attacks deliberately exploit protocol weaknesses or application vulnerabilities using minimal traffic to evade detection. The objective is to subtly degrade system performance over an extended period, making the attack difficult to identify and mitigate promptly. Techniques such as Slowloris, which send partial HTTP requests to occupy server resources without completing connections, exemplify this category.
- High-volume DoS attacks aim to overwhelm a system by generating an excessive volume of traffic or requests within a short period of time. These

attacks are typically more visible and can quickly incapacitate targeted services. They often leverage networks of compromised machines, known as botnets, to amplify their impact by flooding the target with connection requests or data packets.

Various tools can be used to conduct DoS attacks by exploiting different layers of network and application protocols to exhaust system resources and render services unavailable to legitimate users. The following tools are among the most commonly referenced DoS attack tools in the literature:

- **Torshammer** [Solarstone, 2014] is a DoS attack tool that targets web and application servers by sending incomplete HTTP POST requests at a slow rate, typically between 0.5 and 3 seconds per request. This behavior keeps server threads open while waiting for additional data, leading to resource exhaustion. As a result, the server becomes unable to handle new legitimate connections, effectively placing it in a denial-of-service state.
- **Slowloris** is an HTTP-based DoS attack that primarily targets threaded web servers by keeping a large number of connections open for extended periods. The attacker sends partial HTTP requests at regular intervals without completing them, forcing the server to maintain these connections and exhaust its pool of available threads. This prevents the server from accepting new legitimate connections, ultimately leading to service unavailability.
- **HULK** [Grafov, 2016] is a DoS attack tool that targets web servers by generating a high volume of unique HTTP requests at irregular intervals. This approach aims to exhaust server resources while bypassing caching mechanisms and signature-based intrusion detection systems, thereby effectively overloading the target server.
- **GoldenEye** is an HTTP DoS testing tool designed to evaluate a web servers susceptibility to denial-of-service attacks. It operates by opening multiple parallel connections to a target URL and leveraging the HTTP Keep-Alive feature to maintain these connections. This behavior stresses the servers ability to handle numerous persistent connections and helps identify potential vulnerabilities to resource exhaustion attacks.

2.3 Attack Detection in Microservices

This section reviews and discusses the main methodological approaches for detecting attacks in microservice-based systems. It covers performance evaluation metrics, Logic Scoring of Preference (LSP), supervised and unsupervised machine learning techniques, and multimodal models. These approaches form the methodological foundation of the detection strategies investigated in this thesis.

2.3.1 Classification Performance Metrics

Evaluating the effectiveness of attack detection models requires a clear and consistent set of performance metrics. Microservice environments introduce unique challenges for classification due to their highly dynamic runtime behavior, including fluctuating workloads, autoscaling mechanisms, and the ephemeral nature of service instances. These factors may produce patterns that resemble malicious activity, making it essential to employ metrics that quantify not only a models ability to detect attacks but also its robustness against false alarms.

To enable a fair and consistent comparison across the LSP-based, supervised, and unsupervised machine learning models evaluated in this thesis, we rely on standard classification metrics that capture both discriminative capability and predictive reliability. Each model processes a sequence of samples collected during an execution run and assigns each sample to one of two classes: regular (annotated with 0) or attack (annotated with 1). Based on the predicted label and the corresponding ground truth, each sample falls into one of the following categories:

- **True Positives (TP)**: an attack sample correctly classified as attack.
- **False Positives (FP)**: a regular (non-attack) sample incorrectly classified as an attack.
- **True Negatives (TN)**: a regular (non-attack) sample correctly classified as regular.
- **False Negatives (FN)**: an attack sample incorrectly classified as regular.

Using this classification, several performance metrics can be computed. In this thesis, we adopt widely used evaluation metrics [Alpaydin, 2020]:

- **Recall**: rate of actual attack samples that are correctly classified: $recall = TP/(TP + FN)$.
- **Precision**: proportion of predicted attacks that are correctly classified: $precision = TP/(TP + FP)$.
- **F1-score**: harmonic mean between precision and recall: $F1 = 2 \cdot precision \cdot recall/(precision + recall)$.
- **Informedness**: probability that the model will make an informed decision rather than a random guess, considering both recall (i.e., sensitivity) and Specificity: $Informed. = TP/(TP + FN) + TN/(TN + FP) - 1$.
- **Markedness**: difference in predictive value for positive and negative classes, reflecting the model's ability to correctly identify both attack and regular samples: $markedness = (TP + TN)/(TP + FP + TN + FN)$.

2.3.2 Logic Scoring of Preference

The **LSP** method has been developed to assess and compare complex systems [Dujmovic, 2018]. It can be used to specify patterns derived from intuitive human reasoning to allow computing a global score that expresses a certain quality of the system [Dujmovic, 2018]. The LSP technique is composed of the following key steps: (i) decision problem, (ii) suitability attribute tree, (iii) elementary attribute criterion function, and (iv) aggregation structure composed of weights and operators [Shen et al., 2021].

The **Step 1** involves clearly defining the decision problem and specifying the goals. **Step 2** is to construct the suitability attribute tree. The tree is a hierarchical structure that decomposes the decision problem into categories that can be further decomposed into subcategories, and continues decomposing until reaching elementary attributes, which can be directly measurable [Friginal et al., 2011]. For instance, Oliveira et al. [Oliveira et al., 2020] proposed an attribute tree to assess the trustworthiness of web services frameworks. Trustworthiness is decomposed into two categories: Performance and resource consumption. Respectively, these categories are decomposed into the elementary attributes: throughput and response time, and CPU usage and memory allocation.

The **Step 3** refers to the specification of the criterion function of each elementary attribute, in which the preferred values for each feature are defined. The criterion definition maps an input value into the suitability degrees of each elementary attribute. Dujmovic presents 12 different types of elementary criteria, of which the three most frequent forms [Dujmovic, 2018] are the following:

Type S : Preferred small values (i.e., the lower, the better).

Type L : Preferred large values (i.e., the higher, the better).

Type R : Preferred range of values (i.e., the desired values are between a range).

The **Step 4** defines the logic-aggregation structure responsible for calculating the overall Score S. Two main elements must be defined here: the weights and logical aggregation operators (or only operators). Figure 2.4 illustrates an example of an LSP aggregation structure. The final aggregation 1 is decomposed into the two categories 1.1 and 1.3, and the elementary attribute a_3 (1.2). Category 1.1 is decomposed into the attributes a_1 (1.1.1) and a_2 (1.1.2). Category 1.3 is decomposed into the elementary attribute a_4 and sub-category 1.3.2. Finally, category 1.3.2 is decomposed into attributes a_5 (1.3.2.1) and a_6 (1.3.2.2).

The definition of weights is to determine the significance of each attribute in the aggregation [Friginal et al., 2011]. The weights are adjusted following the perspective and requirements of the evaluator(s) or stakeholder (s), and they are adjusted to reflect the logic of human reasoning by typically using a normalized numeric value [Shen et al., 2021]. The following seven methods that can be used to define the weights are presented in [Dujmovic, 2018]: (1) Importance decomposition method; (2) Direct weight assessment; (3) Weights based on ranking; (4) Weights based on menu; (5) Collective weight determination; (6) Weights

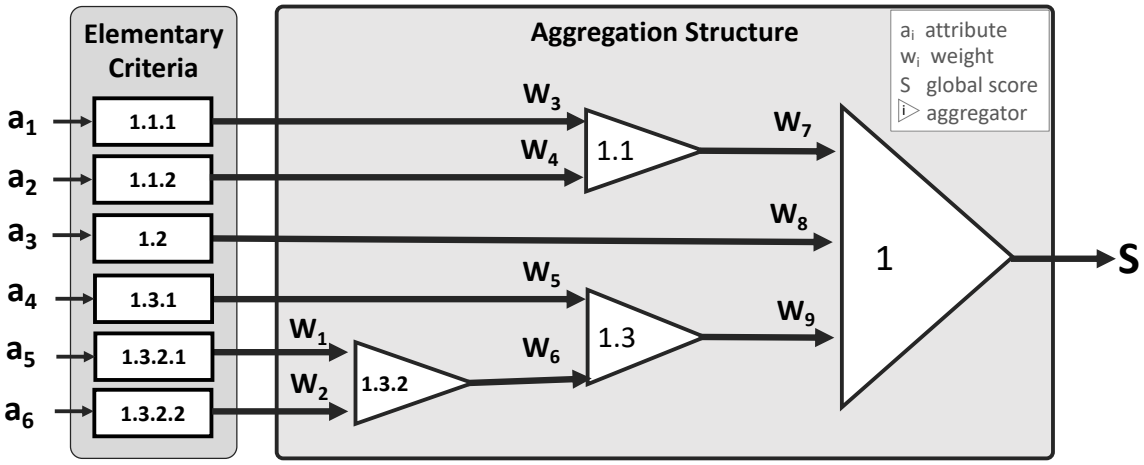


Figure 2.4: Example of LSP aggregation structure.

obtained from pairwise comparisons; (7) Weights based on preferential neuron training. Some of these seven methods can be used separately or combined, depending on the project's specificities. The appropriate weight assignment method selection is determined based on the evaluators' perspectives and practical considerations. Methods 5) and 6) can only be used by two or more evaluators, while the remaining five are to be applied by only one evaluator. Further explanations of the methods can be found in [Dujmovic, 2018].

The *logical aggregation operators* represent the relationship between the attributes a_i for each category until all categories are aggregated into the global score S [Friginal et al., 2011]. The LSP method is based on Soft Computing and uses a set of logical operators for the aggregation [Shen et al., 2021], which are combined into seven groups of Generalized Conjunction/Disjunction (GCD). As an example, Table 2.1 presents one of these groups: the GGCD.17. We can see that the operators belong to one of the three types of logic model [Dujmovic, 2018]:

1. The *substitutability (OR or ω)* is an operator for when a requirement that is highly satisfied can replace another lowly satisfied requirement.

2. The *Simultaneity (AND or α)* are operators for when all requirements must be satisfied.

3. The *Neutrality* refers to the arithmetic mean and expresses a balance between substitutability and simultaneity.

Finally, the aggregation can be calculated via a formula, such as the *Weighted Arithmetic Means (WAM)* or *Weighted Power Means (WPM)*. The choice of the formula depends on the evaluators' needs. However, the WPM is commonly used when dealing with non-neutral aggregators (i.e., those exhibiting substitutability or simultaneity behavior) [Polska et al., 2021]. When utilizing the WPM, the logical operator is mapped to a *power exponent r*, which is then used in the mean calculation. For instance, Table 2.2 correlates the Andness (α) and Orness (ω) values of the aggregators (column Aggregators [17]) with the corresponding WPM exponent values (r_1 to r_5). These exponents quantify the degree of simultaneity

Table 2.1: GGCD.17 operators.

Type of logic model	Form	Mode	Level	Symbol	Andness	Orness
Substitutability	Pure dis.	Hard	Extreme	D	0	1
	Partial disjuntion	Hard	Very high	D++	0.0625	0.9375
			High	D+	0.125	0.875
			Mid-high	D+-	0.1875	0.8125
			Medium	DA	0.25	0.75
		Soft	Mid-low	D-+	0.3125	0.6875
			Low	D-	0.375	0.625
			Very low	D--	0.4375	0.5625
Neutrality				A	0.5	0.5
Simultaneity	Partial conjunction	Soft	Very low	C--	0.5625	0.4375
			Low	C-	0.625	0.375
			Mid-low	C-+	0.6875	0.3125
		Hard	Medium	CA	0.75	0.25
			Mid-high	C+-	0.8125	0.1875
			High	C+	0.875	0.125
			Very high	C++	0.9375	0.0625
	Pure com.	Hard	Extreme	C	1	0

or substitutability of the operator, where $r > 1$ indicates simultaneity and $r < 1$ indicates substitutability.

Table 2.2: WPM Aggregators and Expoents.

Mean local		**Aggregators**	**WPM exponent**			
Andness	**Orness**	**[17]**	r_2	r_3	r_4	r_5
0.02083	0.97917	D	$+\infty$	$+\infty$	$+\infty$	$+\infty$
0.06250	0.93750	D++	46.176	30.857	30.229	31.211
0.12500	0.87500	D+	17.787	13.672	13.593	14.022
0.18750	0.81250	D+-	9.600	8.016	8.041	8.277
0.25000	0.75000	DA	5.898	5.214	5.255	5.392
0.31250	0.68750	D-+	3.842	3.540	3.574	3.650
0.37500	0.62500	D-	2.550	2.423	2.443	2.481
0.43750	0.56250	D–	1.662	1.618	1.625	1.638
0.50000	0.50000	A	1.000	1.000	1.000	1.000
0.56250	0.43750	C–	0.452	0.494	0.497	0.494
0.62500	0.37500	C-	-0.112	-0.033	-0.060	-0.066
0.68750	0.31250	C-+	-0.908	-0.500	-0.401	-0.359
0.75000	0.25000	CA	-2.226	-1.268	-1.016	-0.899
0.81250	0.18750	C+-	-4.721	-2.554	-1.990	-1.721
0.87500	0.12500	C+	-10.549	-5.184	-3.883	-3.269
0.93750	0.06250	C++	-32.090	-13.361	-9.465	-7.698
1	0	C	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Logic Scoring of Preference (LSP) has been widely used in different domains to model, compare, and evaluate complex systems with multiple criteria. Building on the LSP concepts, Oliveira et al. proposed an approach for benchmarking the security of Simple Object Access Protocol (SOAP) web service frameworks [Oliveira et al., 2020]. The approach uses the Logic Scoring of Preference (LSP)

method [Dujmovic, 2018] to calculate a trustworthiness score that ranks different frameworks by security. The authors constructed a QM which aggregates the Performance (response time and throughput) and Resource Consumption (CPU usage and memory allocation) for each step of their experiment (i.e., during regular usage of the framework, during DoS attacks, and pos attacks), i.e., they collected runtime data from the frameworks during regular and attack moments to after calculating the scores. Therefore, they were able to assess and compare four SOAP web service frameworks, and the solution may provide guidance for selecting web services based on trustworthiness.

A proposal, based on LSP, for trustworthiness assessment of web applications from a security perspective is presented in [Lemes et al., 2019]. The authors focused on developing a quality model based on practices described by the Open Worldwide Application Security Project (OWASP), such as input validation and data source validation. Therefore, they used services with and without known security vulnerabilities to build the quality model. Results show that the proposed model can identify and select the most trustworthy applications. Another security-related work is presented in [Dasso and Funes, 2020]. The authors develop a model to measure compliance with the OWASP Web Security Testing Guide checklist. The checklist items serve as attributes in the LSP model, and the categories are based on the list as well. The authors do not present any case study to validate the method.

Basso et al. analyzes the usefulness of LSP in constructing Quality Models to characterize the trustworthiness of online services or systems based on privacy [Basso et al., 2019]. The authors used Re-identification Risk and Information Loss as attributes that are aggregated into a global score. The model was applied to different datasets to evaluate a system or a single service, showing that it could adequately characterize privacy and open the possibility of characterizing other trustworthiness properties, such as security and dependability.

An approach for measuring a website user's experience is proposed in [Casare et al., 2020]. The proposal is based on a set of attributes identified during a literature review, aggregated using the LSP method. The attributes are aggregated into subcategories (e.g., usability, performance), and the subcategories are then aggregated into two main categories (i.e., usability and accessibility), allowing the calculation of a global score that reflects the user's trust in a site. The method was applied to seven websites, allowing users to select the most trustworthy ones.

A proposal, based on LSP, for prioritizing the use of network services in an organization (i.e., with Quality of Service (QoS) in mind) is proposed in [Leyva Vazquez et al., 2021]. The authors use linguistic labels to describe the process of assigning priorities to network services, aiming to reduce subjectivity and uncertainty. The method uses hierarchical aggregation operators, based on a 2-linguistic-tuple model, to prioritize services. The authors applied their proposal in a case study involving three services, and the method ranked the services by priority. The data are collected from six participants, not from real-time properties about the services.

The authors in [Kudermetov et al., 2022] compare the use of the LSP method with several other multi-criteria decision-making methods in the context of QoS-based web service selection. The work confirms the consistency of the LSP method by evaluating the similarity of its ranking result to several other well-known methods, including Simple Additive Weighting (SAW), Analytic Hierarchy Process (AHP), or Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS), to name a few.

Overall, LSP has various applications across domains, primarily for benchmarking, evaluating, and comparing systems using multiple criteria. While the presented works have successfully used LSP to construct QMs to evaluate trustworthiness [Basso et al., 2019; Lemes et al., 2019], security [Casare et al., 2020; Lemes et al., 2019; Oliveira et al., 2020], privacy [Basso et al., 2019], and QoS [Kudermetov et al., 2022; Leyva Vazquez et al., 2021], they have mostly focused on a single system or service. We propose that a technique like LSP is applicable in a microservices context, where multiple services are involved and may be affected by complex inter-relations, and as a tool for detecting attacks at runtime. Such attacks may exploit residual vulnerabilities that escaped detection, and signaling the occurrence of an attack may be useful for protecting the system.

2.3.3 Machine Learning

Machine learning (ML) has become one of the most widely adopted approaches for security monitoring and attack detection in modern distributed systems, including microservice-based architectures. The highly dynamic, large-scale, and heterogeneous nature of microservices, characterized by fluctuating workloads, frequent deployments, autoscaling behavior, and complex service dependencies, renders traditional rule-based and signature-based detection techniques insufficient in many scenarios. In such environments, static rules quickly become obsolete, and handcrafted signatures fail to generalize to novel or evolving attacks.

Machine learning techniques address these limitations by automatically learning patterns, correlations, and behavioral profiles from data, enabling the detection of both known attack signatures and previously unseen anomalies. This capability is particularly valuable in microservice systems, where attacks may manifest as subtle deviations in performance metrics, request patterns, or inter-service communication rather than as easily recognizable network-level signatures.

In general, ML approaches for attack detection can be categorized into supervised and unsupervised learning paradigms. Supervised learning relies on labeled datasets and is effective when representative attack data are available, whereas unsupervised learning focuses on modeling normal behavior and identifying deviations as potential anomalies. In this thesis, both paradigms are investigated to address different attack detection scenarios in microservice environments.

Supervised Learning

Supervised learning algorithms are trained on labeled datasets, in which each instance is associated with a known class label, such as benign or malicious. During training, the model learns a mapping function between input features (e.g., CPU usage, response time, request rate) and output labels. This learned mapping is subsequently used to classify unseen data. Supervised learning is particularly effective when high-quality, representative labeled datasets are available, as it enables precise discrimination between normal and attack behavior.

In microservice-based systems, supervised learning models are typically trained using features derived from service-level metrics, container-level resource usage, or application-layer monitoring data. These features capture the runtime behavior of individual services and the collective behavior of the application. However, obtaining labeled attack data in microservice environments is often challenging due to the cost of attack injection, the diversity of deployment configurations, and the evolving nature of attacks.

The supervised algorithms investigated in this thesis are described as follows:

- **Random Forest (RF)** is an ensemble learning method that constructs a large number of decision trees during training and outputs the class that corresponds to the majority vote of the individual trees. By combining multiple trees and introducing randomness in both feature selection and data sampling (bagging), Random Forest reduces overfitting and improves overall generalization performance.
- **Support Vector Machines (SVM)** aims to find the optimal hyperplane that maximally separates data points of different classes in a high-dimensional feature space. By maximizing the margin between classes, SVM seeks to achieve good generalization to unseen data. Using kernel functions, SVMs can model complex, non-linear decision boundaries.
- **K-Nearest Neighbors (KNN)** is a distance-based, instance-based learning algorithm that classifies a data point based on the majority label among its k nearest neighbors in the feature space. Unlike parametric models, KNN does not build an explicit model during training; instead, it stores the training data and performs classification at inference time.
- **XGBoost (XGB)** is an optimized implementation of gradient boosting that builds an ensemble of decision trees sequentially. Each new tree is trained to correct the errors of the previous ensemble, allowing the model to improve its predictions progressively. XGBoost incorporates regularization mechanisms to prevent overfitting and is specifically designed for high computational efficiency and scalability.
- **Convolutional Neural Network (CNN)** are deep learning models originally developed for image processing but increasingly applied to time-series and sequential data. By applying convolutional filters, CNNs can automatically learn hierarchical feature representations and capture local patterns in

the input data. CNNs can be employed to capture local temporal patterns, burst behaviors, and short-term correlations that may indicate coordinated or evolving attack activities.

Unsupervised Learning

Unsupervised learning algorithms operate on unlabeled data and aim to discover the underlying structure, distribution, or patterns within the dataset. In the context of security monitoring, these techniques are widely used for anomaly detection, as they learn models of normal system behavior and flag deviations as potential attacks. This paradigm is particularly relevant in microservice environments, where new and previously unseen attacks may occur and labeled data may be scarce or unavailable.

Given the continuous evolution of microservice applications and the frequent introduction of new services, configurations, and deployment patterns, defining a complete set of attack signatures in advance is impractical. Unsupervised learning enables the detection of abnormal behavior without relying on prior knowledge of specific attack types.

The unsupervised algorithms considered in this thesis are the following:

- **Isolation Forest (ISOF)** is an anomaly detection algorithm based on the principle that anomalies are easier to isolate than normal data points. It constructs an ensemble of random trees by recursively partitioning the data. Data points that require fewer splits to be isolated are considered more likely to be anomalies.
- **One-Class SVM (OCSVM)** learn a decision boundary that encloses the majority of the training data, which are assumed to represent normal system behavior. During inference, any data point that falls outside this boundary is classified as anomalous.
- **Local Outlier Factor (LOF)** is a density-based anomaly detection algorithm that measures the local deviation of a data point with respect to its neighbors. A point is considered an outlier if its local density is significantly lower than that of its surrounding points.
- **Autoencoder (AE)** is neural networks trained to reconstruct their input data. They learn a compressed latent representation of normal behavior and attempt to reproduce the input at the output layer. When presented with anomalous data, the reconstruction error typically increases, indicating a deviation from learned normal patterns.
- **Variational Autoencoder (VAE)** is a probabilistic extension of standard autoencoders. Instead of learning a fixed latent representation, VAEs learn a distribution over the latent space, enabling better modeling of data uncertainty and variability.

Related Work

A substantial body of research has explored the use of machine learning techniques for detecting attacks and anomalies in microservice applications. These works vary in terms of data sources, attack types, learning paradigms, and levels of abstraction, ranging from application-layer monitoring to container-level metrics and network traffic analysis.

Baarzi et al. [Baarzi et al., 2020] proposed an application-agnostic, non-intrusive, and unsupervised approach to protect microservice systems against DoS attacks at the application layer. Their approach relies on historical resource usage data, including CPU and memory consumption, collected under varying legitimate workloads. By modeling normal behavior and detecting deviations, the method can identify potential DoS attacks. Although the approach accounts for instance-level behavior, it focuses on individual services rather than assessing attack impact and detection across the entire microservice application, which is a key focus of this thesis.

Ficco et al. [Ficco et al., 2025] contributed with a benchmark dataset comprising multiple attack types and service configurations, enabling systematic evaluation of different detection strategies. Their dataset includes scenarios such as high-volume DDoS, low-volume DDoS, SYN floods, and GET floods. The authors evaluated several supervised machine learning models, including SVM, Random Forest, and KNN, reporting classification accuracies of approximately 80%.

Cao et al. [Cao et al., 2022] proposed the use of state machine anomaly models to learn runtime behaviors in cloud environments and detect attacks across multiple microservice applications. Their approach models normal execution flows and identifies deviations indicative of malicious activity. The results demonstrate high detection accuracy.

Bremler-Barr et al. [Bremler-Barr et al., 2024] investigated how autoscaling mechanisms themselves can be exploited by attackers, demonstrating that microservices operating with independent scaling policies can become desynchronized under overload conditions. This desynchronization leads to performance degradation and Economic Denial of Sustainability (EDoS). While this work provides valuable insights into attack characterization and economic impact, it does not address attack detection, which is the primary focus of the present thesis.

Several studies focus on network-layer intrusion detection, which differs from the application-layer monitoring perspective adopted in this thesis. Benedetti et al. [Benedetti et al., 2025] proposed an Intrusion Detection Agent (IDA) for runtime identification and classification of DDoS campaigns in multi-container environments. Their approach uses AI-based agents to classify DoS attacks based on behavior-related measurements such as RAM usage and network traffic volume. Data were collected using the TrainTicket application under attacks including high-volume DDoS, low-volume DDoS, GET floods, and SYN floods. Using decision trees, they achieved an accuracy of approximately 0.9. However, their focus is primarily on network and host-level indicators rather than service-level application behavior.

Morsli et al. [Morsli et al., 2025] introduced a Multidimensional Intrusion Detection System (MIDS) for containerized environments, combining network traffic features with container-level metrics to improve detection accuracy. They generated datasets using Kubernetes-based deployments of DVWA and Bank of Anthos, and simulated attacks such as DoS, brute-force, and SQL injection. By merging network flows with container metrics and applying supervised learning algorithms, including SVM, XGBoost, and deep neural networks, they demonstrated significant improvements in F1-score compared to unimodal approaches. Although effective, their work is primarily focused on infrastructure-level monitoring rather than on application-layer microservice interactions.

Flora et al. [Flora et al., 2023] proposed an intrusion detection approach for scalable and elastic microservice applications based on host-level system call analysis. In subsequent work In [Flora and Antunes, 2024], they explored data generation and benchmarking strategies for intrusion detection. These approaches focus on low-level system behavior and are effective for detecting certain classes of intrusions. Still, they do not directly model service-to-service communication or application-layer attack patterns.

Abdennebi et al. [Abdennebi et al., 2025] presented Sec-Llama, a compact large language model (LLM) for detecting network intrusions in Kubernetes clusters. To address the high computational and memory demands of LLMs, the authors employed memory-efficient data-driven techniques, including byte-based transformations of raw network flows. While this work demonstrates the feasibility of using LLMs for intrusion detection, it remains focused on network-layer data and does not address application-layer attack detection in microservices.

Several works have applied anomaly detection techniques to identify failures or abnormal behavior in microservice and containerized environments. [Du et al., 2018] injected faults to determine relevant metrics for anomaly detection in containers. To do this, they collected CPU and memory usage metrics from the containers and used supervised ML algorithms to detect the anomalies. [Samir and Pahl, 2020] also monitored the container's CPU and memory usage to detect and locate abnormalities using Spearman's rank correlation coefficient and Hierarchical Hidden Markov Models (HHMM). [Grohmann et al., 2019] attempted to predict the system's performance degradation by collecting several performance metrics (such as CPU and memory usage and I/O device throughput) and using different supervised machine learning techniques to evaluate their approach. While these studies contribute to understanding resource-based anomalies, they tend to focus on faults rather than on deliberate attack scenarios.

Several studies have focused on trace analysis for anomaly or failure detection in microservice applications. [Panahandeh et al., 2024] proposed a practical anomaly detection approach using distributed tracing and profiling data to build a Context Propagation Graph (CPG). These approaches are particularly relevant for understanding causal relationships and propagation paths, but are mainly oriented toward fault diagnosis rather than security attack detection.

2.3.4 Multimodal Models

Multimodal machine learning (MML) refers to the design and use of models that jointly process and learn from multiple heterogeneous data modalities. In the context of microservice-based systems, relevant modalities include performance metrics, logs, traces, network flows, system calls, and application-level events. Each modality captures different and complementary aspects of system behavior. Consequently, combining multiple modalities provides a more comprehensive and robust view of the system, particularly valuable for attack detection and anomaly analysis in complex distributed environments.

Microservice architectures naturally generate rich multimodal data due to their layered structure, distributed execution, and extensive instrumentation. Attacks may manifest differently across modalities, for example, as abnormal request paths in traces, unusual resource consumption in metrics, or suspicious patterns in logs. Relying on a single modality may therefore provide an incomplete or misleading picture. Multimodal learning addresses this limitation by integrating heterogeneous information sources to improve detection performance.

Multimodal learning introduces several fundamental challenges and capabilities [Barua et al., 2023]:

- **Representation** concerns how to encode and summarize heterogeneous modalities in a way that preserves their semantic meaning while enabling joint learning. In microservices, this includes representing time-series metrics, unstructured logs, and graph-based traces in a unified feature space. Effective representations must capture both complementary and redundant information across modalities.
- **Translation** refers to mapping information from one modality to another, such as generating textual explanations from metrics or inferring system state from network traffic.
- **Alignment** focuses on identifying correspondences and relationships between elements of different modalities. In distributed systems, this often involves temporal alignment (e.g., aligning logs, traces, and metrics by timestamp) and entity alignment (e.g., mapping events to specific services, containers, or instances).
- **Fusion** addresses how information from multiple modalities is combined to perform prediction or detection tasks. Fusion is the central mechanism enabling multimodal models to integrate heterogeneous data and exploit inter-modal dependencies.
- **Co-learning** refers to the transfer of knowledge between modalities, their representations, and their predictive models, allowing one modality to enhance learning in another.

Fusion is one of the core components of multimodal learning and is particularly relevant to the development of the work in this thesis. Multimodal fusion strategies are typically categorized as early, late, or hybrid:

- **Early Fusion (Low-Level Fusion):** in early fusion, raw features or low-level representations from different modalities are concatenated and jointly processed by a single model. This approach enables early learning of cross-modal interactions but requires careful feature normalization, synchronization, and alignment. In microservice environments, early fusion can combine metrics, log embeddings, and trace features into a unified representation for attack classification.
- **Late Fusion (Decision-Level Fusion):** late fusion combines the outputs or decisions of separate unimodal models, for example, through averaging, majority voting, or weighted schemes. This strategy is modular and robust to missing modalities, making it suitable for large-scale systems where some data sources may be unavailable, delayed, or noisy.
- **Hybrid Fusion:** hybrid approaches combine elements of early and late fusion, allowing interaction at intermediate layers while preserving modality-specific processing. These architectures offer a balance between expressive power and robustness and are increasingly adopted in deep learning-based multimodal systems.

Common fusion mechanisms include averaging, majority voting, weighted schemes, attention-based fusion, and learned gating mechanisms [Baltrušaitis et al., 2018]. Attention mechanisms, in particular, allow models to dynamically weight modalities based on their relevance, which is highly beneficial in dynamic microservice workloads where the importance of different data sources may change over time.

Jing et al. [Jing et al., 2022] addressed the challenge of fault identification by proposing a multimodal approach based on LightGBM. Their method utilizes a dataset comprising performance indicators (CPU, memory, and network packets), log data, and call chain data. The choice of LightGBM is motivated by its efficiency in memory usage during model training. The authors demonstrate that their approach effectively identifies faults and outperforms traditional methods such as XGBoost and GBDT. However, the focus of this work is on fault diagnosis rather than security attack detection.

Zhao et al. [Zhao et al., 2024] introduced CHASE, a causal heterogeneous graph-based framework for root cause analysis. Similar to other advanced frameworks, it leverages performance metrics, logs, and traces, but distinguishes itself by incorporating service instances directly into the analytical model. Multimodal fusion is achieved by transforming metrics and logs into a uniform structure and using trace data to construct a multimodal invocation graph. In this graph, service instance nodes are connected based on traces, with other modalities linked as attributes. This enables instance-level anomaly detection and causal analysis.

Chen et al. [Chen et al., 2022] proposed a deep attentive anomaly detection model focusing on the temporal aspects of system data. The system monitors Key Performance Indicators (KPIs) and logs, which are categorized into load metrics, traffic metrics, and log sequences. The multimodal fusion process involves extracting log templates using the Drain algorithm and aligning them with metrics

by timestamp. Separate LSTM networks are trained for each data group, and their hidden states are concatenated. An attention network then processes this joint representation to learn inter-dimensional and temporal dependencies before producing anomaly scores.

Wang et al. [Wang et al., 2024]. proposed the MADMM mechanism for anomaly detection through multi-feature extraction. Their approach uses metrics and logs to provide a comprehensive view of the microservice environment, enabling the detection of abnormal patterns across modalities. Zhang et al. [Zhang et al., 2023] developed DiagFusion, a framework for robust failure diagnosis that maps metrics, logs, and traces into a single vector space. By training a Graph Neural Network (GNN) on historical patterns and constructing a dependency graph from deployment data, the framework captures spatial features and failure-propagation paths, enabling instance-level localization and failure-type identification.

A more granular approach to data fusion is found in AnoFusion, proposed by Zhao et al. [Zhao et al., 2023]. This unsupervised method detects instance failures by converting metrics, logs, and traces into individual time-series data channels. These channels are then integrated into a heterogeneous graph that represents the system state at each moment t. The fusion is managed by a Graph Transformer Network (GTN) that learns node representations and captures correlations across modalities, followed by a Graph Attention Network (GAT) that assigns importance scores. Finally, a Gated Recurrent Unit (GRU) is employed to predict future graph states and identify anomalies.

The utility of trace-centric data is further explored by Ding et al., who developed a semi-supervised learning framework for trace anomaly detection [Ding et al., 2024]. In this work, the multimodal data is derived entirely from traces, including processing time, network latency, response status codes, and invocation patterns. A TransformerEncoder-based message-passing neural network is used to fuse these multimodal attributes, learning a unified representation of the trace graph. Similarly, the TraceGra framework [Chen et al., 2023] uses trace data to build an attribute graph, with performance metrics such as CPU, memory, and disk usage as node attributes. This enables simultaneous detection of response-time and invocation-path anomalies via unsupervised joint learning.

2.4 Summary

This chapter reviewed the fundamental concepts and the state of the art in securing microservice-based applications. It provides several conclusions that guide the development of this thesis. Microservice architectures, while offering significant advantages in scalability, flexibility, and independent deployment, introduce substantial security and monitoring challenges due to their distributed, heterogeneous, and highly dynamic nature. In particular, the decomposition of applications into multiple loosely coupled services expands the attack surface and complicates the construction of attack detection models.

The literature review showed that existing research explores a broad spectrum of techniques, ranging from multicriteria decision-making methods such as LSP to supervised and unsupervised machine learning models and, more recently, multimodal approaches. The review also demonstrated that while ML is a dominant approach, supervised models are frequently hindered by the scarcity of labeled attack data in microservice environments. Conversely, unsupervised methods, while promising for detecting novel attacks, require careful tuning to distinguish legitimate workload fluctuations and autoscaling events from malicious activity.

