An Empirical Study on the Resilience of Cloud-Native Systems Using Dynamic Scaling Strategies

by

Behrad Moeini

Thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the degree of

Master of Computer Science

in

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

Abstract

Cloud-native systems are essential in customer service, handling complex interactions and fluctuating loads. This thesis provides an empirical study on how companies can assess a predictive analysis method to maintain their cloud-based services focusing on challenges such as increased response times, higher failure rates during peak usage, and inefficient resource allocation.

We perform an empirical study answering important resource management questions and addressing the needs to meet predefined service level objectives (SLOs) for response time and failure rate. Our predictive framework integrates proactive predictive models, real-time monitoring, and Kubernetes' Horizontal Pod Autoscaler (HPA) to dynamically allocate resources effectively.

Our empirical study aims to achieve the following: (1) determine achievable SLOs with fixed resources; (2) identify minimum resources needed to meet desired SLOs; and (3) estimate maximum user capacity with given resources. Using decision tree regression models, we analyzed configurations focusing on CPU and memory utilization.

Our findings show that the number of replicas significantly affects response time and failure rates. CPU utilization thresholds were slightly more effective than the memory thresholds. The optimized models achieved high predictive accuracy, with R-square values up to 0.85.

Our research presents a study on how engineers can perform predictive resource management tasks for cloud-native systems – AI-driven chatbots being an example of such systems – to ensure resilience and responsiveness.

Acknowledgements

I would first like to express my deepest gratitude to my supervisor, **Dr. Shiva Nejati**, for her exceptional guidance, encouragement, and unwavering support throughout the course of this research. Her expertise and mentorship have been instrumental in shaping both the technical direction of this work and my development as a researcher.

I am also sincerely thankful to **Dr. Mehrdad Sabetzadeh** for his valuable insights, constructive feedback, and the thoughtful discussions that greatly enhanced the quality and clarity of this thesis.

To my wife, **Faezeh**, thank you for your unwavering support, patience, and love. Your belief in me, even during the most challenging times, has given me strength and clarity. I am endlessly grateful for your presence by my side.

Finally, I am deeply thankful to my parents, **Faramarz** and **Jila**, whose lifelong support and unconditional belief in my potential have been the foundation of everything I pursue. Their trust in me continues to inspire and sustain my efforts.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Cloud-native architectures are systems designed from the ground up to run in, and make full use of, cloud environments. This architectural paradigm relies on modular components, often packaged as containers, managed through orchestration platforms, and integrated with continuous delivery practices to achieve scalability, resilience, and automation [1].

Scaling these architectures effectively in such distributed and containerized infrastructures introduces a unique set of challenges [2, 3]. Unlike traditional applications, these platforms rely on robust architectures that dynamically adjust computing resources to handle unpredictable spikes in user traffic. Key requirements include maintaining low latency, high availability, and minimized failure rates without over-provisioning, which would raise operational costs [4].

Elasticity, a cornerstone of cloud computing, allows services to automatically adjust resources based on demand [5]. In the context of these modern, container-orchestrated architectures, elasticity must be paired with intelligent resource allocation strategies to prevent overuse of CPU, memory, and networking resources. This orchestration demands fine-grained control of components, such as containers and load balancers, and sophisticated methods to automate resource scaling in response to fluctuating workloads.

Achieving real-time optimization in such systems involves balancing Service Level Objectives (SLOs) like response time and failure rate. For instance, maintaining high responsiveness while ensuring minimal downtime requires predictive models that anticipate resource needs, adapting the infrastructure preemptively to prevent performance degradation [6]. This thesis addresses these operational challenges by exploring methods to predict and meet SLOs reliably, focusing on strategies that dynamically allocate resources to sustain both efficiency and high user satisfaction. Through experimental analysis, we investigate predictive resource allocation models that enable cloud-native systems to maintain optimal performance, ensuring that systems remain resilient and responsive in diverse operational contexts.

Despite these architectural and operational advancements, cloud-based applications still encounter critical issues under varying demand conditions. The following section outlines the specific scalability and performance challenges faced by these systems.

1.2 The Problem: Scalability and Performance in Cloud-Native Systems

This research provides an empirical study aiming to enable engineers and cloud service providers to predict the performance of their services and ensure that their systems achieve their expected Service Level Objectives (SLOs). These distributed cloud-based platforms face significant challenges in maintaining consistent performance and scalability across varying levels of demand. These challenges often manifest in several critical issues, particularly during peak traffic periods:

- **Increased Response Times:** During peak usage, cloud-native systems often experience delays in response times, which undermines the user experience as users expect quick and responsive interactions.
- **Higher Failure Rates:** Systems under heavy load are more prone to errors and service disruptions, resulting in reduced reliability and trust in the service.
- **Resource Management Challenges:**
  - **Limited Resource Availability:** Securing sufficient resources during high-demand periods can be difficult and costly, especially in cloud environments with shared infrastructure.
  - **Resource Wastage:** Over-allocating resources to ensure peak performance often leads to wasted capacity during low-demand periods, which drives up operational costs unnecessarily.

To address these challenges, decision trees were chosen as the predictive model for their interpretability and actionable insights, which are critical for service providers managing complex systems like Kubernetes-based systems. Unlike black-box models, decision trees provide transparent decision rules that service providers can easily understand and implement. For example, our results demonstrate how the decision tree model identifies key parameters, such as the maximum number of replicas, and their thresholds to maintain Service Level Objectives (SLOs). By analyzing features like replica counts and resource thresholds, the model aids cloud-based service providers in configuring Kubernetes autoscalers that are both resource-efficient and performance-driven. Empirical findings reveal that decision trees effectively predict response time and failure rate targets, enabling proactive resource allocation strategies that reduce latency and failure rates without unnecessary over-provisioning. These characteristics make decision trees a practical choice for enabling dynamic, SLO-driven scaling in cloud-native systems. This thesis conducts an empirical study that uses a predictive approach using decision trees to resource management, enabling engineers to:

- Anticipate user demand based on expected user volumes and historical data
- Set target performance levels (e.g., response time, failure rate) as key objectives
- Assess and utilize available resources efficiently to meet demand without overspending

By incorporating predictive models, our study aims to improve cloud-native systems performance under various load conditions. The proposed solution seeks to balance consistent user experience with resource efficiency, ensuring that resources are neither overused

nor underused, thus reducing costs and optimizing performance at all levels of demand. Chatbots provide an excellent test scenario due to their fluctuating and often unpredictable user request patterns, making them ideal candidates for demonstrating the benefits of predictive scaling. A chatbot application, previously developed and deployed in a Kubernetes-based environment at the Sedna Lab [7], serves as the case study. This application, described in earlier work [8], provides a concrete example to validate the predictive resource management approach.

1.3 Significance of the Research

This research addresses important gaps in proactive resource allocation and efficient system adaptation for cloud-based systems. Traditional approaches often use fixed setups or single performance measures, which don't adapt well to the fast-changing needs of cloud-native systems in busy environments. Our work improves on these methods by using a predictive model for resource management, keeping response times low and enhancing user experience even during high traffic.

This research addresses critical gaps in proactive resource allocation and efficient system adaptation for cloud-based systems. Unlike traditional approaches, which rely on fixed setups or single performance measures and struggle to adapt to the dynamic demands of cloud-native environments, our predictive model offers significant improvements. As shown in Table 1.3, our approach enables resource management, maintains low response times, and enhances user experience, particularly during periods of high traffic.

While we refer to our proposed solution as an "approach," it is important to clarify that this term encompasses both a predictive modeling strategy and an empirical evaluation framework. Specifically, our approach includes not only the technical development of decision tree models but also the design of systematic experiments to evaluate their effectiveness under controlled conditions. This integration allows us to assess the models using real-time performance metrics, such as response time and failure rate, across varying resource configurations and user loads. By aligning the model evaluation with empirical research practices—such as variable control, repeated trials, and statistical validation—our study bridges the gap between algorithmic insights and real-world system behavior.

The chatbot case study used in this study was selected for its suitability as an empirical testbed for evaluating predictive resource allocation. We intentionally chose an open-source chatbot with a simple architecture to ensure that the results focus on resource management rather than chatbot complexity. Using a more sophisticated chatbot, such as one with deep NLP models or multi-turn dialogue processing, could introduce additional variability in response times and resource consumption, making it harder to isolate the effects of scaling strategies. By minimizing this noise, our study provides a clear and reproducible evaluation of predictive resource management in cloud-native systems. Furthermore, the open-source nature of the chatbot ensures that our methodology is easy to replicate, enhancing the reliability and scientific impact of our findings.

This chatbot case study is especially valuable from a scientific standpoint for several reasons. First, chatbot systems are inherently latency-sensitive and experience rapid, unpredictable load fluctuations—conditions that test the limits of resource management and auto-scaling strategies. Second, the nature of chatbot interactions, typically involving

real-time language processing, creates measurable performance indicators (such as response time and failure rate) that map directly onto key Service Level Objectives (SLOs). This direct mapping makes it possible to evaluate the effectiveness of predictive resource allocation strategies with minimal ambiguity. Third, this particular chatbot was deployed in a Kubernetes-based environment with precise control over resource thresholds, replica limits, and monitoring metrics. This setup allowed us to conduct a wide range of controlled experiments while preserving ecological validity. These characteristics collectively make the chatbot domain not only relevant but also scientifically rigorous for empirical exploration of dynamic resource scaling in cloud-native systems.

Area of Comparison	**References**	**Our Approach**
Predictive Scaling	Zhang et al. (2024) [9]: Scaling based on a standard pre-set thresholds	Predictive resource allocation based on demand forecasting
Multi-Cloud Efficiency	Repetto (2023) [10]: Distributed scaling for network resilience	Proactive, latency-aware distribution to ensure low response times
Service Level Objectives	Zhu et al. (2023) [11]: Multi-metric optimization without SLO integration	SLO-based framework for maintaining service reliability
Real-Time Metrics	Kang et al. (2023) [12]: Optimization for batch deep learning	Dynamic adjustment based on cloud native system SLO metrics

Table 1.1: Comparison of traditional and predictive resource management approaches.

1.3.1 Novel Contributions

Our research makes three main contributions to scaling cloud native systems on the cloud:

- **Proactive Predictive Resource Allocation:** Existing methods for resource scaling, such as Zhang et al. [9], rely on reactive, pre-set thresholds, where resources are added or removed once specific limits are reached. This approach can be slow to respond to sudden traffic surges, leading to performance dips. Our study, in con-

trast, focuses on the application of predictive models and using frequent and detailed demand patterns to forecast resource needs over short time horizons. Our findings demonstrate the ability of predictive methods to inform more proactive scaling strategies in the implementation. By providing insights into demand fluctuations and forecasting accuracy, this work contributes to the development of systems that could minimize response delays and improve efficiency through informed decision-making.

- **Latency-Aware Load Balancing for Cloud-native systems:** Prior research, such as Repetto [10], has primarily focused on multi-cloud distribution methods that optimize for network resilience without specifically addressing the needs of latency-sensitive tasks. In this work, we analyze the impact of latency-aware load balancing strategies tailored for AI-driven chatbots operating in multi-cloud environments. By studying real-time latency data and its effects on response times, we provide empirical insights into how latency-sensitive approaches could enhance user experience.

Overall, our framework provides a flexible, scalable solution tailored to the needs of cloud-native applications, supporting reliable customer interactions across various industries. With this foundation and motivation established, the next step is to outline the structure of this thesis and guide the reader through the subsequent chapters.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Related Works**: — This chapter reviews existing literature on chatbot scalability, resource allocation strategies in cloud-native environments, and predictive modeling techniques. We identify the gaps in current approaches, particularly in real-time predictive resource management for AI-driven chatbots, setting the stage for our research.

- **Chapter 3: Background** — We provide foundational knowledge on cloud-native architectures, Service Level Objectives (SLOs), performance prediction in distributed systems, and dynamic resource management in AI systems. This background equips the reader with the necessary context to understand the challenges and solutions presented in our work.

- **Chapter 4: Metrics-Driven Kubernetes Auto-Scaler** — This chapter details the design of the scalable cloud native application(chatbot) system. We discuss how Kubernetes and its Horizontal Pod Autoscaler (HPA) are leveraged to achieve dynamic scalability. The architectural choices and strategies implemented to address the identified challenges are thoroughly examined.

- **Chapter 5: Empirical Evaluation** — We outline the experimental setup, including the server configurations, variables tested, and data collection methods. This chapter explains how we formulated our research questions, the decision tree regression models used for prediction, and the evaluation metrics for assessing performance.

- **Chapter 6: Empirical Study Results** — We present and analyze the findings from our experiments. The effectiveness of our predictive resource management framework

in meeting the defined SLOs is evaluated. We discuss how the results validate our approach and their implications for both academia and industry.

- **Chapter 7: Conclusion** — We are presenting the conclusion and key takeaways from our results. We are getting close to the part. We are also defining some future works and limitations, and wrapping up the chapter with closing remarks.

In summary, this thesis addresses the need for scalable and reliable AI-driven chatbot systems in cloud-native environments. By developing a predictive resource management framework and rigorously testing it through experiments, we demonstrate how dynamic scaling can meet performance and reliability targets while optimizing resource use. The insights and methodologies presented contribute valuable knowledge to the field, paving the way for more resilient and efficient applications in the rapidly evolving landscape of cloud-native technologies.