A central gap identified in current research is the limited application of multimodal models for security purposes. While existing works successfully integrate heterogeneous data, they are predominantly oriented toward fault diagnosis and reliability, leaving the potential of data fusion for security attack detection largely underexplored. Furthermore, the LSP method emerges as a versatile alternative for modeling human-like reasoning and aggregating complex system criteria, offering a level of interpretability often missing in black-box ML models.

Building on these observations, the following chapters investigate concrete detection strategies for microservice environments. The subsequent chapters build upon these insights to propose a framework that supports the development of attack detection solutions for microservice applications, enables the generation of realistic attack data, and explores different techniques (such as LSP and ML) to develop detection models, allowing an analysis of their effectiveness and trade-offs in detecting DoS attacks in microservice applications.

Chapter 3

Framework to Develop Attack Detection Models for Microservice Applications

Developing an attack-detection mechanism for microservice applications requires accounting for key architectural and operational aspects, such as the application's specific architecture and configuration (e.g., response to increasing load), the occurrence of genuine peaks in client workloads, and the potential presence of attacks. One of the major challenges is the lack of publicly available datasets (e.g., for training Machine Learning (ML) models). Consequently, the development of novel models typically requires conducting large-scale experiments and collecting data to build relevant datasets. Such datasets should account for the system's normal operational conditions and how it reacts to specific attack classes. The other challenge is the complexity of developing and evaluating attack-detection models in dynamic, distributed microservice applications. Addressing this requires a systematic structure for model definition, comparison, and experimentation across diverse attack and operational scenarios.

To address these challenges, we propose a comprehensive framework for cybersecurity research in microservice applications, with a focus on attack detection. The framework enables the generation of realistic attack datasets and supports the systematic development and evaluation of detection models tailored to microservice environments. It is structured according to the Organisation for Economic Co-operation and Development (OECD) framework [OECD, 2022] and consists of two integrated modules:

1. *Data Generation*: provides the necessary components to generate representative workloads and attack data, and aims to address the lack of significant datasets for microservice application research.

2. *Model Development and Evaluation*: supports the definition of novel attack detection models using various methods (for example, ML, Multi-Criteria Decision-Making (MCDM)) and their evaluation within the scope of microservice applications.

This chapter is organized as follows. Section 3.1 presents the framework overview. Section 3.2 focuses on the data generation module. Section 3.3 details the module for model development and evaluation. Section 3.4 summarizes the chapter.

3.1 Framework Overview

The OECD Framework for the Classification of Artificial Intelligence (AI) Systems [OECD, 2022] served as a foundation for structuring our framework. It organizes AI-based systems into five high-level dimensions: People & Planet, Context, Data & Input, Model, and Task & Output (as shown in Figure 3.1). Figure 3.2 presents our proposal, which consists of two principal modules: *(A) Data generation* aligned with People & Planet, Context, and Data & Input, and *(B) Model development and evaluation* aligned with Model and Task & Output.

Module (A) - Data Generation module addresses three of the OECD dimensions:

- *People & Planet* covers the actors impacted by the system, which, in our framework, refers to the users and attackers of a microservice application. The Workload Execution (1) and Attack Injection (2) capture this, which respectively emulate user behavior and attacks.
- *Context* encompasses the economic and sectoral environment of the system. For microservice applications, this relates to the application's criticality, the computational capacity required for attack detection, and the types of attacks it must handle. The Microservice Application (3) itself is not tightly coupled to the framework's scope, allowing it to remain agnostic to the specific application being targeted and enabling adaptation to different contexts. This separation fosters application transferability. The framework supports applications that allow horizontal and vertical scaling, accommodating complex elastic environments.
- *Data & Input* focuses on the data the model uses to construct a representation of the environment. In our approach, data collection is essential for developing effective attack detection models. Monitoring (4) and Data Collection (5) collect runtime data from the application under both regular

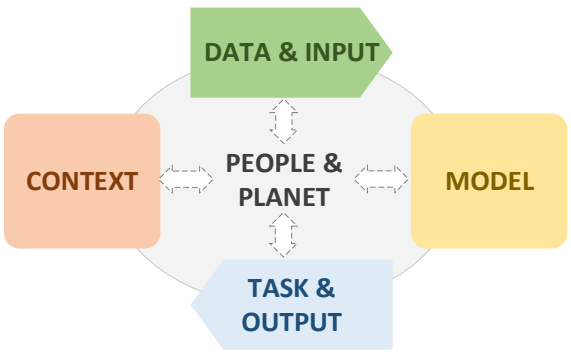


Figure 3.1: OECD framework (Adapted from [OECD, 2022]).

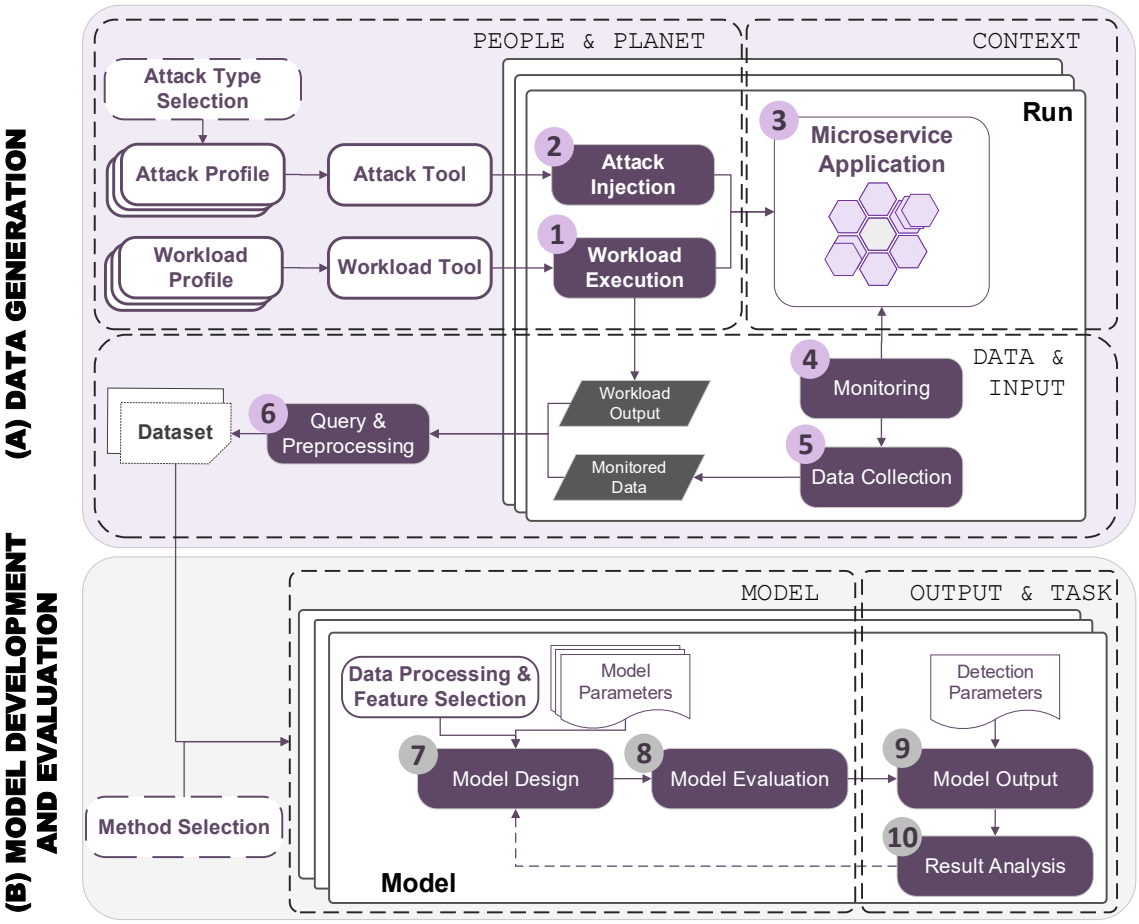


Figure 3.2: The proposed framework design.

workload and attack conditions. Subsequently, the Query & Preprocessing 6 component prepares this data for model development.

Module (B) - Model Development and Evaluation module addresses the remaining dimensions:

- *Model* involves the processes and methods used in the system. Our framework covers this through the Model Design 7 and Model Evaluation 8 components, which manage the development of attack detection models.
- *Task & Output* refers to the specific tasks the system performs and the results it produces. The Model Output 9 and Result Analysis 10 components analyze attack-detection outcomes, ensuring the model meets its objective of identifying attacks.

Fundamentally, the framework design is platform-agnostic and modular, conceived to decouple its components from specific microservice applications, execution environments (from local virtual machines to public cloud platforms, e.g., Amazon EKS, Google GKE, Azure AKS), as well as particular classes or types of attacks.

3.2 Data Generation

Workload Execution 1 is responsible for emulating the behavior of real users interacting with the microservice application. The workload is generated using a *Workload Tool* that executes a *Workload Profile*, which comprises two aspects:

(a) *Workload Intensity Specification:* Responsible for defining the volume and intensity of the workload over time.

(b) *User Behavior Modeling:* Defines the sequence of actions and the specific service operations the users interact with, simulating realistic user behavior.

The strategy to specify the workload intensity depends on the availability of load data and the research objectives. We consider the following three strategies:

1. *Real workloads*, which are derived from actual workload data collected from the target application, providing an accurate representation of the intensity. These workloads reflect the number of users and requests observed in the monitored system, but are often unavailable.

2. *Realistic emulated workloads*, which are designed to closely mimic typical user intensity over time, incorporating adjustments to fit the experimental environment. These emulate real workloads collected from other systems but can be adapted or scaled to suit the research needs.

3. *Synthetic workloads*, which are entirely constructed based on predefined parameters, allowing for controlled experimentation. These workloads offer flexibility, enabling researchers to model various usage scenarios, but they may not closely reflect real operational conditions.

When specifying the intensity for realistic, emulated, and synthetic workloads, specific patterns can be identified or constructed, respectively. These patterns include [Ding and Huang, 2021]: the *gentle type*, where the load remains constant or exhibits minimal variation over time; the *burst type*, characterized by one or more sudden spikes in load; the *rise type*, where the load gradually increases; and the *decline type*, where the load gradually decreases. Workload intensity can follow a single pattern or a combination of patterns (e.g., it might start with a rise, feature bursts of activity, and finish with a decline [Xiao and Hu, 2022]).

Regarding modeling user behavior, there are two main types of user profiles: (i) *Observed Profiles*, where data are collected from the target application and applied to model user behavior based on actual interaction patterns; and (ii) *Simulated Profiles*, where one or more types of user behavior profiles are defined in the absence of real data. For example, benchmarks for microservice applications such as TeaStore [Von Kistowski et al., 2018] and TrainTicket [Zhou et al., 2018] include three typical (simulated) user profiles: Browser/Visitor, Buyer, and Order to represent diverse workload characteristics [Avritzer et al., 2020].

Profiles depend on the specific microservice application, necessitating experimental strategies tailored to its actual operations, as each application has different services and workflows [Calzarossa et al., 2016]. Some applications already provide predefined profiles, such as the browsing profile of the TeaStore [Von Kistowski et al., 2018]. To define user behavior profiles, the following steps should be considered:

1. Develop a comprehensive understanding of the microservice application, its services, and its operations.
2. Map the actions that trigger each service and its operations, identifying what users may interact with.
3. Create user profiles that include potential behaviors for each type of user.
4. Ensure that different user profiles are appropriately represented, acknowledging that some profiles may be more prevalent than others. For example, in an online store, a larger proportion of users may browse products than complete purchases.

A proper definition of the workload profile is essential for generating realistic data to address the challenges of monitoring microservice applications. For instance, to differentiate between legitimate load spikes and attacks, the workload profile must include load spikes that reflect increased resource usage. Similarly, to study the dynamic scaling of services, the workload intensity must lead the application to scale up or down during experiments. Additionally, since we are dealing with microservice applications, it is essential to define workloads that exercise all the services that compose the application. If workloads do not exercise all services in a representative manner, it can result in models that are unable to handle attacks targeting a specific service.

The duration of the workload execution is another crucial parameter to consider. To ensure the system reaches a stable state before meaningful data collection, a warm-up stage is required. Following this, workload data should be collected during different phases of the experiment, including before and after the attacks. This allows for accurate measurement of the system's behavior during regular operations (pre-attack) and after the attack occurs (post-attack) [Oliveira et al., 2020]. Finally, the Workload Execution outputs the *workload output* file, which contains the client-side run results used to analyze measurable attributes such as the total number of request samples, response time, and throughput over time.

The **Attack Injection** 2 component emulates attacks against the microservice application. To perform the attacks, an *Attack Tool* (i.e., a software unit that generates traffic or behavior consistent with the selected attack type) is needed, which should be selected based on the *Attack Type Selection*. For each type of attack (e.g., high-volume Denial of Service (DoS) attack, Adversary-in-the-Middle (AitM), or Application Programming Interface (API) Abuse [Jayalath et al., 2024]), we have to define the *Attack Profile* to be able to perform the attack injection. The Attack Profile describes the attackers' behavior and determines the configuration of the

tool used to perform the attacks. It specifies when the attacks start and end, and which service operations should be targeted.

Diverse attack profiles are essential for generating a comprehensive dataset. For example, in the context of DoS attacks, these profiles should cover a range of durations and frequencies, as attackers continually adapt their strategies to evade detection and mitigation mechanisms [Somani et al., 2017; Zhijun et al., 2020]. To create a varied and representative dataset, we suggest the following types of attack frequencies:

- *Short Attack*: A brief, defined attack duration, *e.g.*, as short as a couple of 2 minutes.
- *Persistent Attack*: A prolonged attack that can last several minutes or the entire duration defined for the attacks.
- *Intermittent Attack - Variable Duration*: Repeated attacks every X minutes, with each having a different duration.
- *Intermittent Attack - Variable Interval*: Attack duration remains constant, but the interval between attacks varies.

Including diverse attack profiles ensures the dataset reflects a wide range of potential attack scenarios, enhancing the ability of the detection model or tool to detect and respond to increasingly sophisticated attack techniques. The attack profiles must also target different services (e.g., one targeting the UI service and another the persistence service) to capture a broader range of potential impacts on microservice performance. It is also crucial to record precise timestamps for the start and end of each attack (and for each occurrence in the case of intermittent attacks), as this is needed to accurately label observations monitored from the application side as either attack or non-attack (regular) behavior.

The Attack Injection component is modular and adaptable. It is designed to handle different attack types targeting microservice applications. Table 3.1 shows examples of supported attack types that can be integrated within this framework. For instance, API abuse can be injected using fuzzing tools or scripted requests that target specific service endpoints. Service spoofing and AitM can be simulated by deploying malicious service replicas. Supply chain attacks can be modeled by deploying compromised containers via the orchestration layer. Each scenario can be defined by a specific attack profile and executed during a designated attack window in a run. Thus, the framework enables the creation of diverse and realistic attack types, allowing future research to explore a broader range of microservice threats.

To perform an experimental run, both the Workload Execution and Attack Injection are executed, targeting the **Microservice Application** 3. There are two types of runs: *Golden Run*, where only the workload is executed, and *Attack Run*, where both the attack injection and workload execution are performed. To construct the dataset, we must perform multiple golden runs and multiple attack runs, as depicted in the multi-layered *Run* in Figure 3.2. Based on our experimental and

Table 3.1: Examples of Supported Attack Types and Integration within the Framework.

Attack Type	**Description**	**Attack Tools**	**Integration with Proposed Framework**
High-Volume DoS	Floods target with excessive requests to overwhelm its resources.	Hulk, GoldenEye	Already implemented and used for dataset generation (Chapters 4 and 7).
Low-Volume DoS	Slow or periodic requests designed to exhaust target's resources over time.	Torshammer, Slowloris	Already implemented and used for dataset generation (Chapters 4 and 7).
API Abuse	Exploits insecure APIs to harm the system, steal data, corrupt systems (eg., Broken Object Level Authorization (BOLA)).	OWASP ZAP, Zed Attack Proxy (ZAP), Burp Intruder (Burp Suite)	Already implemented and defined via a custom profile executed by the Attack Injection module. Available in [Castro et al., 2024].
Service Spoofing	Impersonation of legitimate services to deceive internal components or redirect traffic.	Custom scripts, Fake containers with spoofed service names	Custom deployment scripts inject malicious, spoofed service replicas into te execution environment.
AitM Attacks	Intercepts, observes, or modifies inter-service traffic by positioning an adversary between two communicating services (e.g., unencrypted Remote Procedure Call (gRPC)).	Mitmproxy, Ettercap	Insertion of an Adversary-in-the-Middle proxy container configured to intercept designated service-to-service environment.
Supply Chain	Introduction of vulnerabilities or malicious code via compromises dependencies (e.g., malicious containers, vulnerable base images, compromised libraries).	Compromised Docker images, CI/CD injection	Implemented by deploying a vulnerable or pre-compromised container image via the orchestration layer to simulate a supply chain breach
Orchestration Attacks	Exploits weaknesses in the cluster management plane or container runtime to gain unauthorized access or privilege escalation (e.g., Kubernetes/Docker weaknesses).	kube-hunter, Peirates	Simulation via Cluster Role Manipulation or executing privileged commands from within an infected container

empirical experience and supported by previous work [Oliveira et al., 2020], we propose dividing a single run into five stages:

1. *Initialization (Warm-up)* stage, where the system performance may exhibit significant variability, making it unsuitable for analysis.
2. *Pre-attack* stage during which only regular requests are generated, establishing a baseline of regular behavior.
3. *Attack window* stage during which one or multiple attacks can occur while regular requests are generated, allowing the assessment of the impact of the attack on the application.
4. *Post-attack* stage, where only regular requests are made, establishing system behavior after an attack, which may generate false positives in detection.
5. *Completion (End)* stage, designed to allow any pending requests to be processed and completed. These requests are not analyzed due to their significant variability.

Users of our framework are responsible for determining finer-grain aspects, such as the duration of each stage in a run. These details may vary depending on the infrastructure and context, such as the number of machines, the type of microservice application, or the type of attacks being considered.

The **Monitoring** 4 component comprises one or more tools or monitors to observe the system at the application level by collecting specific system attributes, such as resource consumption (e.g., memory consumption and the number of threads), the state of the services (e.g., active running services and timestamp of replica creation), and traffic-related data (e.g., request duration and request size) [Waseem et al., 2021]. The monitoring tools may collect information about each running service instance and each node that contains one or more services. They may also gather data on the application's response time and throughput (when handling user requests), as well as the response time and throughput of requests among the services composing the application [Wang et al., 2021].

One challenge in detecting attacks in microservice applications is determining what specific features should be tracked [Waseem et al., 2021]. Selecting the right features is crucial in production environments, where performance is critical and monitoring every feature is impractical due to the associated overhead [Giamattei et al., 2023]. To address this challenge, our framework enables the use of various monitoring tools during experiments to capture data. Then, by analyzing the generated data during the model development and evaluation phase (see Section 3.3), the researchers can identify the most effective features for attack detection, enabling the optimization of the feature set for production environments.

In practice, during each run, the monitoring tools collect data for each service in the microservice application. Such data are then stored by the **Data Collection** 5 component. The data should preferably be stored in a time-series database as the chronological aspect is essential for efficient organization and analysis.

Query and Preprocessing 6 is responsible for querying the stored *monitored data* and gathering the *workload output* files provided by the workload execution tool. The workload output is needed to provide timestamps for each run, as the workload profile determines the entire data generation process, including the start and end times. Using these timestamps, we can query the monitors' data for the respective run. Although these processes are independent, timestamps enable synchronization between the workload output and the monitored data, allowing for data merging if desired. After retrieving the data, the required preprocessing steps are performed to build the dataset; e.g., it is crucial to label the samples correctly as regular (pre- and post-attack) or as an attack. This labeling is achieved by saving the timestamps of each attack's start and end, allowing precise identification of the periods during which the application was under attack.

3.3 Model Development and Evaluation

Model development and evaluation begin with the *Method Selection*, which involves selecting the method to build models for detecting attacks. Each method type implies a different set of models, represented by the multiple layers of the Model in Figure 3.2. These models must be designed with the application's context in mind. In other words, the context of the microservice application (e.g., low detection precision may not be acceptable due to limited budget), the type of attacks being addressed (e.g., very destructive attacks may demand high recall, regardless of the false positive rate), and the specific singularities regarding the analysis of microservice applications monitoring data (i.e., the need to analyze a large amount of data generated by the several services within the application), may call for a certain family of models. Such variety requires different families of methods (e.g., ML, MCDM, stochastic models) or different algorithms within the same method (e.g., different ML algorithms).

Model Design 7 for runtime attack detection in microservice applications poses several challenges compared to that in monolithic architectures. An attack-detection model must account for the multiple services an application can have. These services have unique functions and can be impacted differently by user requests and attacks. Furthermore, a model must easily adapt to the dynamic nature of microservice applications, where services are continually created and destroyed [Raeiszadeh et al., 2025; Zhang et al., 2023].

Our framework allows setting different *Model Parameters* for a given model. These determine the model's operational characteristics during attack detection, influencing aspects such as learning behavior, feature importance, and decision thresholds. This includes hyperparameters commonly used in ML models, such as learning rates and regularization terms, as well as specific configurations relevant to other techniques, such as weights, attribute criteria, and operators in multi-criteria decision-making. Some techniques can help in parameter selection, such as grid search in ML models. Other strategies, such as Bayesian optimization and genetic algorithms, can also be effective for performing hyperparameter tuning in diverse detection scenarios [Alibrahim and Ludwig, 2021].

Data Processing and Feature Selection are crucial for transforming raw data into a suitable format for building the model. Data processing typically includes feature conversion, normalization, and handling missing values. Feature conversion transforms raw measurements into meaningful indicators. For example, converting CPU usage reported as a cumulative counter into per-interval utilization, or aggregating service communication metrics to derive features such as the rate of different HTTP response codes or the average request duration. Normalization is then applied to scale heterogeneous metrics, and inconsistent records are corrected or removed to ensure data quality. This selection can be based on related work, including techniques such as filter methods (e.g., correlation analysis), wrapper methods (e.g., recursive feature elimination), and embedded methods (e.g., feature importance from decision trees) [Venkatesh and Anuradha, 2019]. In the case of ML models, the dataset is split into training and test sets to evaluate the model's performance accurately.

Model Evaluation 8 consists of assessing the model's performance using the testing set generated by the Data Generation module. This set can either encompass the entire dataset when no training is required or exclude the training data. The model is then executed on the input data to evaluate its attack-detection performance. The details of the evaluation process vary depending on the technique employed. For instance, MCDM or rule-based models may be assessed by comparing predicted conditions against predefined thresholds. In contrast, machine learning models typically rely on quantitative performance metrics (e.g., accuracy, precision, recall) computed from predicted versus expected labels. Techniques like k-fold or leave-one-out cross-validation [Alpaydin, 2020] can ensure robust performance evaluation by iteratively training and testing the model on various subsets of the data. After that, the model's detection results constitute the **Model Output** 9.

This final result is then assessed using various metrics and visual representations during the **Result Analysis** 10. Metrics such as Recall, Precision, and F1-score provide quantitative insights into the model's performance, supporting a comprehensive understanding of its effectiveness. On the other hand, visual representations like box plots and ROC curves offer graphical interpretations of the model's behavior across different data samples from different runs with various attack and workload profiles. This helps identify outliers and understand the distribution of results. A functional model is established if the results are satisfactory; otherwise, the process iteratively returns to the model design component to improve detection capabilities.

3.4 Summary

This chapter presented a comprehensive framework that addresses the lack of realistic datasets and structured methodologies for detecting attacks in microservice applications. The proposal comprises two integrated modules: (i) *Data Generation*, which enables the realistic and flexible emulation of legitimate workloads and diverse attack profiles while collecting representative runtime data from mi-

croservice applications; and (ii) *Model Development and Evaluation*, which supports the systematic construction and assessment of attack-detection models tailored to microservice environments.

The proposed framework supports attack data generation as well as the development and evaluation of models for detecting attacks on microservice applications. Its versatility enables the incorporation of various attack types, supports the development of models using diverse methods, and facilitates comprehensive evaluations across distinct scenarios, helping optimize attack detection performance. The subsequent chapters leverage this framework, systematically demonstrating its practical applicability across increasing levels of system and data complexity.

The framework is first applied to foundational scenarios. Chapter 4 uses the Data Generation module to create a first, static dataset, focusing on core microservice behaviors to support a fundamental analysis of attack effects in a multi-service environment without autoscaling. This dataset then serves as the basis for Chapters 5 and 6, where the Model Development module is used to explore different detection techniques, specifically Logic Scoring of Preference (LSP) and ML, for attack detection in these static conditions.

Building on this initial work and incorporating the experience gained along the way, the framework is then applied to dynamic environments. Chapter 7 leverages the Data Generation module to produce a more complex dataset by introducing new workload profiles that actively trigger service autoscaling and by integrating a broader set of data modalities. Then, the Model Development module is used to develop a novel, specialized model capable of addressing the challenges posed by system scalability and the fusion of these distinct data modalities.

Chapter 4

Generating Data in Static Microservice Applications

The complexity and dynamic nature of modern microservice environments introduce distinctive security challenges, making high-quality, representative data essential for developing effective attack-detection models. By instantiating the Data Generation module of the framework introduced in Chapter 3, we focus on producing labeled datasets to support research on runtime attack detection.

We begin by describing the target microservice application and the overall experimental setup. We then detail how legitimate workloads are generated to exercise the application under realistic operating conditions and how attack traffic is injected into the system. Although the framework is generic and supports multiple attack types through configurable profiles and tools, it is illustrated and validated here using Denial of Service (DoS) attacks, which directly threaten application availability by degrading or preventing responses to legitimate users. In particular, we consider two categories: *high-volume* attacks, which overwhelm the system by issuing large numbers of requests that exhaust available resources, and *low-volume* attacks, which slowly send and maintain open requests to deplete connection-related resources [Jaafar et al., 2019].

The experimental setup comprises the TeaStore microservice application [Von Kistowski et al., 2018], tools for DoS emulation, and monitoring components for data collection and storage. We describe in detail the procedures for workload execution, attack injection, monitoring, and data preprocessing, discuss threats to validity, and conclude with a summary of the main findings. The complete experimental configuration and the generated dataset are publicly available [Castro et al., 2023c].

In summary, the contributions of this chapter are two-fold:

- A detailed description of the emulation of legitimate workloads and attack traffic in a microservice application to generate representative data for runtime attack detection research.

- An extensive experimental campaign demonstrating the instantiation of the framework for both high-volume and low-volume DoS attacks, including the release of a publicly available dataset.

The chapter is organized as follows. Section 4.1 describes the target microservice application and experimental setup. Section 4.2 details workload execution and attack injection. Section 4.3 presents monitoring, data collection, and preprocessing. Section 4.4 discusses the main findings. Section 4.5 discusses threats to validity. Finally, Section 4.6 summarizes the chapter.

4.1 Target Microservice Application

The experimental environment comprises a microservice application under test, the tools used to generate both legitimate and malicious traffic, and the infrastructure required to deploy and monitor the system. Figure 4.1 provides an overview of the architecture and the role of each Virtual Machine (VM).

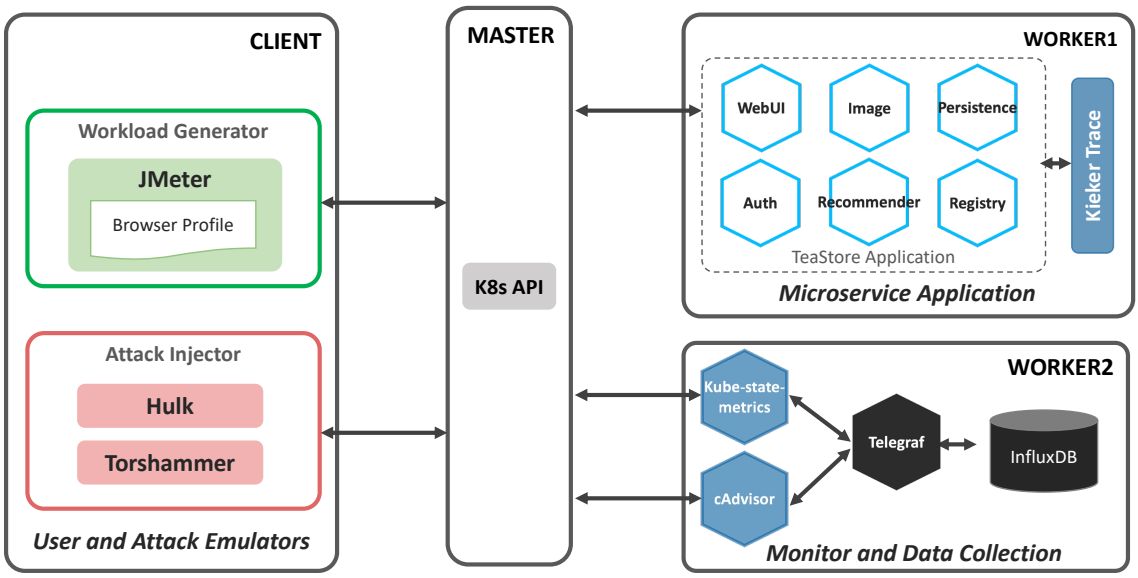


Figure 4.1: Experimental Setup.

The target system is TeaStore [Von Kistowski et al., 2018], a microservice application for benchmarking, evaluating performance modeling, and researching resource management, also used in similar contexts [Caculo et al., 2020; Grohmann et al., 2019]. The TeaStore system contains five services, each serving a specific purpose. The WebUI service is the user interface that provides front-end functionality. The Auth service verifies user login credentials and manages session data. The Image-Provider serves the images displayed on the WebUI. Persistence provides access to the store's database. Finally, the Recommender uses rating algorithms to generate personalized product recommendations for the user. The application also includes a service registry and a backing database. Each service can be replicated, scaled, or removed at runtime.

Table 4.1: Summary of Performance Test (300 users).

		Response Time				
Config	# Samples	Average	Min	Max	Std. Dev.	Throughput
C1	170390	6713	4	32800	4428,26	70,79
C2	306614	3664	2	26882	3200,01	127,54
C3	469869	2387	2	18617	2102,61	195,53
C4	960228	1171	2	8117	1032,33	399,85
C5	1544956	729	2	6021	642,06	643,26
C6	2276090	492	1	6194	518,93	948,15

TeaStore was deployed to a Kubernetes cluster configured for controlled, reproducible experiments. We used the TeaStore deployment with client-side load balancer Ribbon. The original deployment manifests do not include specifications for resource requests and limits, which are essential for ensuring reproducible and controlled performance evaluations in Kubernetes clusters. In Kubernetes, a resource request specifies the minimum CPU and memory a container requires to run reliably. The scheduler uses this value to allocate pods to nodes with sufficient available resources. A resource limit, on the other hand, sets an upper bound on the CPU and memory a container is allowed to consume. If a container exceeds its CPU limit, it is throttled; if it exceeds its memory limit, it may be terminated due to an out-of-memory error.

To determine appropriate resource configurations for each TeaStore microservice, we conducted a series of configuration experiments. Specifically, we set the CPU request and limit to 420 millicores and 500 millicores, respectively, and the memory request and limit to 1 GB and 3 GB, respectively. We then performed a set of experiments incrementally increasing the number of replicas per service, up to a maximum of 5. Following these experiments, we analyzed the CPU consumption of each service and the response times recorded by the workload generator (JMeter), as shown in Table 4.1. This analysis enabled us to assess the resource demands of each microservice within our deployment context and to define the final resource configurations presented in Table 4.2.

Workload generation and attack injection were executed from the Client VM. Legitimate user behavior was emulated using JMeter [Apache Software Foundation, 2022], while high-volume and low-volume DoS attacks were performed us-

Table 4.2: Requests and Limit of CPU and Memory of TeaStore services.

Service	Request CPU	Request Mem.	Limit CPU	Limit Mem.
Auth	400m	1Gi	800m	3Gi
Image	400m	1Gi	800m	3Gi
Persistence	500m	1Gi	1000m	3Gi
Recommender	210m	1Gi	420m	3Gi
WebUI	750m	1Gi	1500m	3Gi

ing Hulk [Grafov, 2016] and Torshammer [Solarstone, 2014], respectively. The remaining VMs hosted the components of the Kubernetes cluster: the Master VM managed the cluster, Worker1 ran the TeaStore services, and Worker2 hosted the monitoring and data-collection tools.

All VMs were provisioned on a host machine equipped with an Intel Core i5-10400 processor (12 logical cores, 2.90 GHz) and 64 GiB of RAM. Four virtual machines were provisioned with the following allocations:*Master* VM was allocated 2 CPU cores and 8GB of RAM, *Worker1* had 6 CPU cores and 32GB of RAM, *Worker2* had 2 CPU cores and 4GB of RAM, while the *Client* had 1 CPU core and 14GB of RAM. These allocations were selected based on available hardware and preliminary experiments profiling the resource consumption of the microservice application, the workload generator, and the attack tools.

4.2 Workload Execution and Attack Injection

The Data Generation module of the framework (Section 3.2) defines five execution stages for each experiment: initialization, pre-attack, attack, post-attack, and completion. These stages ensure a controlled and repeatable execution environment in which both normal and malicious system behavior can be observed under equivalent operational conditions. We respected this structure in our experiments. Each experimental run lasts 35 minutes and follows the general profile shown in Figure 4.2. The run begins with a 10-minute initialization period, during which the workload generator reaches steady state. This is followed by a 5-minute pre-attack phase during which only legitimate traffic is processed, a 10-minute attack window, and a 5-minute post-attack stage. The experiment concludes with a 5-minute completion phase to ensure that the system returns to a stable condition. This design enables us to capture clean observations of system behavior before, during, and after the attack.

Each run produces two categories of data: one set generated under legitimate workload only, and another generated when both workload and attacks are active. To ensure diversity in the collected data, we executed multiple experimental runs that combine different attack profiles and attack types. The 10-minute attack window was selected to accommodate the varying temporal characteristics and frequencies of the different DoS attacks. Consequently, each attack run yielded two distinct types of samples: (1) non-attack samples generated while running only the regular workload, and (2) attack samples generated during the co-existence of the regular workload and the injected attacks. The subsequent sections detail the definition of the legitimate workload and the design and implementation of the attack injection component.

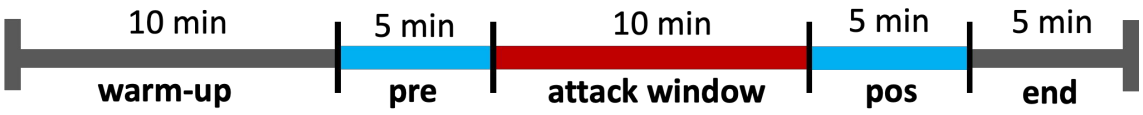


Figure 4.2: Stages of an experimental run.

4.2.1 Workload Generator

Apache JMeter is a widely used tool for load testing and performance evaluation of software applications [Apache Software Foundation, 2022]. It enables the simulation of multiple concurrent users accessing application endpoints, thereby approximating realistic usage scenarios. JMeter provides various metrics to assess system performance, including response time and throughput. To execute a test, a configuration script must be defined that specifies key parameters, such as the target URLs, the number of concurrent users, the ramp-up period (i.e., the time over which the load is gradually increased), and the total test duration.

To generate the workload (i.e., to emulate real users interacting with the application), we used the TeaStore Browsing profile provided by the TeaStore project [Von Kistowski et al., 2018] as the user behavior modeling. Essentially, this script emulates users browsing the store, logging in, adding items to their cart, and discarding them. Lastly, the users log out of the platform. However, further details are needed to run the scripts based on available resources (i.e., the workload intensity specification). For example, we also needed to define the number of users we would use when generating the attacks. To define this, we ran several tests with different numbers of users and analyzed CPU and memory consumption. Additionally, we used a JMeter plugin, *Constant Throughput Timer*, to ensure a constant number of requests per minute. We defined a ramp-up of 1 second, meaning that all the users start simultaneously at the beginning of the run. As we discard the warm-up period, there is no need for a gradual ramp-up.

As a result of this analysis, we defined 300 users. We also analyzed different throughput values per second to stabilize the workload generator. Our goal was to define a value that would consume about 50% to 70% of the available resources, leaving enough for the attackers. Therefore, we compared constant throughput of 6000, 10000, and 12000 requests per minute. The value of 10000 per minute was selected because it met our goal. To define the warm-up period, we ran the script from the previous steps five times. We analyzed the response times recorded by JMeter to identify the minute at which the standard deviation stabilized relative to the rest of the run. This value was determined to be ten minutes.

4.2.2 Attack Injector

The Attack Injector component is designed to emulate application-level DoS attacks. Specifically, we considered two distinct types of DoS: high-volume and low-volume. The high-volume, commonly called a flooding attack, employs regular POST and GET requests to mimic legitimate users. The objective is to overwhelm the target system by consuming its available resources when generating large numbers of requests. On the other hand, a low-volume attack, also known as the slow request attack, sends several HTTP requests slowly. The attacker continually dispatches new requests while keeping the previous ones open, thus exploiting system resources. This process aims to exhaust the resources available for initiating new connections, effectively rendering the application incapable of accepting new requests from legitimate users [Jaafar et al., 2019].

To reflect the fact that DoS attacks may vary in duration, frequency, and target endpoints, we defined four attack profiles. These profiles determine the attackers behavior during the attack window stage, including targeted pages, attack frequency, and attack duration. Figure 4.3 illustrates the profiles. Profile **A-STD** lasts the entire 10 minutes, and we have three sub-profiles, each focusing on attacking a specific page: Homepage, Product Page, and Category Page. These pages use different services to answer user requests. We used the following variants for profile **A-STD**: **STD-Home**, which targets the homepage (the application entry point); **STD-Prod**, which targets the product page that displays information and recommendations for a single product; and **STD-Cat**, which targets a category page listing multiple products. Profiles **B-2M** and **C-INT3** were executed against the homepage. Profile **D-ALT3** implements a three-phase attack within a single run: the first phase targets the category page (first 2 min.), the second phase targets the homepage (3 min.), and the third phase targets the product page (1 min.).

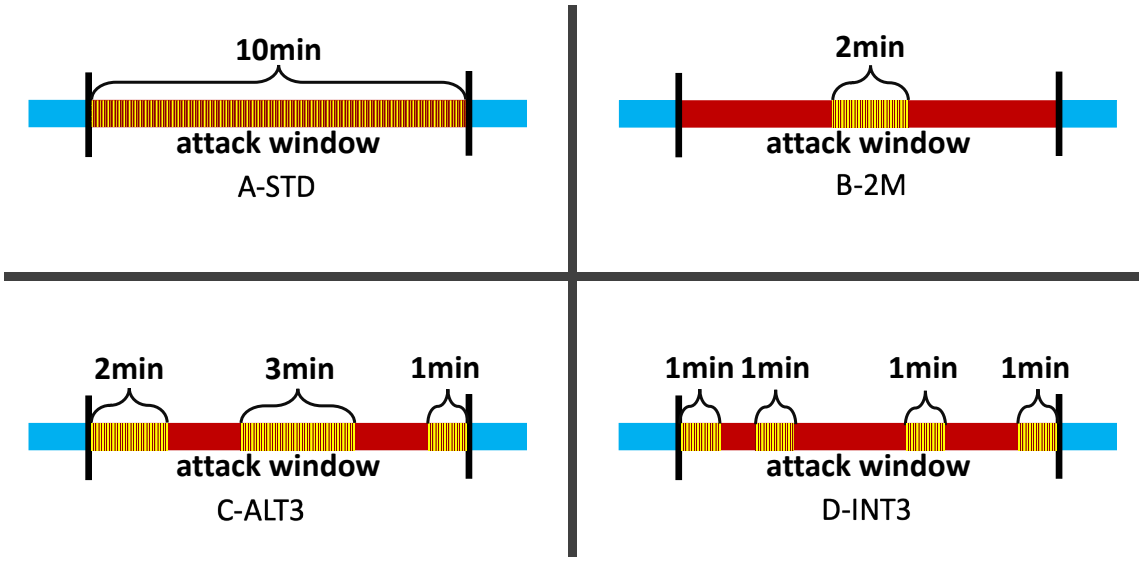


Figure 4.3: Attack profiles.

Once the attack types were selected, we searched for attack tools to perform them in research papers, related web articles, and code repositories, such as GitHub. We collected a total of seventeen tools. However, when we tried to install them, some were not working as expected in our setup, leaving us with four tools: GoldenEye and Hulk for high-volume attacks, and Slowloris and Torshammer for low-volume attacks. To select the ones we would use to perform the attacks, we verified the impact of each tool on resource consumption and response time in our setup. Figure 4.4 shows how each tool impacted the CPU usage of the TeaStore application, and Table 4.3 presents the response time and throughput of client requests during the attack. For comparison, the resources available for each service are those listed in Table 4.2.

Based on the results, we selected Hulk for high-volume attacks and Torshammer for low-volume attacks. Slowloris had basically no effect on the microservice application. In contrast, GoldenEye had a major impact at the beginning of the attack but did not sustain it as the other tools did. Moreover, analyzing Figure

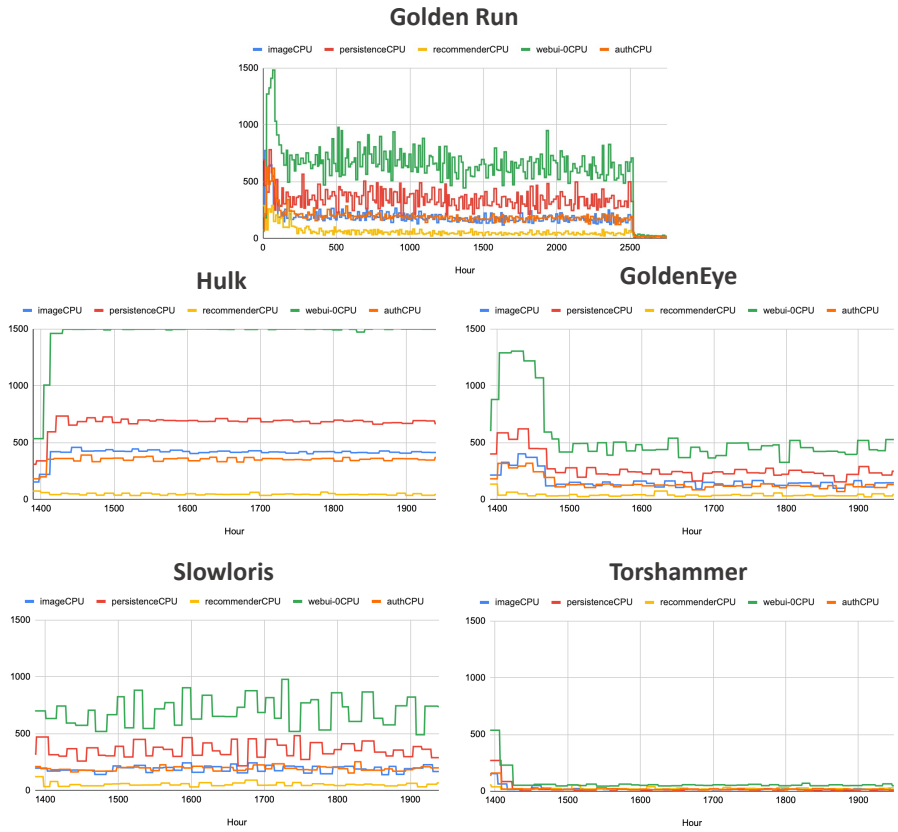


Figure 4.4: Attack tools CPU consumption.

4.4 it is possible to verify how each type of attack impacted application's CPU usage. Hulk increases the application's CPU usage to its limit, as shown in the WebUI, reaching 1500 millicores (i.e., the defined WebUI limit). On the other hand, Torshammer renders the application unable to respond to user requests, reducing CPU usage to practically zero.

Finally, Table 4.4 summarizes the profiles with their sub-profiles, the number of attack runs generated by the Hulk and Torshammer tools, targeted pages, attack frequency, and attack duration. It is important to highlight that Torshammer does not execute all profiles because it cannot target specific application pages. Torshammer operates by attacking the service endpoint at the IP and port level, rather than issuing requests to a particular URL. As a consequence, it is not possible to configure Torshammer to attack different pages within the same experiment, and only the profiles targeting the homepage were used with this tool. As

Table 4.3: DoS Attack Tools.

Tool	# Samples	Response Time	Throughput
Golden Run	120151	23	199.91681
GoldenEye	116175	251	193.29640
Hulk	119987	498	199.26397
Slowloris	120175	49	199.96938
Torshammer	649	462880	1.07205

shown in Table 4.4, a total of 90 experimental runs with different attack types were executed: 60 with the Hulk attack tool and 30 with the Torshammer tool. In addition, 10 golden runs were generated, executing only the regular workload without running any attack profiles.

Table 4.4: Attack Profiles

Profile	Number of Runs per Tool		Pages	Frequency (attack window)	Attack Duration
	Hulk	Tors-hammer			
STD-Home	10	10	Homepage	one attack at min. 0	10 min.
STD-PROD	10	0	Product	one attack at min. 0	10 min.
STD-CAT	10	0	Category	one attack at min. 0	10 min.
2M	10	10	Homepage	one attack at min. 0, 3 or 8	2 min.
INT3	10	10	Homepage	attack at min. 0, 2, 6 and 9	1 min.
ALT3	10	0	Homepage, PROD, CAT	attack at min. 0, 4 and 9	2 min., 3 min., 1 min.
Golden Run	10 runs without attack				

4.3 Monitoring, Data Collection, Query & Preprocessing

To support data generation and collection, it is necessary to use a monitoring tool that captures fine-grained resource usage from each microservice at runtime, operates continuously throughout long-running execution, and integrates easily with Kubernetes. Therefore, after data generation, it is necessary to have tools that can be easily integrated and export the generated data for subsequent preprocessing and labeling.

Given these needs, we surveyed existing observability and monitoring tools commonly used in microservice and cloud-native environments. In particular, we focused on lightweight, widely adopted tools that provide container-level performance metrics. Based on this search, we selected cAdvisor, an open-source monitoring solution that integrates naturally with Kubernetes deployments [Google, 2022] and is used in microservice workloads [Du et al., 2018; Jamshidi et al., 2018; Viktorsson et al., 2020]. cAdvisor provides detailed insight into the resource usage and performance characteristics of running containers. cAdvisor collects, aggregates, processes, and exports information from running containers, such as CPU usage, memory utilization, network statistics, and file system statistics.

Once we selected the monitoring tool, the next step was to choose a suitable agent capable of efficiently collecting and storing the desired attributes of the monitoring application deployed in Kubernetes. For this, we selected the Telegraf agent [InfluxData Inc, 2023b]. We opted for Telegraf, a versatile server agent that

utilizes plugins to collect, process, and transmit features and data from diverse sources. Telegraf integrates perfectly with our chosen database, InfluxDB [InfluxData Inc, 2023a]. InfluxDB was selected for its seamless integration with Telegraf and its overall architecture. It can be easily deployed on Kubernetes as a service, ensuring easy scalability.

After collecting the data, the next step was data preprocessing. At first, the dataset containing the collected data was structured as follows: feature name, timestamp, value, and container name (representing the service within Kubernetes), as shown in 4.5.a. Note that we use the terms 'feature' and 'attribute' as synonyms: ML methods use 'features', while the LSP method uses 'attributes'. These data were initially linearly arranged in a Comma-Separated Values (CSV) file, with features and their values listed sequentially over time. However, to analyze the data, we needed to reorganize them by transposing the feature names into columns and merging the samples by timestamp. This way, each sample contains all feature values for a particular service at a specific time. Table 4.5.b displays a cutout of the dataset (without all columns and rows). Additional merging was necessary to prepare these data for use by ML algorithms. For a given timestamp, samples from all services were aggregated into a single sample, so we expanded the feature names to indicate the service. For example, the feature 'container_sockets' was expanded to include the service names such as 'container_sockets_webui' and 'container_sockets_persistence'. This organizes the data as in the example in Table 4.5.c, which shows only a selection of columns. This process ensured that every sample in the dataset included the values for all features across all services.

Table 4.5: Data preprocessing.

(a) Data as collected

name	time	counter	container
container_cpu_cfs_periods_total	1652046164	8608	teastore-webui
container_cpu_cfs_periods_total	1652046182	8790	teastore-webui
container_cpu_cfs_periods_total	1652046201	8981	teastore-webui

(b) Samples merged per service

time	container	container_cpu_cfs periods_total	timestamp	container_cpu _cfs_throttled _seconds_total	container_cpu _cfs_throttled _periods_total
2022-05-08 21:42:44	teastore-webui	8608	1652046164	834.855	5024
2022-05-08 21:43:02	teastore-webui	8790	1652046182	859.901	5168

(c) Samples merged per application

time	container _sockets _webui	container _threads _webui	type	label	throttled % _webui	cpu_cfs _throttled _seconds _webui	cpu _system _webui
2022-05-08 21:43:21	404	421	hulk	pre	0.524	16.158	0.188
2022-05-08 21:43:34	449	421	hulk	pre	0.772	18.202	0.221

It is important to note that since these features come from a microservice application, the actual set of monitored features is multiplied by the number of running service instances. For example, in our setup, which has one instance per service, we have one feature for the CPU usage of each of the five TeaStore services (e.g. WebUI_CPU_usage, Persistence_CPU_usage).

The dataset is thus composed of multiple files, each representing a golden or attack run. Each run comprises samples organized chronologically, with each sample representing a single observation. Since each run's data collection period lasts 20 minutes (pre-attack, attack, and post-attack stages), with cAdvisor providing monitoring data at an average interval of 15 seconds, this results in approximately 77 (±3.55) samples per run. Note that the 15-second interval was not deliberate; rather, it represents the average frequency at which cAdvisor collects and reports data. While reporting times can vary, this does not affect the results, as the latest value can always be used for each sample, ensuring a consistent dataset. Each sample includes the value of each feature, a timestamp, and a label indicating whether it is an attack or a regular sample. In total, the 100 runs comprise 7,709 samples.

Some samples may have missing data for one or more features. This may be because, at the time of collection, the cAdvisor monitor did not update the value of that feature for a particular service within the 15-second time window. In such cases, we performed data imputation. We used the 'bfill' method [Team, 2024] for the features that vary over time, which filled the missing values with the last known value. For instance, if a missing value occurred for the number of active threads and the previous value was 257, this method filled the missing value with 257.

The features collected by cAdvisor can be of two types: Gauges and Counters. Gauges provide the value of the feature at the time of measurement (e.g., the number of sockets is 280 at time t and 260 at time $t + s$), with no further calculation required. On the other hand, a Counter accumulates the feature values over time (e.g., CPU usage is 400 millicores at time t and 450 millicores at time t+s, meaning that CPU usage between t and t+s is 50 millicores). Therefore, for the counter features, we had to calculate the proper value at all measurement intervals. We computed the normalized value for the features that are based on the available resources as follows:

$$\text{CPU usage} = \frac{\text{cpu\_usage\_seconds\_total}}{\text{spec\_CPU\_quota}} \tag{4.1}$$

$$\text{CPU Throttled} = \frac{\text{container\_cpu\_cfs\_throttled\_periods\_total}}{\text{container\_cpu\_cfs\_periods\_total}} \tag{4.2}$$

$$\text{Memory usage} = \frac{\text{container\_memory\_usage\_bytes}}{\text{container\_spec\_memory\_limit\_bytes}} \tag{4.3}$$

4.4 Findings and Implications

A **significant portion of the effort was devoted to preparing the experimental setup**. This involved tuning microservice resource configurations (i.e., defining resource requests and limits for each service), conducting experiments to determine the optimal workload calibration for the available infrastructure, and designing and validating the attack profiles and tools.

The controlled setup proved effective in maintaining a stable operational baseline while preserving sufficient headroom for attack impact. **By calibrating the workload to approximately 50-70% of resource utilization, the system remained responsive under normal conditions while still exhibiting measurable degradation under attack**. This balance was essential for producing datasets that capture both benign and malicious behavior without prematurely saturating the system.

Since this work focuses on collecting application-level features in a Kubernetes environment, **the monitoring pipeline was straightforward to deploy**, enabling continuous fine-grained monitoring across all microservices. However, this process generated a large volume of data, meaning **more effort was required during data collection and preprocessing to build a consistent dataset**. In particular, it was necessary to ensure that each sample contained synchronized feature values across services. Additionally, due to disk space limitations, the data extraction process had to be performed promptly to avoid data loss. This constraint also led us to discard some collected features which, although not required for the initial experiments, could potentially be useful for future model development.

The selected attack tools demonstrated clearly distinguishable system-level effects. As shown in the CPU consumption analysis (Figure 4.4), high-volume attacks generated by Hulk drove CPU utilization toward the configured limits, particularly in the WebUI service. In contrast, the low-volume Torshammer attack rendered the application unresponsive while CPU usage dropped significantly. These contrasting behavioral signatures are valuable for attack detection models because they expose fundamentally different resource consumption patterns associated with high- versus low-volume DoS attacks.

The use of multiple attack profiles increased the behavioral diversity of the dataset. The inclusion of STD, INT3, 2M, and ALT3 profiles introduced variability in attack duration, frequency, and targeted endpoints. This variability is important for training and evaluating detection techniques, as it reduces the risk of overfitting to a single profile and better reflects real-world attack dynamics.

4.5 Threats to Validity

Potential threats to validity associated with the experimental design and data-generation process presented in this chapter are discussed next.

As this is the first discussion of threats to validity in this work, the three primary categories used consistently throughout the dissertation are defined here.

Construct validity refers to the extent to which the independent and dependent variables are accurately measured, that is, how well the study captures what it intends to measure. **Internal validity** concerns whether the observed effects can be attributed to the experimental factors rather than to uncontrolled or confounding variables. **External validity** addresses the extent to which the findings and generated data can be generalized beyond the specific experimental setting considered.

Construct Validity. We used the TeaStore browser profile for user emulation, as recommended in [Von Kistowski et al., 2018]. This workload profile is representative since it includes requests to various TeaStore web pages. Using other workload profiles, however, may impact the services differently. Therefore, future work should assess the impact of different workload profiles.

Internal Validity. Experiments were conducted in a controlled environment utilizing a local physical machine. To ensure similar conditions across all experiments, the VMs were reset to identical states for each run. However, during each run, one VM (such as the Master VM) may be affected by another VM (such as the Client VM). Additionally, we cannot guarantee the perfect functioning of the VMs and Kubernetes clusters used. However, we analyzed the runs to identify potential issues and repeated the experiments when necessary.

External Validity. We chose the TeaStore application as a representative microservice application for our experiments. The TeaStore application is a well-established reference application in microservice research, widely used by researchers in numerous papers and studies [Caculo et al., 2020; Grohmann et al., 2019]. However, we cannot claim that the generated data can be generalized to other microservice applications.

4.6 Summary

This chapter detailed the complete instantiation of the Data Generation module of our proposed framework, focusing o DoS attacks within a stable, non-scaling microservice environments. We described the deployed experimental setup, the generation of legitimate workloads using JMeter (configured for constant throughput), and the injection of both high-volume (Hulk) and low-volume (Torshammer) DoS attacks across four attack profiles. The comprehensive experimental campaign yielded 100 runs (10 golden and 90 attack runs) and a publicly available, high-quality labeled dataset of 7,709 samples. This dataset serves as a foundational resource for evaluating the effectiveness of the attack detection mechanisms presented in Chapters 5 and 6.

The detailed methodology and outputs presented in this chapter make key contributions by providing two assets to the research community. First, we offer a detailed, fully replicable process for emulating both legitimate and malicious traffic in a realistic microservice application, including all necessary configuration files and environmental variables. Second, we present an extensive, well-documented experimental campaign, supported by the release of a public dataset.

In summary, **determining appropriate configurations for data generation requires substantial effort**. The primary challenge lay in preparing the experimental setup, which involved defining suitable request rates and resource limits for each service of the TeaStore application, as well as carefully designing the workload and attack profiles. This preparation phase proved considerably more time-consuming than the actual data generation process.

Chapter 5

Exploring LSP for DoS Attack Detection

Traditional detection techniques often struggle to capture system-wide behavior, as each service may exhibit different responses under stress or attack. Logic Scoring of Preference (LSP), a multi-criteria decision-making technique based on graded logic [Shen et al., 2021], offers a structured and interpretable way to combine heterogeneous indicators of system behavior. Its flexibility in representing partial evidence and weighting multiple criteria makes it a promising candidate for security monitoring in complex architectures. Although LSP has been used in software engineering to evaluate software systems [Casare et al., 2020; Friginal et al., 2011; Oliveira et al., 2020], its potential for detecting attacks in multi-service environments remains unexplored.

To investigate the applicability of LSP for detecting Denial of Service (DoS) attacks in microservice applications, where the distributed, heterogeneous nature of services complicates security analysis, we adapt the method by treating each service as a contributor to the overall system state. In this setting, each service provides measurable behavioral attributes, such as CPU usage or number of active threads, that reflect its operational condition at a given moment. LSP enables aggregating these heterogeneous indicators into a single global score that characterizes the applications behavior. This unified view facilitates the identification of deviations from normal conditions that may signal an attack. Adopting the perspective of a *critical system*, in which availability and timely detection are essential, we further examine how different LSP components, such as weights, operators, and aggregation means [Dujmovic, 2018], influence the behavior and effectiveness of the resulting detection model.

The contributions of this chapter are the following:

- An analysis and comparison of different methods for defining weights, aggregation operators, and techniques for calculating means, aimed at defining models for microservice applications.

- Two case studies demonstrating the usefulness of the proposed approach in detecting both high- and low-volume DoS attacks, evaluating the model's performance and transferability.

The remainder of this chapter is organized as follows. Section 5.1 describes the proposed LSP-based approach to detect attacks in microservices. Section 5.2 presents the first case study, which focuses on high-volume DoS attacks and researches the applicability of LSP. Section 5.3 introduces a complementary study that examines low-volume DoS attacks and assesses model transferability. Section 5.4 discusses the main findings. Threats to validity are addressed in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.1 LSP for Attack Detection

Our LSP-based approach for detecting DoS attacks against microservice applications consists of the following steps:

Step 1. Decision problem definition.

Step 2. Identification of the elementary attributes and creation of the attribute tree.

Step 3. Specification of the elementary attribute criteria.

Step 4. Definition of the aggregation structure, which involves defining the attributes' weights, selecting the logical aggregation operators, and determining the aggregation formula.

5.1.1 Decision Problem Definition

Monitoring and securing microservice applications can be challenging because they are typically composed of many service instances and have inter-relations between services. In the context of critical services (e.g., business-critical), security is of utmost importance, and resources to address all aspects related to security monitoring, attack detection (and attack prevention) generally exist. Our goal is to use the LSP method to define a model capable of detecting attacks on such *critical* microservice applications (i.e., a model that must not miss any attacks, even if false alarms are occasionally generated). We monitor microservice applications at the application level, enabling us to collect data on several attributes we can analyze to detect attacks.

5.1.2 Attributes and Attribute Tree Definition

A set of *elementary attributes* (e.g., CPU usage, number of active threads) must be identified to evaluate the system, corresponding to the application-level metrics collected during execution. To avoid constructing an unnecessarily complex

model, a subset of relevant attributes can be selected, for example, based on prior work or by identifying functionally equivalent metrics. Once the elementary attributes are defined, the next step is to construct the attribute tree.

Creating the suitability attribute tree in the context of our work brings in novel challenges related to the complexity inherent in the entire microservice application and its associated characteristics. The application is decomposed into different services (e.g., a storage-centric service, a user-interface service, a monitoring service), with each service further decomposing into one or more service instances (i.e., a user-centric service may run across several instances to allow for scalability). This hierarchical structure forms the **suitability attribute tree**. Figure 5.1 illustrates how we define the tree in three levels: the single service instance, the service, and the entire microservice application.

The *Service Instance* corresponds to a unit of a running service, a container (e.g., a single running instance of a UI service). The first tree level comprises the elementary attributes, namely the leaves (a_1, ..., a_n). The attributes are aggregated into categories, ultimately yielding a single score, denoted by *Score* I_j. This score then serves as the input (e.g., *Score* I_1, ..., *Score* I_n) to the second tree level, named *Service*. A single *Service* (e.g., UI service) can have numerous instances; thus, in the second tree level, the scores of all instances of a service (*Score* I_1, ..., *Score* I_n) are aggregated into a single score (e.g., the scores of all UI service instances are aggregated into *Score* S_j). Finally, at the third tree level, the scores (*Score* S_1, ..., *Score* S_n) of all services composing the *microservice application* are aggregated into a *global score* used to detect whether the application is under attack.

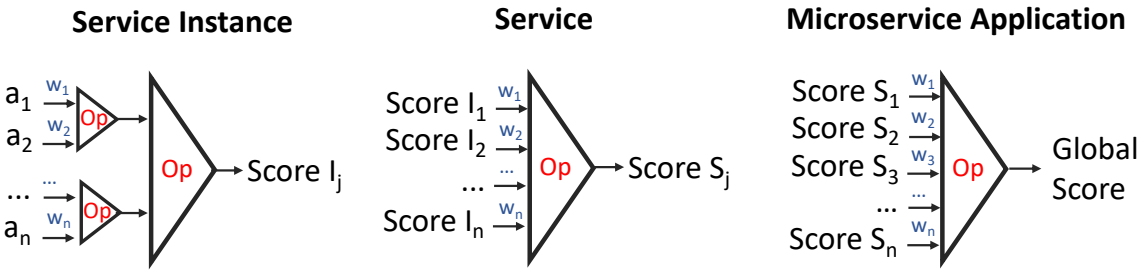


Figure 5.1: Microservice application tree structure.

5.1.3 Specification of Attribute Criteria

The criteria definition maps input values into their corresponding suitability degrees (e.g., Type S, Type L, Type R, as discussed in Section 2.3.2). These attributes can follow preferred small, large, or range values based on the application-layer attributes. It is important to note that different attacks can affect attributes differently (and the same attack can affect them at different times, depending on the application's state). Therefore, analyzing how (and to what extent) each attribute is affected in a specific attack scenario is crucial (e.g., to determine if small values are preferred or within what range values may fluctuate). We use a linear interpolation function [Dujmovic, 2018] to map attribute values to their preferred values. This function simplifies the application of the LSP method [Friginal et al., 2011] while yielding reasonable results [Dujmović and Fang, 2004].

For preferred large-value attributes (the-higher-the-better), we use Equation (5.1). This equation represents a function $s(x)$ with three conditions based on the value of a variable a_i and two given constants, A and B. The first condition declares that if a_i is less than A, the output is set to zero, indicating that the value is below the minimum acceptable threshold. In the context of this work, this is considered an indication of an attack. The second condition states that if a_i is between A and B, the function's output is a linear transformation of the input value a_i. More precisely, the function scales the input value a_i into the range [0, 1] based on the interval between A and B. The resulting value is given by (a_i - A)/(B - A). The third condition states that if a_i is greater than B, the function returns 1, indicating that the attribute's value is unaffected by any malicious activity.

$$s(a_i) = \begin{cases} 0, & a_i < A \\ \dfrac{a_i - A}{B - A}, & A \leq a_i \leq B \\ 1, & a_i > B \end{cases} \tag{5.1}$$

Equation (5.2) describes the function for preferred small values. It follows the same principles as large-values, but in the range [1, 0], where values closer to C are preferred (e.g., the smaller the CPU usage value, the better).

$$s(a_i) = \begin{cases} 1, & a_i < C \\ \dfrac{D - a_i}{D - C}, & C \leq a_i \leq D \\ 0, & a_i > D \end{cases} \tag{5.2}$$

Equation (5.3) defines the function for the preferred range of values. The parameters A, B, C and D define the boundaries of the function:

- [B,C]: the preferred range, where the score is 1.
- [A,B): the lower transition zone, where the score increases linearly from 0 to 1.
- (C,D]: the upper transition zone, where the score decreases linearly from 1 to 0.
- Values outside the interval [A,D] are considered unacceptable and scored as 0.

$$s(a_i) = \begin{cases} 0, & a_i < A \text{ or } a_i > D \\ \dfrac{a_i - A}{B - A}, & A \leq a_i < B \\ 1, & B \leq a_i \leq C \\ \dfrac{D - a_i}{D - C}, & C < a_i \leq D \end{cases} \tag{5.3}$$

Defining the correct A, B, C, and D values for each service's attributes can be challenging. Each service has its own goals and is affected differently by the application's usage. For example, a user interface service may be more requested and consume more CPU than a service that authenticates users. In the absence of a detailed specification or formal information on regular and exceptional operational conditions (e.g., expected sporadic peaks in service usage), these values must be empirically determined by running the application under representative conditions and collecting the necessary data.

5.1.4 Aggregation Structure Definition

Score aggregation requires several design decisions, including assigning appropriate weights to each attribute, selecting suitable operators, and choosing the corresponding weighted mean formula for aggregation.

Weight Definition

The weight of each attribute represents its importance for the attack detection process. There are seven well-known methods for weight definition [Dujmovic, 2018]. Of these seven, we chose the following three because utilizing multiple methods allows for a better understanding of their impact on aggregation results and the overall sensitivity to different weight values: (i) *Direct weight assessment*, (ii) *Weights based on ranking*, and (iii) *Importance decomposition method*.

The **direct weight** method is the simplest way to assign attribute weights. In this method, the weights are defined directly based on the knowledge of the problem and attributes, without further calculation. Researchers and stakeholders may use their expertise to assign weights to each attribute.

In the **weights based on ranking** method, the attributes are ranked in order of importance from the most important to the least important. The weight for each attribute is then calculated using the following Equation 5.4, where i denotes the attribute's position in the ranking. For instance, the most important attribute is assigned a ranking of 1, the second most important attribute is ranked 2, and so on.

$$W_i = \frac{2(n+1-i)}{n(n+1)} \tag{5.4}$$

Finally, the **importance decomposition** method provides both the weight for each attribute and the corresponding *operator* for aggregation. In this method, each attribute is assigned a label representing its overall importance (i.e., its significance to the aggregation, as defined in Table 5.1). This label is associated with a corresponding numeric value S (e.g., S is 14 for Very High) [Dujmovic, 2018]. The weight for each attribute is calculated using the Equation 5.5, where W_i represents the weight for attribute i, and S_i is the overall importance value for that attribute, divided by the sum of the S values of all attributes.

$$W_i = \frac{S_i}{\sum_{j=1}^{n} S_j} \tag{5.5}$$

Next, Equation (5.6) can be used to calculate α and define the operator. Here, S_{max} is the maximum possible degree of importance (16, based on Table 5.1), and n is the number of attributes used. The resulting α indicates the *andness* (conjunction degree) or *orness* (disjunction degree) and can be mapped to an operator using a mapping such as the one in Table 2.1, presented in Section 2.3.2. For instance, if α = 0.0625, the respective operator symbol is D++.

$$\alpha = \frac{\sum_{i=1}^{n} S_i}{n * S_{\max}} \tag{5.6}$$

Table 5.1: Overall importance for importance decomposition method.

Overall importance	S
Highest	16
Slightly below highest	15
Very high	14
Slightly above high	13
High	12
Slightly below high	11
Medium-high	10
Slightly above medium	9
Medium	8
...	...
Lowest	0

Operator Selection

The operators define the relationships between all attributes used in a given aggregation. In our approach, to simplify the aggregation structure definition, we use the set of 17 operators named *GGCD.17* previously presented in Table 2.1, Section 2.3.2. This set of operators is recommended for most applications due to its balanced combination of flexibility, precision, and simplicity [Dujmovic, 2018].