Chapter 2

Related Work

Recently, cloud-native systems have seen rapid advancements, especially with the rise of AI-driven models, cloud-native infrastructures, and scalable solutions [11–22]. This chapter explores related studies, focusing on scalability, performance, resource management, predictive models, and operational thresholds, and compares these studies to our approach.

2.1 Scalability in Cloud-Native Architectures

Scalability and performance optimization are crucial for handling dynamic user demands in chatbot systems. Kubernetes-based infrastructure, as discussed by Milroy et al. [13], has shown potential for scaling HPC environments, but such studies lack real-time predictive scalability models specific to chatbot systems. Repetto [10] analyzed real-time resilience against network attacks, though without incorporating predictive scaling, which we address through AI-integrated cloud-native chatbot systems.

2.2 Resource Allocation for Cloud-Native Systems

Resource management in cloud-native systems is fundamental to ensuring consistent performance and cost-efficiency. Khan et al. [16] introduced a framework for SLA-aware resource recommendations, aiming to optimize resource allocation in cloud functions. While this framework enhances resource predictability, it does not address the specific latencies of cloud-native services. Kumar et al. [17] expanded on resource optimization through proactive scaling, utilizing custom resource definitions to adjust for demand . Our research diverges by applying these strategies specifically to AI chatbots, balancing resource efficiency with real-time user engagement.

Additionally, Theodoropoulos et al. [18] presented an overview of resource observability for cloud-native applications, advocating for enhanced monitoring to prevent system overload. Their approach lacks predictive resource modeling for AI systems, a feature our research includes to maintain optimal resource utilization and reduce response time variability under high loads.

2.3 Predictive Models for Cloud Service Scalability

Predictive models play an essential role in proactive scaling for cloud-native systems. To assess their effectiveness, we first established baseline models for resource allocation and customer support estimation. These baseline models relied on simple decision trees trained on historical data but lacked fine-tuned hyperparameters. Deng et al. [23] discussed load prediction for optimizing resource allocation in time-sensitive applications. However, it

does not consider the unique conversational workload patterns of chatbot systems. In contrast, our research develops predictive models tailored to chatbot traffic, enabling more accurate resource scaling and reduced latency.

Joshi et al. [20] explored declarative orchestration for machine learning models, employing Bayesian optimization for resource autoconfiguration. While effective for general AI applications, their work does not address the distinct interaction patterns in chatbot systems. Our work builds on their orchestration strategies by incorporating interaction frequency and conversational context, enhancing model efficiency under varying load conditions.

2.4 User Load Testing and Capacity Estimation

Determining optimal user capacity without compromising system stability is another focal area. Studies on cloud-native applications frequently employ load testing to establish thresholds for optimal performance. Lertpongrujikorn et al. [21] conducted load testing in serverless environments, deploying Knative for resource management, and concluded that thresholds are critical to avoid performance degradation. However, they did not apply these findings to conversational applications with fluctuating demand.

2.5 Service-Level Objectives (SLO) Management

Service-level objectives (SLOs) are critical for maintaining operational standards in cloud-native applications. Toffetti et al. [24] presented a self-management model for ensuring

SLOs in cloud-native systems, using load-based adaptations. While comprehensive, their approach does not incorporate predictive mechanisms, leading to reactive rather than proactive adjustments.

We adopt a predictive approach to SLO adherence, leveraging real-time metrics to anticipate and mitigate potential bottlenecks. This aligns with the work of Podolskiy et al. [25], who focused on SLO preservation through adaptive resource sharing but without focus on cloud-native systems which work directly with end-users. Our research fills this gap by integrating real-time adjustments, ensuring consistent SLO adherence and enhanced user satisfaction.

Area of Focus	**Existing Work**	**Our Research**
Predictive Resource Scaling	Predictive scaling with real-time metrics for cloud-native systems [9, 26–30]	Proactive Planning before deployment
Latency Optimization	General cloud optimization [11, 12, 19, 31–33]	Specific to chatbot response times
User Load Testing	Applied to serverless environments [19, 21, 22, 34, 35]	Focused on conversational workload patterns
SLO Management	Reactive self-management [3, 24, 25, 36]	Proactive adjustments specific to cloud-native systems

Table 2.1: Comparison of Optimization Techniques Across Studies

2.6 Discussion

Existing works primarily analyze cloud-native, **serverless** architectures [19, 22, 34, 35], where computational resources are automatically allocated and scaled by cloud providers. In contrast, our chatbot system operates in a **self-hosted, local environment**, where re-

source provisioning follows static policies. Therefore, direct comparisons with prior baseline models were not feasible due to substantial differences in system architecture, deployment environments, and operational assumptions. One reason why most prior research relies on serverless architectures is the ability to dynamically allocate and scale computing resources without manual intervention [34]. Cloud providers manage resource provisioning based on demand, making serverless deployments ideal for handling unpredictable workloads. However, this comes at the cost of **latency variability, vendor dependency, and limited control over execution environments** [37]. Our study focuses on a **local, self-hosted** chatbot deployment, where these constraints are eliminated, but scalability must be explicitly managed through predictive scaling techniques.

This distinction has a significant impact on scalability behavior. Many serverless implementations rely on **on-demand instance scaling** [34, 38], whereas our chatbot system requires **explicit resource allocation and predictive scaling** using decision trees. Consequently, comparing response time, failure rate, and resource utilization between these fundamentally different architectures would be misleading.

Furthermore, prior research optimizes general-purpose cloud applications with **uniform traffic patterns** [37], while our chatbot workload exhibits **interactive, real-time user engagement** with dynamic traffic fluctuations. Given these constraints, we adopted a self-contained evaluation framework to measure the effectiveness of our predictive models in the specific context of **local AI-driven chatbot deployments**.

2.7 Conclusion

The literature on cloud-native systems reveals significant progress in areas such as scalability, performance optimization, and resource management. However, most existing studies lack integration of real-time predictive models and comprehensive SLO optimization, which are critical for maintaining consistent cloud-native system performance under fluctuating demand. Our study addresses these gaps by providing an empirical study to predict resource management, and demonstrates how the predictive information obtained from the study can help service providers efficiently manage their resources while ensuring that SLOs are consistently met.

Chapter 3

Background

In this chapter, we discuss the background necessary to follow the concepts that are foundational to this thesis. This includes the core concepts of cloud-native environments, Service Level Objectives (SLOs), performance prediction in distributed systems, scalability in AI-driven chatbots, dynamic resource management, system reliability, and predictive modeling for resource optimization. These topics provide the context necessary to understand the challenges and solutions presented in this research.

3.1 Scalability Challenges in AI-Driven Chatbots

Scalability is a critical challenge for AI-driven systems, especially chatbots, which must handle variable and often unpredictable user loads. Traditional rule-based chatbots, as shown in Table 3.1, are relatively lightweight and can be scaled more easily. However,

modern AI-driven chatbots, which rely on complex machine learning models (such as Natural Language Processing models), require significantly more computational resources, as they offer dynamic, contextual responses rather than static responses.

Scalability methods can be further understood through Figure 3.1, which illustrates the differences between horizontal and vertical scaling in cloud-native environments. Horizontal scaling involves adding more instances (pods) to distribute the load, making it suitable for systems like rule-based chatbots. In contrast, vertical scaling, which increases the resources for a single instance, is often necessary for AI-driven chatbots due to their higher computational demands.

Feature	**Rule-Based Chatbot**	**AI-Driven Chatbot**
Resource Requirement	Low	High
Response Adaptability	Fixed responses	Dynamic, contextual responses
Scalability	Easier to scale	Challenging, high computational needs

Table 3.1: Differences Between Rule-Based and AI-Driven Chatbots

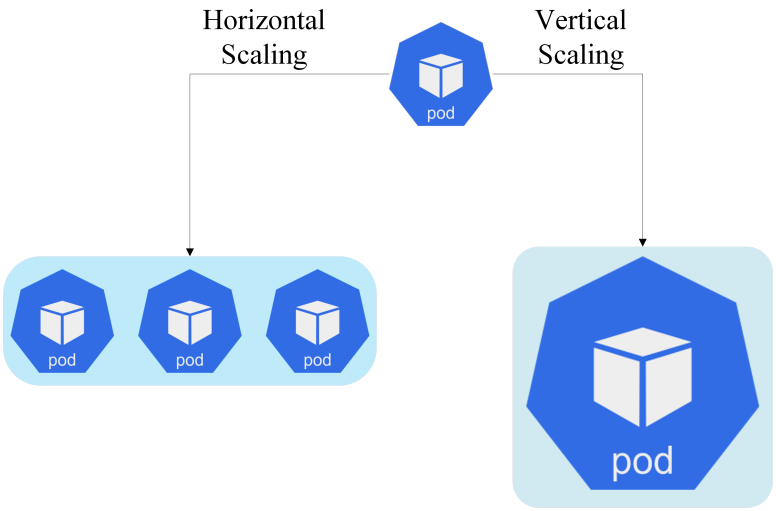

Figure 3.1: Scalability in Cloud-Native Environments: Horizontal vs. Vertical Scaling

3.2 Cloud-Native Environments and Scalability

A cloud-native environment refers to an ecosystem that takes full advantage of cloud computing models, allowing applications to scale dynamically and be highly resilient [39]. Unlike traditional cloud computing [40], which primarily focuses on moving existing applications to a cloud infrastructure, cloud-native environments are designed with the cloud in mind from the beginning.

Cloud-native systems use microservice architecture, containerization, and orchestration platforms like Kubernetes. Microservices break down applications into smaller, manageable components, each of which can be independently scaled [39]. Containerization allows for consistent environments, making it easier to deploy and scale applications. Kubernetes, for instance, handles orchestration, allowing services to scale dynamically based on resource

demands [41]. Kubernetes ensures that microservices can be replicated, scaled, and load balanced across different environments [42].

A comparison of traditional environments with cloud-native environments, highlighting the differences in scalability, resource management, resiliency, and deployment strategy, is provided in Table 3.2. This table outlines the fundamental advantages that cloud-native architectures bring to application design and scalability.

Feature	**Traditional Environment**	**Cloud-Native Environment**
Scalability	Limited, typically vertical [43]	Dynamic, horizontal [44]
Resource Management	Static allocation [43]	Dynamic allocation [44, 45]
Resiliency	Single point of failure [24]	Microservices for high availability [45]
Deployment Strategy	Monolithic updates [24]	Continuous integration and deployment [44]

Table 3.2: Comparison of Traditional vs. Cloud-Native Environments

In the context of this thesis, scalability is a critical concern. As chatbot usage can fluctuate drastically depending on user traffic, it is essential to ensure that the system can scale up resources during high-demand periods while scaling down during low-demand periods to conserve resources [46]. This thesis focuses on optimizing this dynamic scalability, ensuring that resource allocation is not only efficient but also cost-effective

3.3 Service Level Objectives (SLOs) and Service Level Agreements (SLAs)

Service Level Objectives (SLOs) define the measurable goals for a system's performance and availability. These objectives are foundational to ensuring reliability and compliance in service delivery, especially within cloud-native environments where applications must scale dynamically and meet varying demands [47]. SLOs are typically embedded within Service Level Agreements (SLAs), which are formal contracts that outline service expectations, including aspects like uptime, response times, and reliability [48]. SLAs ensure accountability and provide clear repercussions if service levels are not met, making them essential for both providers and consumers in cloud-native ecosystems.

SLAs provide high-level guarantees, while SLOs serve as the actionable metrics to achieve these guarantees. For instance, an SLA might guarantee 99.9% uptime for a service, while an SLO will measure the exact uptime performance. In cloud-native environments, SLOs can be applied to track critical performance metrics, such as response time and error rates, ensuring that the system maintains consistent performance under varying loads [49]. Table 3.3 provides common examples of SLO metrics and target values relevant to cloud-native systems, where high reliability and rapid response times are imperative [50].

SLO Metric	**Target Value**
Response Time	90% of requests in < 20 seconds [47]
Availability	99.9% uptime [51]
Failure Rate	$< 1\%$ of requests

Table 3.3: Examples of SLO Metrics and SLAs in Cloud-Native Systems

In cloud-native environments, SLOs play a critical role due to the inherent scalability and distributed nature of these systems. Traditional environments often rely on monolithic structures, where performance and availability metrics are easier to measure but less adaptable to fluctuations in demand. In contrast, cloud-native environments use microservices and containerized architectures, which allow for independent scaling of components and thereby improve flexibility and resilience [49].

Within the context of this thesis, SLOs are used to optimize the scalability and resilience of a chatbot system. By setting and monitoring SLOs, such as response time and availability, the system can dynamically scale resources in response to real-time traffic, ensuring that it meets performance targets even during peak periods. As shown in Table 3.3, these SLOs serve as key metrics for maintaining service quality, which is crucial for the real-time, high-availability demands of interactive chatbot applications.

Through continuous monitoring and adherence to defined SLOs, this thesis demonstrates an efficient and scalable framework for managing resources in a cloud-native chatbot environment. By optimizing resource allocation based on real-time SLO data, the system can proactively adjust capacity, ensuring reliability and performance without unnecessary overprovisioning, thus achieving both operational efficiency and customer satisfaction.

3.4 Performance Prediction and Resource Allocation in Distributed Systems

Performance prediction in distributed systems refers to the ability to anticipate system behavior under different load conditions. In cloud-native environments, where resource allocation is dynamic, predicting performance is critical for ensuring that services meet their Service Level Objectives (SLOs). Effective performance prediction enables systems to allocate resources proactively, preventing performance degradation during high-demand periods [52].