The selection of the operator group represents only the initial phase of the operator definition. The subsequent and challenging task is specifying the appropriate logical aggregation operators for all existing aggregations (i.e., defining all *Op* labels shown in Figure 5.1). Furthermore, once the operator is defined, it is necessary to determine its level (for example, one of the four levels of hard partial disjunction in Table 2.1). The resulting model can have many possible combinations. Therefore, two methods are generally employed for operator definition: (1) directly defining the operators based on our knowledge of the problem and data, or (2) using the operators derived from the earlier mentioned *importance decomposition weight method*.

Aggregation Formula Selection

The final step is to compute the aggregation, which requires a weighted mean calculation [Miranda et al., 2019], such as arithmetic, geometric, or harmonic. To assess the impact of different methods for calculating means, we selected three methods to compare. The Weighted Power Means (WPM) is the most frequently used in LSP due to its ability to effectively capture the concept of overall importance, as observed in human evaluation reasoning [Dujmovic, 2018]. The other two means, Weighted Arithmetic Means (WAM) [Oliveira et al., 2020] [Dujmović and Allen III, 2021] and Weighted Harmonic Means (WHM) [Shen et al., 2021] [Dujmovic, 2007], are also employed in some works. The corresponding formulas are as follows:

- *WAM*:
$$y = \sum_{i=1}^{n} w_i x_i, \tag{5.7}$$
where w_i represents the weight for each attribute value x_i.

- *WPM*:
$$y = \left(\sum_{i=1}^{n} w_i x_i^r \right)^{1/r}, \tag{5.8}$$
where w_i represents the weight for each attribute value x_i, and r is the value obtained from the logic operator as presented in Table 2.2, Section 2.3.2.

- *WHM*:
$$y = \frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} \frac{w_i}{x_i}}, \tag{5.9}$$
where w_i represents the weight for each attribute value x_i.

5.2 Detecting High-volume DoS Attacks

This case study focuses on high-volume DoS attacks and compares the impact of different LSP component definitions on detection performance. All detailed results and further information necessary to replicate the experiments are available at [Castro et al., 2023c]. Our experimental goals are the following:

- Understanding the general applicability of LSP in the detection of DoS attacks in the context of microservices.
- Providing practical insights regarding the different methods for defining several components of the LSP technique (weights, operators, and mean calculations).
- Understanding to what extent an LSP-based model can be adjusted to improve particular metrics of interest, such as recall.

5.2.1 Definition of the LSP Models

Having set the **Step 1. Decision problem definition**, we proceeded with the **Step 2. Elementary attributes and creation of the suitability attribute tree**. First, we selected the elementary attributes based on previous work in similar contexts [Castro et al., 2022; Du et al., 2018; Grohmann et al., 2019]. The selected attributes are: Number of Sockets, Number of Threads, CPU usage by the user (CPU user), CPU usage by the system (CPU system), Memory usage, Number of throttled period intervals (Throttled%), and duration that a container has been throttled (CFS_throttled).

After selecting the attributes, we constructed the respective tree shown in Figure 5.2. The tree presents all the elementary attributes and the aggregation that produces the score S_j. Here, j represents WebUI, Auth, Persistence, Image, and Recommender services. The scores for each service are then aggregated into the global score.

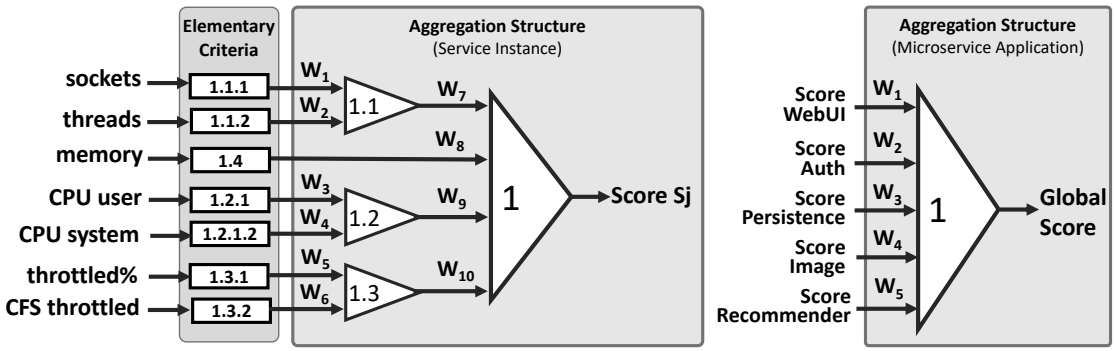


Figure 5.2: Suitability attribute tree for detecting DoS attacks.

Before we proceed with the next steps, we must detail the values used for the attribute criteria, weight, and operators. These values, initially defined as **Set A**, were used in the first experimental iteration, which focused on the high-volume attack scenario (the focus of this section). However, following the analysis of these initial experiments, a second set of values, **Set B**, was defined. This refinement was implemented specifically to increase recall.

Next, we go through the **Step 3. Specification of the elementary attributes criteria**. Table 5.2 shows the attributes *criteria A* and *B*, where for each service, the [min, max] value of all elementary attributes is defined. In the case of high-volume DoS attacks, all attributes follow the *the-lower-the-better* function, described in Equation 5.2.

To define the values for *Criteria A*, we visually analyzed the fluctuation of the values of each attribute during a few runs. We selected a random run for each attack profile (e.g., the fifth run of the INT3 profile, previously described in Subsection 4.2.2) and reviewed the attribute values to observe their behavior under regular and attack circumstances. Figure 5.3 shows an example of the plot to be analyzed that depicts the CPU usage per service for a standard profile run. This analysis, along with our empirical knowledge, allowed us to define the attribute

Table 5.2: Criteria for high-volume DoS attacks.

Criteria A

Measure	WebUI	Auth	Image	Persistence	Recomme.
Sockets	[600,1000]	[40, 140]	[50, 150]	[50, 180]	[10, 70]
Threads	[400, 500]	[250, 270]	[250, 270]	[200, 240]	[70, 100]
CPU user	[0.6, 0.8]	[0.3, 0.6]	[0.5, 0.8]	[0.4, 0.7]	[0.2, 0.5]
CPU system	[0.15, 0.25]	[0.05, 0.1]	[0.11, 0.16]	[0.12, 0.19]	[0.03, 0.08]
Memory usage	[0.3, 0.4]	[0.17, 0.18]	[0.2, 0.25]	[0.14, 0.16]	[0.1, 0.11]
Throttled%	[0.4, 0.8]	[0.05, 0.1]	[0.05, 0.5]	[0.05, 0.2]	[0.1, 0.11]
CFS throttled	[10, 30]	[0, 2]	[0, 20]	[1, 5]	[0, 1]

Criteria B

Measure	WebUI	Auth	Image	Persistence	Recomme.
Sockets	[590, 900]	[50, 80]	[50, 150]	[100, 200]	[10, 750]
Threads	[330, 450]	[250, 270]	[250, 270]	[200, 240]	[70, 100]
CPU user	[0.5, 0.8]	[0.34, 0.4]	[0.5, 0.8]	[0.4, 0.6]	[0.2, 0.25]
CPU system	[0.2, 0.25]	[0.05, 0.09]	[0.11, 0.16]	[0.14, 0.2]	[0.02, 0.04]
Memory usage	[0.38, 0.41]	[0, 0.1]	[0.2, 0.25]	[0.14, 0.16]	[0, 0.11]
Throttled%	[0.4, 0.8]	[0.17, 0.18]	[0.05, 0.2]	[0.02, 0.17]	[0, 0.1]
CFS throttled	[12, 35]	[0, 2]	[0, 5]	[1, 5]	[0, 1]

criteria (i.e., minimum and maximum values) for each service, as well as the type of criterion function to be associated with each attribute.

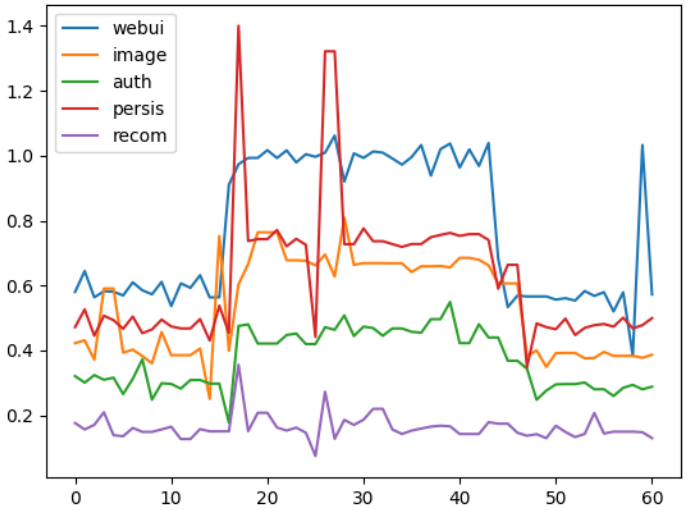


Figure 5.3: CPU usage per service graph.

Regarding the *Criteria B*, we analyzed boxplots for all attributes. Each attribute was analyzed using two boxplots: one with regular values and the other with attack momentum values. Before drawing the boxplots, we removed outliers by applying a z-score transformation. Based on [Chikodili et al., 2020], a cut-off value of 3 was used. I.e., any value with a distance value greater than 3 was eliminated. Figure 5.4 shows an example of the analyzed boxplots, where the Y-axis shows the attribute values and the X-axis shows the different services. Boxplots display the distribution of the data, allowing us to visualize the center, spread, and outliers. Thus, the boxplots enabled the identification of differences and pat-

terns between the services and attributes, allowing us to define a more narrow and precise interval to improve attack detection in the critical system scenario (i.e., improving *recall*).

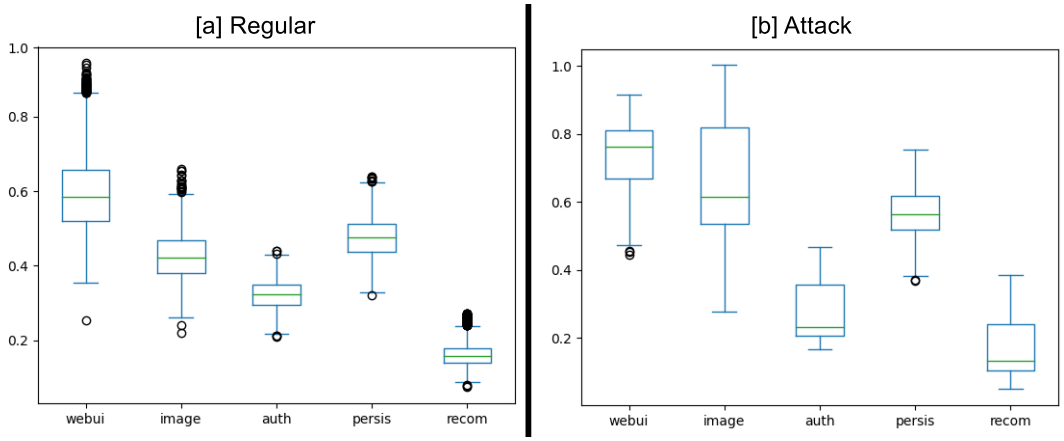


Figure 5.4: CPU usage per service graph for (a) Regular and (b) Attack samples.

The final step is the **Step 4. Definition of the aggregation structure**, i.e., weights and operators. We apply the *direct definition* (Direct in Table 5.3), *weights based on ranking* (Ranking), and the *importance decomposition* (Decomp.) methods to **define the weights**. To **define the operators**, we defined them directly (Dir.) based on our empirical knowledge, and we also applied the *importance decomposition* (Dec.) method. The resulting defined weights and operators at the instance level can be visualized in Table 5.3, while those for the service level are presented in Table 5.4.

Table 5.3: Weights and operators for high-volume DoS attacks: instance-level.

Weight A and Operator A																
Attributes	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.	Classification	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.
Sockets	0.7	1	0.67	15	0.6	C-	C+-	Parallel Processing	0.3	3	0.2	15	0.29	DA	D+-
Threads	0.3	2	0.33	10	0.4										
CPU user	0.7	1	0.67	16	0.67	C-	CA	CPU usage	0.4	1	0.4	16	0.31		
CPU system	0.3	2	0.33	8	0.33										
Throttled%	0.7	1	0.67	14	0.64	C-	C-+	CPU thrtottled throttled	0.2	2	0.3	12	0.24		
CFS_throttled	0.3	2	0.33	8	0.36										
Memory usage									0.1	4	0.1	8	0.16		
Weight B and Operator B															
Attributes	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.	Classification	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.
Sockets	0.9	1	0.67	16	0.7	CA	C+-	ParallelProcessing Processing	0.35	2	0.3	15	0.34	D-	D+-
Threads	0.1	2	0.33	7	0.3										
CPU user	0.9	1	0.67	16	0.67	CA	CA	CPU usage	0.45	1	0.4	16	0.36		
CPU system	0.1	2	0.33	8	0.33										
Throttled%	0.6	1	0.67	13	0.62	CA	C-+	CPU thrtottled throttled	0.15	3	0.2	10	0.23		
CFS_throttled	0.4	2	0.33	8	0.38										
Memory usage									0.05	4	0.1	3	0.07		

All the combinations used in this experiment are presented in Table 5.5. We considered the attribute criteria, operators, and weight definitions to create various configurations. This resulted in eight groups, each with three unique configurations, for a total of 24 possible configurations. For example, configuration *cfg20* belongs to group G and uses the attribute criteria B, the weights A defined by the *ranking* method, and the operators A defined directly. Regarding the threshold definition (the value used for comparison with the global score to determine

Table 5.4: Weights and operators for high-volume DoS attacks: service-level.

Weight A and Operator A							
Services	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.
webui	0.3	1	0.33	16	0.31		
auth	0.2	4	0.13	7	0.14		
image	0.2	2	0.27	14	0.27	D–	DA
persistence	0.2	3	0.2	10	0.2		
recommender	0.1	5	0.07	4	0.08		
Weight A and Operator A							
Services	Direct Weight	Ranking R	Ranking Weight	Decomp. S	Decomp. Weight	Operators Dir.	Operators Dec.
webui	0.4	1	0.33	16	0.33		
auth	0.1	4	0.13	5	0.1		
image	0.3	2	0.27	14	0.29	D–	DA
persistence	0.2	3	0.2	10	0.2		
recommender	0.1	5	0.07	4	0.08		

the presence of an attack), we empirically analyzed the effect of using values 0.4, 0.5, and 0.6 before running our experimental campaign (Section 5.3 further explores the impact of different thresholds). We analyzed the prediction results of the standard-prod runs obtained with the configuration *cfg1*, ultimately selecting 0.5 due to its superior results. Thus, if the global score is ≥ 0.5, the application is deemed not under attack. Conversely, if the score is < 0.5, the application is considered to be under attack.

Table 5.5: All configuration combinations.

Group	A			B			C			D		
Config.	cfg1	cfg2	cfg3	cfg4	cfg5	cfg6	cfg7	cfg8	cfg9	cfg10	cfg11	cfg12
Criterion	A	A	A	A	A	A	B	B	B	B	B	B
Weight	Direct_A	Ranking_A	Decom_A	Direct_A	Ranking_A	Decom_A	Direct_B	Ranking_B	Decom_B	Direct_B	Ranking_B	Decom_B
Operator	DIR_A	DIR_A	DIR_A	DEC_A	DEC_A	DEC_A	DIR_B	DIR_B	DIR_B	DEC_B	DEC_B	DEC_B
Group	E			F			G			H		
Config.	cfg13	cfg14	cfg15	cfg16	cfg17	cfg18	cfg19	cfg20	cfg21	cfg22	cfg23	cfg24
Criterion	A	A	A	A	A	A	B	B	B	B	B	B
Weight	Direct_B	Ranking_B	Decom_B	Direct_B	Ranking_B	Decom_B	Direct_A	Ranking_A	Decom_A	Direct_A	Ranking_A	Decom_A
Operator	DIR_B	DIR_B	DIR_B	DEC_B	DEC_B	DEC_B	DIR_A	DIR_A	DIR_A	DEC_A	DEC_A	DEC_A

5.2.2 Experimental Results

The most significant results from our initial experimental case study are discussed next. The metrics used to evaluate the models are those presented in Subsection 2.3.1. The complete set of results is available at [Castro et al., 2023c].

Weighted Mean Formulas and Operators Impact

Three weighted mean formulas, namely *WAM*, *WPM*, and *WHM*, were employed and compared in our experiments. Table 5.6 presents the average results for each metric type across all experiments.

Table 5.6: Average means results.

	precision	recall	f1	markedness	informedness
WAM	0.43	0.98	0.56	0.37	0.22
WPM	0.64	0.79	0.65	0.55	0.5
WHM	0.37	1	0.52	0	0

As shown in Table 5.6, while WHM had a recall of 1.0, its markedness and informedness were both 0.0, indicating that it classified most instances as attacks, making it unsuitable as the exclusive mean formula. WAM outperformed WPM in recall but exhibited poor precision and an F_1 score. Furthermore, since the WAM method represents neutrality in LSP, it does not exhibit significant variation across the various configurations shown in Table 5.5; the only influencing factor is the change in weight values. Figure 5.5 illustrates the scores over time using the WPM [a] and the WAM [b]. We can observe that WPM scores show a more significant drop during the attack and tend to cross the threshold line with greater amplitude.

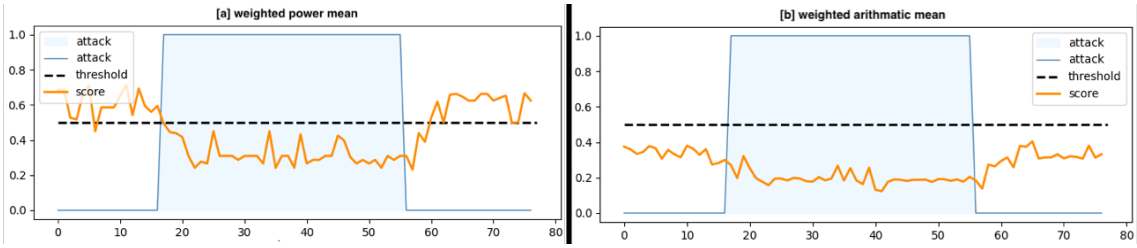


Figure 5.5: WPM and WAM scores over time of a standard attack profile attacking a TeaStore product page in CFG10.

Table 5.7 compares the average results for two configurations (cfg7 and cfg10) using both WAM and WPM. These configurations are similar, differing only in their operators: cfg7 employs the direct operator B (DIR_B), while cfg10 uses the importance-decomposition operator B (DEC_B).

Table 5.7: Comparing WAM and WPM means.

cfg7	precision	recall	f1	markedness	informedness
WAM	0,54	0,98	0,66	0,54	0,51
WPM	0,41	0,98	0,55	0,31	0,17
cfg10	**precision**	**recall**	**f1**	**markedness**	**informedness**
WAM	0,54	0,98	0,66	0,54	0,51
WPM	0,81	0,88	0,83	0,78	0,81

As shown in Table 5.7, WAM values remain essentially unchanged across both configurations. In contrast, WPM values clearly change across both configurations. These results show us that **the definition of operators in LSP, i.e., simultaneity and substitutability operators, can impact the result**. Therefore, the WAM is simple and sufficient for decision problems where neutrality applies (or is close to it). Conversely, the WPM method is preferable when simultaneity and

substitutability operators are important. Accordingly, we only discuss the results obtained using the WPM aggregation in the remainder of this section.

Weight Methods

We analyzed the effects of using direct, ranking, and importance-decomposition methods for weight definition to assess their differences. Additionally, we sought to identify any potential issues with the direct method and determine if it should be deemed less reliable.

The direct, ranking, and importance decomposition methods exhibited varying performance depending on the configuration used (as shown in Table 5.5), with no clear winner across all configurations. Table 5.8 presents the average results for the three weight methods on the left side, while the right side shows the best results for each type of weight definition.

Table 5.8: Weight methods results.

Average results				**Best results**			
Weights	**precision**	**recall**	**f1**	**Configs**	**precision**	**recall**	**f1**
Direct	0.65	0.74	0.62	cfg10	0.81	0.88	0.83
Ranking	0.64	0.82	0.67	cfg11	0.75	0.86	0.77
Decom.	0.64	0.79	0.65	cfg12	0.72	0.89	0.77

As shown on the left-hand side of Table 5.8, the average results indicate that the ranking method achieved the highest recall of 0.82, followed by the decomposition method with 0.79 and the direct method with 0.74. This may provide a general view of each method's applicability across various scenarios; however, in practice, we need to analyze the best results obtained with a single configuration. So, we show the best results on the right-hand side of Table 5.8, which were obtained with *cfg10* for direct, *cfg11* for ranking, and *cfg12* for the importance decomposition method. We can see that the direct method in *cfg10* achieved the best-balanced results across all metrics. This can be attributed to the direct method's greater flexibility and ease of use, which enable straightforward prioritization of the attributes most influential in attack detection.

Although the direct method is often regarded as the easiest way to achieve better results, it heavily relies on the person defining the weights. According to the results presented in Table 5.8, the ranking and importance decomposition methods are not far behind and have the potential to yield results as good as those of the best-performing methods with further refinement. Nonetheless, the **direct method** remains viable as it **may help us achieve our goals more efficiently**. Moreover, the ranking and importance decomposition methods may be useful when a deep understanding of the system and its behavior patterns is lacking. **Each method has its strengths and limitations**, and the selection of the most appropriate method may depend on the specificities of the context where it is being used.

LSP Applicability to Our Scenario

The performance metric values collected during our experiments range from 0.8 to 0.9, with some fluctuation. It is important to note that the numbers shown represent the average across all attack profiles. Some attack profiles, such as the 2-Minute (2M) attacks, pose significant detection challenges for the model (due to their relatively short attack window of only 2 minutes). Conversely, our experiments with the longest attack profiles, such as standard and standard-prod, showed that excellent results could be achieved, particularly with model cfg10, with all metrics exceeding 0.93. The results for both of these cases (i.e., 2M and PROD) are shown in Table 5.9.

Table 5.9: Standard and 2M attack profiles cfg10 WPM average results.

Profile	precision	recall	f1
PROD	0.93	0.99	0.96
2M	0.74	0.78	0.73

Our experiments show the applicability of LSP to our scenario, achieving relatively high metric values. Some of the most interesting results were obtained from group D configurations using the WPM formula and importance decomposition for the operators, as shown in Table 5.7 and 5.8. In Group D, all aggregation elements were defined according to Set B of criteria, weights, and operators, highlighting their importance and the ways they can affect the aggregation result. Defining everything with a focus on improving a particular metric (in our case, recall) is possible in LSP, at least considering our own scenario. Overall, the results emphasize the challenges and the importance of carefully selecting and defining LSP elements to achieve better results.

5.3 Detecting Low-volume DoS Attacks and Model Transferability

Building upon the previously defined LSP configuration, we apply new models to explore the applicability of LSP to low-volume DoS attacks and to assess model transferability, that is, the ability of a single configuration to detect both attack types. In doing so, we preserve the established attribute tree while redefining the attribute criteria, weights, and operators.

Low-volume attacks

For the low-volume attacks, we present a single configuration for the attribute elementary criteria, weights, and operators in the LSP model. While the impact of different configurations was analyzed for high-volume attacks, repeating that analysis here would significantly lengthen this section. Instead, we later present the analysis of the impact of varying thresholds for the global score. The weights

and operators are visualized in Table 5.10 and they match the tree structure in Figure 5.2, while Table 5.11 presents the attribute elementary criteria.

Table 5.10: Weights and Operators for Low-Volume DoS Attacks.

Attributes	**Weights**	**Operators**	**Weights**	**Operator**	**Scores**	**Weights**	**Operators**
Sockets	0.5	C-	0,15	D+-	WebUI	0.5	DA
Threads	0.5				Auth	0.15	
CPU user	0.7	C-	0.4		Persisten.	0.15	
CPU system	0.3				Image	0.1	
Throttled%	0.7	C-	0.1		Recomm.	0.1	
CFS throttled	0.3						
Memory			0.35				

In this case, we experimented with global score thresholds of 0.9, 0.85, and 0.80 to assess how they affect the LSP model's detection performance. The results for each threshold value are presented in Table 5.12. As can be observed, the threshold of 0.90 achieved a high recall of 0.97 but a precision of 0.70. When the threshold was 0.85, the recall was slightly lower (0.94), but the precision was considerably higher (0.87). The 0.80 threshold yielded near-zero performance across all metrics.

Table 5.11: Criteria for Low-volume DoS Attacks.

Measure	**Criterion Function**	**WebUI**	**Auth**	**Image**	**Persistence**	**Recomme.**
Sockets	lower-the-better	[450,580]	[40, 140]	[50, 150]	[100, 180]	[20, 70]
Threads	range	[300,400, 400,500]	[100,250, 250,270]	[100,250, 250,270]	[150,240, 240,300]	[20,70, 70,100]
CPU user	higher-the-better	[0.3, 0.5]	[0.2, 0.3]	[0.2, 0.4]	[0.2, 0.4]	[0.1, 0.2]
CPU system	higher-the-better	[0.1, 0.15]	[0.03, 0.05]	[0.05, 0.11]	[0.1, 0.15]	[0.02, 0.05]
Memory	higher-the-better	[0, 0.2]	[0, 0.18]	[0, 0.2]	[0, 0.13]	[0.0, 0.1]
Throttled%	higher-the-better	[0.1, 0.4]	[0.1, 0.2]	[0.1, 0.2]	[0.05, 0.1]	[0, 0.1]
CFS throttled	higher-the-better	[0, 10]	[0, 2]	[0, 0]	[0, 5]	[0, 1]

Table 5.12: LSP Model for Low-Volume Dos Attacks Results.

Threshold	**Precision**	**Recall**	**F1**	**Markedness**	**Informedness**
0.90	0.70	**0.97**	0.77	0.64	0.78
0.85	**0.87**	0.94	**0.89**	**0.85**	**0.87**
0.80	0.08	0.04	0.04	0.07	0.04

Transferability between LSP models

Before addressing transferability, a dedicated model for high-volume attacks was defined and refined based on extensive knowledge of the LSP. Unlike the model used in the *first case study*, this revision focused on recalibrating specific attribute criteria values and weightswhile maintaining the same set of logical operatorswith the primary objective of improving the precision metric (see Tables 5.3 and 5.7 for comparisons). Therefore, Table 5.13 presents the criteria used in this model. The weight and operators are presented in Table 5.14.

Table 5.13: Criteria for High-volume DoS Attacks.

Measure	Criterion Function	WebUI	Auth	Image	Persistence	Recommender
Sockets	lower-the-better	[450,1000]	[40, 140]	[50, 150]	[100, 180]	[10, 70]
Threads	lower-the-better	[400, 500]	[250, 270]	[250, 270]	[200, 240]	[70, 100]
CPU user	lower-the-better	[0.6, 0.8]	[0.3, 0.6]	[0.5, 0.8]	[0.4, 0.8]	[0.2, 0.5]
CPU system	lower-the-better	[0.15, 0.25]	[0.05, 0.1]	[0.11, 0.16]	[0.12, 0.2]	[0.05, 0.08]
Memory	lower-the-better	[0.3, 0.4]	[0.18, 0.2]	[0.2, 0.25]	[0.14, 0.16]	[0.1, 0.11]
Throttled%	lower-the-better	[0.4, 0.8]	[0.1, 0]	[0.05, 0.5]	[0.05, 0.2]	[0.1, 0.11]
CFS throttled	lower-the-better	[10, 30]	[0, 2]	[0, 20]	[0, 5]	[0, 1]

LSP enables the construction of a model that aggregates the outputs from two or more models. Therefore, LSP allowed us to develop a unified model composed of models for high-volume and low-volume attacks. We used the models we previously defined for high-volume and low-volume DoS attacks. Subsequently, we merged the global scores from both models to create a new, final global score. For low-volume attacks, we used the values shown in Table 5.10, while for high-volume attacks, we used the CFG1 configuration that yielded the best F_1-score. To perform the aggregation, we applied the neutrality operator, defined as the arithmetic mean, which reflects a balance between substitutability and simultaneity. Given that the models operated under different thresholds for attack classification (0.85 for low-volume attacks and 0.70 for high-volume attacks), we balanced these thresholds using weights of [0.4, 0.6] for low-volume and high-volume attacks, respectively. The final aggregation is represented by Equation (5.10).

$$R = \begin{cases} 0, & 0.6 \times LV\_Score + 0.4 \times LV\_Score \geq 0.85 \\ 1, & 0.6 \times LV\_Score + 0.4 \times HV\_Score < 0.85 \end{cases} \tag{5.10}$$

where R is the final result for an observation, and LV_Score and HV_Score are the global score for low- and high-volume models, respectively.

The results in Table 5.15 show the average performance for high- and low-volume attacks, as well as the overall average for both attack types. The results showed that applying a higher threshold to the high-volume model increased recall but led to more false positives, thereby lowering precision. On the other hand, all performance metrics for low-volume attacks declined, likely because the low-volume model was focused on recall. Nevertheless, the overall performance for combined attacks yielded an average precision of 0.74, recall of 0.91, and F_1-score of 0.79. The current models were developed independently, each focusing on a

Table 5.14: Weights and Operators for High-Volume DoS Attacks.

Attributes	Weights	Operators	Weights	Operator
Sockets	0.7	C-	0.3	D+-
Threads	0.3			
CPU user	0.7	C-	0.4	
CPU system	0.3			
Throttled%	0.7	C-	0.2	
CFS throttled	0.3			
Memory			0.1	

Scores	Weights	Operators
WebUI	0.3	DA
Auth	0.2	
Persisten.	0.2	
Image	0.2	
Recomm.	0.1	

specific attack type, which resulted in different threshold values for global scores (0.70 for high-volume attacks and 0.85 for low-volume attacks). In addition, both models were defined under a *higher-is-better* interpretation for classifying regular system behavior. The models were designed with a 'higher-is-better' criterion for classifying the application as regular, limiting the effective use of a disjunction operator in the final aggregation, which could account for higher substitutability between the two models.

Table 5.15: LSP results for high-, low-volume, and combined attacks.

	Precision	Recall	F1	Markedness	Informedness
High-volume	0.74±0.13	0.95±0.06	0.81±0.09	0.70±0.11	0.77±0.06
Low-volume	0.73±0.16	0.83±0.22	0.76±0.18	0.68±0.19	0.72±0.21
Overall Avg.	**0.74±0.19**	**0.91±0.17**	**0.79±0.19**	**0.70±0.22**	**0.75±0.21**

5.4 Findings and Implications

The LSP method offers excellent flexibility, allowing model elements to be rebalanced to meet specific requirements, such as increasing recall for critical systems. In our scenario, the **direct weight method proved effective**, eliminating the need for more complex, meticulous weight methods.

A significant advantage of the LSP method is that it **does not require historical data for model training**; understanding the application's typical behavior is sufficient for establishing attribute criteria that identify normal behavior. Therefore, our results suggest that the LSP method can be successfully used to detect DoS attacks in stable microservice applications.

LSP enables aggregating existing LSP modes into new models, providing flexibility to update and add models as needed to address emerging threats without requiring training data. This adaptability enables the same LSP configuration to detect various types of attacks, including previously unseen ones. However, as shown, for the LSP models to perform effectively, they must be constructed with attributes and thresholds within similar ranges.

Defining an appropriate LSP model is even more challenging for microservice applications because it is necessary to define suitable elementary attribute criteria for each type of service. Our recommendation is to use LSP models when sufficient human resources are available to analyze system behavior, define model components precisely, and create and tune multiple models iteratively.

The current models were developed independently, each focusing on a specific attack type, resulting in different final-score thresholds (0.70 for high-volume and 0.85 for low-volume). Ideally, both models should produce similar global scores with more closely aligned thresholds. This can be achieved by redefining the LSP components to ensure the values are more closely ranged. Finally, the models were designed using a 'higher-is-better' criterion to classify the application as

regular. This limited the use of an appropriate disjunction operator, which could have accounted for higher substitutability between the two models.

The aggregation of independently defined LSP models introduces structural constraints that affect model transferability and expressiveness. The independent development of the high- and low-volume models led to different threshold values for the global score. This disparity affects how the models can be combined into a unified detection framework. Moreover, the adoption of a *higher-is-better* interpretation for regular behavior constrains the choice of aggregation operators, favoring neutral combinations over operators that assume higher substitutability between model outputs.

5.5 Threats to Validity

Construct Validity. The LSP method supports a wide range of models with different structures, attributes, weights, criteria, and operators. This high degree of flexibility introduces a threat, as an insufficient exploration of the parameter space could lead to suboptimal model selection. We mitigated this by thoroughly exploring and comparing multiple methods for defining the most critical components (weights, operators, and weighted means), establishing a suitable set of LSP components for DoS attack detection.

Internal Validity. Our experiments demonstrated the effectiveness of the LSP method in detecting a small set of attacks (high-volume and low-volume DoS). Since other attack types and configurations may affect the system differently, the model's strong performance is confirmed only for the profiles evaluated in this study. We adopted only the recommended browser profile from [Von Kistowski et al., 2018] for user emulation, which may not reflect other usage patterns that could influence service behavior and potentially alter the attribute scores. Although we attempted to mitigate this by employing a disjunction operator for aggregation, full validation under varied user profiles is left for future work.

External Validity. Our experimental design required us to limit several variables. Specifically, we focused on a single application (TeaStore) and stabilized only one instance per service, which restricts the generalizability of our findings to other microservice architectures. Our goal was to assess the fundamental applicability of the LSP method in this environment. However, further work is needed to evaluate this applicability in more dynamic settings, where microservice applications can scale up or down quickly, increasing the number of services involved in the score computation. The method should also be assessed in different operational scenarios, such as low-budget applications where a high number of false positives would be unacceptable. Exploring these conditions is essential to understand the approach's limits and broader potential fully.

5.6 Summary

This chapter explored the applicability and potential of the LSP method as a Multi-Criteria Decision-Making (MCDM) approach for detecting DoS attacks in microservice applications. The approach hierarchically aggregates measurable application-level attributes into a single global score, providing a clear indication of potential attack conditions.

An empirical analysis compared different strategies for defining weights (Direct, Ranking, and Importance Decomposition), operators (Direct and Importance Decomposition), and weighted means (WAM, WPM, and WHM). The results show that the weighted power mean (WPM) outperformed the other aggregation methods, as it better captured the influence of logical operators such as simultaneity and substitutability, which are central to LSP. Overall, the method demonstrated its viability for DoS detection, achieving average recall values above 80%.

In addition, a unified model was constructed by aggregating independently defined high-volume and low-volume detectors using an arithmetic mean. For combined attack scenarios, this configuration achieved an average precision of 0.74, a recall of 0.91, and an F_1-score of 0.79, illustrating LSPs ability to compose and adapt models without retraining.

In summary, LSP provides a flexible and effective solution for DoS detection in microservice environments, allowing model components to be tuned to specific security requirements. At the same time, these conclusions are bound by the experimental conditions considered and should not be directly generalized to more dynamic or large-scale environments.

While the LSP-based approach showed promising results and important advantages such as interpretability and independence from training data, it is essential to compare it with data-driven techniques. Therefore, the next chapter investigates supervised and unsupervised ML models under the same experimental conditions, enabling a comparison between LSP and ML models, further discussed in Section 6.4.

Chapter 6

Developing ML Models for DoS Attack Detection

The use of Machine Learning (ML) has become widespread for detecting threats in modern computing systems [Ahmad et al., 2021; Ahsan et al., 2022; Alzaabi and Mehmood, 2024]. However, identifying Denial of Service (DoS) attacks in microservice architectures remains particularly challenging due to their distributed and dynamic nature [Chamberlain et al., 2025; Haindl et al., 2024; Lee et al., 2023].

Building on the Model Development and Evaluation module of the framework introduced in Chapter 3, we investigate the application and evaluation of various ML models in a controlled, reproducible experimental environment, enabling a systematic assessment of their effectiveness for runtime DoS attack detection. This analysis complements the previously studied LSP-based approach by examining a data-driven modeling paradigm under the same experimental conditions. A suite of ML models is developed, evaluated, and compared, providing a structured basis for analyzing trade-offs between Multi-Criteria Decision-Making (MCDM) and ML techniques in microservice attack detection.

The main contributions of this chapter are as follows:

- We provide the first thorough comparative analysis of supervised ML and unsupervised ML for DoS detection in the context of microservices applications, highlighting their strengths and limitations.
- Based on our findings, we provide a set of actionable recommendations for both practitioners and researchers on selecting appropriate ML algorithms and input features to detect high- and low-volume DoS attacks.

The chapter is organized as follows. Section 6.1 describes how we selected the features to use as input for the ML models. Section 6.2 presents the Supervised ML Models training and testing, and results. While Section 6.3 presents the Unsupervised ML Models training and testing, and results. Section 6.4 discusses the main findings. Finally, Section 6.5 presents the threats to validity, and Section 6.6 summarizes the section.

6.1 Feature Selection

To identify the most relevant features for detecting DoS attacks in microservice applications, we first conducted an exploratory analysis and subsequently assessed feature importance using the Random Forest algorithm. In our exploratory analysis, we analyzed how the values of each feature fluctuated over time. This involved creating graphs to visually assess the behavior of each feature across runs, specifically comparing attack periods with regular workloads. This way, we identified the features that exhibited value changes during attacks. In addition, we calculated the average and standard deviation of these features, providing a better understanding of their behavior. We also compared values across different features, allowing us to identify those that, despite different names, exhibited the same patterns and values. For example, if the memory_usage_bytes value in a sample were 1000, memory_working_set_bytes would also be 1000; when the former increased to 1200, the latter would also be 1200. To avoid redundancy, we selected only one feature in such cases.

We also explored feature importance using the Random Forest algorithm to determine the significance of each feature in the model's decision-making process. This helped us to understand which features contribute the most to accurately detecting high- and low-volume DoS attacks by measuring the impact of individual features on the model's performance.

Based on the exploratory analysis and feature importance analysis, we selected the features with the greatest impact on detection performance (Table 6.1). As shown in the 'Model' column, the supervised models used a total of 35 features. In contrast, the unsupervised models, except the autoencoder models, used either 1 or 2 features per service, resulting in 5 or 10 features at a time across the five services. This grouping strategy was adopted because such models, particularly traditional ones like Isolation Forest, tend to perform better with fewer input features. However, deep learning-based models, such as Autoencoders, can handle a larger number of features. More details are in section 6.3.

Table 6.1: Selected Features.

Name	Description	Model
cpu_cfs_periods_total	Elapsed enforcement period intervals	Sup., Unsup.
cpu_cfs_throttled_seconds	Duration container has been throttled	Sup., Unsup.
cpu_user_seconds_total	User CPU time consumed	Sup.
cpu_system_seconds_total	CPU time consumed	Sup.
cpu_usage_seconds_total	Combined CPU time consumed of user and system	Unsup.
memory_usage_bytes	Memory usage	Sup., Unsup.,
sockets	Open sockets for the container	Sup., Unsup.,
threads	Threads running inside the container	Sup., Unsup.

6.2 Supervised ML Models

This section presents the development and evaluation of supervised ML models for DoS attack detection in microservice applications. The following subsections describe the experimental setup for training and testing these models with the goal of distinguishing between regular and attack behaviors across different attack profiles, followed by the performance results in high-volume, low-volume, and combined attack scenarios.

6.2.1 Training and Testing Supervised ML Models

When developing ML models, two important steps are Model Training and Testing. During training, a model is trained on a portion of the dataset to identify patterns. This involves iteratively adjusting the model parameters to improve its predictive performance. Model testing assesses the trained model on a separate portion of the dataset.

Figure 6.1 illustrates the process used for training and testing the supervised models, which consisted of three stages. In the first stage, all runs (golden runs and attack runs) were loaded. Since each run was stored in a separate file, we compiled a list of filenames for all runs in the dataset and shuffled the filenames using Python's *random.shuffle* method, with a fixed seed value (we used 42 in these experiments), ensuring that all ML experiments consistently use the same shuffled dataset. The second stage is the training process, while the third stage is dedicated to testing the model. Given the limited number of runs, we structured the training and test sets according to the Experiment-wise Leave-One-Out Cross-Validation (ELOOCV) procedure [Campos et al., 2023].

Using ELOOCV allowed us to train the classification models on a large portion of the dataset in each iteration. Specifically, we trained the model on all golden runs and all attack runs except one, which was reserved as the test set. This process was repeated, ensuring that each attack run was used exactly once in the test set. For example, when training with low-volume attack runs and golden runs, the dataset consisted of 40 runs (approximately 3,080 samples). Thus, we trained the model on 39 runs (approximately 3,003 samples) and tested it on a single attack run (approximately 77 samples). In this case, the process was repeated 30 times, one for each attack run. Thus, the results presented in this chapter represent the average performance across all 60 runs for high-volume attacks and across all 30 runs for low-volume attacks. It is important to note that the model classifies each sample in the test run as either regular or attack, and not the run as a whole.

We selected five supervised ML algorithms to build models for detecting DoS attacks: K-Nearest Neighbors (KNN), Random Forest (RF), Support Vector Machines (SVM), XGBoost (XGB), and Convolutional Neural Network (CNN). While the first four followed standard implementation, the CNN required additional configuration. First, we reshaped the data to a one-dimensional format to ensure compatibility with the CNN architecture. Furthermore, the defined model consisted of a sequential structure that began with a convolutional layer applying the

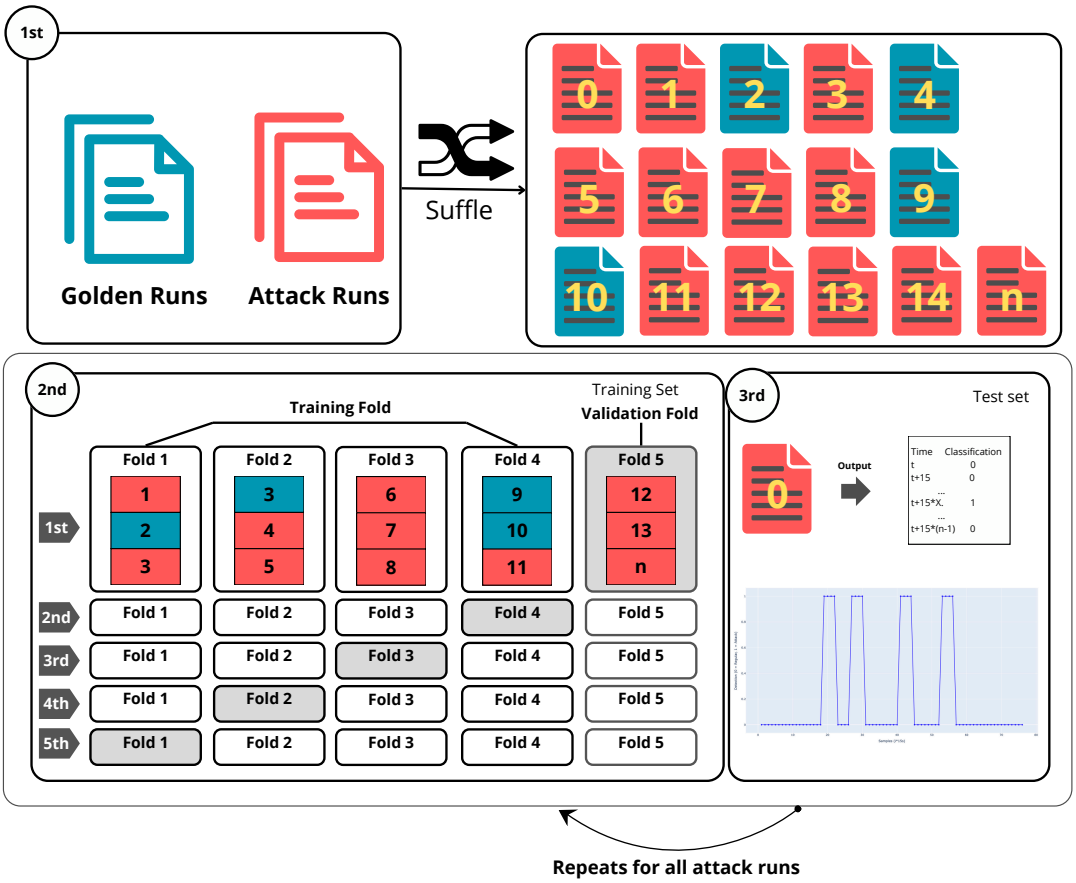


Figure 6.1: Supervised ML model training and testing.

ReLU activation function. The input shape was set to match the reshaped data dimensions. Following the convolutional layer, a max pooling layer was added to reduce the spatial dimensions of the output, enhancing feature extraction with a pool_size of 2. A dropout layer was included to mitigate overfitting. The output from the pooling layer was then flattened and passed through a dense layer with 100 units and a *ReLU* activation function. To further address overfitting, another dropout layer was applied before the final output layer, which used a sigmoid activation function to produce binary classifications. The model was compiled with the *Adam optimizer* and trained using *binary cross-entropy loss*, with accuracy as the performance metric.

We employed a Grid Search Cross-Validation approach using the *GridSearchCV* method from sklearn during training. For this, we used the *GroupKFold* method with 5 splits for 5-fold cross-validation. Since the runs were independent, each run was treated as a group. This ensured that when *GridSearchCV* divided the data into folds, all samples from the same run remained in the same fold. Note that this process was repeated for each ELOOCV fold, ensuring that parameter tuning was specific to the training data of each fold. Grid search for parameter tuning identified the best parameter settings for each model, thereby improving classification performance.

The parameters used for each supervised ML algorithm in the grid search are detailed in Table 6.2 and correspond to the *Model Parameter* module shown in Figure

3.2(B). After the parameter tuning, the best model was tested on the previously excluded attack run. Specifically, the model classified each of the 77 test samples as either regular or an attack. Therefore, if the model were deployed to identify attacks at runtime, it could indicate whether the application is under attack every 10 or 15 seconds.

Table 6.2: Supervised ML Algorithms and Hyperparameters.

Algorithm	**Parameters**
RF	n_estimators: [100, 200, 300], max_features: [0.1, 'sqrt', 'log2'], min_samples_leaf: [1, 2, 5], min_samples_split': [2, 4, 5], random_state: [42]
SVM	C: [1, 10, 100], kernel: ['linear', 'rbf', 'poly'], gamma: ['scale', 'auto'], degree: [2, 3],
KNN	n_neighbors: [3, 5, 10, 20], weights: ['uniform', 'distance'], metric: ['euclidean', 'manhattan']
XGB	max_depth: [3, 5, 7], learning_rate: [0.1, 0.01, 0.001], subsample: [0.5, 0.7, 1], random_state: [42]
CNN	filters: [16, 32], kernel_size: [3], dropout_rate: [0.2, 0.5], learning_rate: [0.0001, 0.001], epochs: [25], batch_size: [77], kernel_-regularizer: [l2(0.001)]

6.2.2 Results for Supervised ML Models

The results for high-volume DoS attacks are presented in Table 6.3, where we see that XGBoost yielded the best results across all performance metrics, with recall, precision, and F1-score of 0.98, and markedness and informedness of 0.97. Following XGBoost, RF showed strong performance, with results ranging from 0.96 to 0.98 across the various metrics. SVM also performed well, with scores ranging from 0.94 to 0.97, followed by the CNN, which had a recall of 0.96 but a markedness of 0.93. Finally, the KNN achieved an F1-score of 0.92, but had the lowest markedness of 0.85. Despite being last, KNN still delivered good results, particularly regarding recall and F1-score. Note that Subsection 2.3.1 presented the performance metrics used in this chapter.

Table 6.3: Supervised ML Performance for High-Volume DoS Attacks.

	Precision	Recall	F1-score	Markedness	Informedness
RF	0.97±0.07	0.98±0.03	0.98±0.05	0.96±0.07	0.97±0.03
SVM	0.96±0.13	0.97±0.13	0.96±0.13	0.94±0.18	0.95±0.13
KNN	0.91±0.15	0.93±0.13	0.92±0.14	0.85±0.19	0.91±0.13
XGB	**0.98±0.04**	**0.98±0.03**	**0.98±0.02**	**0.97±0.04**	**0.97±0.03**
CNN	0.95±0.14	0.96±0.13	0.95±0.13	0.93±0.19	0.95±0.13

The results indicate that XGBoost is highly effective at correctly identifying and classifying high-volume DoS attacks, similar to RF, which is also based on decision trees. On the other hand, SVM, KNN, and CNN models performed poorly,

particularly in detecting attacks from the ALT3 and INT3 profiles. This difficulty contributed to a higher standard deviation in their performance metrics, indicating less consistent performance than with XGBoost and RF.

Table 6.4 shows the results for low-volume DoS attacks. These results differ slightly, especially in recall. When compared to high-volume DoS attacks, the three best-performing algorithms, RF, SVM, and XGB, had worse performance in terms of recall. However, the other two, KNN and CNN, had slightly better recall. Still, XGBoost achieved the best results across all performance metrics, except recall, tying with RF for precision (0.98) and with SVM for informedness (0.97). In low-volume DoS attacks, the precision was slightly higher than the recall for all the supervised ML algorithms, except for SVM, which had slightly higher recall than precision. Like high-volume DoS models, these models struggled to detect attacks with the ALT3 profile. Remember that the low-volume DoS attacks lacked an INT3 profile.

Table 6.4: Supervised ML Performance on Low-Volume DoS Attacks.

	Precision	Recall	F1-Score	Markedness	Informedness
RF	**0.99±0.02**	0.95±0.07	0.97±0.04	0.94±0.03	0.94±0.07
SVM	0.93±0.18	0.94±0.18	0.94±0.18	0.92±0.20	0.93±0.18
KNN	0.97±0.04	0.94±0.07	0.95±0.05	0.93±0.05	0.93±0.07
XGB	0.98±0.03	**0.97±0.06**	**0.98±0.04**	**0.98±0.04**	**0.96±0.06**
CNN	0.98±0.04	0.97±0.05	0.97±0.04	0.97±0.04	0.96±0.06

When comparing the low- and high-volume attack results, we observed that the average recall for all ML algorithms was slightly lower for low-volume attacks, whereas precision was marginally higher. This suggests that low-volume DoS attacks may be harder to detect than high-volume DoS attacks.

We also built ML models using both attack types, with the results presented in Table 6.5. The training was conducted according to the process shown in Figure 6.1, where the attack run files of both types were combined and shuffled. Consistent with the previous findings, XGBoost (XGB) achieved the highest performance across all metrics, followed closely by the CNN, SVM, and RF algorithms. KNN, however, performed worse, with metric values around 0.90.

Table 6.5: Supervised ML Performance for Combined DoS Attacks.

	Precision	Recall	F1-score	Markedness	Informedness
RF	0.97±0.06	0.97±0.08	0.97±0.07	0.96±0.08	0.96±0.09
SVM	0.97±0.04	0.98±0.04	0.97±0.03	0.96±0.05	0.97±0.05
KNN	0.92±0.15	0.92±0.16	0.92±0.15	0.90±0.19	0.90±0.16
XGB	**0.98±0.04**	**0.98±0.04**	**0.98±0.03**	**0.97±0.04**	**0.97±0.05**
CNN	0.97±0.06	0.98±0.05	0.97±0.05	0.96±0.06	0.96±0.05

To provide additional insights, we conducted an in-depth analysis of the ML models' performance across specific attack profiles, enabling us to identify the profiles that posed the greatest challenges for the models. These results, divided by attack profile, are presented in Tables 6.6 and 6.7 for high- and low-volume

attacks, respectively. For high-volume attacks, XGB struggled with the INT3 profile, and RF achieved the lowest precision with 2M, followed closely by INT3. When it came to recall, RF struggled the most with INT3. On the other side, KNN, SVM, and CNN encountered the most challenges with the ALT3 profile. For low-volume attacks, most algorithms struggled with the INT3 profile. The only exception was SVM, for which the 2M profile was the most challenging. When analyzing the models' performance across both attack types, SVM, XGB, and CNN showed consistent difficulty with the INT3 profile on both precision and recall. RF maintained similar precision across the ALT3, INT3, and 2M profiles, with ALT3 being the most challenging for recall. Lastly, KNN faced the most difficulty with ALT3 for precision and with INT3 and 2M for recall.

Table 6.6: Supervised ML Performance for High-Volume DoS Attacks per Attack Profile.

High-volume					
RF	precision	recall	f1	markedness	informedness
PROD	0.99	0.99	0.99	0.98	0.98
CAT	1.00	0.99	0.99	0.99	0.99
INT3	0.93	0.87	0.90	0.90	0.85
ALT3	0.96	0.97	0.96	0.94	0.94
2M	0.91	0.94	0.92	0.91	0.93
STANDARD	0.98	0.98	0.98	0.96	0.96
SVM	precision	recall	f1	markedness	informedness
PROD	0.97	0.98	0.98	0.95	0.95
CAT	0.98	0.91	0.93	0.91	0.89
INT3	0.87	0.89	0.88	0.84	0.85
ALT3	0.82	0.84	0.83	0.78	0.79
2M	0.80	0.80	0.80	0.79	0.79
STANDARD	0.97	0.98	0.98	0.95	0.95
KNN	precision	recall	f1	markedness	informedness
PROD	0.99	0.99	0.99	0.98	0.98
CAT	1.00	1.00	1.00	0.99	0.99
INT3	0.94	0.93	0.93	0.92	0.91
ALT3	0.88	0.85	0.86	0.85	0.84
2M	0.89	0.88	0.83	0.88	0.86
STANDARD	0.98	0.97	0.97	0.95	0.95
XGB	precision	recall	f1	markedness	informedness
PROD	1.00	0.99	0.99	0.99	0.99
CAT	1.00	1.00	1.00	1.00	1.00
INT3	0.96	0.96	0.96	0.95	0.95
ALT3	0.97	0.97	0.97	0.95	0.95
2M	0.99	0.98	0.98	0.99	0.98
STANDARD	0.98	0.99	0.99	0.97	0.98

Table 6.7: Supervised ML Performance for High-Volume DoS Attacks per Attack Profile.

Low-volume					
RF	precision	recall	f1	markedness	informedness
INT3	0.99	0.83	0.90	0.94	0.82
2M	0.98	0.92	0.94	0.97	0.92
STANDARD	0.99	0.99	0.99	0.98	0.98
SVM	precision	recall	f1	markedness	informedness
INT3	0.92	0.82	0.87	0.87	0.80
2M	0.89	0.97	0.90	0.89	0.92
STANDARD	0.99	0.98	0.99	0.97	0.97
KNN	precision	recall	f1	markedness	informedness
INT3	0.93	0.93	0.92	0.91	0.91
2M	0.83	0.84	0.83	0.82	0.83
STANDARD	0.99	1.00	0.99	0.99	0.99
XGB	precision	recall	f1	markedness	informedness
INT3	0.98	0.92	0.95	0.96	0.92
2M	0.98	0.92	0.94	0.97	0.92
STANDARD	0.99	0.99	0.99	0.98	0.98

6.3 Unsupervised ML Models

In this section, we explore the application of unsupervised ML for DoS attack detection. Unlike supervised approaches, these models are trained exclusively on golden runs, enabling anomaly detection without requiring labeled attack data. We evaluate a set of traditional and deep learning-based unsupervised algorithms and analyze their performance under high- and low-volume attack scenarios. The following subsections detail the training and testing process and present the experimental results.

6.3.1 Training and Testing Unsupervised ML Models

The unsupervised models were trained exclusively on the golden runs, enabling us to develop anomaly detection models without requiring attack data during training. Specifically, the training set consisted of the ten golden runs, around 770 samples, and the testing set incorporated all attack runs, i.e., around 4,620 samples from high-volume and 2,310 samples from low-volume attack runs.

We selected five unsupervised algorithms: Isolation Forest (ISOF), One-Class SVM (OCSVM), Local Outlier Factor (LOF), Autoencoder (AE) and Variational Autoencoder (VAE). Note that for ISOF and LOF, we used the same model for both low- and high-volume DoS attacks. However, further analysis was needed to improve OCSVM's results. Specifically, since the OCSVM's results for high-volume attacks were unsatisfactory, we analyzed different kernel parameters and

normalized the data using Z-score. We found that the models that achieved better results for high-volume attacks differed from those that performed well for low-volume attacks. Therefore, for high-volume we used normalized data and an RBF kernel, while for low-volume attacks we used non-normalized data and a linear kernel. The parameters for unsupervised ML algorithms are in Table 6.8.

The AE with all features includes an encoder with three Dense layers (32, 16, and 8 neurons), followed by a 4-neuron bottleneck. The decoder mirrors this structure in reverse (8, 16, 32 neurons). For the grouped features model, due to reduced input size, the encoder uses layers with 8, 4, and 2 neurons, and omits the bottleneck. All layers use ReLU activation. Models were compiled with the Adam optimizer and Mean Squared Error (MSE) loss. Anomalies were detected based on reconstruction error (Mean Absolute Error), with a threshold set at the 99th percentile of the training loss, selected after analyzing the long-tailed error distribution and comparing performance across multiple percentiles. The VAE mirrors the AE structure, but replaces the bottleneck with Dense layers to compute the latent mean (z_mean) and log variance (z_log_var), and uses a single-sampling approach.

Table 6.8: Unsupervised ML Algorithms and Parameters.

Algorithm	**Parameters**
ISOF	n_estimators: 100, max_samples: auto, random_state: [7, 21, 42, 123, 999], contamination: auto
OCSVM	kernel: [linear, rbf, sigmoid, poly], gama: auto
LOF	novelty: True
AE	learning_rate: 0.0001, epochs: 100, batch_size: 64, validation_split: 0.2, loss_function: mse
VAE	learning_rate: 0.0001, epochs: 100, batch_size: 64, validation_split: 0.2, loss_function: [mse, KL divergence]

For deterministic models like OCSVM and LOF, the training and testing process was straightforward: we trained the models on the golden runs and tested them on the attack runs. However, a slightly different strategy was required for the ISOF algorithm due to its inherent randomness during training. Specifically, we divided the attack runs into five folds, using each fold as the test set while training the model on the golden runs with one of five different *random_state* (seed) parameters. The seeds used for each experiment are in Table 6.8. This ensured a robust evaluation of the model's performance across different random initialization scenarios. For the AE and VAE, which were implemented using TensorFlow, TensorFlow was configured with the option enable_op_determinism.

Note that we selected groups of features to train the models using unsupervised algorithms rather than supervised ones. Initially, we experimented with the same set of features, but because traditional unsupervised models are inherently unsupervised, this yielded poor results, with recall values as low as 1%. To improve the performance, we reduced the number of features by defining four groups, where each feature is collected per service, resulting in 5 or 10 input features depending on the group size. The groups are: (1) CPU usage (CPU: 1 Œ 5); (2) CPU

and memory usage (C&M: 2 Œ 5); (3) sockets and threads (S&T: 2 Œ 5); and (4) CPU CFS periods and throttled seconds (C&T: 2 Œ 5). In contrast, deep learning-based models (AE and VAE) can handle a larger number of features, and we also trained them using the full feature set.

6.3.2 Results for Unsupervised ML Models

Table 6.9 presents the best results of combining the unsupervised ML algorithms with the different sets of features. For high-volume DoS attack detection, the VAE-S&T model consistently outperformed other configurations across most metrics. It achieved the highest scores for precision (0.88), F1-score (0.92), and markedness (0.85), and the second highest for informedness (0.90). Although it was not top-ranked in recall (0.99), the difference from the top result was marginal. The AE-S&T model followed closely, achieving the best result for informedness and the second-best for all other metrics.

The ISOF model with the S&T feature group excelled in recall (0.99), despite ranking lower on other metrics and having a lower precision of 0.67. In contrast, ISOF-CPU achieved a more balanced outcome, ranking third in precision (0.80), F1-score (0.84), markedness (0.74), and informedness (0.78), demonstrating better performance than its S&T model. The LOF algorithm performed best when combined with the S&T feature across most metrics, except for precision, where LOF-CPU slightly outperformed it (0.74 vs. 0.72). This shows the S&T feature group is particularly effective for identifying high-volume attacks, especially when paired with deep learning models such as AE and VAE.

Table 6.9: Performance of the Combinations of Unsupervised ML Algorithms and Selected Groups of Features.

		High-volume attacks			
Ranking	Precision	Recall	F1-Score	Markedness	Informedness
1st	**VAE-S&T (0.88±0.14)**	**ISOF-S&T (0.99±0.02)**	**VAE-S&T (0.92±0.1)**	**VAE-S&T (0.85±0.23)**	AE-S&T (0.91±0.14)
2nd	AE-S&T (0.86±0.15)	AE-S&T (0.99±0.02)	AE-S&T (0.92±0.130	AE-S&T (0.84±0.24)	VAE-S&T (0.90±0.14)
3rd	ISOF-CPU (0.80±0.17)	LOF-S&T (0.99±0.03)	ISOF-CPU (0.84±0.16)	ISOF-CPU (0.74±0.25)	ISOF-CPU (0.78±0.18)
		Low-volume attacks			
Ranking	Precision	Recall	F1-Score	Markedness	Informedness
1st	**AE-CPU (0.84±0.17)**	**ISOF-C&M (0.99±0.04)**	**AE-CPU (0.86±0.10)**	**AE-CPU (0.78±0.34)**	**VAE-CPU (0.76±0.10)**
2nd	VAE-CPU (0.84±0.17)	OCSVM-CPU (0.98±0.04)	VAE-CPU (0.86±0.15)	VAE-CPU (0.78±0.33)	AE-CPU (0.83±0.20)
3rd	LOF-CPU (0.71±0.19)	LOF-C&M (0.96±0.05)	LOF-CPU (0.80±0.16)	LOF-CPU (0.69±0.21)	LOF-CPU (0.81±0.20)

In low-volume attack detection, AE-CPU achieved the best overall performance, ranking first in precision (0.84), F1-score (0.86), and markedness (0.78), and sec-

ond in informedness (0.83), and obtained a high recall of 0.92. VAE-CPU also showed strong results, comparable to those of AE-CPU. The difference was only in the third or fourth decimal place.

Despite not topping recall, the LOF-CPU model provided the most balanced performance among the non-deep learning alternatives, ranking third in precision (0.71), F1-score (0.80), markedness (0.69), and informedness (0.81). Even though the LOF algorithm did not reach the top of the recall ranking, it still delivered a solid recall of 0.96. This result contrasted with the best result obtained by LOF in the high-volume attacks, where the S&T feature was more effective. The ISOF algorithm was not in the top ranks, but it also yielded the best results using the CPU feature. For recall, the ISOF model using the C&M feature was ranked first with 0.98, but it performed poorly on the other metrics, such as the F1-score of 0.46. This indicates that the ISOF with C&M achieved high recall only because it incorrectly labeled most samples as attacks. The ISOF with CPU did not achieve the highest recall, but it still performed well at 0.95. The OCSVM algorithm showed high recall, ranking second, but performed poorly on the other metrics. For example, OCSVM-C&M had a precision of 0.47, an F1-score of 0.59, markedness of 0.37, and informedness of 0.58.

VAE and AE models detected high-volume attacks effectively, but their performance dropped significantly for low-volume attacks. VAE-CPU had the best precision (0.78) but only reached an *F1 Score* of 0.43, revealing its difficulty maintaining balance under low-volume conditions. Overall, for low-volume attacks, CPU usage was the most crucial feature across all unsupervised ML algorithms. As with supervised ML models, unsupervised ML models detected low-volume DoS attacks more slowly than high-volume attacks, and the most challenging attack profiles to detect were INT3, 2M, and ALT3.

We also evaluated the performance of models using the complete set of features. While ISOF, LOF, and One-Class SVM struggled in this configuration, AE and VAE achieved strong results, particularly in high-volume attack detection. VAE had a precision of 0.86, a recall of 0.93, and an F1-score of 0.89, while AE followed closely with scores of 0.83, 0.98, and 0.89, respectively. However, both models underperformed on low-volume attacks. For these attacks, VAE achieved a precision of 0.57, a recall of 0.94, and an F1-score of 0.68, while AE achieved 0.42, 0.95, and 0.53, respectively. Notably, when considering a single model for detecting both high- and low-volume attacks, the VAE trained with the complete feature set achieved the best overall performance, with a precision of 0.76, recall of 0.93, and F1-score of 0.82. These results suggest that VAEs and AEs could benefit from an expanded, more informative feature space.

6.4 Findings and Implications

To better understand the relative strengths and limitations of the different detection paradigms, we compare the results obtained in this chapter for supervised and unsupervised ML models with those reported in Chapter 5 for the Logic Scoring of Preference (LSP) models. Table 6.10 summarizes the performance metrics

of the best-performing supervised and unsupervised ML models alongside the corresponding LSP models.

Table 6.10: Results of the Best Performing Models (HV and LV stand for high-voume and low-volume DoS attacks, respectively).

Model	Attack	Precision	Recall	F1	Marked.	Informed.
XGBoost	HV	**0.98**	0.98	**0.98**	**0.97**	**0.97**
XGBoost	LV	**0.98**	0.94	**0.96**	**0.97**	**0.94**
XGBoost	Both	**0.98**	**0.98**	**0.98**	**0.97**	**0.97**
VAE-S&T	HV	0.85	**0.99**	0.91	0.83	0.90
AE-CPU	LV	0.84	0.92	0.86	0.78	0.76
VAE	Both	0.76	0.93	0.82	0.72	0.79
ISOF-CPU	Both	0.75	0.92	0.81	0.72	0.78
LSP (CFG1)	HV	0.85	0.80	0.77	0.75	0.74
LSP (CFG4)	HV	0.72	0.92	0.77	0.69	0.74
LSP (t=0.85)	LV	0.87	**0.94**	0.89	0.85	0.87
LSP	Both	0.74	0.91	0.79	0.70	0.75

Considering different attack profiles allows designing solutions that deal with a broad range of attack possibilities. Results show that classification performance varies across different attack profiles and attack types. This phenomenon led to the use of different models and/or model parameters to achieve appropriate attack-detection performance.

Standard attack profiles were successfully detected. These are persistent 10-minute attacks during which the system eventually fails to deliver services, allowing the models to identify them effectively. Detection performance varied among profiles, with STD-Home and STD-PROD achieving the best results.

INT3 attack profile was the most challenging for most models, which suggests that, in general, all types of models struggle to detect short-duration attacks effectively. One possible explanation is that these attacks can momentarily disrupt the system, making the detection harder. Even after the attack ceases, the application's residual effects may persist, leading to incorrect sample classification. Additionally, the rapid recurrence of the attacks soon after the system begins to recover complicates detection efforts further. These observations emphasize the need for more effective strategies to identify short, intermittent attacks and mitigate their effects on the system.

XGBoost is a recommended algorithm for developing attack-detection models for microservice applications. XGBoost achieved F1-scores of 0.98 for high-volume DoS attacks, 0.96 for low-volume DoS attacks, and 0.98 when trained on both types of attacks. ISOF-CPU attained F1-scores of 0.83, 0.80, and 0.81 for high-volume, low-volume, and both types of attacks, respectively, which is still good but significantly worse than XGBoost. LSP models performed worse than

unsupervised ML models for high-volume and both types of attacks, but outperformed them for low-volume attacks.

Unsupervised ML algorithms can be used to build anomaly detection models for both high- and low-volume DoS attacks with satisfactory performance. When using grouped features, the ISOF algorithm with the CPU feature produced the best results. However, when using the full feature set, the VAE model outperformed all others, achieving a precision of 0.76, a recall of 0.93, and an F1-score of 0.82. More in-depth analysis showed that the CPU feature was the most effective for attack detection: low-volume attacks reduced CPU usage, while high-volume attacks increased it. In comparison, the number of sockets increased during both high-volume and low-volume attacks, making it less sensitive and consequently less effective for attack detection.

VAE and AE are promising unsupervised deep ML algorithms for detecting DoS attacks. For high-volume attacks, VAE achieved a recall of 0.99 and an F1-score of 0.91, effectively identifying most attack instances, making it suitable when minimizing false negatives is critical. Future work focused on expanding the feature space has the potential to improve precision in the case of high-volume attacks and enhance the detection of low-volume attacks.