Resource allocation techniques in distributed systems can vary significantly, with common approaches including static, dynamic, and predictive allocation. Table 3.4 compares these methods, highlighting their advantages and disadvantages. Static allocation is straightforward but inefficient under variable loads. Dynamic allocation optimizes resource use but adds complexity in prediction and orchestration. Predictive allocation, which uses AI-driven models, aims to balance resource availability with demand, though it requires accurate prediction models to function effectively.

Technique	**Advantages**	**Disadvantages**
Static Allocation	Simple and predictable [53]	Inefficient under variable load [52]
Dynamic Allocation	Efficient resource use [54]	Complexity in prediction and orchestration [55]
Predictive Allocation	Optimizes performance [56]	Requires accurate prediction models [52]

Table 3.4: Resource Allocation Techniques in Distributed Systems

3.5 Dynamic Resource Management in AI Systems

Dynamic resource management refers to the real-time adjustment of resources based on changing conditions. This is especially relevant in cloud-native environments, where resource demands fluctuate with user traffic and workload complexity [53, 55].

Dynamic resource management in AI systems often involves using sophisticated tools like the Kubernetes Horizontal Pod Autoscaler (HPA), and Cloud Autoscaler. Table 3.5 compares these tools based on metrics tracked, granularity, and cost efficiency [52, 54, 57]. Kubernetes HPA, for instance, allows scaling at the pod level based on metrics like CPU and memory utilization, providing high granularity and cost-efficiency.

Feature	**Kubernetes HPA**	**VM-Based Autoscaler**	**Group-Based Autoscaler**
Metrics Tracked	CPU, Memory	Customizable metrics	CPU, Memory, Custom
Granularity	Pod level	VM instance level	Instance group level
Cost Efficiency	High	Medium	High

Table 3.5: Features of Dynamic Resource Management Tools

In the context of this thesis, dynamic resource management enables the chatbot system to adapt its resources based on real-time metrics. By using Kubernetes HPA, the system dynamically scales based on traffic patterns, ensuring responsiveness and minimizing costs during low-demand periods. This approach aligns with the scalability objectives defined for cloud-native environments [58, 59], emphasizing cost efficiency and performance reliability.

3.6 Failure Rates and System Reliability in Distributed AI Models

In distributed AI systems, reliability is critical, especially for applications like chatbots that require high availability. System failures can arise from various factors, including network latency, resource contention, node failure, and insufficient scaling. Understanding and addressing these factors is essential for maintaining performance under varying loads [60].

One significant factor impacting reliability is network latency, which increases response times and degrades user experience. Solutions such as Content Delivery Networks (CDNs) and edge computing help mitigate this issue by reducing the physical distance data must travel [61]. Resource contention occurs when multiple processes compete for limited resources, leading to node failures or slow performance. Implementing dynamic resource allocation and load balancing strategies can help reduce contention and improve overall system efficiency. Node failure can also severely impact system reliability; therefore, maintaining sufficient redundancy and monitoring is crucial to prevent such failures. Finally, insufficient scaling during peak demand can result in high failure rates. Predictive autoscaling models enable systems to allocate resources based on anticipated load, ensuring stability during traffic surges [62].

Table 3.6 summarizes these factors and their corresponding mitigation strategies. Figure 3.2 visually represents the causes of system failure in the fishbone diagram, illustrating their relationships and potential impacts on system reliability.

Factor	**Effect on Reliability**	**Mitigation Strategy**
Network Latency	Increased response times	Content Delivery Network (CDN) and edge computing [63]
Resource Contention	Node failure, slow performance	Dynamic resource allocation, load balancing [64]
Node Failure	Reduced system reliability	Redundancy and proactive monitoring [65]
Insufficient Scaling	High failure rate during peak times	Autoscaling with predictive modeling [62]

Table 3.6: Factors Affecting System Reliability and Mitigation Strategies in Distributed AI Models

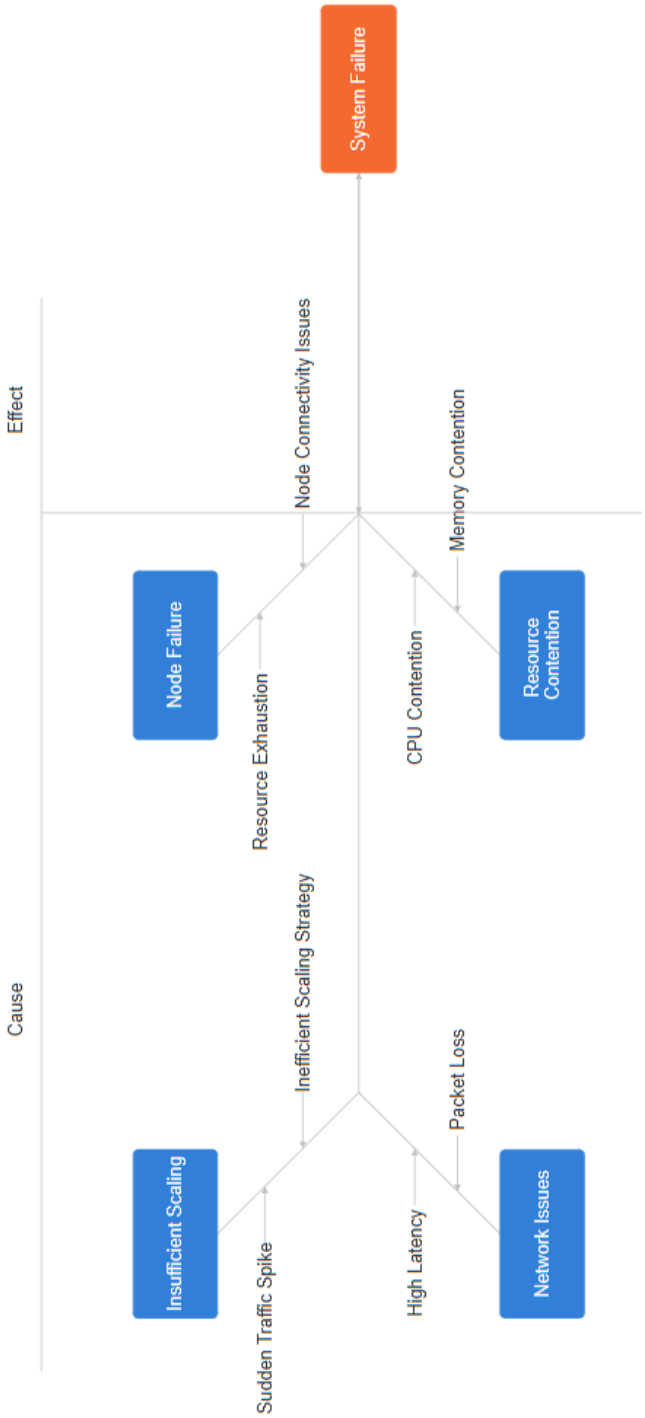

Figure 3.2: System Reliability and Failure Rate Analysis in AI Systems (Fishbone Diagram)

3.7 Predictive Modeling in Resource Optimization

Predictive modeling is essential for efficient resource allocation in cloud-native environments. By analyzing historical data and real-time metrics, these models can forecast resource needs, allowing systems to scale resources proactively. This approach reduces resource waste during low-traffic times and ensures sufficient resources during peak periods, thereby optimizing both cost and performance [52].

Different predictive models are suitable for specific scenarios in resource optimization. Decision trees offer interpretability and are effective for handling non-linear relationships, but can overfit in complex environments [66]. Linear regression is simpler and works well for linear trends but is less effective in complex environments. Neural networks, while more complex and requiring larger datasets, excel in large-scale applications due to their ability to model complex patterns.

Table 3.7 compares these predictive models, outlining their strengths, limitations, and best use cases in resource allocation.

Model Type	**Pros**	**Cons**	**Best Use Case**
Decision Tree	Interpretable	Overfitting risk	Non-linear relationships
Linear Regression	Simple	Poor for complex data	Linear trends
Neural Networks	Handles complexity	Needs large datasets	Large-scale applications

Table 3.7: Comparison of Predictive Models for Resource Allocation

3.8 Decision Tree for Regression

Before applying optimized decision trees, we initially developed baseline models for both resource allocation (RQ1) and customer support capacity (RQ3). These baseline models used unoptimized decision trees with default settings, providing a preliminary assessment of resource requirements. Decision trees for regression are hierarchical models used to predict continuous target variables. The training process aims to partition the input space to minimize the prediction error at each split [67, 68].

3.9 Training a Decision Tree for Regression

Given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathbf{x}_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \mathbb{R}$ is the target variable, the goal is to recursively split the data to minimize the squared error at each node [69]. Let $\mathcal{D}_t$ denote the subset of the dataset at node t. At each node, the algorithm selects the feature j and threshold θ that minimize the mean squared error (MSE) after the split.

The subset $\mathcal{D}_t$ is split into:

$$\mathcal{D}_{t,\text{left}} = \{(\mathbf{x}_i, y_i) \in \mathcal{D}_t : x_{i,j} \leq \theta\}, \quad \mathcal{D}_{t,\text{right}} = \mathcal{D}_t \setminus \mathcal{D}_{t,\text{left}}.$$

The prediction at each node is calculated as the mean of the target values:

$$\hat{y}_t = \frac{1}{|\mathcal{D}_t|} \sum_{i \in \mathcal{D}_t} y_i.$$

The loss function at node t is defined as the total squared error:

$$\mathcal{L}(t) = \sum_{i \in \mathcal{D}_t} \left(y_i - \hat{y}_t\right)^2.$$

For a candidate split (j, θ), the weighted loss after the split is:

$$\text{Loss}(j, \theta) = \frac{|\mathcal{D}_{t,\text{left}}|}{|\mathcal{D}_t|} \mathcal{L}(\mathcal{D}_{t,\text{left}}) + \frac{|\mathcal{D}_{t,\text{right}}|}{|\mathcal{D}_t|} \mathcal{L}(\mathcal{D}_{t,\text{right}}).$$

The optimal feature j and threshold θ are chosen to minimize this loss:

$$(j^*, \theta^*) = \arg\min_{j,\theta} \text{Loss}(j, \theta).$$

This process continues recursively until a stopping criterion is met, such as reaching a maximum depth, a minimum number of samples per leaf, or a minimum reduction in loss.

3.9.1 Prediction Using a Decision Tree for Regression

To make a prediction for a new input $\mathbf{x} \in \mathbb{R}^d$, the decision tree traverses from the root to a leaf node. At each internal node, the feature j and threshold θ determine the direction taken by the traversal:

$$\text{If } x_j \leq \theta, \text{ move to the left child; otherwise, move to the right child.}$$

This process continues until a leaf node is reached, at which point the prediction is the

mean target value at the leaf:

$$\hat{y} = \frac{1}{|\mathcal{D}_{\text{leaf}}|} \sum_{i \in \mathcal{D}_{\text{leaf}}} y_i.$$

3.9.2 Mathematical Regularization for Regression Trees

To prevent overfitting, regression trees often use regularization techniques to simplify the model:

1. **Maximum Depth** ($d_{\max}$): The maximum number of levels in the tree is limited.
2. **Minimum Samples per Leaf** ($n_{\min}$): A minimum number of samples is required in each leaf node.
3. **Pruning**: Subtrees that contribute minimally to reducing the loss are removed. The cost-complexity pruning criterion minimizes:

$$\text{Cost}(T) = \sum_{t \in \text{leaves}(T)} \mathcal{L}(\mathcal{D}_t) + \lambda |T|,$$

where $|T|$ is the number of leaf nodes, and λ is a penalty for model complexity.

3.9.3 Feature Importance in Regression Trees

Feature importance measures how much each feature contributes to reducing the loss across the tree. For a feature j, its importance is calculated as:

$$\Delta \mathcal{L}_j = \sum_{t \in T_j} \Big(\mathcal{L}(\mathcal{D}_t) - \frac{|\mathcal{D}_{t,\text{left}}|}{|\mathcal{D}_t|} \mathcal{L}(\mathcal{D}_{t,\text{left}}) - \frac{|\mathcal{D}_{t,\text{right}}|}{|\mathcal{D}_t|} \mathcal{L}(\mathcal{D}_{t,\text{right}}) \Big),$$

where T_j is the set of nodes where feature j is used for splitting. The total importance is normalized to compare features.

Chapter 4

Metrics-Driven Kubernetes Auto-Scaler

In this chapter, we address the scalability requirements of our chatbot system to ensure seamless performance under varying user demands. Modern chatbots often encounter fluctuating workloads, ranging from routine interactions to spikes during promotional events or peak usage times. To meet these challenges, we leverage Kubernetes' dynamic resource management capabilities, focusing on its Horizontal Pod Autoscaler (HPA). By enabling the cloud-native systems to scale horizontally in response to real-time resource utilization, we ensure optimal system performance, efficient resource use, and cost-effectiveness.

The chapter explores the integration of Kubernetes into our cloud-native system, detailing the architectural setup, auto-scaling mechanisms, and load balancing strategies. Specifically, we illustrate how Kubernetes supports adaptive scaling through pod configurations and dynamic traffic routing, which are critical for maintaining high availability and user satisfaction during peak loads.

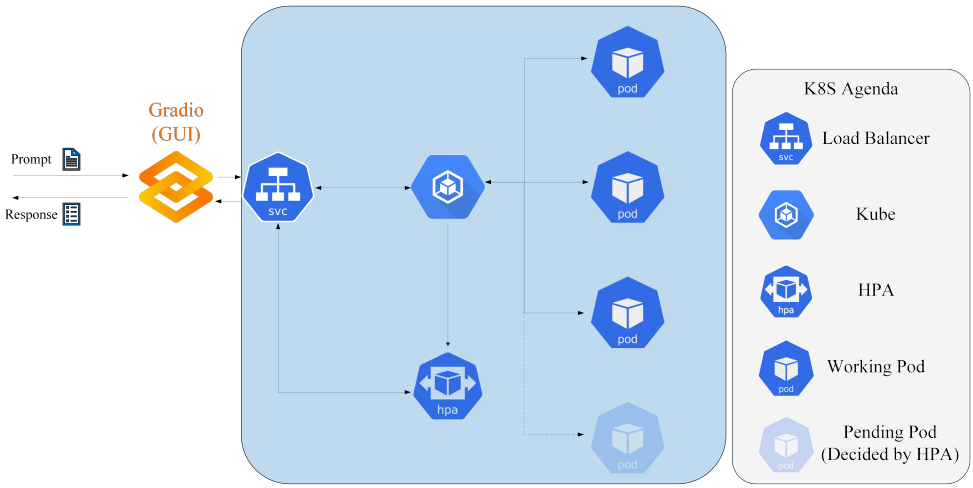

Figure 4.1: High-level architecture of the Kubernetes-based cloud-native system

Figure 4.1 provides a visual representation of the cloud-native system's architecture managed via Kubernetes, illustrating the dynamic scalability and load management critical for handling varying user loads efficiently. Each pod, depicted in the diagram, runs an instance of the cloud-native application. The scalability of these pods is dynamically managed by the Horizontal Pod Autoscaler (HPA), which adjusts the number of active pods based on real-time resource utilization to maintain optimal performance. As pods are scaled, Kubernetes services dynamically adjust routing rules to distribute user requests evenly across all active pods. This effective load balancing prevents any single pod from becoming overwhelmed, thus maintaining high system efficiency and user response quality.

4.1 Scalability in Cloud-Native Systems

Scalability is essential for a cloud-native's ability to handle increased workloads effectively or to expand its capabilities smoothly. Specifically, scalability ensures that cloud-native systems maintain high performance standards, especially when facing fluctuating user demands. For example, during product launches or major promotional events, customer service bots may experience sudden and significant spikes in user interaction. If a cloud-native system is initially set up to manage only up to 100 simultaneous interactions but experiences 500 during peak events, it may slow down significantly or become unresponsive.

There are two main approaches to scaling cloud-native systems: horizontal and vertical scalability. Horizontal scalability involves adding more servers or instances to the existing pool—essentially scaling out. This is beneficial because it allows the cloud-native system to flexibly adjust the number of active servers based on real-time user demand, scaling up during busy periods and down during quieter ones. Vertical scalability, on the other hand, involves upgrading the existing servers with more powerful hardware—scaling up. This approach might require a significant upfront investment in more powerful hardware, which might not be fully utilized except during peak times. Both scaling approaches is illustrated in Figure 4.2.

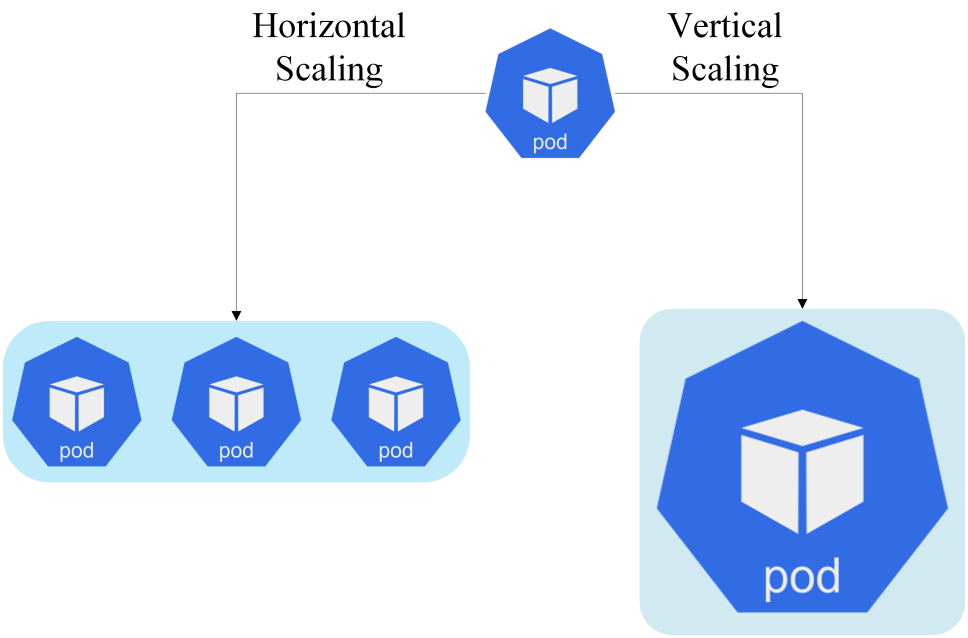

Figure 4.2: Horizontal vs. Vertical Scaling

The chatbot used in our study designed on the PrivateGPT framework, horizontal scalability is more advantageous. It allows us to dynamically manage resources, adding or reducing server instances as needed without the need for constant hardware upgrades. This not only optimizes resource utilization but also keeps operating costs more manageable. Vertical scalability, while effective for constant high loads, does not offer the flexibility needed to handle the variable loads typical of chatbot interactions, making it a less suitable option for our needs.

4.2 Architecture Overview

To achieve the scalability described previously, the chatbot employs a Kubernetes-driven architecture that supports dynamic scaling with minimal manual intervention. This architecture centers around Kubernetes pods, which house the chatbot's application instances. Each pod is designed to operate independently, which allows for efficient scaling as user demand fluctuates. The use of Kubernetes not only simplifies scaling operations but also ensures that the system can adapt quickly to changes in load without compromising performance.

4.2.1 Chatbot Model and Purpose

The chatbot deployed in our study is built upon the **PrivateGPT** framework, an open-source large language model (LLM) optimized for private, local inference. Rather than training a model from scratch, we fine-tuned an existing open-source LLM using a domain-specific dataset to enhance its relevance to our use case. The chatbot was designed to assist with automated customer support by providing contextual responses based on predefined knowledge bases. It supports **natural language interactions**, processes user queries dynamically, and adapts responses based on inferred intent. Unlike cloud-based LLMs, which rely on remote processing, our implementation ensures **data privacy** by running entirely on local infrastructure. This allows for compliance with stringent data security requirements while maintaining low-latency interactions.

4.2.2 Auto-scaling

The key to the chatbot's scalability lies in the Kubernetes Horizontal Pod Autoscaler (HPA), which intelligently adjusts the number of active pods based on real-time resource usage, ensuring optimal performance across varying loads.

Mechanism Of Action: The HPA functions through a series of steps to accurately scale the system:

- **Metric Observation**: The HPA continuously monitors the average CPU and memory usage across all pods. This constant surveillance allows it to react promptly to changes in demand.
- **Calculation of Desired Replicas**: If the average CPU utilization exceeds the predefined thresholds, the HPA calculates the necessary number of replicas to handle increased load without overloading the system. The formula used is:

$$\text{Desired Replicas} = \left\lceil \frac{\text{Current Replicas} \times \text{Current Resource Utilization}}{\text{Target Resource Utilization}} \right\rceil \tag{4.1}$$

  - Current Replicas: The number of pods currently deployed.
  - Current Resource Utilization: The observed average resource usage.
  - Target Resource Utilization: The desired resource usage per pod to maintain optimal performance.

- **Enforcement of Replica Limits**: The HPA ensures that the number of pods remains within the set minimum and maximum limits to maintain efficiency and

cost-effectiveness. The formula ensures any calculated increase in pods is feasible within the system's operational constraints.

For example, if the system starts with 10 pods each at 60% CPU usage and the target utilization is 50%, the HPA would calculate:

$$\text{Desired Replicas} = \left\lceil \frac{10 \times 60}{50} \right\rceil = \lceil 12 \rceil = 12 \tag{4.2}$$

This would trigger an increase to 12 pods under the current conditions.

4.2.3 Load Balancing During Auto-Scaling

Effective load balancing is crucial as the number of pods changes. Kubernetes manages this through its service definitions, which act as reliable intermediaries for directing traffic. These services ensure that:

- Requests are distributed evenly across all pods, preventing any single pod from becoming a bottleneck.
- The system can adapt seamlessly as pods are added or removed, maintaining service continuity and responsiveness.
- As pods are scaled up or down, Kubernetes services adjust their routing rules to ensure that all pods receive a proportionate amount of traffic, optimizing the use of available resources and maintaining high performance.

This dynamic adjustment is crucial during peak load times when the number of pods may fluctuate significantly.

This chapter detailed the scalable architecture of the Kubernetes-based cloud-native system, with a particular focus on the chatbot we used. It emphasized the role of the HPA and load balancing in adapting to varying user demands. In the next chapter, we will connect this architectural framework to specific research questions concerning the chatbot's performance. This transition will delve into how the system sustains efficiency and responsiveness under dynamic loads, paving the way for a more detailed analysis of its scalability and overall effectiveness.

Chapter 5

Empirical Evaluation

This chapter is dedicated to evaluating the conducted research and investigating the balance between computational resources, number of users, and service requirements. In this research, we have identified two SLOs, which will be detailed first. Our research is limited to these SLOs. Following the presentation of the SLOs, we define three research questions. We then explain how we plan to address each research question considering these SLOs.

The SLOs under consideration are:

- **SLO1 (Response Time Objective):** (X%) of requests to be responded to within 20 seconds.
- **SLO2 (Failure Rate Objective):** Failure rate of requests below a certain threshold (Y%).

In this research, we investigate three research questions that a provider of chatbot

services must address to optimize the use of their computational resources. These questions are essential for understanding how to balance the available resources with the number of users that can be efficiently supported without compromising service quality. Each question targets a different aspect of resource allocation and chatbot performance, directly impacting the scalability and reliability of service delivery.

- **RQ1: Resource Allocation for Specific Traffic Volumes**: Given a fixed set of resources (CPU, Memory, and a maximum number of replicas), what are the achievable Service Level Objectives (SLOs) for a specified number of users? This question aims to determine the optimal configuration of resources needed to maintain service quality under varying user loads. It helps in understanding the limits of current resource allocation and how it can be adjusted to meet service demands.

- **RQ2: Resource Requirements for Desired SLOs**: What are the minimum resource requirements (CPU, Memory, and number of replicas) to meet predetermined SLOs for a given customer load? This question explores the lowest possible resource configuration that still achieves the desired service standards. Identifying these minimum requirements helps providers in optimizing their operational costs while ensuring customer satisfaction.

- **RQ3: Customer Support Capacity with Given Resources and SLOs**: With a defined set of resources and desired SLOs, what is the maximum number of customers that can be supported while maintaining service quality? This question is significant for scalability operations, as it helps predict the maximum service capacity possible

without sacrificing quality, informing strategic decisions in resource management and customer service planning.

RQ	Inputs	Predicted Output
RQ1	Threshold type, value, max replicas	% responses <20s, failure rate
RQ2	CPU, memory, failure rate, SLOs	Combined Score (performance + efficiency)
RQ3	Threshold type, value, max replicas	Max supported users (requests)

Table 5.1: Decision Tree Regressor: Inputs and Predicted Outputs

5.1 Experimental Setting

Our experimental design is structured to systematically assess the chatbot's responsiveness, scalability, and resource utilization under a controlled yet realistic load. To achieve this, we employed the following procedure:

1. Preparation: The chatbot was deployed on an HTTP-capable server, using Locust to simulate user interactions via request streams from a separate device.

2. Scalability: An auto-scaling feature was enabled to adjust server computational resources based on thresholds on CPU or memory usage.

3. Load Generation: Test runs involved subjecting the chatbot to a uniform request flow for a fixed duration.

4. Monitoring and Recording: Key metrics such as response times, failure rates, CPU/memory usage, and replica count were tracked during each experiment.

5. Variation of Conditions: Experiments were varied in configurations, including request numbers and auto-scaling thresholds, to analyze chatbot performance across different scenarios.

5.1.1 Differences Between Local and Cloud-Based Deployments

Most chatbot systems in prior research have been deployed using **cloud-based architectures** or **Function-as-a-Service (FaaS) models** (e.g., AWS Lambda, Google Cloud Functions). These environments are designed for **on-demand scalability and distributed execution**. In contrast, the chatbot in this study operates in a **local server environment**, leading to several key architectural differences:

- **Computational Environment:** - Local deployment runs entirely on **dedicated hardware**, ensuring **predictable execution** with a fixed allocation of CPU, memory, and GPU resources. - Cloud/FaaS-based systems distribute computations dynamically, allocating resources as needed across a shared infrastructure.
- **Scaling Mechanism:** - Kubernetes' **Horizontal Pod Autoscaler (HPA)** adjusts the number of chatbot instances based on real-time CPU and memory usage. - Cloud-based solutions typically use **event-driven autoscaling**, where resources are allocated dynamically based on incoming requests.
- **Network and Data Flow:** - All chatbot interactions and model inferences occur **locally**, avoiding the need to transmit user data over external networks. - Cloud

deployments require **external API calls**, where requests are processed by remote servers before returning responses.