Supervised ML outperforms unsupervised ML, but requires the generation of attack data. The supervised ML algorithms outperformed both unsupervised ML algorithms and LSP models, with XGBoost performing best, followed by RF, across both high- and low-volume DoS attacks. However, unlike unsupervised ML and LSP, supervised ML algorithms require attack data to build the models, which may not be available for all applications.

ML models can detect both high- and low-volume attacks using the same model, unlike the LSP method, which requires defining different attribute criteria, weights, operators, and thresholds for each type of attack. When attack data are unavailable, and resources are limited, we recommend using unsupervised ML algorithms (such as ISOF), which produce a single model with a small number of features for both high- and low-volume attacks.

When selecting the type of model, in addition to performance, several trade-offs need to be considered. Using ML is more straightforward than LSP, but LSP is easier to customize by changing the weights and operators. Thus, the selection of the operators can alter the model's purpose. For example, CFG1 is suitable for applications that require high precision, while CFG4 is more appropriate for critical applications that require high recall. On the positive side, LSP models are highly sensitive to changes. For instance, even a slight change in the threshold value for the global score can significantly change the outcome.

6.5 Threats to Validity

Construct Validity. We used multiple supervised and unsupervised ML algorithms, but many others could be used. These unexplored algorithms may lead

to different results. To prevent overfitting, we used cross-validation strategies, such as GroupKFold, and tuned hyperparameters using grid search. Regarding class imbalance, our dataset consists of 31% attack samples and 69% regular samples. This moderate imbalance did not significantly affect the models' performance. Each run produces approximately 77 samples. To ensure data sufficiency, we adopted ELOOCV. This approach maximizes the training data by reserving one attack run for testing and using the rest for training. ELOOCV is particularly effective when the number of samples per run is small, reduces overfitting, and improves model generalization across attack instances. To further validate the sufficiency of training data, we conducted an experiment in which the training set size was gradually increased from 10% to 100% of the available data. The observed changes in performance were negligible (under 3%).

Internal Validity. The randomness in ML models can affect performance due to variations in random initialization, data splits, and optimization processes. To mitigate this threat, we used the following practices: (1) *Random State Initialization* - we set a fixed seed value of 42 across experiments to ensure reproducibility of results and consistency in model initialization. This was used for supervised models during the shuffling and grouping of files in the ELOOCV process, ensuring each model was trained and tested on the same groupings. For ISOF, it was used to ensure consistent results across different executions; (2) *Deterministic Execution in TensorFlow* - when using TensorFlow for AE and VAE models, we enforced deterministic behavior using the *enable_op_determinism* command; and (3) *Performance Reporting* - following best practices, we reported the average performance over multiple runs, along with standard deviations to capture variability.

External Validity. We focused on a single application (TeaStore) and stabilized to a single instance per service, limiting the generalizability of our findings to other microservice applications. Further research is required to assess applicability in more dynamic environments, where microservice applications can rapidly scale up or down, increasing the number of services involved in score computation.

6.6 Summary

This chapter presented the development and evaluation of ML models for detecting DoS attacks against microservice applications. We developed and compared supervised algorithms (KNN, RF, SVM, XGB, CNN) and unsupervised algorithms (ISOF, OCSVM, LOF, AE, VAE). Supervised models were trained and validated using ELOOCV, while unsupervised models were trained exclusively on benign 'golden runs' to simulate a more realistic anomaly detection scenario.

The evaluation demonstrated that supervised models generally outperformed unsupervised models. Specifically, XGBoost achieved the most consistent performance across high-volume, low-volume, and combined attack datasets (F1-score around 0.98), followed closely by Random Forest. Among unsupervised models, deep learning methods (VAEs and AEs) achieved the best results, though their efficacy depended strongly on the feature group and attack type. The VAE-S&T combination (F1-score 0.92) was optimal for high-volume attacks, whereas AE-

CPU (F1-score 0.86) was best for low-volume attacks. The analysis confirmed that while traditional unsupervised models require careful feature engineering to be effective, deep learning-based anomaly detection models such as VAEs and AEs can leverage a larger, more complex feature space.

The chapter also discussed the implications and recommendations of using ML algorithms to build models for detecting attacks in microservice applications. This also compares the ML models with the LSP models developed in Chapter 5. This chapter and Chapter 5 illustrate and validate the Framework proposed in Chapter 3 on DoS attacks, as well as the importance of defining multiple types of attack profiles that were defined in Chapter 4.

In this and the previous chapter, we investigated model development for DoS detection in static microservice environments. However, modern systems are inherently dynamic and scalable, which raises new challenges for runtime detection. Therefore, the next chapter examines how unsupervised attack detection models can be applied to dynamic, scalable microservice environments.

Chapter 7

Detecting DoS Attacks in Scalable Microservice Applications

Microservice applications produce heterogeneous monitoring data spanning multiple sources, including resource usage, network behavior, and service interactions, all of which can exhibit legitimate fluctuations. Distinguishing between genuine workload spikes and malicious traffic is non-trivial [Caculo et al., 2020], especially given that modern deployments heavily rely on autoscaling mechanisms that dynamically adjust the number of service instances at runtime in response to unpredictable and fluctuating demand [Bremler-Barr et al., 2024].

Traditional detection methods often rely on supervised Machine Learning (ML), which requires large amounts of labeled attack data [Wang et al., 2025]. However, in production environments, such datasets are scarce, labeling is costly, and new attack types frequently emerge, restricting the practical adoption of supervised ML techniques. Unsupervised ML, by contrast, builds models of normal behavior and detects deviations without requiring prior attack labels [Zoppi et al., 2021]. This makes unsupervised methods well-suited for microservices, where behavior can evolve dynamically and deviations may arise from previously unseen conditions, including Denial of Service (DoS) attacks or legitimate usage changes [Wang et al., 2025].

In addition to the challenges that affect supervised machine learning, previous research also suggests that a single data modality may not be sufficient to capture attacks in scalable microservice applications [Bremler-Barr et al., 2024; Flora et al., 2023; Zhao et al., 2023]. A multimodal approach that fuses complementary signals across different system layers [Zhang et al., 2023] provides a richer context for more effective attack detection. We therefore argue that multimodal unsupervised anomaly detection offers a promising approach for detecting DoS attacks in elastic, dynamic microservice environments.

The works in chapters 5 and 6 demonstrated that while existing models can effectively detect DoS attacks in static environments, they were not inherently designed and evaluated to handle the core challenges of scalability or to fuse the multimodal data of scalable microservice applications. This way, we propose a novel multimodal unsupervised anomaly detection model that integrates four

data modalities (system resources, network traffic, service and instance states, and service interactions) to detect cyberattacks in microservice applications. To effectively fuse these modalities, our hierarchical approach employs specialized autoencoder models at the instance-, service-, and application-levels. This design enables the model to adapt to runtime variability caused by autoscaling and dynamic workloads, thereby learning to distinguish between benign operational fluctuations and malicious anomalies such as high- and low-volume DoS attacks.

We validated our approach through an extensive experimental campaign conducted according to our framework. Our testbed combines the TeaStore microservice application deployment with monitoring tools, workload generators, and DoS attack tools that emulate both high- and low-volume attacks. All artifacts, including generated datasets, and data generation tools, are publicly available at [Castro et al., 2026].

The main contributions of this chapter are as follows:

- A **hierarchical data organization strategy** that aligns multi-source monitoring data with microservice architectural layers; this decomposition enables consistent anomaly detection across instance-, service-, and application-levels, regardless of runtime autoscaling events.
- A **large-scale experimental campaign** with realistic workloads and different attack profiles, providing a performance assessment of the approach.
- **Empirical insights** into the behavior of multimodal DoS detection models in microservice applications, demonstrating how threshold selection, workload diversity, aggregation level, and multimodal fusion influence the detection performance.
- The **dataset and workload generation framework** for DoS detection in microservice applications, including three workload profiles and all artifacts required to reproduce and extend the experiments.

The outline of this chapter is as follows. Section 7.1 details the proposed multimodal anomaly detection model, explaining its hierarchical architecture and data fusion strategies. Section 7.2 describes the experimental testbed used to validate the approach, including the workload and attack generation. Subsequently, Section 7.3 presents the experimental results, analyzing the impact of detection parameters and multimodal fusion on performance. Section 7.4 synthesizes our main findings and discusses their implications. Section 7.5 addresses the threats to validity, and Section 7.6 summarizes the chapter.

7.1 Multimodal Anomaly Detection Model

Our proposed anomaly detection approach integrates heterogeneous monitoring data collected across three architectural levels: instance, service, and application.

Different autoencoder architectures are employed at these levels due to their suitability for unsupervised anomaly detection in complex, dynamic environments. Autoencoders effectively capture non-linear relationships and compress high-dimensional monitoring data into compact latent representations, which aids the integration of heterogeneous feature modalities [Schneider et al., 2022]. Their architectural flexibility supports specialized variants (e.g., convolutional, recurrent, variational), allowing us to apply different types based on the model.

The core contribution of this work lies in the hierarchical organization of the attack detection to address the dynamic, unknown structure of autoscaling microservice applications. The model operates under workload variability and autoscaling, using unsupervised learning to detect both *high-* and *low-volume* DoS attacks. Figure 7.1 illustrates the overall model architecture, showing how data from multiple modalities are fused hierarchically to produce the final anomaly-detection decision.

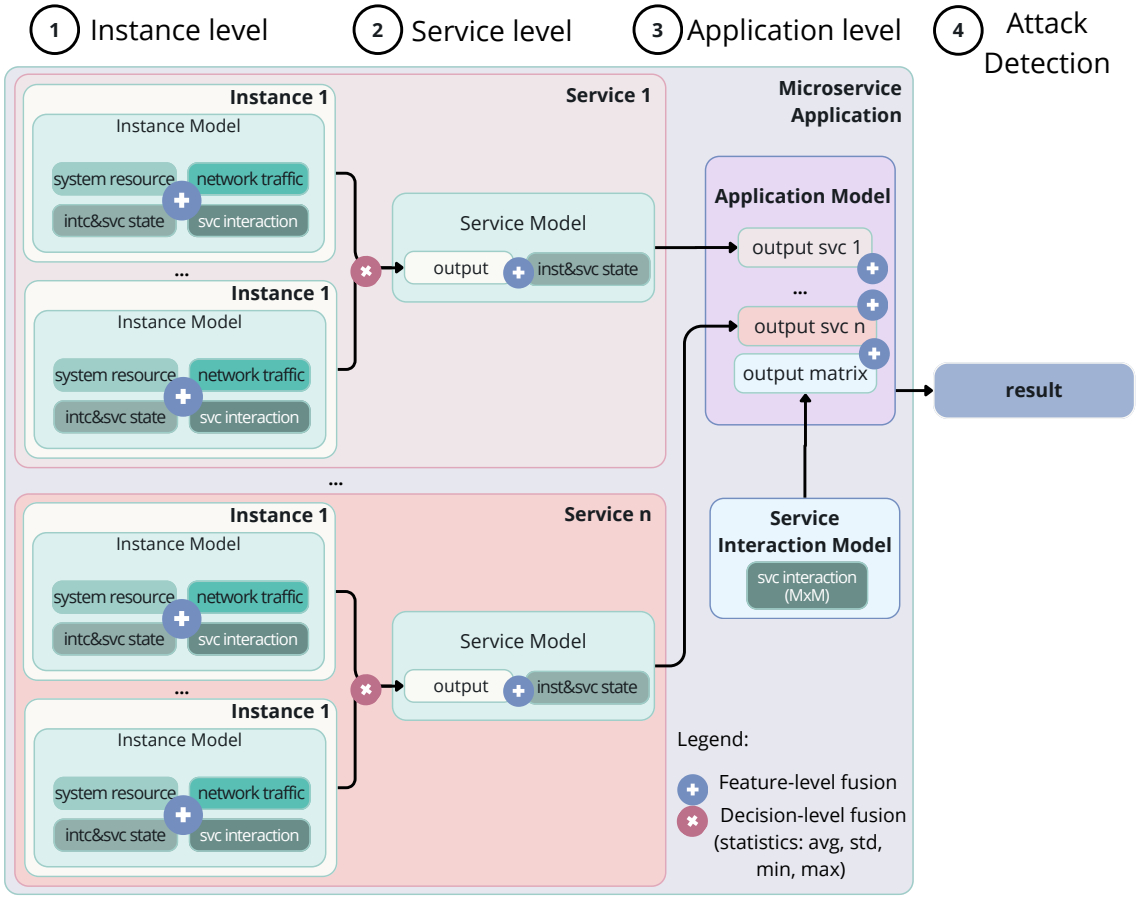


Figure 7.1: Proposed multimodal, hierarchical anomaly detection approach.

7.1.1 Hierarchical Levels

Microservice applications are composed of multiple microservices, each responsible for a specific functionality. These microservices are typically deployed with one or more running instances, also referred to as replicas or containers, to ensure

scalability. To effectively represent and analyze their behavior, our model mirrors this architectural hierarchy across three levels: instance, service, and application.

Instance-level (fine-grained view) represents the most granular level, focusing on individual microservice instances. Each instance operates independently and may exhibit unique behavioral patterns depending on its workload, resource allocation, and network interactions. Monitoring at this level enables the collection of fine-grained data, which is crucial for identifying localized anomalies that may not be apparent at higher levels of abstraction. At this level, we consider four data modalities:

- **System resource:** Consumption of system resources, such as CPU usage and number of threads, by each instance.
- **Network traffic:** Communication behavior, including received and transmitted packets and bytes.
- **Service and instance state:** Lifecycle and health indicators, such as restart counts, readiness, and liveness status.
- **Service interaction:** Patterns of communication with other components, such as response code distribution and service dependencies.

Although conceptually separated, these features may originate from different monitoring components, requiring careful design of the fusion strategy to ensure consistent interpretation.

Service-level (aggregated view) aggregates observations across all instances of a given service, providing a consolidated view of service behavior. This level addresses the challenge of dynamic instance counts caused by autoscaling, as capturing service-wide anomalies can be masked when examining individual instances in isolation. At the service-level, we enhance instance-level output with service-state features, such as the current instance count and availability status, to maintain a coherent representation of service health during scaling events.

Application-level (global view) integrates service-level outputs and models of service interactions (e.g., call graphs and request flows). This view captures coordinated anomalies across multiple services (e.g., a distributed DoS that targets multiple services or leverages service dependencies). Inputs at this level may include relationships and interactions among services (e.g., number of requests exchanged between them). This global perspective captures coordinated anomalies distributed across multiple services, enabling the detection of complex DoS behaviors that exploit service dependencies.

7.1.2 Multimodal Data Fusion

The fusion process in the proposed model is hierarchical, applying different types of fusion at each level to progressively combine information from heterogeneous

sources and multiple system components, ensuring that the resulting representations capture both local and global application behaviors.

At the instance-level, a feature-level fusion is performed during data processing. Features collected from the four modalities (system resource, network traffic, instance state, service interaction) are merged into a unified dataset per instance. This integration consolidates resource consumption, network, and deployment information into a single feature space, allowing the model to learn correlations among heterogeneous system characteristics within the same timestamp.

Subsequently, the outputs from all instances of the same service type are combined using decision-level fusion. Since microservices' elasticity dynamically changes the number of instances over time, we propose adopting a statistical aggregation approach to ensure model adaptability and comparability across runs. For each service, the reconstructed error values obtained from all instances are summarized using four statistics: minimum, maximum, average, and standard deviation. This aggregation provides a stable representation of service behavior regardless of the scaling events.

At the service-level, the aggregated decision-fusion results are further enriched through a feature-level fusion step, combining them with service-state modality features. The resulting dataset represents both the behavioral patterns inferred from individual instances and the structural state of each service (e.g., replica availability, deployment health, container status). This combined representation serves as input to the service-level autoencoder model.

At the application-level, the final model integrates the outputs from all service-level models and the output from the model that captures inter-service communication. Each service contributes a single feature derived from its reconstruction error computed by the service-level autoencoder as an anomaly score, which is concatenated with the Service Interaction model output to form the application-level autoencoder input.

7.1.3 Autoencoder Models

We define different Autoencoder (AE) architectures for the three hierarchical levels: instance, service, and application.

At the instance- and service-levels, the models have the same architecture. They follow a fully connected *dense autoencoder* design with a symmetric encoderdecoder structure. The encoder progressively compresses the input feature space through three hidden layers with decreasing dimensionalities (1/2, 1/4, and 1/8 of the input dimension), using the ReLU activation function. A bottleneck layer, defined as 1/16 of the input dimension, captures the latent representation of each observation. The decoder mirrors the encoder, expanding the representation back to the original input size through a sequence of dense layers with ReLU activations, and a linear activation in the output layer to reconstruct continuous values. The model is optimized using the Adam optimizer with a learning rate of 0.001 and mean squared error (MSE) as the loss function.

We adopt a *convolutional autoencoder* to process two-dimensional feature matrices that capture inter-service communication patterns at the application-level. Each matrix represents all requests exchanged between every pair of services in the application at a specific timestamp, forming an $M \times M$ matrix, where M is the number of services. The encoder consists of two convolutional blocks, each followed by max-pooling layers that reduce the spatial resolution while increasing the number of filters (16 and 8, respectively). A 4-filter bottleneck layer represents the latent embedding of communication features. The decoder performs symmetrical upsampling and convolution operations to reconstruct the input, with a final cropping layer ensuring dimensional alignment. The output layer uses a sigmoid activation to produce normalized values between 0 and 1.

Finally, at the application-level, we use a simplified *dense autoencoder* architecture that aggregates high-level behaviors across services. The encoder includes a single hidden layer with half of the input dimension, followed by a bottleneck layer that maintains the same dimensionality as the input to preserve global information. The decoder mirrors this configuration, reconstructing the input using a single dense hidden layer and a linear output layer. Similar to the previous models, it uses the Adam optimizer and MSE loss.

7.1.4 Training and Testing

The training and testing procedures follow the hierarchical structure of the proposed approach. Only data from golden runs are used to train the models, while attack runs are reserved exclusively for testing. This separation allows each model to learn the system's normal behavior and later identify deviations indicative of potential attacks. Before training, all input features are normalized for comparable scales across variables. Non-numeric entries are converted to numeric, and missing values are replaced with zeros. To ensure compatibility between datasets, training and test data are aligned by feature names to ensure identical dimensionality and ordering.

At the instance-level, a separate dense autoencoder is trained for each service type, using a dataset that combines features from all monitoring sources at this level. Each model receives as input the features of all instances belonging to a given service, collected during golden runs. The autoencoders are trained to minimize the mean squared reconstruction error. Once trained, the reconstruction error for each instance is computed and stored for subsequent decision-level fusion, which aggregates instance-level outputs to feed the next level's models.

At the service-level, a dense autoencoder is trained for each service type. Its input consists of the fused instance-level outputs, complemented with the service-level features derived from the monitoring data. As in the previous level, the model learns to reconstruct normal patterns and computes a reconstruction error for each service. These reconstruction errors are then used as inputs for the next hierarchical level.

At the application-level, the model integrates both service-level outputs and a matrix representation of the service interaction modality, capturing inter-service

communication patterns. The matrix data is reshaped into $M \times M$ tensors and normalized to the [0,1] range, forming the input to a convolutional autoencoder. The reconstruction error from this model is fused with the service-level outputs to form the input to the final autoencoder. This top-level model provides the overall reconstruction error representing the global application state. A detection threshold, defined as the 95th percentile of the reconstruction error distribution observed during training, is used to classify anomalous behaviors.

During testing, the trained models are applied following the same hierarchical process. The testing dataset, composed exclusively of attack runs, is evaluated run by run. At the instance-level, each instance of a specific service type is processed by its corresponding model, leading to reconstruction errors over time. These errors are then aggregated using the decision-level fusion strategy to produce service-level input values.

Next, at the service-level, the fused outputs are used to compute a new set of reconstruction errors for each service type, capturing deviations from the normal operational profile. Finally, the application-level models (both the convolutional model for service interaction and the global dense autoencoder) are applied. Their reconstruction errors are compared against the training threshold, and samples exceeding this threshold are classified as anomalies. This final stage produced the global anomaly decision for each sample in each run, indicating if the application, as a whole, exhibits behavior consistent with a DoS attack.

7.1.5 Performance Evaluation

The performance metrics used for this chapter (–)specifically Precision, Recall, and F1-score) are standard for classification tasks and were previously defined in detail in Subsection 2.3.1. This subsection, therefore, describes the specific Attack Detection Mechanism used in this chapter.

In realistic environments, attack detection is rarely done per sample. Since features are continuously collected and may fluctuate, evaluating each timestamp independently leads to unstable, noisy detection results. To address this, we adopted a **window-based detection mechanism** that aggregates consecutive predictions over time.

We characterize the **detection window** by two parameters: (i) Window size (ws): number of consecutive samples considered together for evaluation; (ii) Threshold (t): minimum proportion of samples within a window that must be predicted as attacks to trigger an detection for that window. Given these parameters, a detection is triggered when the number of samples predicted as attacks, divided by the window size, is greater than or equal to t. This approach smooths short-lived fluctuations in model predictions and enables the detection mechanism to respond to sustained anomalous behavior rather than to isolated misclassifications.

We partition each run into consecutive **detection blocks** of fixed duration. Each block is labeled as either normal or attack based on the ground truth. Detection blocks are adopted because attacks are not isolated events and typically mani-

fest as anomalous behavior over time. Evaluating detection at a block level better approximates real-world conditions and aligns with operational needs where timely detection enables mitigation actions without requiring perfect per-sample classification. In our experiments, we considered detection blocks of 60 seconds and 120 seconds. These values relate to the duration of our attack profiles, which include continuous attacks lasting 120 seconds or similar idle intervals.

Detection performance is evaluated at the **block level**, considering continuous sequences of attack or detection events. (i) *Attack block* is a continuous sequence where the ground-truth label $y_{true} = 1$. (ii) *Detection block* is a continuous sequence where the model prediction $y_{pred} = 1$.

The following definitions apply.

- **True Positive (TP)**: Attack blocks that overlap with at least one detection block.
- **False Positive (FP)**: Detection blocks that occur outside any attack block.
- **False Negative (FN)**: Attack blocks that are not detected by any detection block.
- **True Negative (TN)**: Normal (non-attack) blocks that are correctly not detected as attacks.

To better approximate operational detection scenarios, we introduce **temporal tolerance zones** around attack events. These zones define **grace periods** during which specific false positives or false negatives are not penalized.

- **For the FPs**: (i) Any detection block occurring before the first attack ($y_{true} = 1$) that is predicted as an attack is considered a FP; and (ii) Any detection block wrongly detected as an attack occurring after the last attack plus a grace period is also considered a FP.
- **Regarding the FNs**: Attack blocks that are not matched by any detection block within their duration are considered FN. If a false detection happens in a grace zone, it is not considered a false positive or negative.

The grace period allows for realistic tolerance to detection delays at the beginning and residual effects after an attack. We used a 60-second grace period, but it is configurable. This choice reflects the empirical observation that, at the onset of an attack, the system may not immediately exhibit significant degradation. After attack termination, residual effects (e.g., elevated resource usage or latency) may persist temporarily before returning to normal operation. For all performance metrics (precision, recall, and F1-score), we report 95% confidence intervals (CIs) computed using a nonparametric bootstrap with B=1000 iterations.

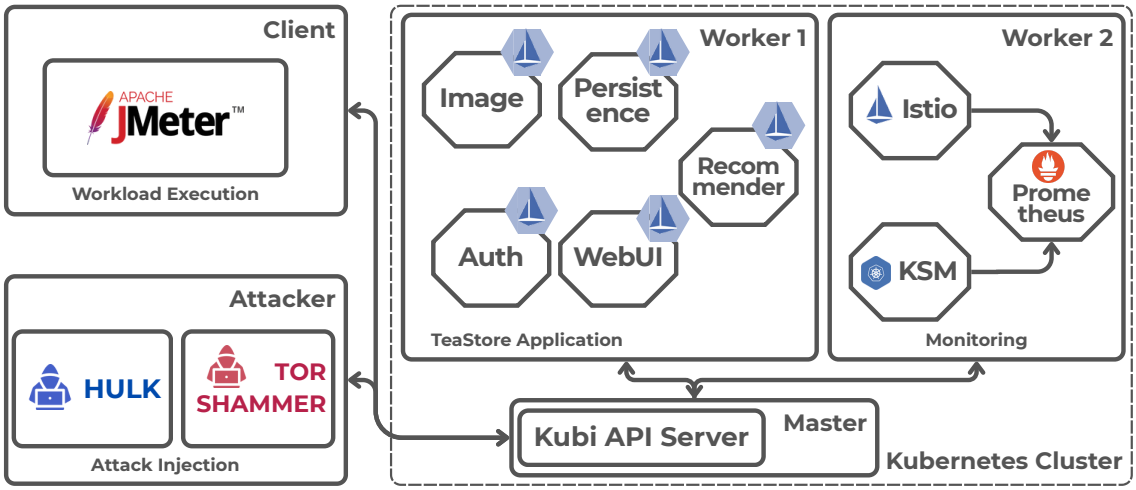


Figure 7.2: Experimental testbed for data generation.

7.2 Experimental Testbed

This study utilizes an extended version of the experimental testbed described in Chapter 4. The base configuration of the Virtual Machines (VMS), Kubernetes, and the core TeaStore application [Von Kistowski et al., 2018] remains consistent. The following extensions were made to the testbed to model a more dynamic and scalable environment.

7.2.1 Experimental Environment

While the VM layout is similar to our first testbed presented in Chapter 4.1, we introduced a dedicated Attacker VM to isolate attack traffic from the legitimate Client VM (JMeter), as illustrated in Figure 7.2. The virtual machines were configured as follows: Master (2 CPU cores, 8 GB RAM), Worker1 (6 cores, 32 GB RAM), Worker2 (2 cores, 4 GB RAM), Client (1 core, 14 GB RAM), and Attacker (1 core, 4 GB RAM). The impact of hardware configuration on the experiments is discussed in Section 7.5.

The most significant change is the activation of Autoscaling. The deployment was reconfigured to be elastic, with each TeaStore service managed by the Horizontal Pod Autoscaling (HPA). Table 7.1 summarizes these configurations. For all services, the CPU request and limit parameters define the guaranteed and maximum computing resources per pod (i.e., per service instance), respectively. Memory requests were fixed at 1 GiB, and limits were set at 3 GiB. Each service was initialized with a minimum of 1 and a maximum of 3 instances, except for the Recommender service, which was limited to 1 instance due to its lower resource requirements. These choices reflect a balance between scalability and available computational resources, ensuring stable operation under varying workloads. These configurations were defined through empirical experimentation, in which different parameter values were tested to identify settings compatible with the

available hardware capacity and capable of supporting elastic scaling without saturating resources.

Autoscaling decisions were based on CPU usage, using the default Kubernetes configuration. A service scaled up (i.e., increased the number of running instances to handle higher demand) when its average CPU usage across instances exceeded 70% of the defined CPU request, and scaled down when below this threshold. To prevent frequent oscillations under fluctuating workloads, stabilization windows of 30 seconds for scale-up and 60 seconds for scale-down were set. These parameters were empirically determined by analyzing workload behavior and considering our hardware constraints.

Table 7.1: CPU and HPA of TeaStore services in Kubernetes.

	CPU		HPA	
Services	Req.	Lim.	Min. Rep.	Max. Rep.
WebUI	900m	1080m	1 GiB	3 GiB
Auth	200m	240m	1 GiB	3 GiB
Image	200m	240m	1 GiB	3 GiB
Persistence	320m	384m	1 GiB	3 GiB
Recom.	230m	400m	1 GiB	1 GiB

7.2.2 Data Generation

To construct a dataset, multiple experimental runs were performed. A subset of these runs, referred to as golden runs, involved only workload execution and served as a baseline for normal behavior. The remaining runs combined workload execution with attack injection targeting the microservice application. Following our framework described in Chapter 3, each run was divided into the five stages shown in Figure 7.3. We adjusted the durations of these stages relative to those detailed in Chapter 4 to accommodate the system's dynamic behavior, ensuring sufficient time for autoscaling mechanisms (scale-up and scale-down events) to complete. The stage durations are as follows:

- **Initialization (Warm-up)**: 10 minutes (Data not considered for analysis).
- **Pre-attack:** 17.5 minutes (Normal operation baseline).
- **Attack window:** 20 minutes (Workload plus attack).
- **Post-attack:** 17.5 minutes (Normal operation baseline).
- **Completion (End):** 5 minutes (Data not considered for analysis).

Workload Execution

We emulated realistic user behavior with JMeter, configured with a custom workload profile comprising two components: a) *Workload Intensity Specification*, which defines the request volume and variation over time, and b) *User Behavior Model*, which specifies the types of users and their interactions with the application.

Workload Intensity Specification. To emulate realistic patterns, we derived workload profiles from real-world data. We used 13 days of public analytics from the U.S. Postal Service website (*analytics.usa.gov*), which provides the number of active users per webpage at 30-minute intervals. After analysis, we identified distinct temporal patterns corresponding to different user activity dynamics. Based on this, we designed three representative workload profiles: *Workload A (WL.A)*, *Workload B (WL.B)*, and *Workload C (WL.C)* as depicted in Figure 7.4. WL.A begins with rising user activity, then gradually declines, and ends with a light rise and a stable, gentle phase. WL.B starts with a gentle load, followed by two distinct bursts. WL.C opens with a burst, transitions into a gentle phase, and concludes with less intense burst. These workload profiles differ in their peak intensity: WL.A reaches ∼18,000 simultaneous users, WL.B ∼22,500, and WL.C ∼20,000.

To adapt these real-world patterns to the experimental context, the workloads were rescaled to match the 55-minute effective duration of each run (excluding warm-up and completion, see Figure 7.3). In practice, the original intensity data were interpolated and normalized into 55-minute-long intervals (i.e., one per minute) and linearly transformed according to:

$$y = \left(\frac{x - x_{min}}{x_{max} - y_{min}} \right) \cdot (y_{max} - y_{min}) + y_{min} \tag{7.1}$$

where x is the original workload value with bounds $x_{\text{min}} = 1,000$ and $x_{\text{max}} = 25,000$, while y_{max} defines the upper limit for JMeter parameters (e.g., thread count or throughput). This ensures proportional scaling between real-world traffic magnitudes and the simulated workload constraints.

The workload profiles were implemented in JMeter using two plugins: *jp@gc - Ultimate Thread Group* to define the number of concurrent users (threads), and *jp@gc - Throughput Shaping Timer* to regulate request throughput (requests per second). The transformation (Equation 1) effectively maps the real-world user peaks of 25,000 down to a maximum of 700 concurrent users (threads) and 500 requests per minute. This proportional scaling preserves the original data's temporal patterns while adjusting the intensity to a level suitable for controlled experimentation within our setup.

User Behavior Modeling. To emulate users' behavior, we used three user profiles (Browser, Buyer, and Order) that reflect typical e-commerce interactions and

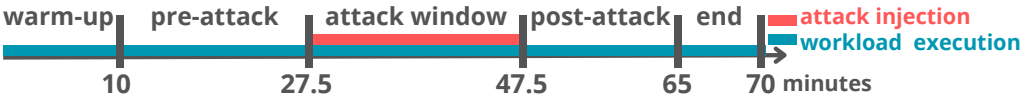


Figure 7.3: Stages of an experimental run.

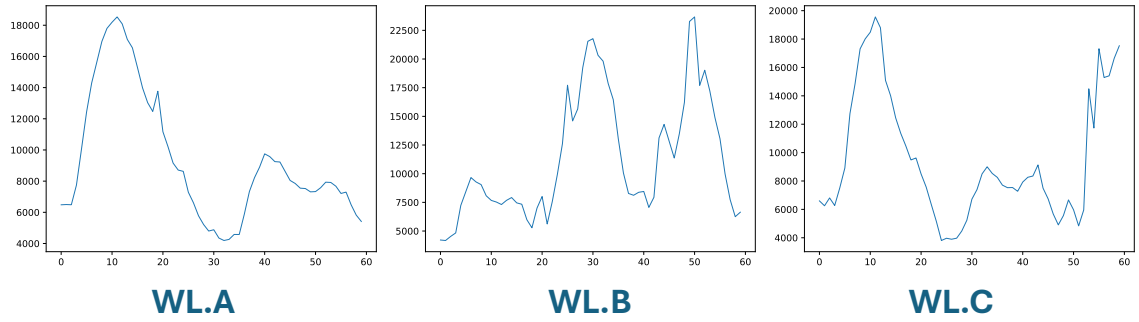


Figure 7.4: Workload Profiles (x-axis: time in min, y-axis: number of active users).

have been used in similar contexts [Avritzer et al., 2018, 2020; Camilli and Russo, 2022]. The *Browser* profile represents users who log in, navigate pages, and view product details. The *Buyer* profile extends this behavior by adding items to the cart, updating quantities, and placing orders. The *Order* profile models users who mostly log in to view past purchases. We also used two JMeter plugins. The *Weighted Switch Controller* assigned probabilities to each profile: 40% to Browser, 30% to Buyer, and 30% to Order. The *Poisson Random Timer* introduced random pauses between actions to simulate human-like delays during page browsing. The timer was configured with $\lambda = 4000$ ms and a constant delay offset of 0 ms, generating random think times typically below 4 seconds.

Attack Injection

We emulate malicious activity against the microservices, considering two classes of DoS attacks:

1. *Low-volume (slow) DoS* consumes server resources by maintaining many partially open or very slow connections, causing resource exhaustion while generating relatively few packets per second. We used the *Torshammer* tool to emulate this kind of attack.

2. *High-volume (flooding) DoS* generates large volumes of requests in a short period (flooding), rapidly exhausting computing or networking resources. For this type, we used the *Hulk* tool, which issues many HTTP requests to provoke CPU and application-level overloads.

Both attack tools operate with different targeting granularities. Torshammer (*low-volume*) targets the service IP/port; i.e., it attacks the server socket interface and therefore affects the service as a whole. In contrast, Hulk (*high-volume*) was configured to target specific HTTP endpoints (URLs), enabling attacks that focus on particular services. The two attack classes stress the system differently; *low-volume* attacks tend to exhaust connection tables and thread pools over time while producing low network throughput, whereas *high-volume* spike CPU and request-handling load, resulting in higher throughput.