- **Resource Constraints:** - The local server operates within **fixed hardware limits**, meaning system capacity must be estimated and provisioned in advance. - Cloud/-FaaS environments scale **elastically**, allowing infrastructure to expand or contract based on real-time demand.

- **Execution Consistency:** - Local execution ensures **consistent runtime performance**, unaffected by multi-tenant resource sharing or external service availability. - Cloud-based models can exhibit performance fluctuations due to **network latency, virtual machine scheduling, and API rate limits**.

These differences influence how the chatbot manages workloads and optimizes computational resources. The local setup ensures a stable and controlled execution environment, whereas cloud-based and FaaS models emphasize dynamic scaling and distributed execution.

The decision to use a local deployment environment was intentional and grounded in the goals of this study. A local setup allowed for complete control over hardware specifications, system load, and resource metrics, minimizing external variables such as network latency, noisy neighbors, or unpredictable auto-scaler behaviors often found in public cloud environments. This control ensured reproducibility and consistency during experimentation, making it possible to isolate the effects of resource configurations on Service Level Objectives (SLOs).

Nonetheless, it is important to acknowledge that deploying the same chatbot in a cloud environment could lead to different outcomes. Cloud-based systems benefit from elastic scalability and geographic distribution, potentially improving responsiveness and fault tolerance. However, they also introduce cost variability, hidden latency, and reliance on shared infrastructure, which may obscure the direct relationship between resource limits and performance.

By starting with a controlled local setup, this study establishes a clear baseline for understanding predictive resource management. Future work can extend this framework to cloud deployments, adapting the methodology to account for external variability while building on the validated models and analysis presented here.

5.1.2 Server Hardware Specifications

The chatbot was deployed on Kubernetes on a server with the following specifications:

Component	Specification
CPU	2x Intel Xeon Gold 6338, 32C/64T, 2.0 GHz, 48MB, 205W
Memory	512GB RDIMM, 3200MT/s, Dual Rank
GPU	4x NVIDIA Ampere A40, 300W, 48GB

Table 5.2: Server Hardware Specifications Details

5.1.3 Experiment Constant Parameters

All experiments shared a set of common configurations:

Parameter	Value
Duration	10 minutes
Minimum Number of Replicas	1
Starting Number of Replicas	1

Table 5.3: Experiment Constant Parameters

5.1.4 Experiment Variables

The experiments systematically varied the following key variables:

Variable	**Details**
Max Number of Replicas	2 to 10
Resource Thresholds	Memory: 250, 500, 750 MB; CPU: 5%, 7.5%, 10%
Number of Requests	25, 50, 75

Table 5.4: Key Experiment Variables

The variables presented in Table 5.4 were chosen to assess the chatbot's scalability and performance across various conditions.

5.2 Data Collection

We conducted experiments using various combinations of variables outlined in Table 5.4. For each experiment, we run the experiment and recorded the logs by setting a 'Max Number of Replicas', choosing a 'Resource Threshold' (either Memory or CPU) with one of the corresponding threshold values, and determining a 'Number of Requests' to represent the incoming user request volume.

This experimental matrix creates 162 unique configurations. To mitigate the variation in individual test runs, we conducted five iterations for each configuration. In Table 5.5 a subset of the data from one such experiment is presented.

ID	Completed?	Prompt Sent Time	Response Received Time	Response Time	# of Replicas	Memory Usage	CPU Usage	Prompt	Response
0	Yes	57:12.6	57:22.2	9.6s	1	184.95	5.22%	Who is Harry Potter?	...
1	Yes	57:24.6	57:34.3	9.702016s	1	233.43	8.26%	What is Hogwarts?	...
2	Yes	57:36.6	57:46.4	9.814034s	2	261	8.60%	Who is Ron Weasley?	...
3	Yes	57:48.6	57:58.5	9.93456s	2	239.91	8.08%	Can you describe Hermione Granger?	...
4	Yes	58:00.6	58:10.7	10.062147s	2	239.77	4.48%	What are the four houses of Hogwarts?	...

Table 5.5: Sample raw data from chatbot experiments, featuring a memory threshold of 250 MB

In Table 5.5, we present the first 5 requests sent during our experiments. The first column, 'ID', identifies each request. The 'Completed?' column indicates whether the chatbot successfully responded to each request; it displays 'No' for responses that resulted in an error or were empty. 'Prompt Sent Time' records the exact time each request was sent, while 'Response Time' records the time when responses were received (this is marked as 'NaN' for failed requests). The 'Response Time' column calculates the duration between the 'Prompt Sent Time' and the 'Response Received Time' (again, 'NaN' for failed requests). The '# of replicas' column shows the chatbot's replica count at the request's send time. 'Memory Usage' and 'CPU Usage' columns track the chatbot's memory and CPU utilization, respectively, at the moment each request was sent. The 'Prompt' column displays the specific question sent in each request. Given that the chatbot is trained on the text of Harry Potter books, we designed 50 unique Harry Potter-related questions to ensure diversity in the requests for each experiment. The 'Response' column captures the chatbot's reply to each prompt. Due to space constraints, the responses are not included.

After completing the experiments, we gathered data from more than 800 log files generated during the experiments. From this data collection, we created what we refer to as

a 'summary dataset.' This dataset is organized such that each row corresponds to one of the configurations experimented. Given that we conducted five experimental runs for each configuration, the values presented in the dataset are the averaged results of these runs. In the following, we will provide a detailed explanation of the summary dataset and how it was constructed:

In Table 5.6, we provided first 5 rows of the summary dataset we created. Each row represents an average results we created for the 5 experiments we have done with each configuration. The first column, 'ID', identifies each configurations. The column 'Max Replicas' shows the maximum number of replicas that were chatbot allowed to create in Kubernetes of itself. The column 'Threshold Param' is the parameter that were used to measure if we need to add more replicas. In this study, we used Memory and CPU. The column 'Threshold val' is the value of the threshold parameter that is monitored and if the system exceeds this Threshold val, if allowed, another replicas is going to be created. We assume each user sends a single request per experiment, meaning the 'Requests' column reflects the number of users as well. The columns so far define the configuration of the experiment. The columns from now on are the results. The column 'Fail Rate(%)' is the average percentage over 5 experiments of the requests that has been responded empty results. The column 'Error Rate(%)' is the average percentage over 5 experiments of the requests that has been responded by Error. The column 'Resp $<$ 5s(%)' is the average percentage over 5 experiments of the requests that has been completed(not error or empty response) and responded in less than 5 seconds. The column 'Resp $<$ 15s(%)' is the average percentage over 5 experiments of the requests that has been completed(not error or empty response) and responded in less than 15 seconds. The column 'Resp $<$ 20s(%)' is the

ID	Max Replicas	Threshold Param	Threshold val	Requests	Fail Rate(%)	Err Rate(%)	Resp <5s(%)	Resp <10s(%)	Resp <15s(%)	Resp <20s(%)}
0	2	Memory	250	25	11.72	6.354	0.109	1.565	46.618	61.8
1	2	Memory	250	50	13.224	5.465	0.3	1.411	42.552	64.568
2	2	Memory	250	75	12.72	4.9	0.044	1.639	41.194	68.865
3	2	Memory	500	25	11.195	4.746	0.325	1.39	45.994	69.841
4	2	Memory	500	50	13.122	5.983	0.107	1.399	43.748	71.29
5	2	Memory	500	75	12.782	4.219	0.116	1.643	43.764	64.785
6	2	Memory	750	25	11.773	3.853	0.012	1.631	46.733	65.048
7	2	Memory	750	50	12.888	5.778	0.06	1.474	47.539	70.625
8	2	Memory	750	75	13.204	3.862	0.014	1.365	43.186	63.054
9	2	CPU	5	25	12.398	5.928	0.113	1.617	42.293	69.581

Table 5.6: First rows of the summary dataset

average percentage over 5 experiments of the requests that has been completed(not error or empty response) and responded in less than 20 seconds.

After completing our data collection, we have compiled a dataset that enables us to address our research questions.

Measurement and Role of Metrics: Throughout the experiments, we systematically measured several key metrics for each chatbot request: **response time**, **failure status**, **CPU usage**, **memory usage**, and the **number of replicas** active at the time of the request. These metrics were gathered using Kubernetes resource monitoring and log collection tools, recorded at the moment each request was processed.

The **response time** was measured as the interval between the prompt being sent and the chatbot's reply being received. A request was marked as **failed** if it resulted in an empty or erroneous response. CPU and memory usage were captured from Kubernetes pod metrics, and represent the resource consumption at the time of each request.

These raw measurements fed directly into the construction of our summary dataset. For each experimental configuration, we aggregated results over five repetitions to compute average metrics such as *Fail Rate(%)*, *Err Rate(%)*, and *Resp < Xs(%)*. These aggregated

values serve as target variables or performance indicators in our decision tree regressors for each research question (RQ1–RQ3). Additionally, individual raw file measurements were used to compute a **Combined Score** for RQ2, integrating performance and resource efficiency to evaluate each configuration holistically.

By tightly coupling raw operational metrics to high-level performance targets, our analysis ensures that model predictions and statistical conclusions reflect actual system behavior under varying load and resource conditions.

Model Selection: We choose decision tree regressor for models in this study. This model is selected for its transparent decision-making process, allowing easy interpretation and traceability of its decisions—key factors for model validation and establishing trust. Decision trees are well-suited for managing non-linear relationships between features and targets, making them ideal for complex datasets where linear models might underperform. Additionally, decision trees require less preprocessing, efficiently handle both numerical and categorical data, and are robust against outliers that could otherwise skew the performance of models like linear regression.

With this dataset and the decision tree regressor model, we are prepared to explore each research question.

5.3 RQ1: Resource Allocation for Specific Traffic Volumes

Before optimizing our predictive models, we established baseline models for both RQ1 (resource allocation) and RQ3 (customer support capacity). These initial models used default decision tree parameters and provided a reference for performance improvements. The baseline models showed that while they captured broad trends in response time and failure rates, they lacked predictive precision, requiring hyperparameter tuning. For RQ1, we use a decision tree regressor to predict two target variables: (1) the percentage of chatbot responses returned within 20 seconds, and (2) the failure rate of chatbot requests. These reflect the two SLOs being evaluated. The input features used by the model are:

- **Threshold Parameter** (either CPU or Memory),
- **Threshold Value** (e.g., 5%, 250MB),
- **Maximum Number of Replicas**.

Each experiment configuration is defined by these three input values, and the model learns to predict the corresponding SLO metrics based on them.

To address Research Question 1, we individually train a regressor model for each SLO to predict the respective target values. Since the target variables (response time and failure rate) are continuous, we use a regression approach rather than classification. Decision trees for regression were chosen to capture nonlinear dependencies between resource allocation and chatbot performance.

5.3.1 Data-Driven Approach to Predicting Responsiveness

Objective: The first SLO focuses on assessing whether the system can handle a specific percentage of requests(X%) within 20 seconds. We aim to predict this SLO's target value(X%), the given parameters are 'Threshold Parameter', 'Threshold Value', and 'Maximum Number of Replicas'.

Dataset Overview: The dataset, represented in Table 5.6, includes the 'Resp $<$ 20s(%)' metric, representing the mean proportion of requests completed within 20 seconds over five experimental runs. This serves as the dependent variable in our analysis, with 'Threshold parameter', 'Threshold value', and 'Maximum Replicas' as independent variables.

Preprocessing: Categorical data within the 'Threshold parameter' was transformed via one-hot encoding to facilitate machine learning analysis, converting categorical inputs into a binary vector representation. Table 5.3.1 demonstrates the one-hot encoding transformation of the 'Threshold parameter', with binary variables 'CPU' and 'Memory'. In this encoding, a '1' under 'CPU' indicates the threshold parameter is set to CPU, whereas a '1' under 'Memory' shows the parameter is set to Memory.

CPU threshold	**Memory threshold**	**...**	**Resp $<$ 20s(%)**
1	0	...	...
0	1	...	...
...	...	...	...

Table 5.7: One-Hot Encoding of 'Threshold Parameter'

Model Training: For the model training process, we first split the dataset into 80% for training and 20% for testing. Without initially limiting the depth or complexity of

the decision tree, we establish a baseline performance. This approach allows for a clear starting point for further optimization efforts. The decision tree obtained from our analysis is considerably large, making it impractical to visualize in its entirety. Therefore, Figure 5.1 provides a visual representation of only a fragment of the baseline model—specifically the root and its immediate branches.

Hyperparameter Optimization: Through grid search and cross-validation, we fine-tuned the model by adjusting parameters `max_depth`, `min_samples_split`, and `min_samples_leaf` to improve performance and prevent overfitting.

The step-by-step procedure used that was explained above for preprocess the data, train the initial model, and then tune its hyperparameters is described in Algorithm 5.1.

The decision tree model using the optimized hyperparameters shown in Table 5.8, is visualized in Figure 5.2.

5.3.2 Data-Driven Approach to Predicting Failure Rates

Objective: The second SLO targets maintaining the system's failure rate under a defined threshold (Y%). We plan to predict this target using 'Threshold Parameter', 'Threshold Value', and 'Maximum Number of Replicas'.

Dataset Overview: Table 5.6 presents the dataset including 'Fail Rate(%)' and 'Err Rate(%)', which are the average percentages of failed or erroneous requests across five experiments. These metrics serve as labels, while 'Threshold parameter', 'Threshold value', and 'Maximum Replicas' act as independent variables.

Preprocessing: We selected a decision tree regressor for our model. The data was preprocessed using one-hot encoding to convert categorical variables into binary vectors, as shown in Table 5.3.1, which is also utilized for the first SLO.

Model Training: The dataset was partitioned into two subsets: 80% for training and 20% for testing. We initialized with an unrestricted decision tree to set a benchmark for performance, facilitating subsequent refinements. Figure 5.3 shows the initial decision tree, serving as our preliminary model.

Hyperparameter Tuning: We employed grid search and cross-validation to refine `max_depth`, `min_samples_split`, and `min_samples_leaf`, enhancing model efficacy and mitigating overfitting. Table 5.8 details the adjusted hyperparameters.

Parameter	Optimized Value
`max_depth`	4
`min_samples_leaf`	1
`min_samples_split`	17

Table 5.8: Optimized hyperparameters for the decision tree model in RQ1-SLO2

Figure 5.4 shows the decision tree model post-optimization.

After training the model following the steps outlined in Algorithm 5.2, we tested it by using test data to see if it could accurately predict the SLOs target values. Since our data covered limited configurations, we also tried the model with different configurations that were not part of our original experiments. This allowed us to determine whether the model could accurately predict beyond the training parameter range. We have detailed the predicted values in the "Results" chapter.

5.4 RQ2: Resource Requirements for Desired SLOs

Objective: This section determines the minimum resources—CPU, memory, and number of replicas—required to achieve our SLOs. Fundamentally, we have two SLOs with set targets, and our aim is to identify the least resources necessary for the chatbot to operate within these targets.

Preprocessing: In this section, we use both the summary dataset, as illustrated in Table 5.6, and the original raw files as presented in Table 5.5. Each entry in the summary table represents the average of five raw data files collected during the experiments, which are outlined in Table 5.5

We use one-hot encoding for features and narrow down records to those meeting SLO targets. We select records where the percentage of requests responded to within 20 seconds surpasses the SLO1 target and records with failure rates under the SLO2 threshold. Following this filtration process, one of two outcomes occurs:

- If no records are left after we filter the data, we conclude that we cannot tell whether the chatbot can meet the SLOs with the current resources based on the available data.
- If there are records that pass the filter, this shows some records where the chatbot meets the SLOs. We then look more into these records to understand what resources make this possible. In this section, From now on, we assume that there are some data records.

In our dataset, each summary row combines the results from five different experiments. We start by removing any data records that don't fit our SLO requirements, focusing only on those configurations. Then, we look closely at five separate raw data files linked to each summary dataset row.

To effectively evaluate the performance and resource efficiency of the chatbot under these conditions, we propose the use of a metric known as the "Combined Score."

Combined Score: Combined Score metric is calculated for each raw data file, providing an evaluation of the chatbot's capabilities. The Combined Score is defined by two components:

- Performance Score: This evaluates how efficiently the chatbot handles requests, focusing on quick responses and minimal failures.
- Resource Efficiency Score: This assesses the chatbot's use of CPU and memory, favoring configurations that achieve more with less resource consumption.

For each raw data file in a data record, we calculate a Combined Score to assess overall chatbot efficiency and performance:

$$\textit{Combined Score} = (1 - \lambda) \cdot \textit{Performance Score} + \lambda \cdot \textit{Resource Efficiency} \tag{5.1}$$

where $\lambda = 0.5$ is used to balance the importance of resource efficiency against performance.

We assess each raw file with two primary scores: the Performance Score and the Resource Efficiency score. The Performance Score rewards configurations that quickly respond

to requests (responses in less than 20 seconds) and have a low failure rate. On the other hand, the Resource Efficiency score penalizes configurations that require high CPU and memory usage, encouraging more resource-efficient configurations.

Performance Score: The Performance Score is a metric designed to measure the chatbot's effectiveness in processing requests. It is calculated as follows:

$$\textit{Performance Score} = (100 - \text{avg}(\textit{Fail Rate(\%)})) + \text{avg}(\textit{Resp} < \textit{20s (\%)}) \tag{5.2}$$

This score combines the average failure rate and the average response time under 20 seconds for each raw file of experiment. This score aggregates the average failure rate and the average percentage of responses within 20 seconds across the raw data files.

Resource Efficiency: The Resource Efficiency score evaluates the chatbot's utilization of CPU and memory resources:

$$\textit{Resource Efficiency} = (100 - \text{avg}(\textit{CPU Usage})) + (100 - \text{avg}(\textit{Normalized Memory Usage})) \tag{5.3}$$

Here, CPU utilization is percentage so there is no need for normalization but memory usage is normalized to a 0-100 scale, with higher scores indicating more efficient resource use.

Given multiple data records, each with 5 raw data files, we calculated Normalized Memory Usage for each raw file based on a maximum expected usage of 750 MB:

$$\textit{Normalized Memory Usage} = \left(\frac{\textit{Memory Usage}}{750}\right) \times 100 \tag{5.4}$$

Analytical Use of the Combined Score: After calculating the Combined Score for each raw data file within a configuration, we obtain a list of five values per configuration (one per experimental run). These lists form the basis of our statistical comparison. Specifically, we use the Combined Score as the dependent variable in a one-way ANOVA to test whether different configurations result in significantly different overall performance.

The ANOVA test treats each configuration as a separate group and compares the mean Combined Scores across these groups. This allows us to identify whether any configuration significantly outperforms the others in terms of balancing performance (low failure rate and fast responses) and resource efficiency (low CPU/memory usage). The test assumes independence across experimental runs, approximate normality of scores, and homogeneity of variances—all of which were verified, as discussed earlier.

Once ANOVA indicates significant differences (p-value < 0.05), we proceed to rank configurations based on their mean Combined Scores, prioritizing lower means (indicating higher efficiency and performance) and inspecting confidence intervals to ensure robustness in our ranking.

Note:

- *Fail Rate(%)* and *Resp < 20s (%)* metrics are based on the chatbot's performance goals.
- *CPU Usage* and *Normalized Memory Usage* represent the usage percentages of CPU and memory, respectively, adjusted to a common scale for comparison.
- The calculation of the Combined Score for each raw data file, as outlined in Algo-

rithm 5.3, allows us to individually assess the chatbot's performance and resource efficiency across different operational scenarios.

ID	Max Replicas	Threshold Param	Threshold val	Requests	Fail Rate(%)	Err Rate(%)	Resp <20s(%)	Combined Scores
0	2	Memory	250	25	11.72	6.354	61.8	[45.16, 45.2, 45.1, 45.05, 45.25]
1	2	Memory	250	50	13.224	5.465	64.568	[36.29, 36.3, 36.25, 36.2, 36.35]
9	2	CPU	5	25	12.398	5.928	69.581	[38.52, 38.5, 38.55, 38.6, 38.45]

Table 5.9: Summary Dataset with Combined Scores

As shown in Table 5.9, we present a sample of the summary dataset now enriched with the computed combined scores for each configuration. This table illustrates how the combined scores provide a direct comparison across different experimental runs. Our next objective is to discern the most reliable configuration. To achieve this, we will employ statistical testing methods, specifically the t-test and ANOVA (Analysis of Variance), to compare and rank the configurations systematically, as outlined in Algorithm 5.4.

Validation of ANOVA Assumptions: Before proceeding with the ANOVA, we verified that our dataset satisfies its statistical assumptions. First, **independence** was ensured by designing the experiments so that each configuration's outcome was unaffected by the others; each configuration was run in isolation with no shared state or temporal overlap. Second, for **normality**, we conducted a Shapiro–Wilk test on the combined scores across all configurations, which showed that the residuals approximately follow a normal distribution ($p > 0.05$). Lastly, we assessed the **homogeneity of variances** using Levene's test, confirming that the variance across different configurations did not significantly differ ($p > 0.05$). These results validate the use of ANOVA for comparing the configurations' performance based on the Combined Score metric.

To evaluate the impact of resource configurations on SLO compliance, we define the key independent and dependent variables used in our analysis.

- **Independent Variables:**
  - Memory Allocation (MB): Limits assigned per container.
  - CPU Threshold (%): Usage level that triggers scaling.
  - Max Number of Replicas: Upper bound on auto-scaling.
- **Dependent Variables:**
  - Response Time (s): Time from request to chatbot reply.
  - Failure Rate (%): Proportion of requests resulting in errors.
  - Combined Score: Weighted metric balancing response time, failure rate, and

These variables define the experimental conditions and provide the foundation for statistical analysis using ANOVA which we do with the following steps:

1. **Calculate Averages:** For each configuration (row), we compute the average combined score. This average tells us the typical performance level of each configuration.
2. **Statistical Analysis:** We use an ANOVA test, which is a type of statistical test that checks if there's a significant difference between the averages of a group of numbers. ANOVA is particularly well-suited for this analysis because we are comparing performance across more than two resource configurations. Unlike a t-test, which

is limited to comparing two groups, ANOVA allows us to evaluate whether statistically significant differences exist among multiple groups simultaneously. This is essential for identifying which configurations perform better in terms of the combined score. Each configuration group in our dataset contains five independent measurements of the Combined Score, corresponding to five experimental repetitions for each resource setup. Since each has multiple independent observations, ANOVA is the appropriate statistical test. It allows us to assess whether the differences in group means—reflecting performance and efficiency—are statistically significant across the full set of configurations. Using a t-test would be insufficient in this context, as it only supports pairwise comparisons. It tells us whether any of our configurations perform significantly better or worse than the others.

3. **Check Significance:** After running the ANOVA test, we look at the p-value it gives us. If this p-value is less than 0.05, it means that at least one configuration is significantly different from the others.

4. **Rank Configurations:** We list the configurations in order based on their average combined scores, from lowest to highest. The one with the lowest average is the best because it indicates the fastest or most efficient performance.

5. **Determine Confidence:** We also calculate what's called a confidence interval for each average score. This interval is a range that we are fairly sure contains the true average performance of the configuration. It helps us understand how precise our average score is.

6. **Final Ranking:** Using the ANOVA results and the confidence intervals, we final-

ize the rankings. A configuration with a significantly lower average and a narrow confidence interval is considered better.

Conducting the ANOVA test detailed in the Algorithm 5.4 allows us to rank the configurations and select the top-ranked as our answer. The rankings and detailed results are presented in the 'Results' chapter.

5.5 RQ3: Customer Support Capacity with Given Resources and SLOs

Objective: We want to predict the maximum number of users that can be supported with using 'Threshold Parameter', 'Threshold Value', and 'Maximum Number of Replicas' and SLO requirements. For RQ3, the decision tree regressor is used to predict the maximum number of supported chatbot users while meeting both SLOs. The target variable is the number of requests successfully handled. The input features are identical to those used in RQ1:

- **Threshold Parameter** (CPU or Memory),
- **Threshold Value**,
- **Maximum Number of Replicas**.

These features represent the autoscaler configuration, and the model is trained to estimate the user capacity the system can sustain under those settings while complying with the SLOs.

Dataset Overview: Table 5.6 presents the dataset, where the 'requests' metric shows the number of requests for each configuration. We want to predict the dependent variable using three factors: 'Threshold parameter', 'Threshold value', and 'Maximum Replicas'.

Model Training and Hyperparameter Optimization: The dataset was split into 80% for training and 20% for testing. During the training process, grid search and cross-validation were employed to fine-tune the decision tree parameters, `max_depth`, `min_samples_split`, and `min_samples_leaf`. This approach aimed at optimizing model performance and preventing overfitting. The optimized hyperparameters obtained through this process are summarized in Table 6.4.1.

Parameter	**Value**
`max_depth`	5
`min_samples_leaf`	2
`min_samples_split`	15

Table 5.10: Optimized hyperparameters for the decision tree model in RQ3

The decision tree model, utilizing the optimized hyperparameters outlined in Table 6.4.1, is shown in Figure 5.5.

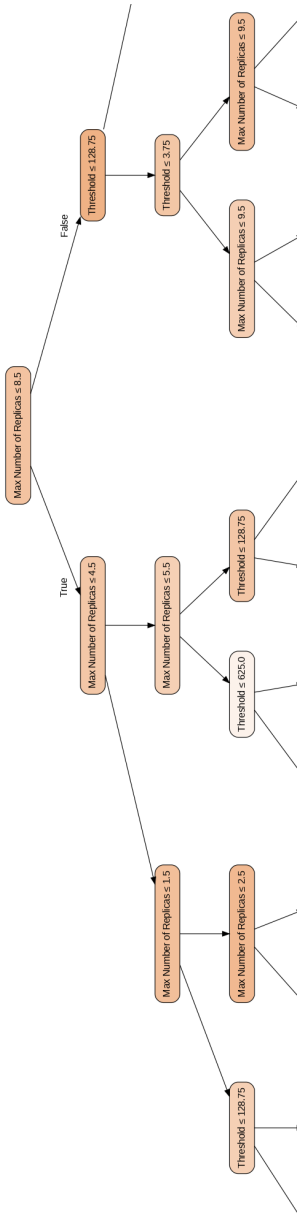

Figure 5.1: Partial view of the initial decision tree model for RQ1-SLO1, illustrating the root and its immediate branches.