Based on the types of attacks described in Chapter 3 and following the attack profiles defined in Chapter 4, we updated the duration of the attack profiles to emulate distinct malicious activity patterns (see Figure 7.5):

1. Persistent (A-STD): single sustained attack of 10 minutes.
2. Short (B-4M): short, single-burst attack lasting 4 minutes.
3. Intermittent variable duration (C-ALT3): a sequence of repeated attacks with varying durations (4, 6, and 2 minutes), each separated by 4 minutes.
4. Intermittent variable interval (D-INT3): repeated attacks with fixed duration (2 minutes) separated by variable intervals (2, 6, and 4 minutes).

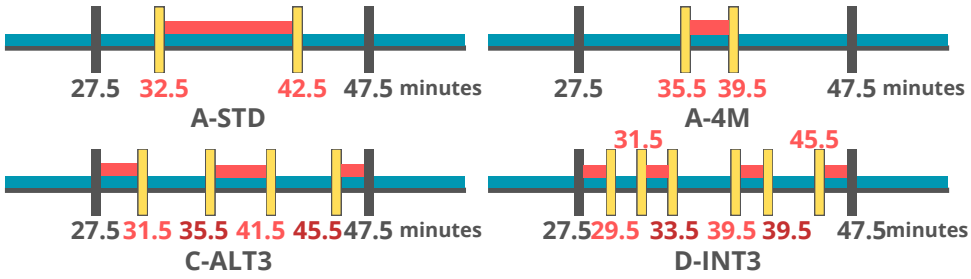


Figure 7.5: Attack profiles.

Data Generation Process

Data generation was automated using orchestration scripts that performed the following steps for each run: (i) reboot and prepare VMs, (ii) deploy TeaStore, (iii) start workload generation (JMeter) according to the chosen workload profile, (iv) execute attack scripts when applicable, and (v) collect monitoring features and logs.

For each unique combination of workload profile (WL.A, WL.B, WL.C), attack class (*low-volume/high-volume*), attack profile (A-STD, B-4M, C-ALT3, D-INT3), and target variant (home, product, and category pages), we executed ten repetitions. After filtering out runs with issues (e.g., incomplete data collection), 252 valid attack runs remained. We also generated 125 valid golden runs. All generated runs were annotated with metadata that identified the run type, workload profile, attack tool, attack profile, target, and run identifier. Furthermore, attack scripts wrote precise start and end timestamps for each attack phase, which (together with the run metadata) were stored in Comma-Separated Values (CSV) files and later used to label the datasets.

7.2.3 Data Collection, and Preprocessing

The data collection and preprocessing were significantly complex due to the multimodal data sources. During each run, the system was monitored with:

- **cAdvisor** [Google, 2022] is a native Kubernetes component responsible for collecting fine-grained container-level resource usage features, including CPU, memory, network, and file system utilization. It provides detailed visibility into the performance of individual instances.
- **Istio** [Istio, 2025], configured as a service mesh, monitors inter-service communication within the TeaStore application. By injecting a sidecar proxy into each service instance, it captures data related to request rates, latencies, response codes, and traffic volumes.
- **Kube-state-metrics (KSM)** [Kubernetes, 2025] is a service that exports features describing the state of Kubernetes objects, such as deployments and pods (instances).

For data collection and storage, we used **Prometheus** with an integrated **Istio** deployment. Prometheus scrapes features from all three monitors at 15-second intervals and stores them in a time-series database. After each run, the first and last timestamps were extracted from the JMeter output file, which contained detailed request-level information. These timestamps defined the run's temporal window and served as query boundaries for Prometheus.

Each query produced a separate data file per service, containing the combined features from **cAdvisor**, **Istio**, and **KSM**, including data from all active instances during execution. The results were stored in JavaScript Object Notation (JSON) format and annotated with metadata (service, run identifier, workload, and attack profile). This process ensured that the collected data were aligned with the timing of the workload execution, facilitating accurate labeling and analysis in subsequent stages.

The collected JSON files were transformed into structured tabular data through a dedicated preprocessing pipeline. The main script iterated over all runs and delegated parsing to specialized modules, one for each data source. The module **cAdvisor** filtered out non-informative or constant features, retaining only those with temporal variability. The resulting data includes attributes such as timestamp, pod (instance) name, and service identifier. The **Istio** module processed the features and interactions unrelated to the target workloads (i.e., services), and discarded them. Each record includes the feature name, timestamp, source and destination workloads, request protocol, response code, instance name, and reporter. The **KSM** module processed the derived features to capture higher-level system behaviorfor example, the difference between desired and available replicas or the ratio of ready replicas.

The outputs of each module were stored in CSV, with each row representing a timestamped feature value enriched with contextual metadata (e.g., service, pod, source), ensuring consistency across heterogeneous data sources and facilitating feature alignment during model training. Each collected sample (i.e., row) was labeled by run type. For golden runs, all rows were labeled as *normal*. For attack runs, a dedicated labeling script referenced a CSV file containing the precise start and end timestamps of each attack phase. Based on these timestamps, the script

annotated each sample as either *normal* or *attack*, ensuring temporal consistency in the labeling.

7.2.4 Data Processing Pipeline and Feature Extraction

The raw monitoring data underwent a structured processing pipeline to produce datasets suitable for model development, consolidating features from the three sources used during data generation: **cAdvisor**, **Istio**, and **KSM**. This stage included orchestration, temporal alignment, and feature engineering tasks. For each experimental run (either golden or attack), the data were processed per service and timestamp, deduplicated, and time-aligned to produce three synchronized datasets that ensure a consistent temporal view across all features.

Table 7.2 presents the features for each monitoring modality and hierarchical level. This serves as a reference for the processing and feature engineering steps described in this subsection. Cumulative counter features from **cAdvisor** and **Istio** were transformed into rate-based features using first-order differencing, a common time-series transformation technique [Hamilton, 2020]:

$$x_{diff}[t] = x[t] - x[t-1], \tag{7.2}$$

where t is the sampling instant. This conversion normalizes counter growth over time, yielding more stable, interpretable rate-based indicators of system activity.

KSM exported both instance- and service-level information. The data were split into two views: features containing pod identifiers were included in the instance-level dataset. At the same time, those referring to deployment objects were used in the service-level dataset. Derived indicators, such as the difference between desired and available replicas and the ratio of ready replicas, were computed to capture autoscaling dynamics and infrastructure state.

Istio provided telemetry for service-to-service communication, enabling both instance-level and application-level features. The workflow generated datasets at two granularities: the *instance-level* captures the behavior of individual microservice instances. In contrast, the *application-level* aggregates traffic across instances of the same service. At both levels, redundant or inconsistent records were removed, timestamps were aligned, and derived attributes were computed to enrich interpretability and predictive potential. Specifically, the following four families of features were derived from Istio:

1. **Per-instance request averages from paired sum/count features.** For each of {request_bytes, request_duration_milliseconds, response_bytes}, Istio exposes paired _sum and _count counters. For every $(pod\_name, reporter \in \{source, destination\})$ we compute per-interval increments Δsum and $\Delta count$ (with zero-run redistribution), and then the per-interval average:

$$avg_t = \begin{cases} \frac{\Delta sum_t}{\Delta count_t}, & \Delta count_t > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{7.3}$$

Table 7.2: Used features, grouped by level.

Feature	Category	Level
container_blkio_device_usage_total	System Resource	Instance-level
container_cpu_cfs_periods_total	System Resource	Instance-level
container_cpu_cfs_throttled_periods_total	System Resource	Instance-level
container_cpu_cfs_throttled_seconds_total	System Resource	Instance-level
container_cpu_system_seconds_total	System Resource	Instance-level
container_cpu_usage_seconds_total	System Resource	Instance-level
container_cpu_user_seconds_total	System Resource	Instance-level
container_file_descriptors	System Resource	Instance-level
container_fs_writes_total	System Resource	Instance-level
container_last_seen	System Resource	Instance-level
container_memory_cache	System Resource	Instance-level
container_memory_failures_total	System Resource	Instance-level
container_memory_mapped_file	System Resource	Instance-level
container_memory_rss	System Resource	Instance-level
container_memory_usage_bytes	System Resource	Instance-level
container_memory_working_set_bytes	System Resource	Instance-level
container_processes	System Resource	Instance-level
container_sockets	System Resource	Instance-level
container_spec_cpu_period	System Resource	Instance-level
container_spec_cpu_quota	System Resource	Instance-level
container_spec_cpu_shares	System Resource	Instance-level
container_spec_memory_limit_bytes	System Resource	Instance-level
container_start_time_seconds	System Resource	Instance-level
container_threads	Network Traffic	Instance-level
container_network_receive_packets_total	Network Traffic	Instance-level
container_network_transmit_bytes_total	Network Traffic	Instance-level
container_network_transmit_packets_total	Network Traffic	Instance-level
istio_tcp_sent_bytes_total	Network Traffic	Instance-level
istio_tcp_received_bytes_total	Network Traffic	Instance-level
istio_tcp_connections_opened_total	Network Traffic	Instance-level
istio_tcp_connections_closed_total	Network Traffic	Instance-level
kube_pod_container_state_started	Service and Instance State	Instance-level
kube_pod_container_status_ready	Service and Instance State	Instance-level
kube_pod_container_status_restarts_total	Service and Instance State	Instance-level
kube_pod_container_status_running	Service and Instance State	Instance-level
kube_pod_container_status_terminated	Service and Instance State	Instance-level
kube_pod_container_status_waiting	Service and Instance State	Instance-level
istio_request_bytes_avg_destination	Service Interaction	Instance-level
istio_request_bytes_avg_source	Service Interaction	Instance-level
istio_request_duration_milliseconds_avg_destination	Service Interaction	Instance-level
istio_request_duration_milliseconds_avg_source	Service Interaction	Instance-level
istio_response_bytes_avg_destination	Service Interaction	Instance-level
istio_response_bytes_avg_source	Service Interaction	Instance-level
istio_requests_total_destination_lt200	Service Interaction	Instance-level
istio_requests_total_destination_2xx	Service Interaction	Instance-level
istio_requests_total_destination_3xx	Service Interaction	Instance-level
istio_requests_total_destination_4xx	Service Interaction	Instance-level
istio_requests_total_destination_gt=500	Service Interaction	Instance-level
istio_requests_total_source_lt200	Service Interaction	Instance-level
istio_requests_total_source_2xx	Service Interaction	Instance-level
istio_requests_total_source_3xx	Service Interaction	Instance-level
istio_requests_total_source_4xx	Service Interaction	Instance-level
istio_requests_total_source_gt=500	Service Interaction	Instance-level
kube_deployment_metadata_observed_generation_diff	Service and Instance State	Service-level
kube_deployment_replicas_available_percent	Service and Instance State	Service-level
kube_deployment_replicas_diff	Service and Instance State	Service-level
kube_deployment_replicas_ready_percent	Service and Instance State	Service-level
kube_deployment_spec_paused	Service and Instance State	Service-level
kube_deployment_spec_replicas	Service and Instance State	Service-level
kube_deployment_status_condition_available	Service and Instance State	Service-level
kube_deployment_status_condition_progressing	Service and Instance State	Service-level
kube_deployment_status_replicas_updated	Service and Instance State	Service-level
istio_request_total	Service Interaction	Application-level

We then pivot these averages so that each feature appears twiceonce for reporter=source and once for reporter=destination.

2. **Response-code dynamics from istio_requests_total.** We transform the cumulative istio_requests_total counter into per-interval counts stratified by reporter $\in \{source, destination\}$ and HTTP response-code categories: < 200, $2xx$, $3xx$, $4xx$, and ≥ 500. After summing per timestamp and pod, we differ across each (pod, category, reporter) series, clamp negative values to 0, and pivot to produce one column per reporter_category (e.g., source_2xx, destination_4xx). This yields a compact view of success/error dynamics per instance and time.

3. **Service interaction matrices (application-level view).** To characterize application interactions, we isolate istio_requests_total with reporter=source and aggregate, for each service i, the traffic from its source workloads to destination workloads. For each timestamp, we first differ the destination-wise totals to obtain per-interval request counts and then build an $S \times S$ adjacency matrix M_t, where $M_t[i, j]$ is the number of requests emitted by service i to service j at time t (S equal to the number of services in scope). Cells with no observations were treated as zero. Additionally, Istio labels traffic directed outside the mesh or to unknown destinations as "unknown". We treat this label (i.e., all the unknowns) as a destination in the matrix and model it as a service. The exported application-level dataset is the time-ordered list $\{(t, label, M_t)\}$.

4. **TCP-related features.** For the persistence service that interfaces directly with the TeaStore database, we also extract TCP-level counters (e.g., bytes sent/received, connections opened/closed), per instance and timestamp. These features provide a transport-layer perspective.

7.3 Results and Discussion

We evaluated multiple combinations of detection window parameters: window size (ws) and threshold (t). ws specifies the number of consecutive anomaly predictions aggregated to form a detection decision. At the same time, the threshold t represents the minimum proportion of anomalous windows within a block required to raise an alert.) We experimented with ws values from 3 to 6, selected based on the minimum attack duration of 2 minutes in the ALT3 and INT3 profiles. For each ws, all admissible thresholds were evaluated. For example, when $ws = 5$, thresholds of $t = \{0.2, 0.4, 0.6, 0.8\}$ correspond to requiring 1, 2, 3, or 4 anomalous samples within the window, respectively. This evaluation resulted in 14 (ws, t) configurations. Experiments were repeated for each block duration (60s and 120s), resulting in 28 experimental scenarios. Table 7.3 presents the results of all combinations of ws, t, and block duration.

7.3.1 Overall Performance Analysis

Across all 28 scenarios, we observed a consistent trade-off between precision and recall. Higher thresholds (0.75-0.83) generally yielded better precision (up to 0.78), while lower thresholds (0.2-0.33) achieved superior recall (up to 0.98). The optimal balance was found in intermediate thresholds (0.6-0.75), which maximized F1-scores between 0.77 and 0.84. As shown in Table 7.3, the scenario with $ws = 5$ and $t = 0.8$, and $ws = 6$ and $t = 0.83$ achieved the highest F1-score (0.84) for 120s blocks. These configurations also maintained strong performance under 60s blocks (F1-score ≈ 0.78). Additionally, $ws = 4$ and $t = 0.75$ delivered consistently high scores across both block durations, indicating that moderate detection windows and intermediate thresholds yield the most stable and generalizable performance.

Table 7.3: Results across all detection window and threshold configurations for both block durations.

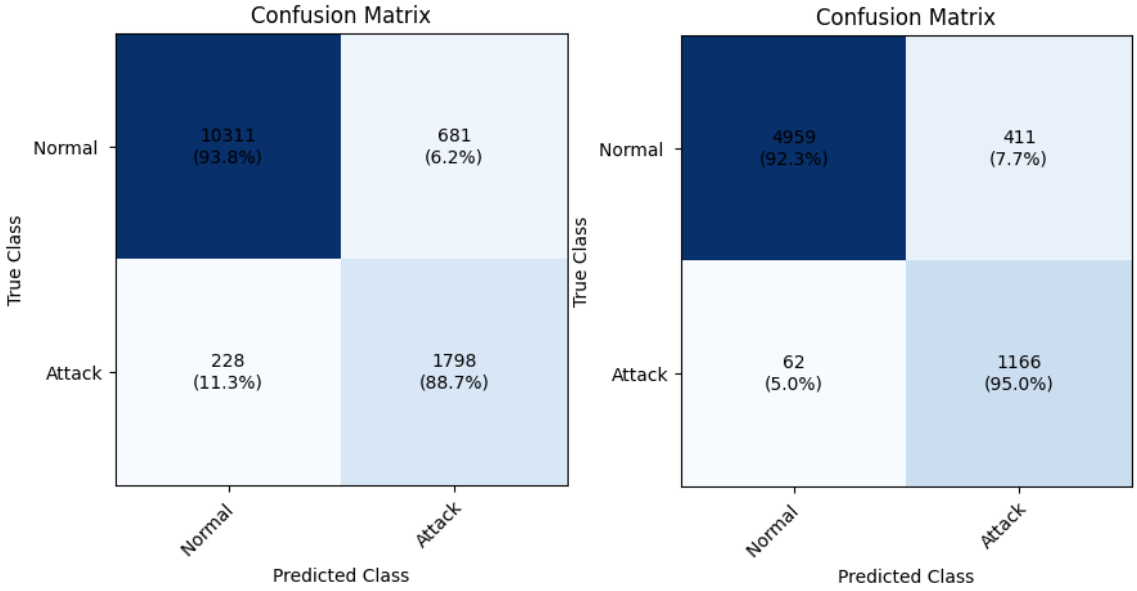
		120 block duration				**60 block duration**			
ws	*t*	Accuracy	Precision	Recall	F1-Score	Accuracy	Precision	Recall	F1-Score
3	0,66	0,91	0,69 (±0.02)	0,96 (±0.01)	0,80 (±0.02)	0,92	0,68 (±0.02)	0,90 (±0.01)	0,77 (±0.01)
3	0,33	0,83	0,54 (±0.02)	0,97 (±0.01)	0,69 (±0.02)	0,86	0,54 (±0.02)	0,92 (±0.01)	0,68 (±0.01)
4	0,75	0,93	0,74 (±0.02)	0,95 (±0.01)	0,83 (±0.02)	0,93	0,72 (±0.02)	0,89 (±0.01)	0,80 (±0.01)
4	0,50	0,90	0,66 (±0.02)	0,96 (±0.01)	0,78 (±0.02)	0,91	0,65 (±0.02)	0,91 (±0.01)	0,76 (±0.01)
4	0,25	0,82	0,52 (±0.02)	0,97 (±0.01)	0,67 (±0.02)	0,84	0,51 (±0.02)	0,93 (±0.01)	0,66 (±0.01)
5	0,80	0,94	0,76 (±0.02)	0,93 (±0.01)	0,84 (±0.02)	0,94	0,73 (±0.02)	0,85 (±0.02)	0,78 (±0.01)
5	0,60	0,92	0,72 (±0.02)	0,95 (±0.01)	0,82 (±0.02)	0,93	0,70 (±0.02)	0,90 (±0.01)	0,79 (±0.01)
5	0,40	0,88	0,62 (±0.02)	0,96 (±0.01)	0,75 (±0.02)	0,90	0,62 (±0.02)	0,92 (±0.01)	0,74 (±0.01)
5	0,20	0,80	0,50 (±0.02)	0,97 (±0.01)	0,66 (±0.02)	0,81	0,47 (±0.01)	0,94 (±0.01)	0,62 (±0.01)
6	0,83	0,95	0,78 (±0.02)	0,91 (±0.02)	0,84 (±0.02)	0,94	0,74 (±0.02)	0,82 (±0.02)	0,78 (±0.02)
6	0,66	0,94	0,75 (±0.02)	0,94 (±0.01)	0,83 (±0.01)	0,93	0,71 (±0.02)	0,86 (±0.02)	0,78 (±0.01)
6	0,50	0,92	0,70 (±0.02)	0,95 (±0.01)	0,81 (±0.02)	0,92	0,68 (±0.02)	0,90 (±0.01)	0,78 (±0.01)
6	0,33	0,86	0,59 (±0.02)	0,96 (±0.01)	0,73 (±0.02)	0,87	0,55 (±0.02)	0,93 (±0.01)	0,69 (±0.01)
6	0,16	0,79	0,49 (±0.02)	0,98 (±0.01)	0,65 (±0.02)	0,80	0,46 (±0.01)	0,95 (±0.01)	0,62 (±0.01)

7.3.2 Impact of Block Duration and Grace Period

To assess the influence of block duration, we selected the scenario $ws = 4$ and $t = 0.75$, which exhibited balanced precision and recall across both durations. Figure 7.6 presents the corresponding confusion matrices. The 120-second configuration (6598 blocks + 210 grace blocks) achieved a higher actual positive rate (0.95) than the 60-second configuration (0.89), indicating that longer observation periods enhance contextual understanding and reduce false positives. Conversely, the 60-second configuration (13021 blocks + 367 grace blocks) yielded slightly higher false-positive rates (0.08 vs. 0.06), as shorter windows are more sensitive to transient workload fluctuations misclassified as attacks.

In general, longer block durations (120s) consistently yielded better overall results, with average F1-score improvements of 0.030.05 compared to 60s configurations. In comparison, shorter block durations (60s) occasionally achieved higher precision but at the cost of a significant compromise in recall.

To assess the impact of the grace period on detection performance, Table 7.4 reports results with and without the grace period for block durations of 60 s and



(a) Block duration: 60 (b) Block duration: 120

Figure 7.6: Confusion matrices of scenario $ws = 4$ and $t = 0.75$.

120 s. Removing the grace period primarily reduced recall, leading to more false negatives due to missed detections during early attack stages. Precision remained largely unaffected, indicating that the grace period does not artificially inflate performance. Instead, the grace period improves robustness by allowing the detector to tolerate short-lived transitions at attack onset, thereby reducing premature false-negative decisions.

Table 7.4: Performance Metrics Comparison by Block Duration and Grace Period ($ws = 4$ and $t = 0.75$).

	Dur.	**Classification Metrics**			**Confusion Matrix**			
	(s)	Prec.	Rec.	F1-Sc	TP	TN	FP	FN
With Grace Period	60	0.72 (±0.02)	0.89 (±0.01)	0.80 (±0.02)	1798	10311	684	228
	120	0.74 (±0.02)	0.95 (±0.01)	0.83 (±0.01)	1166	4959	411	62
Without Grace Period	60	0.72 (±0.02)	0.75 (±0.02)	0.73 (±0.01)	1798	10311	707	595
	120	0.73 (±0.02)	0.8 (±0.02)	0.77 (±0.02)	1166	4959	430	272

Note: TN_grace for 60s and 120s block duration were 367 and 210 blocks, respectively.

7.3.3 Attack Profile Analysis

The scenario with $ws = 5$ and $t = 0.8$ was used to evaluate the detection performance across different attack profiles and attack types. As shown in Table 7.5, the INT3 profile achieved perfect recall (1.00), but 66 blocks fell within grace periods, accounting for approximately 0.05 of all blocks. It also resulted in slightly lower precision (0.79). The ALT3 profile achieved the highest F1-score (0.89), indicating consistent detection. The 4M profile showed high recall (0.96) but lower precision (0.66), which is more likely related to the workload profile, since t. Finally,

the STD profile exhibited lower recall values, likely due to autoscaling responses from microservices that mitigated part of the attack impact.

When considering the type of attack, *low-volume* attacks achieved higher recall (0.97) but lower precision (0.71) compared to *high-volume* attacks (recall: 0.91, precision: 0.81). This can be explained by the different ways these attacks affect the system. *High-volume* attacks resemble short-lived surges of legitimate user traffic, making them harder to distinguish from normal workload fluctuations. In contrast, *low-volume* attacks occupy open connections, prevent the application from handling new requests, and may further degrade the application's performance.

Table 7.5: Result per attack profile and attack type (Mean ±CI).

Attack profile	Prec.	Rec.	F1-Sc.
INT3	0.79 (±0.04)	1.00 (±0.00)	0.88 (±0.02)
ALT3	0.83 (±0.04)	0.95 (±0.04)	0.89 (±0.03)
4M	0.66 (±0.07)	0.96 (±0.03)	0.78 (±0.05)
STD	0.72 (±0.04)	0.85 (±0.04)	0.78 (±0.03)
Attack type	Prec.	Rec.	F1-Sc.
high-volume	0.81 (±0.03)	0.91 (±0.02)	0.86 (±0.02)
low-volume	0.71 (±0.04)	0.97 (±0.02)	0.82 (±0.03)

7.3.4 Workload Profile Impact

Table 7.6 evaluates the generalization capability across different workload profiles. Models trained on all workloads combined achieved the best results (F1 between 0.84 and 0.86), showing the importance of training on heterogeneous execution patterns. The WL.B profile exhibited the highest variability, suggesting it introduces greater runtime fluctuations and overlapping resource usage between normal and attack states, posing greater challenges for reliable detection.

Table 7.6: F1-score when the model was trained and tested with different workload profiles ($ws = 5$, $t = 0.8$, 120s block).

W.P. Trained/Tested	WL.A	WL.B	WL.C	All
WL.A	0.85 (±0.02)	0.87 (±0.03)	0.85 (±0.02)	0.85 (±0.01)
WL.B	0.75 (±0.03)	0.79 (±0.03)	0.80 (±0.02)	0.78 (±0.02)
WL.C	0.79 (±0.02)	0.79 (±0.03)	0.81 (±0.02)	0.80 (±0.01)
All combined	0.84 (±0.03)	0.83 (±0.03)	0.86 (±0.03)	0.84 (±0.02)

7.3.5 Instance- and Service-Level Detection

We further compared detection at the instance- and service-levels using $ws = 5$ and $t = 0.8$. Instance-level detection achieved limited performance (precision:

0.51, recall: 0.28, F1-score: 0.35), reflecting the noisy and fragmented nature of per-instance data. When aggregated at the service-level, the results improved substantially (precision: 0.61, recall: 0.63, F1-score: 0.62). The application-level also improved substantially (precision: 0.76, recall: 0.93, F1-score: 0.85). This demonstrates the advantage of hierarchical aggregation, which effectively captures cross-instance patterns and aligns more closely with application-level DoS manifestations, where attack traffic is often distributed across instances.

7.3.6 Multimodal Model vs. Single Modality Models

As part of the evaluation process, we explored which existing approaches could serve as baselines for comparison with our approach. After careful consideration, it became apparent that such comparisons were not feasible for several reasons. Some works were not adequate baselines because they were focused on detection of generic anomalies or failure identification rather than security attacks [Chen et al., 2023, 2022; Ding et al., 2024; Huang et al., 2023; Panahandeh et al., 2024; Wang et al., 2024; Zhang et al., 2023; Zhao et al., 2024]. In addition, these works utilized multimodal detection that relied on logs or traces, which is incompatible with our data and our focus on low-overhead, application-agnostic monitoring. The works that explicitly targeted DoS attacks in microservice environments [Baarzi et al., 2020; Cao et al., 2022; Castro et al., 2023a, 2025; Ficco et al., 2025] relied either on single-modality or static architectures. Still, none addressed scalability, the central focus of this work.

Even though a direct quantitative comparison is unfeasible, our results align with the current literature's baselines that used an unsupervised approach. In fact, F1-scores reported by recent multimodal microservice anomaly detectors [Chen et al., 2022; Panahandeh et al., 2024; Zhang et al., 2023] were between 0.80 and 0.85, while DoS detection for static microservices configurations [Castro et al., 2023a, 2025] yielded approximately 0.83. Our F1-score of 0.84 is consistent with state-of-the-art results, demonstrating that the DoS attack detection approach proposed in this paper successfully addresses the challenges of runtime elasticity and autoscaling dynamics, which were not addressed in these related works.

Table 7.7 compares the performance of single modalities against the multimodal approach. All models share the same hierarchical architecture as the multimodal model. Still, each single-modality model is trained and evaluated using only one of the four data types: system resources, network traffic, service and instance states, or service interactions. Among single modalities, the *network-based* achieved the highest precision (0.80) but had very low recall (0.30), whereas the *interaction* modality showed the weakest performance (F1 = 0.12). The multimodal model that integrates all four modalities outperformed all single-modality models (F1 = 0.84). This indicates that the multimodal approach captures complementary indicators of attack behavior, improving detection success.

Table 7.7: Performance of single modality models compared to four modality model ($ws = 5$, $t = 0.8$, 120s block).

Modality	Precision	Recall	F1Sc.
System resource only	0.56 ($\pm$0.05)	0.46 ($\pm$0.04)	0.51 ($\pm$0.04)
Network traffic only	0.80 ($\pm$0.06)	0.30 ($\pm$0.04)	0.44 ($\pm$0.05)
Service and instance state only	0.34 ($\pm$0.04)	0.36 ($\pm$0.04)	0.35 ($\pm$0.04)
Service interaction only	0.22 ($\pm$0.06)	0.08 ($\pm$0.03)	0.12 ($\pm$0.04)
All four modalities	0.78 ($\pm$0.02)	0.91 ($\pm$0.02)	0.84 ($\pm$0.02)

7.3.7 Comparison with Baseline Approaches

As part of the evaluation process, we explored which existing approaches could serve as baselines for comparison with our approach. After careful consideration, it became apparent that such comparisons were not feasible for several reasons. Some works were not adequate baselines because they were focused on detection of generic anomalies or failure identification rather than security attacks [Chen et al., 2023, 2022; Ding et al., 2024; Huang et al., 2023; Panahandeh et al., 2024; Wang et al., 2024; Zhang et al., 2023; Zhao et al., 2024]. In addition, these works utilized multimodal detection that relied on logs or traces, which is incompatible with our data and our focus on low-overhead, application-agnostic monitoring. The works that explicitly targeted DoS attacks in microservice environments [Baarzi et al., 2020; Cao et al., 2022; Castro et al., 2023a, 2025; Ficco et al., 2025] relied either on single-modality or static architectures. Still, none addressed scalability, the central focus of this work.

Even though a direct quantitative comparison is unfeasible, our results align with the current literature's baselines that used an unsupervised approach. In fact, F1-scores reported by recent multimodal microservice anomaly detectors [Chen et al., 2022; Panahandeh et al., 2024; Zhang et al., 2023] were between 0.80 and 0.85, while DoS detection for static microservices configurations [Castro et al., 2023a, 2025] yielded approximately 0.83. Our F1-score of 0.84 is consistent with state-of-the-art results, demonstrating that the DoS attack detection approach proposed in this paper successfully addresses the challenges of runtime elasticity and autoscaling dynamics, which were not addressed in these related works.

7.4 Findings and Implications

In this section, we highlight the main findings regarding the design and performance of our multimodal hierarchical approach for DoS attack detection and discuss their implications.

Scenarios with higher thresholds ($t \geq 0.8$) and longer block durations (120s) achieved higher precision, making them suitable for production environments where false alarms are costly. Lower thresholds ($t = 0.20.4$) favored recall, enabling faster and broader anomaly detection, though at the cost of stability. Sim-

ilarly, shorter blocks (60s) yielded faster responses and finer granularity but increased false positives, whereas longer blocks provided more stable, more accurate detection. Therefore, threshold selection should align with the operational priorities: **(i) use lower thresholds for critical systems where the cost and consequences of false negatives are high** and **(ii) high thresholds for non-critical systems that may have a high cost of false positives**.

The use of a grace period **reduced false negatives at attack onset without inflating false positives**, particularly for shorter block durations. This was particularly effective for 60s blocks, which are more susceptible to transient workload variations. In contrast, 120s blocks maintained fewer early false negatives and more stable error distributions even without a grace period.

Training on diverse workloads improves generalization. **Models trained with all workload profiles combined achieved the best performance, with F1-scores above 0.75.** This highlights the importance of exposing models to heterogeneous data to make them resilient to normal workload variability and autoscaling events typical in microservice systems.

The model showed higher sensitivity to *low-volume* attacks, with greater recall but also more false positives. Autoscaling in some attack profiles (e.g., STD and 4M) masked the attack impact, explaining the lower recall observed in such cases.

Instance-level detection suffered from noisy local behavior, whereas service-level aggregation substantially improved precision and recall. **Application-level detection was most effective at capturing distributed DoS patterns**, justifying its slightly higher computational cost with better performance.

Multimodal fusion was critical for reliable detection. Single-modality models (e.g., network or resource-only features) captured only limited aspects of the attacks. Combining the modalities increased F1-scores considerably. Since DoS attacks manifest across multiple layers of the system, the complementary information was essential for robust detection.

7.5 Threats to Validity

Construct Validity. Since no real DoS attack data on microservices is publicly available, the evaluation of the proposed approach was conducted using simulated DoS attacks, similar to prior work in this area [Castro et al., 2025; Ficco et al., 2025; Flora et al., 2023]. To emulate diverse malicious activity patterns, we generated low- and high-volume attacks that follow four different attack profiles. To enable reproducible evaluation, we used widely used, well-established tools and made the datasets publicly available. Dataset labels were assigned based on the timestamps recorded in the attack scripts to distinguish between regular and attack periods. Although these timestamps may have slight delays (under 1 second), the effect is negligible given the 15-second feature-collection interval (the average frequency at which cAdvisor provides data). Moreover, since the models

were designed for anomaly detection, cross-validation was not needed, as they were trained exclusively on golden runs and tested on separate attack runs.

Internal Validity. Slight variations in system performance or VM interference may have influenced some runs. This was mitigated by reviewing all run outputs, verifying correct client-side behavior (e.g., expected ratio of successful to unsuccessful transactions), and ensuring proper labeling of all samples (e.g., attack runs contained attack-labeled data). The available hardware constrained scalability. Specifically, limited CPU cores restricted the number of service instances that could be scaled. However, this did not affect the detection model's performance, as the selected unsupervised algorithm was compatible with the hardware's capabilities. Additionally, since randomness in model initialization and optimization may affect results, random seeds were fixed, and TensorFlow deterministic operations were enabled to foster reproducibility.

External Validity. The evaluation was conducted using the TeaStore benchmark, a widely used microservice-based application in related research [Caculo et al., 2020; Castro et al., 2025; Flora et al., 2023; Panahandeh et al., 2024; **?**]. While this limits claims about model performance at large scale, the proposed hierarchical detection architecture is designed to be independent of the number and lifetime of service instances. Therefore, scaling to larger deployments will primarily affect data volume and inference cost, without requiring changes to the detection structure. Evaluating the approach on additional, larger-scale microservice applications is an important direction for future work. This study considers two representative classes of DoS attacks (*low- and high-volume*). Although these do not cover the full spectrum of real-world attack strategies, our approach, which operates on application-level behavioral anomalies rather than attack-specific signatures, is expected to generalize to other DoS variants.

7.6 Summary

This chapter addressed the significant challenge of detecting DoS attacks in dynamic, scalable microservice applications. The inherent complexity of these systems, legitimate workload fluctuations, and runtime autoscaling render traditional detection methods ineffective.

To overcome these limitations, we proposed and validated a novel, hierarchical, multimodal, unsupervised anomaly detection model. The model is structured to mirror the microservice architecture, operating at the instance-, service-, and application-levels. It employs specialized autoencoder architectures to effectively fuse four distinct data modalities: system resources, network traffic, service and instance states, and service interactions. The unsupervised nature of the approach allows the model to be trained solely on data from golden runs, which represent normal system behavior.

The model's efficacy was evaluated through an experimental campaign. We constructed a testbed integrating the TeaStore microservice application, Kubernetes, and monitoring tools (cAdvisor, Istio, KSM). This setup was used to generate

a new, publicly available dataset, employing realistic workload profiles derived from real-world data and emulating both high-volume (Hulk) and low-volume (Torshammer) DoS attack scenarios.

The results demonstrate the advantage of the proposed hierarchical and multimodal design. The final application-level model achieved an F1-score of 0.84. We empirically confirmed that hierarchical aggregation is critical, as application-level detection (F1=0.85) substantially outperformed service-level (F1=0.62) and instance-level (F1=0.35) models. Also, the multimodal fusion proved essential, with the proposed model significantly outperforming any single-modality model. The findings also underscore the importance of training on diverse workloads.

Chapter 8

Conclusions and Future Work

The research presented in this thesis was motivated by the urgent need for attack-detection solutions that operate effectively in microservice environments. Existing security approaches are fundamentally insufficient, as they do not adequately address the distributed architecture, heterogeneity, and dynamic autoscaling capabilities that distinguish microservices from traditional monolithic systems.

These limitations highlight the need not only for new detection models but also for reproducible experimental methodologies tailored to the unique characteristics of dynamic applications. Given the complexity of this context and the research gaps spanning the entire model development pipeline, from data generation and feature analysis to model design and evaluation, the present work addresses these challenges and contributes both practical and methodological advances to the runtime security of microservice applications.

8.1 Conclusions

This thesis advances the state of the art in attack detection for microservice applications by addressing the end-to-end challenges involved in developing and evaluating runtime detection models. The work began with the introduction of a comprehensive framework designed to support the entire research pipeline, from data generation to model development and evaluation.

The framework addresses the lack of representative datasets and the absence of structured methodologies for constructing detection models tailored to the unique characteristics of microservices. It comprises two integrated modules: a Data Generation module, which provides guidelines for defining realistic workload and attack profiles and for producing labeled datasets; and a Model Development and Evaluation module, which supports the systematic construction and assessment of detection approaches capable of handling microservice dynamism, service granularity, and distributed execution.

To validate and instantiate this framework, the thesis presented a sequence of empirical studies focusing on high- and low-volume Denial of Service (DoS) at-

tacks. The first study involved an experimental campaign conducted in a static microservice environment, where the number of service instances remained constant. This effort resulted in the creation of the first publicly available labeled dataset specifically designed for DoS detection in microservices, to the best of our knowledge. This dataset served as the foundation for the next two studies, which focused on assessing different detection methods.

The first study developed and evaluated an attack-detection model based on Logic Scoring of Preference (LSP) to assess its applicability to detecting DoS attacks in microservice applications. This was the first attempt to build an attack-detection model using LSP in this context. We designed a structured LSP model capable of aggregating measurements from different services, such as the system resource usage. The resulting global score was used to determine whether the application was under attack. Experimental results showed that LSP is capable of detecting both high- and low-volume attacks with strong performance, achieving high precision, recall, and F1-scores. The study also demonstrated that the LSP method is flexible and can adapt to different weighting and operator schemes. While direct weight assignment produced the best overall performance in our scenario, the ranking method proved the easiest to apply, and the importance decomposition method offered intuitive, insightful operator configurations.

The second study focused on the development and evaluation of Machine Learning (ML) algorithms for DoS attack detection in microservice applications, examining a diverse set of supervised and unsupervised techniques. We analyzed algorithms such as Random Forest, XGBoost, SVM, CNN, Autoencoders, ISOF, LOF, and others. The results confirmed that both LSP and ML are viable solutions, with supervised ML achieving the highest accuracy, particularly the XGBoost classifier. Among the unsupervised approaches, ISOF and LOF showed strong results for both attack types when trained on CPU-related features. When comparing ML with LSP, we observed that while LSP requires no attack data during construction and remains highly adaptable, it demands more effort in parameter tuning. Conversely, ML models require attack data but are easier to optimize and compare through established performance metrics.

The final contribution extended the attack-detection problem to dynamic, scalable microservice environments, where autoscaling introduces additional complexity. To tackle this challenge, a new dataset was generated using multiple workload profiles that combine different user behaviors and intensity levels, triggering scale-up and scale-down events at different times. This dataset enabled the evaluation of detection models under realistic elastic conditions. This study introduced a multimodal, hierarchical anomaly detection model specifically designed for autoscaling microservice applications. The model integrates multiple monitoring modalities across instance, service, and application levels, enabling it to capture distributed and evolving attack patterns. Experimental results showed that detection performance is influenced by factors such as detection window duration, threshold selection, aggregation levels, and training diversity. In particular, application-level aggregation and multimodal fusion proved more effective than instance-level or single-modality approaches, while training with diverse

workload profiles improved generalization and ensured reliable detection under varying runtime conditions.

The contributions presented in this thesis demonstrate that effective runtime attack detection in microservice environments requires both methodological structure and adaptable modeling paradigms. By combining a reproducible experimental framework with complementary detection strategies (namely, multicriteria decision-making and machine learning approaches), this work provides a principled foundation for designing, evaluating, and comparing attack-detection models in distributed and elastic systems. The results show that no single paradigm universally dominates; rather, the choice between approaches depends on data availability, operational constraints, and security priorities. These findings establish a systematic basis for advancing runtime security in modern microservice architectures.

8.2 Future Work

Building on the findings and limitations identified throughout this thesis, several avenues for future research emerge. These include expanding datasets and experimental scenarios to cover broader architectural and operational conditions; further refining LSP-based detection models to enhance transferability and automation; extending the proposed multimodal hierarchical anomaly detection model to larger and more heterogeneous deployments; integrating detection mechanisms with self-adaptive control loops to enable autonomous mitigation; and investigating service-level detection and root-cause analysis to improve diagnostic precision. The following directions aim to broaden the applicability of the proposed framework, enhance detection robustness under dynamic conditions, and strengthen runtime security mechanisms in microservice applications:

- **Expanding datasets and experimental scenarios.** To improve model robustness and generalizability, future work should broaden the experimental scope in terms of system scale, attack diversity, and data generation strategies, focusing on the following topics:
  - **Experimenting with additional microservice applications:** Generate datasets using larger and more complex applications with a greater number of services and richer inter-service communication patterns. While TeaStore enabled controlled experimentation, it represents a relatively small-scale system.
  - **Including additional attack types:** Extend the datasets beyond DoS to cover other threats, such as API abuse, adversary-in-the-middle attacks, and scenarios involving injected vulnerabilities at the service or instance level.Broadening attack coverage will increase dataset diversity and improve model generalization.
  - **Generative AI-driven data generation:** Investigate the use of **Generative AI (GAI)** techniques to mitigate dataset scarcity by synthesiz-

ing realistic and environment-agnostic workload and attack scenarios. Fine-tuned large language models or generative models trained on monitoring data (e.g., logs, traces, and system metrics) could support data augmentation, domain adaptation, and the generation of labeled data for new deployments.

- **Advancing LSP-based detection models.** Although LSP demonstrated good performance, further research is needed to enhance its scalability, adaptability, and automation in complex microservice environments:
  - **Evaluating LSP in dynamic environments:** Validate LSP performance using datasets generated from scalable microservice applications with autoscaling. This would assess the robustness of the attribute tree under fluctuating replica counts and distributed resource usage.
  - **Automated and dynamic LSP parameter tuning:** Reduce the manual and expert-driven nature of LSP configuration by exploring automated optimization or ML-assisted calibration techniques. Future work may also consider runtime adaptation of weights, operators, and criteria functions, allowing LSP models to evolve with changing workloads, system conditions, or attack patterns.
  - **Native model transferability:** Develop unified LSP configurations that incorporate transferability from the outset, enabling reusable models capable of detecting multiple attack types with minimal recalibration.
- **Extending the multimodal hierarchical anomaly detection model.** Explore alternative algorithms, such as LSTMs or transformers, in place of Autoencoders, and investigate continual learning techniques to support real-time adaptation. Additional modalities, including logs and distributed traces, may be integrated, potentially leveraging generative or language-model-based approaches for log interpretation. Further research on cross-level and cross-modality fusion strategies could enhance detection performance.
- **Integration with self-adaptive control loops.** Extend the monitoring and analysis components into a complete MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) control loop. This includes design, planning, and execution mechanisms capable of triggering mitigation strategies such as rate limiting, proactive scaling adjustments, or moving-target defense techniques. GAI may assist in decision-making, policy generation, and orchestration within platforms such as Kubernetes.
- **Service-level attack detection and root-cause analysis.** Improving diagnostic precision and operational response requires moving beyond application-level alerts toward fine-grained localization and explanation mechanisms:
  - **Instance- and service-level evaluation:** Conduct a systematic assessment of detection performance at the service and instance levels to complement application-level analysis. Although hierarchical mechanisms were introduced and instance-level scores were computed, service-specific prediction accuracy remains underexplored.

- **Root-cause analysis:** Develop techniques to accurately identify which service, instance, or inter-service interaction is responsible for anomalous behavior. GAI models may support causal reasoning, explanation generation, and summarization of complex failure scenarios, enabling actionable diagnostics and targeted mitigation while reducing the scope of incident response.

These research directions aim to evolve runtime attack detection in microservice environments from isolated detection models toward integrated, adaptive, and autonomous security ecosystems. By combining more representative datasets, transferable modeling paradigms, multimodal and hierarchical analysis, and self-adaptive mitigation mechanisms, future work can move closer to continuous, context-aware protection of distributed applications. Advancing along these lines will help transform runtime security in microservices from reactive anomaly identification to proactive, resilient, and self-improving defense capabilities.

References

Anes Abdennebi, Kara Nadjia, Laaziz Lahlou, Mohamed Younis, and Hakima Ould-Slimane. Sec-llama: A compact fine-tuned llm for network intrusion detection in kubernetes clusters. In *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pages 1–7. IEEE, 2025.

Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, 15(3):1448–1460, 2020.

Zeeshan Ahmad, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, and Farhan Ahmad. Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, 32(1):e4150, 2021.

Mostofa Ahsan, Kendall E Nygard, Rahul Gomes, Md Minhaz Chowdhury, Nafiz Rifat, and Jayden F Connolly. Cybersecurity threats and their mitigation approaches using machine learninga review. *Journal of Cybersecurity and Privacy*, 2(3):527–555, 2022.

Bilal Alhayani, Sara Taher Abbas, Dawood Zahi Khutar, and Husam Jasim Mohammed. Best ways computation intelligent of face cyber attacks. *Materials Today: Proceedings*, 2021.

Hussain Alibrahim and Simone A Ludwig. Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization. In *IEEE Congr. on Evolutionary Computation*, pages 1551–1559, 2021.

Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

Ahmed Alshaibi, Mustafa Al-Ani, Abeer Al-Azzawi, Anton Konev, and Alexander Shelupanov. The comparison of cybersecurity datasets. *Data*, 7(2):22, 2022.

Fatima Rashed Alzaabi and Abid Mehmood. A review of recent advances, challenges, and opportunities in malicious insider threat detection using machine learning methods. *IEEE Access*, 12:30907–30927, 2024.

Edward Amoroso. *Cyber Security*. Silicon Press, 2006.

Apache Software Foundation. Apache jmeter. `https://jmeter.apache.org/`, 2022.

Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137:106600, 2021.

Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and André van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In *European Conference on Software Architecture*, pages 159–174. Springer, 2018.

Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, André van Hoorn, Henning Schulz, Daniel Menasché, and Vilc Rufino. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *J. Syst. Softw.*, 165: 110564, 2020.

Ataollah Fatahi Baarzi, George Kesidis, Dan Fleck, and Angelos Stavrou. Microservices made attack-resilient using unsupervised service fissioning. In *Proceedings of the 13th European workshop on Systems Security*, pages 31–36, 2020.

Ahmed Bali, Yassine El Houm, Abdelouahed Gherbi, and Mohamed Cheriet. Automatic data featurization for enhanced proactive service auto-scaling: Boosting forecasting accuracy and mitigating oscillation. *Journal of King Saud University-Computer and Information Sciences*, 36(2):101924, 2024.

Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

Arnab Barua, Mobyen Uddin Ahmed, and Shahina Begum. A systematic literature review on multimodal machine learning: Applications, challenges, gaps and future directions. *Ieee access*, 11:14804–14831, 2023.

Tania Basso, Hebert Silva, and Regina Moraes. On the use of quality models to characterize trustworthiness properties. In *Software Engineering for Resilient Systems: 11th International Workshop, SERENE 2019, Naples, Italy, September 17, 2019, Proceedings 11*, pages 147–155. Springer, 2019.

Giacomo Benedetti, Luca Caviglione, Alberto Falcone, Massimo Ficco, Massimo Guarascio, and Antonio Guerriero. Detecting ddos attacks in microservice architectures via ai-based agents. In *International Conference on Computational Science and Its Applications*, pages 3–15. Springer, 2025.

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE access*, 10:20357–20374, 2022.

Anat Bremler-Barr, Michael Czeizler, Hanoch Levy, and Jhonatan Tavori. Exploiting miscoordination of microservices in tandem for effective DDoS attacks. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 231–240. IEEE, 2024.

Sriyash Caculo, Kanishka Lahiri, and Subramaniam Kalambur. Characterizing the scale-up performance of microservices using teastore. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 48–59. IEEE, 2020.

Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. Workload characterization: A survey revisited. *ACM Comput. Surv.*, 48(3):1–43, 2016.

Matteo Camilli and Barbara Russo. Modeling performance of microservices systems with growth theory. *Empirical Software Engineering*, 27(2):39, 2022.

João R Campos, Ernesto Costa, and Marco Vieira. Online failure prediction through fault injection and machine learning: Methodology and case study. In *IEEE 34th Int. Symp. on Software Reliability Engineering*, pages 451–461, 2023.

Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–9, 2022.

Andréia Casare, Tania Basso, and Regina Moraes. Trust metrics to measure website user experience. In *The 13th Int. Conf. on Advances in Computer-Human Interactions*, pages 1–8, 2020.

Jessica Castro, Nuno Laranjeiro, and Marco Vieira. Detecting dos attacks in microservice applications: Approach and case study. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, pages 73–78, 2022.

Jessica Castro, Nuno Laranjeiro, and Marco Vieira. Exploring logic scoring of preference for DoS attack detection in microservice applications. In *2023 IEEE International Conference on Web Services (ICWS)*, pages 573–584. IEEE, 2023a.

Jessica Castro, Nuno Laranjeiro, and Marco Vieira. Generating realistic attack data for microservices: Framework and case study. In *Proc. 12th Latin-American Symp. on Dependable and Secure Comput.*, pages 60–69, 2023b.

Jessica Castro, Nuno Laranjeiro, and Marco Vieira. Exploring Logic Scoring of Preference for DoS Attack Detection in Microservice Applications Supplemental Material, March 2023c. URL `https://doi.org/10.5281/zenodo.8007658`.

Jessica Castro, Nuno Laranjeiro, Katerina Goseva-Popstojanova, and Marco Vieira. Developing attack detection models for microservice applications - supplemental material, 2024. URL `https://doi.org/10.5281/zenodo.11204757`.

Jessica Castro, Nuno Laranjeiro, Katerina Goseva-Popstojanova, and Marco Vieira. Developing attack detection models for microservice applications: A comprehensive framework and its illustration and validation on DoS attacks. *IEEE Transactions on Dependable and Secure Computing*, 2025.

Jessica Castro, Katerina Goseva-Popstojanova, Nuno Laranjeiro, and Marco Vieira. Detecting DoS Attacks in Scalable Microservices Using Multimodal Anomaly Detection - Supplemental Material, 01 2026. URL `https://doi.org/10.5281/zenodo.18364296`.

Jonathan Chamberlain, Jilin Zheng, Zeying Zhu, Zaoxing Liu, and David Starobinski. Exploiting kubernetes autoscaling for economic denial of sustainability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(2):1–29, 2025.

Deli Chen, Chunyang Ye, Hui Zhou, Shanyan Lai, and Bo Li. Enhancing microservice migration transformation from monoliths with graph neural networks. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 136–146. IEEE, 2025.

Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. Tracegra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications*, 204:109–117, 2023.

Yufu Chen, Meng Yan, Dan Yang, Xiaohong Zhang, and Ziliang Wang. Deep attentive anomaly detection for microservice systems with multimodal time-series data. In *2022 IEEE International Conference on Web Services (ICWS)*, pages 373–378. IEEE, 2022.

Nwodo Benita Chikodili, Mohammed D Abdulmalik, Opeyemi A Abisoye, and Sulaimon A Bashir. Outlier detection in multivariate time series data using a fusion of k-medoid, standardized euclidean distance and z-score. In *International Conference on Information and Communication Technology and Applications*, pages 259–271. Springer, 2020.

Berkcan Çiftçi and Birol Çiloğlugil. A review of comparative studies on performance evaluation of communication mechanisms for microservices. In *International Conference on Computational Science and Its Applications*, pages 98–113. Springer, 2025.

Cloud Native Computing Foundation. Kubernetes. `https://kubernetes.io/`, 2022.

Aristides Dasso and Ana Funes. Web applications security testing evaluation. In *III Simposio Argentino de Informática Industrial e Investigación Operativa (SIIIO 2020)-JAIIO 49 (Modalidad virtual)*, 2020.

Andrea Detti, Ludovico Funari, and Luca Petrucci. μbench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

Wajjakkara Kankanamge Anthony Nuwan Dias and Prabath Siriwardena. *Microservices security in action*. Simon and Schuster, 2020.

Shuai Ding, E Yuepeng, Jiye Zhang, Liangxiong Li, Lei Zhang, and Jingguo Ge. Trace anomaly detection for microservice systems via graph-based semi-supervised learning. In *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 2375–2380. IEEE, 2024.

Zhijun Ding and Qichen Huang. Copa: A combined autoscaling method for kubernetes. In *IEEE Int. Conf. on Web Services*, pages 416–425, 2021.

Docker, Inc. Docker official website, 2025. URL `https://www.docker.com/`. Acesso em 11 Dezembro 2025.

Qingfeng Du, Tiandi Xie, and Yu He. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In *Algorithms and Architectures for Parallel Processing: 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part IV 18*, pages 560–572. Springer, 2018.

Jozo Dujmovic. *Soft computing evaluation logic: The LSP decision method and its applications*. John Wiley & Sons, 2018.

Jozo Dujmović and William L Allen III. Soft computing logic decision making in strategic conservation planning for water quality protection. *Ecological Informatics*, 61:101167, 2021.

Jozo J Dujmovic. Continuous preference logic for system evaluation. *IEEE Transactions on Fuzzy Systems*, 15(6):1082–1099, 2007.

Jozo J Dujmović and Wen Yuan Fang. An empirical analysis of assessment errors for weights and andness in lsp criteria. In *Modeling Decisions for Artificial Intelligence: First International Conference, MDAI 2004, Barcelona, Spain, August 2-4, 2004. Proceedings 1*, pages 139–150. Springer, 2004.

Ummay Faseeha, Hassan J Syed, Fahad Samad, Sehar Zehra, and Hamza Ahmed. Observability in microservices: An in-depth exploration of frameworks, challenges, and deployment paradigms. *IEEE Access*, 2025.

M Ficco, P Fusco, A Guerriero, R Pietrantuono, M Russo, F Palmieri, and S Russo. A benchmark for DDoS attacks detection in microservice architectures. In *2025 International Conference on Computing, Networking and Communications (ICNC)*, pages 345–349. IEEE, 2025.

Stefan Fischer, Pirmin Urbanke, Rudolf Ramler, Monika Steidl, and Michael Felderer. An overview of microservice-based systems used for evaluation in testing and monitoring: a systematic mapping study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, pages 182–192, 2024.

José Flora and Nuno Antunes. Evaluating intrusion detection for microservice applications: Benchmark, dataset, and case studies. *J. Syst. Softw.*, 216:112142, 2024.

José Flora, Paulo Gonçalves, and Nuno Antunes. Intrusion detection for scalable and elastic microservice applications. In *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 39–45. IEEE, 2023.

Joint Task Force. Security and privacy controls for information systems and organizations. Technical report, National Institute of Standards and Technology, 2020.

Martin Fowler and James Lewis. Microservices: A definition of this new architectural term, 2014. URL `https://martinfowler.com/articles/microservices.html`. Acesso em 11 Dezembro 2025.

Jesús Friginal, David De Andrés, Juan-Carlos Ruiz, and Pedro Gil. Coarse-grained resilience benchmarking using logic score of preferences: ad hoc networks as a case study. In *Proceedings of the 13th European Workshop on Dependable Computing*, pages 23–28, 2011.

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 3–18, 2019.

Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Ivano Malavolta, Tanjina Islam, Madalina Dinga, Anne Koziolek, Snigdha Singh, Martin Armbruster, et al. Monitoring tools for devops and microservices: A systematic grey literature review. *J. Syst. Softw.*, page 111906, 2023.

Google. cAdvisor. `https://github.com/google/cadvisor`, 2022.

Alexey Grafov. Hulk. `https://github.com/grafov/hulk`, 2016. Accessed on March 17, 2023.

Johannes Grohmann, Patrick K Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. Monitorless: Predicting performance degradation in cloud applications with machine learning. In *Proceedings of the 20th international middleware conference*, pages 149–162, 2019.

Philipp Haindl, Patrick Kochberger, and Markus Sveggen. A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures. *IEEE Access*, 12:90252–90286, 2024.

James D Hamilton. *Time series analysis*. Princeton university press, 2020.

Jun Huang, Yang Yang, Hang Yu, Jianguo Li, and Xiao Zheng. Twin graph-based anomaly detection via attentive multi-modal learning for microservice system. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 66–78. IEEE, 2023.

InfluxData Inc. Influxdb. `https://github.com/influxdata/influxdb`, 2023a.

InfluxData Inc. Telegraf. `https://github.com/influxdata/telegraf`, 2023b.

Istio. Istio: Open source service mesh, 2025. URL `https://istio.io`. Accessed: 2025-10-15.

Ghafar A Jaafar, Shahidan M Abdullah, Saifuladli Ismail, et al. Review of recent detection methods for http ddos attack. *Journal of Computer Networks and Communications*, 2019, 2019.

Stephen Jacob, Yuansong Qiao, Yuhang Ye, and Brian Lee. Anomalous distributed traffic: Detecting cyber security attacks amongst microservices using graph convolutional networks. *Comput. Secur.*, 118:102728, 2022.

Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

Nikola Janevski and Katerina Goseva-Popstojanova. Session reliability of web systems under heavy-tailed workloads: an approach based on design and analysis of experiments. *IEEE Transactions on Software Engineering*, 39(8):1157–1178, 2013.

Raveen Kanishka Jayalath, Hussain Ahmad, Diksha Goel, Muhammad Shuja Syed, and Faheem Ullah. Microservice vulnerability analysis: A literature review with empirical insights. *IEEE Access*, 2024.

Ning Jing, Han Li, and Zhuofeng Zhao. A microservice fault identification method based on lightgbm. In *2022 IEEE 8th International Conference on Cloud Computing and Intelligent Systems (CCIS)*, pages 709–713. IEEE, 2022.

Kubernetes. kube-state-metrics, 2025. URL `https://github.com/kubernetes/kube-state-metrics`. Accessed: 2025-10-15.

Ravil Kudermetov, Olga Polska, and Natalia Shcherbak. Confirming the consistency of qos-based web services ranking by logic scoring of preference method. In *2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET)*, pages 478–481. IEEE, 2022.

Wonjun Lee, Yung Ryn Choe, and Raiat Subhra Ghosh. Recurrent neural network and convolutional neural network for detection of denial of service attack in microservices. In *2023 International Conference on Machine Learning and Applications (ICMLA)*, pages 1451–1456. IEEE, 2023.

Cristiano Inácio Lemes, Vincent Naessens, and Marco Vieira. Trustworthiness assessment of web applications: Approach and experimental study using input validation coding practices. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 435–445. IEEE, 2019.

Maikel Yelandi Leyva Vazquez, Miguel Angel Quiroz Martinez, Josseline Haylis Diaz Sanchez, and Jorge Luis Aguilera Balseca. Decision model for qos in networking based on hierarchical aggregation of information. In *Advances in Artificial Intelligence, Software and Systems Engineering: Proceedings of the AHFE 2020 Virtual Conferences on Software and Systems Engineering, and Artificial Intelligence and Social Computing, July 16-20, 2020, USA*, pages 361–368. Springer, 2021.

Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. Microservices: architecture, container, and challenges. In *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)*, pages 629–635. IEEE, 2020.

Nuno Mateus-Coelho, Manuela Cruz-Cunha, and Luis Gonzaga Ferreira. Security in microservices architectures. *Procedia Computer Science*, 181:1225–1236, 2021.

Microservices-demo. Sock Shop: A microservice demo application. `https://github.com/microservices-demo/microservices-demo`, 2018. Accessed: 2025-12-19.

Enrique A Miranda, Mario Berón, Germán Montejano, and Daniel Riesco. Using reverse engineering techniques to infer a system use case model. *Journal of Software: Evolution and Process*, 31(2):e2121, 2019.

Reda Morsli, Nadjia Kara, Hakima Ould-Slimane, and Laaziz Lahlou. Multidimensional intrusion detection system for containerized environments. In *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*, pages 546–554. IEEE, 2025.

Paul K Mvula, Paula Branco, Guy-Vincent Jourdan, and Herna L Viktor. A systematic literature review of cyber-security data repositories and performance assessment metrics for semi-supervised learning. *Discover Data*, 1(1):4, 2023.

Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public rest web service apis. *IEEE Trans. Serv. Comput.*, 14(4):957–970, 2018.

Modupe Odusami, Sanjay Misra, Olusola Abayomi-Alli, Adebayo Abayomi-Alli, and Luis Fernandez-Sanz. A survey and meta-analysis of application-layer distributed denial-of-service attack. *International Journal of Communication Systems*, 33(18):e4603, 2020.

OECD. Digital security and resilience in critical infrastructure and essential services. Technical report, OECD, 2019. URL `https://www.oecd.org/en/publications/digital-security-and-resilience-in-critical-infrastructure-and-essential-services_a7097901-en.html`. Accessed November 10, 2025.

OECD. OECD framework for the classification of ai eur. *OECD Digital Economy Papers*, 323, 2022. doi: https://doi.org/https://doi.org/10.1787/cb6d9eca-en. URL `https://www.oecd-ilibrary.org/content/paper/cb6d9eca-en`.

Rui André Oliveira, Miquel Martínez Raga, Nuno Laranjeiro, and Marco Vieira. An approach for benchmarking the security of web service frameworks. *Future Generation Computer Systems*, 110:833–848, 2020.

Omer YoachimikJorge Pacheco, Omer Yoachimik, Jorge Pacheco, and João Tomé. DDoS threat report for 2023 q4, Jan.s 2024. URL `https://blog.cloudflare.com/ddos-threat-report-2023-q4`.

Mahsa Panahandeh, Abdelwahab Hamou-Lhadj, Mohammad Hamdaqa, and James Miller. Serviceanomaly: An anomaly detection approach in microservices using distributed traces and profiling metrics. *J. Syst. Softw.*, 209, 2024.

OV Polska, RK Kudermetov, and VV Shkarupylo. An approach web service selection by quality criteria based on sensitivity analysis of mcdm methods. *Radio Electronics, Computer Science, Control*, 2:133–143, 2021.

Mahsa Raeiszadeh, Amin Ebrahimzadeh, Roch H Glitho, Johan Eker, and Raquel AF Mini. Asynchronous real-time federated learning for anomaly detection in microservice cloud applications. *IEEE Transactions on Machine Learning in Communications and Networking*, 2025.

Areeg Samir and Claus Pahl. Detecting and localizing anomalies in container clusters using markov models. *Electronics*, 9(1):64, 2020.

Sarah Schneider, Doris Antensteiner, Daniel Soukup, and Matthias Scheutz. Autoencoders-a comparative analysis in the realm of anomaly detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1986–1992, 2022.

IBM Security and Ponemon Institute. Cost of a data breach report 2024. Technical report, IBM, 2024. URL `https://cdn.table.media/assets/wp-content/uploads/2024/07/30132828/Cost-of-a-Data-Breach-Report-2024.pdf`. Accessed November 10, 2025.

Numan Shafi, Muhammad Abdullah, Waheed Iqbal, Abdelkarim Erradi, and Faisal Bukhari. Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices. *Cluster Computing*, 27(4):5195–5215, 2024.

Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A. Ghorbani. Developing realistic distributed denial of service (ddos) attack dataset and taxonomy. In *IEEE 53rd International Carnahan Conference on Security Technology (ICCST)*, Chennai, India, 2019. IEEE. URL `https://www.unb.ca/cic/datasets/ddos-2019.html`.

Shuoge Shen, Suzana Dragićević, and Jozo Dujmović. Gis-based logic scoring of preference method for urban densification suitability analysis. *Computers, Environment and Urban Systems*, 89:101654, 2021.

Solarstone. Torshammer. `https://sourceforge.net/projects/torshammer/`, 2014. Accessed on March 17, 2023.

Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

Gaurav Somani, Manoj Singh Gaur, Dheeraj Sanghi, Mauro Conti, Muttukrishnan Rajarajan, and Rajkumar Buyya. Combating ddos attacks in the cloud: requirements, trends, and future directions. *IEEE Cloud Computing*, 4(1):22–32, 2017.

Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: a systematic mapping study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress, 2018.

Pandas Development Team. Pandas 2.2.2 documentation pandas.dataframe.bfill. `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.bfill.html`, 2024. Accessed: May 06, 2024.

Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016.

Victor Velepucha and Pamela Flores. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*, 11:88339, 2023. doi: 10.1109/ACCESS.2023.3305687.

B Venkatesh and J Anuradha. A review of feature selection and its methods. *Cybern. Inf. Technol.*, 19(1):3–26, 2019.

William Viktorsson, Cristian Klein, and Johan Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *2020 28th International symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*, pages 1–4. IEEE, 2020.

Joakim Von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

Jóakim von Kistowski, Simon Eismann, Yannik Lubas, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Microservice benchmark applications. In *Systems Benchmarking: For Scientists and Engineers*, pages 305–321. Springer, 2025.

Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. Root-Cause Metric Location for Microservice Systems via Log Anomaly Detection. *Proceedings - 2020 IEEE 13th International Conference on Web Services, ICWS 2020*, pages 142–150, 2020. doi: 10.1109/ICWS49710.2020.00026.

Peipeng Wang, Xiuguo Zhang, Zhiying Cao, and Zihan Chen. Madmm: microservice system anomaly detection via multi-modal data and multi-feature extraction. *Neural Comput. and Appl.*, 36(25):15739–15757, 2024.

Peipeng Wang, Xiuguo Zhang, Yutian Chen, and Zhiying Cao. Unsupervised microservice system anomaly detection via contrastive multi-modal representation clustering. *Information Processing & Management*, 62(3):104013, 2025.

Shuo Wang, Zhijun Ding, Ru Yang, and Changjun Jiang. A complex behavioral interaction analysis method for microservice systems with bounded buffers. *IEEE Trans. Cloud Comput.*, 2023.

Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empir. Softw. Eng.*, 26(4):63, 2021.

Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners perspective. *J. Syst. Softw.*, 182:111061, 2021.

Gregory B White, Eric A Fisch, and Udo W Pooch. *Computer system and network security*. CRC press, 2017.

Zhijiao Xiao and Song Hu. Dscaler: A horizontal autoscaler of microservice based on deep reinforcement learning. In *23rd Asia-Pacific Netw. Oper. Manag. Symp.*, 2022.

Shuaiyu Xie, Jian Wang, Bing Li, Zekun Zhang, Duantengchuan Li, and Patrick CK Hung. Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications. *IEEE Transactions on Services Computing*, 17 (2):604–616, 2024.

Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, 2018.

Dongjin Yu, Yike Jin, Yuqun Zhang, and Xi Zheng. A survey on security issues in services communication of microservices-enabled fog applications. *Concurrency and Computation: Practice and Experience*, 31(22):e4436, 2019.

Shenglin Zhang, Pengxiang Jin, Zihan Lin, Yongqian Sun, Bicheng Zhang, Sibo Xia, Zhengdan Li, Zhenyu Zhong, Minghua Ma, Wa Jin, et al. Robust failure diagnosis of microservice system through multimodal data. *IEEE Trans. Serv. Comput.*, 16(6):3851–3864, 2023.

Zhiyu Zhang, Tao Wang, An Li, and Wenbo Zhang. Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 866–871. IEEE, 2022.

Chenyu Zhao, Minghua Ma, Zhenyu Zhong, Shenglin Zhang, Zhiyuan Tan, Xiao Xiong, LuLu Yu, Jiayi Feng, Yongqian Sun, Yuzhi Zhang, et al. Robust multimodal failure detection for microservice systems. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5639–5649, 2023.

Ziming Zhao, Tiehua Zhang, Zhishu Shen, Hai Dong, Xingjun Ma, Xianhui Liu, and Yun Yang. Chase: A causal heterogeneous graph based framework for root cause analysis in multimodal microservice systems. *arXiv e-prints*, pages arXiv–2406, 2024.

Wu Zhijun, Li Wenjing, Liu Liang, and Yue Meng. Low-rate dos attacks, detection, defense, and challenges: A survey. *IEEE access*, 8:43920–43943, 2020.

Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *Proc. 40th Int. Conf. on Softw. Engineering: Comp. Proc.*, pages 323–324, 2018.

Zhi-Hua Zhou. A brief introduction to weakly supervised learning. *National science review*, 5(1):44–53, 2018.

Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli. Unsupervised algorithms to detect zero-day attacks: Strategy and application. *Ieee Access*, 9: 90603–90615, 2021.