Algorithm 5.1 Optimized Decision Tree for Responsiveness

Objective: Predict the SLO target value (X%) based on 'Threshold Parameter', 'Threshold Value', and 'Maximum Number of Replicas'.
procedure PREPROCESS
    One-hot encode 'Threshold Parameter'.
    Transform Raw Data to Summary Dataset:
    Aggregate raw data by computing mean values for each configuration over multiple runs to form the summary dataset.
    The summary dataset includes averages of 'Resp $<$ 20s(%)', 'Fail Rate(%)', 'CPU Usage', 'Memory Usage', etc.
    Normalize and scale relevant features to ensure consistent scales across the dataset.
    Split dataset: 80% for training, 20% for testing.
end procedure
procedure TRAININITIALMODEL
    Train an unrestricted decision tree to establish a baseline.
end procedure
procedure TUNEHYPERPARAMETERS
    Use grid search with cross-validation:
    for `max_depth` in [3, ... , 10] **do**
        for `min_samples_leaf` in [1, ... , 4] **do**
            for `min_samples_split` in [2, ... , 20] **do**
                Train model on the training set.
                Evaluate model on the validation set.
                Record model performance.
            end for
        end for
    end for
    Select the model with the best performance.
end procedure
procedure EVALUATE
    Test the optimized model on the test set.
end procedure

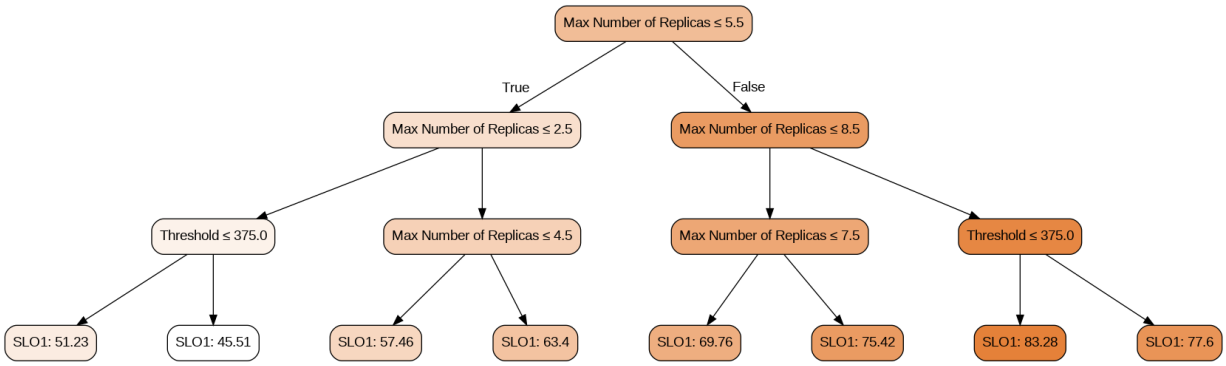

Figure 5.2: Optimized Decision Tree Model in RQ1-SLO1

Algorithm 5.2 Optimized Decision Tree for Failure Rate Prediction

Objective: Predict system's failure rate below a specific threshold (Y%).
procedure PREPROCESS
    Apply one-hot encoding to 'Threshold Parameter'.
    Split dataset: 80% training, 20% testing.
end procedure
procedure TRAININITIALMODEL
    Initialize decision tree without restrictions to establish baseline.
end procedure
procedure TUNEHYPERPARAMETERS
    Use grid search with cross-validation:
    for `max_depth` in [3, ... , 10] **do**
        for `min_samples_leaf` in [1, ... , 4] **do**
            for `min_samples_split` in [2, ... , 20] **do**
                Train model on training set.
                Evaluate model on validation set.
                Record model performance.
            end for
        end for
    end for
    Select model with best performance based on validation.
end procedure
procedure EVALUATE
    Assess optimized model on test set.
end procedure

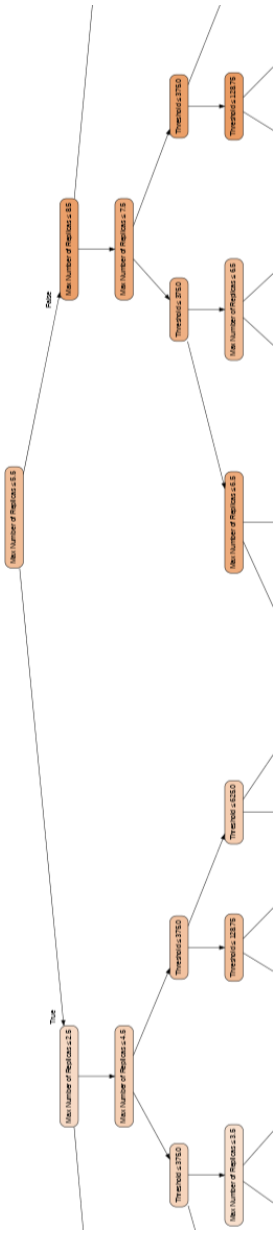

Figure 5.3: Baseline Decision Tree Model in RQ1-SLO2

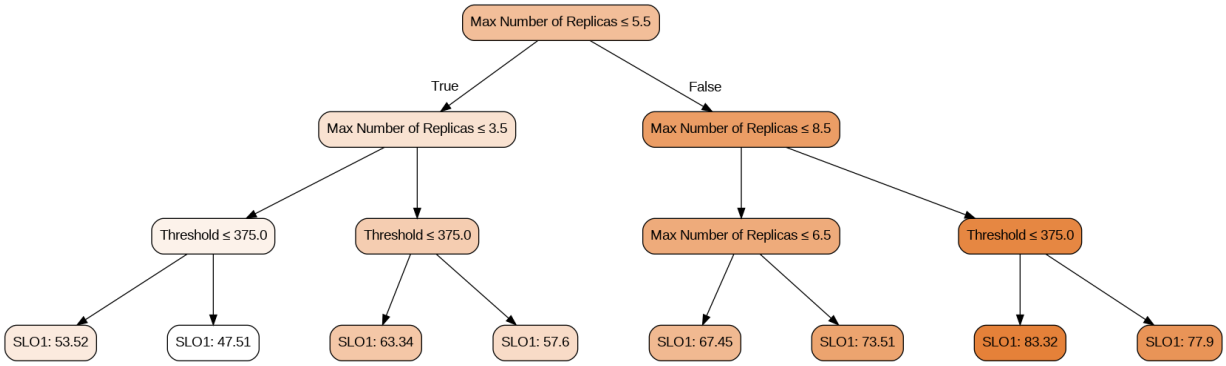

Figure 5.4: Optimized Decision Tree Model in RQ1-SLO2

Algorithm 5.3 Calculate Combined Score for Summary Dataset

Input: Summary dataset
Output: Summary dataset with new column of combined scores for raw data files
procedure ProcessSummaryDataset
    for each row in summary dataset **do**
        Remove data records not meeting SLO requirements
        for each of the 5 raw data files linked to the row **do**
            Calculate the *combined score* reflecting:
                - Performance Score (failure rate, resp $<$ 20s)
                - Resource efficiency (CPU and memory usage)
        end for
        Add a new column to the summary dataset
            - List of 5 combined scores for each raw data file
    end for
end procedure

Algorithm 5.4 Rank Configurations Using ANOVA

Input: Summary dataset with combined scores for each configuration
Output: Ranked list of configurations based on reliability
procedure RANKCONFIGURATIONS
    for each configuration in the dataset **do**
        Calculate the mean of the combined scores
    end for
    Perform ANOVA test across all configurations
    if ANOVA p-value $<$ significance level(0.05) **then**
        There is a statistically significant difference in means
    end if
    for each configuration **do**
        Calculate 95% confidence interval for the mean combined score
    end for
    Order configurations by increasing mean combined score
    Rank configurations based on order and non-overlapping confidence intervals
    return Ranked list of configurations
end procedure

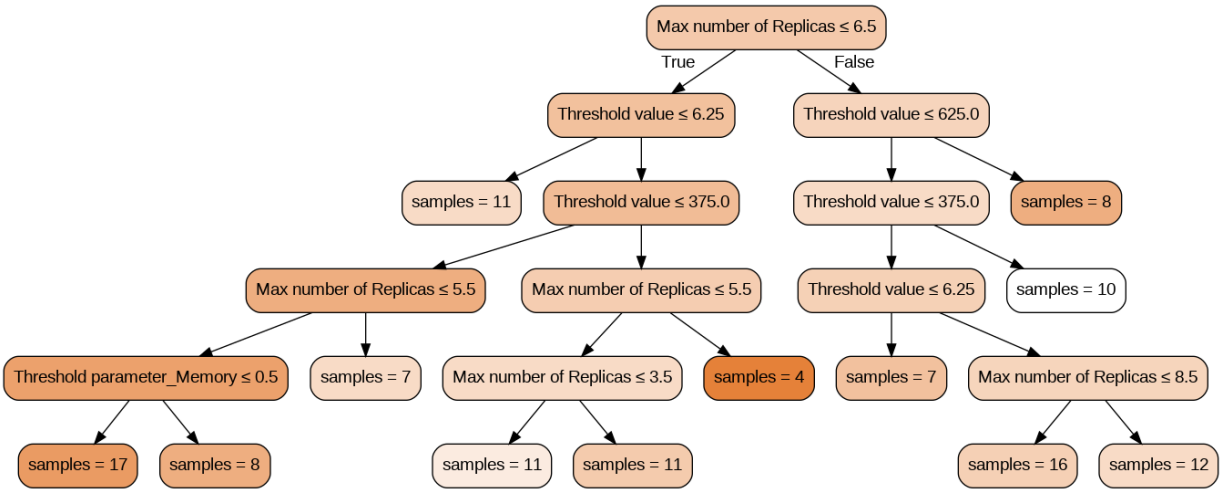

Figure 5.5: Optimized hyperparameters for the decision tree model in RQ3

Chapter 6

Empirical Study Results

6.1 Overview

This chapter presents our experimental findings. We evaluated two main SLOs related to Response Time and Failure Rate. Our research addressed three questions: resource allocation, minimum resource requirements, and user support capacity. The following subsections detail the results and our discussion of these findings.

6.2 RQ1: Predicting SLO Target Achievement Based on Configuration Parameters

This section evaluates our chatbot's performance in predicting SLOs based on three key variables: Threshold Parameter, Threshold Value, and Maximum Number of Replicas.

We built predictive models to determine if these parameters could accurately predict SLO achievement, using experimental data for validation. Below, we describe the methods employed in this evaluation.

6.2.1 Methodology

To create models for each SLO target, we first trained the models and then tested them on a separate dataset to measure their predictive accuracy. The evaluation metrics we used are:

- **Mean Absolute Error (MAE)**: Average prediction error.
- **Root Mean Square Error (RMSE)**: Penalizes larger errors more.
- **R-squared** (R^2): Proportion of variance explained by the model.

6.2.2 Evaluation of Initial and Optimized Models for SLO1

This part evaluates the initial and optimized performance of the model predicting the Response Time SLO. We compare performance metrics before and after optimization, highlighting key influencing factors.

Initial Model Performance:

The initial performance metrics for our model targeting SLO1 are shown in Figure 6.1. The MAE is 14.2%, indicating that the model's predictions are, on average, 14.2% off from the actual percentage of requests processed within 20 seconds. This means that for

every 100 requests, the model's estimate could deviate by about 14 from the actual number of requests processed within the target time. This deviation highlights the necessity for further model refinement to improve prediction accuracy.

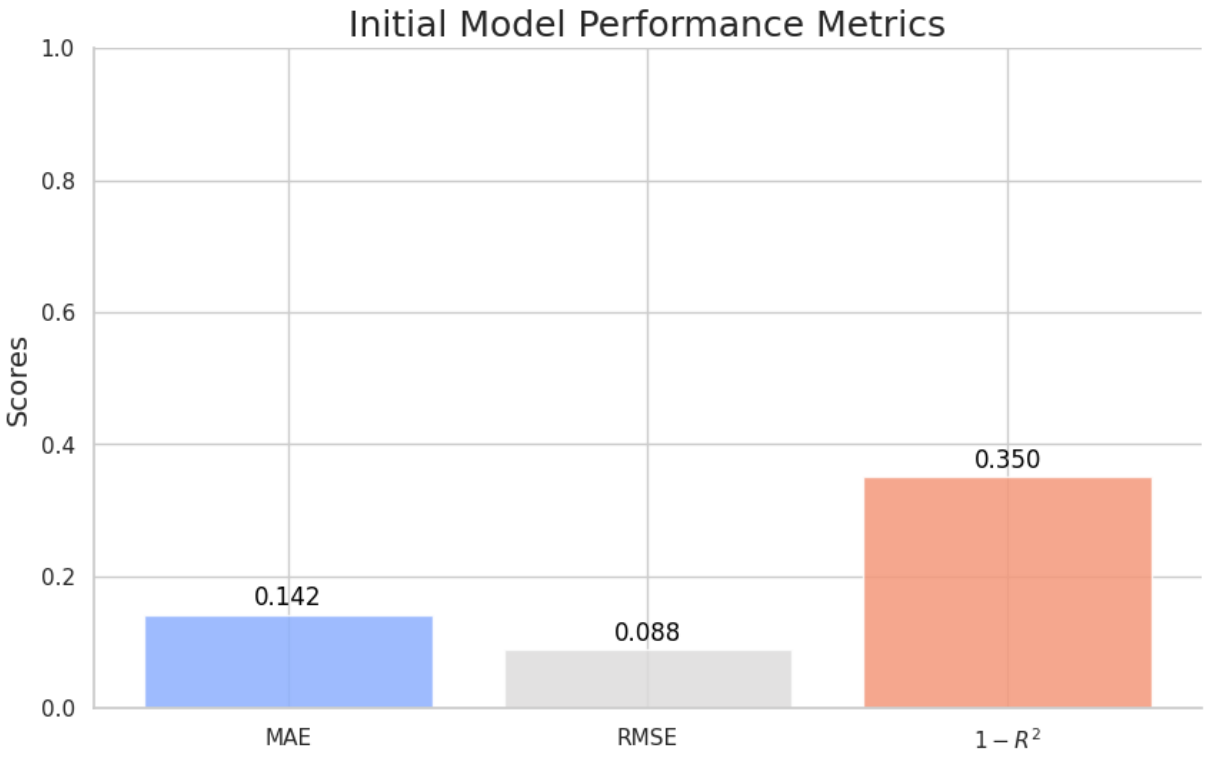

Figure 6.1: Initial model error metrics.

The RMSE is 8.8%, indicating that some predictions have larger errors. These errors suggest that the model might significantly underestimate or overestimate the required resources, particularly during high-traffic periods, potentially affecting response times and user experience.

With an R^2 value of 0.65, the model explains 65% of the variability in processing

requests within the desired time frame. This is a solid starting point, but it also indicates that 35% of the variability remains unexplained. Improving the R^2 value is crucial for achieving more predictable and consistent service quality, which directly impacts customer satisfaction and operational efficiency.

Overall, these error metrics guide our next steps in model optimization. Our goal is to reduce MAE and RMSE while increasing R^2, aiming for more accurate predictions that closely align with real-world outcomes. This approach focuses on enhancing the chatbot's responsiveness to improve user experience and operational efficiency.

Optimized Model Performance:

After fine-tuning the model with the hyperparameters listed in Table 5.8, we observed a marked improvement in the performance metrics, as depicted in Figure 6.2. The Mean Absolute Error (MAE) decreased from 14.2% to 8.5%, indicating that the predicted values are now closer to the actual values. The Root Mean Square Error (RMSE) reduced from 8.8% to 6.2%, suggesting a decrease in the impact of large errors or outliers in the predictions. Furthermore, the coefficient of determination, represented by R^2, improved from 0.65 to 0.85, meaning that the optimized model now explains 85% of the variance in response times, compared to 65% previously. This substantial increase in R^2 value signifies that the model is now much more aligned with the underlying data distribution, providing predictions that are both more accurate and reliable. The enhancements in model performance can largely be attributed to the restriction of the decision tree's complexity, preventing overfitting and ensuring that the predictions are not swayed by anomalous data points.

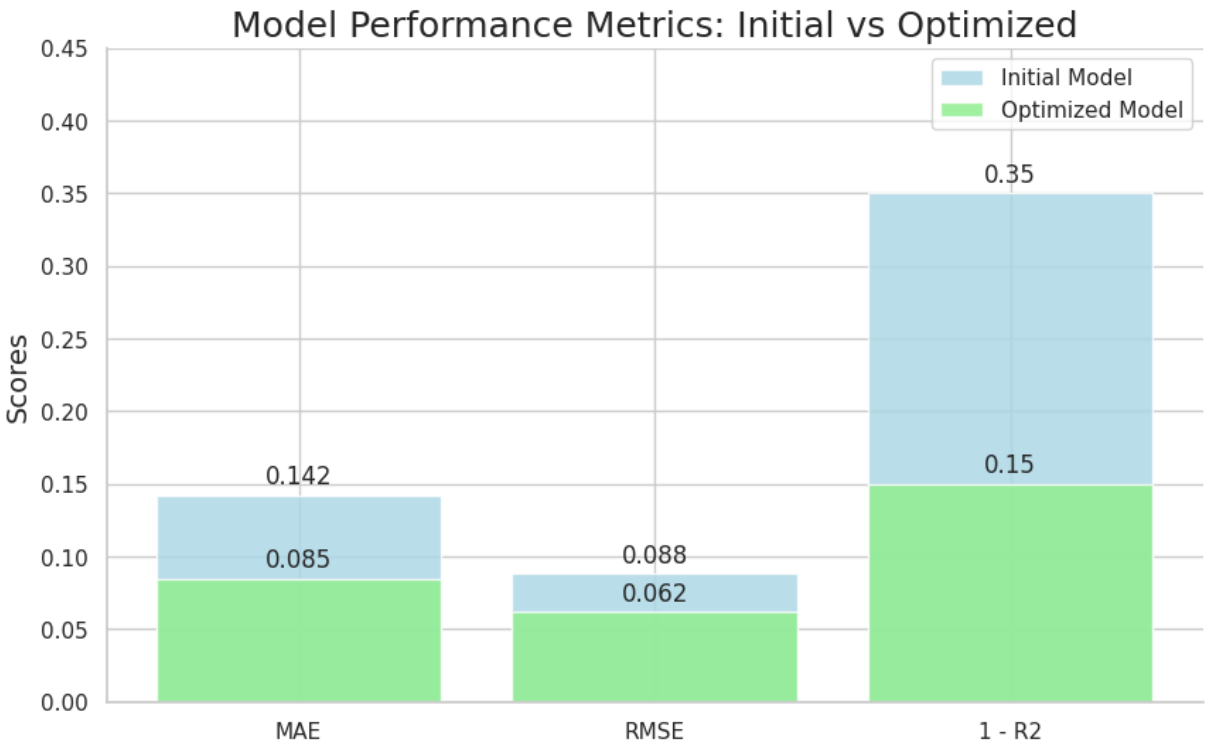

Figure 6.2: Model Performance Metrics Pre- and Post-Optimization for SLO1 Predictions

After cross-validation, we identified the most effective settings for the model. These settings are provided in Table 5.8. With these adjustments, the model's performance for predicting the SLO1 target value improved, as shown by the lower error metrics.

In Figure 6.3, we can see that 'Max number of Replicas' is the most important feature for predicting the SLO1 target value. This finding indicates that the number of replicas is a key determinant in the model's predictions.

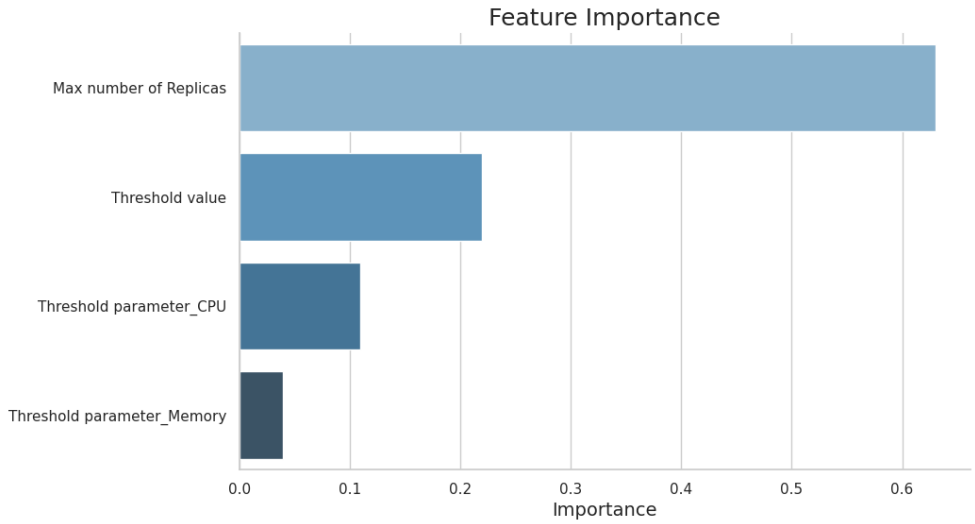

Figure 6.3: Feature Importance of the Optimized Model Predicting the SLO1 Target Value

It is crucial to investigate how changes in the 'Max number of Replicas' affect the SLO1 value. This analysis will confirm the model's reliability and provide a deeper understanding of how system performance scales. The investigation could also reveal the point at which adding more replicas no longer significantly improves the SLO1 target value, which would inform decisions on resource allocation.

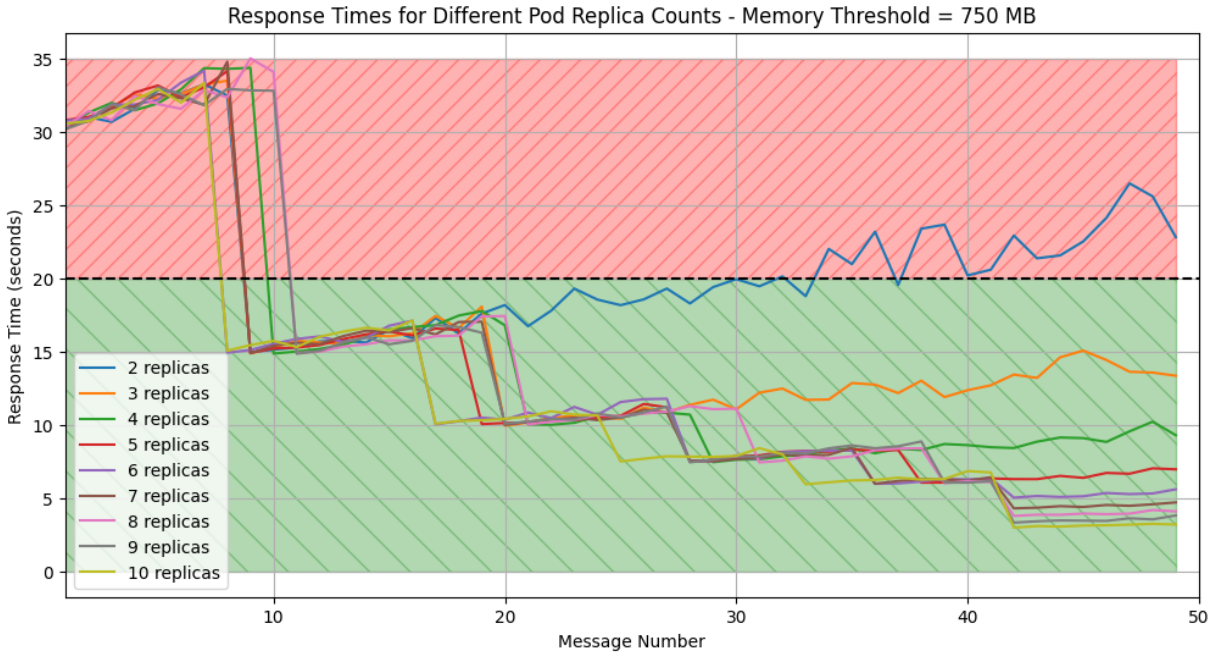

Figure 6.4: Average Response Times for Different Pod Replica Counts with a Memory Threshold of 750MB

Figure 6.4 visualizes the results of our experiments. In these experiments, we set the resource threshold as memory with a value of 750 MB and 50 requests, and varied the maximum number of replicas from 2 to 10. The figure shows how increasing the 'Max number of Replicas' impacts the chatbot's ability to meet the SLO1 target value.

The data reveal a clear trend: more replicas generally lead to faster response times, up to a certain point. For instance, with only two replicas, response times often exceed 20 seconds, indicated in red. In contrast, with four or more replicas, response times consistently stay below the target, shown in green. However, the improvement in response times stalls when increasing from seven to ten replicas, indicating that beyond a certain number, additional replicas do not significantly enhance performance.

The fluctuations seen in 6.4 arise from a combination of Kubernetes scheduling artifacts and the stochastic execution nature of an LLM-based architectures. While the chatbot's backend, operating on deep learning inference pipelines, exhibits non-deterministic execution paths due to dynamic tokenization, layer parallelism, and hardware-level optimizations. Concurrent request processing further amplifies response variations as GPU memory contention and execution order influence latency.

These noises should not be interpreted as anomalies but as intrinsic properties of distributed AI workloads. To ensure robust evaluation, we conducted multiple experiment iterations and applied statistical smoothing techniques, such as averaging and quantile filtering, to reduce short-term noise while preserving meaningful trends.

We also investigated whether CPU or Memory thresholds work better for scaling up the chatbot to keep our response times under 20 seconds. The data in Table 6.2.3 compares the effectiveness of each threshold in achieving this goal.

Max Replicas	**% < 20s with Memory Threshold**	**% < 20s with CPU Threshold**
2	53.5%	54.2%
3	75.6%	76.1%
4	76.4%	76.8%
5	78.2%	78.5%
6	79.0%	79.4%
7	81.3%	81.7%
8	82.6%	83.0%
9	84.1%	84.5%
10	86.7%	87.0%

Table 6.1: Comparison of Request Completion Rates within 20 Seconds

The table shows that the CPU threshold performs slightly better than the Memory

threshold across all numbers of replicas. For instance, with three replicas, 76.1% of requests are completed in under 20 seconds with the CPU threshold, compared to 75.6% with the Memory threshold. This small difference persists as the number of replicas increases. Although the CPU threshold is marginally better for meeting our 20-second response goal (SLO1), we need to consider SLO2 before deciding which threshold is best for RQ1.

6.2.3 Evaluation of Initial and Optimized Models for SLO2

This part analyzes the model's ability to predict the Failure Rate SLO. Initial and optimized model performance metrics are compared, showing the improvements achieved through model refinement and the parameters affecting failure rates. **Initial Model Performance:**

We developed an initial model to estimate the chatbot's failure rate, specifically for SLO2, as detailed in Section 5.3.2 of the *Evaluation* chapter. By testing the initial model using our test sets, we reached to the following error metrics:

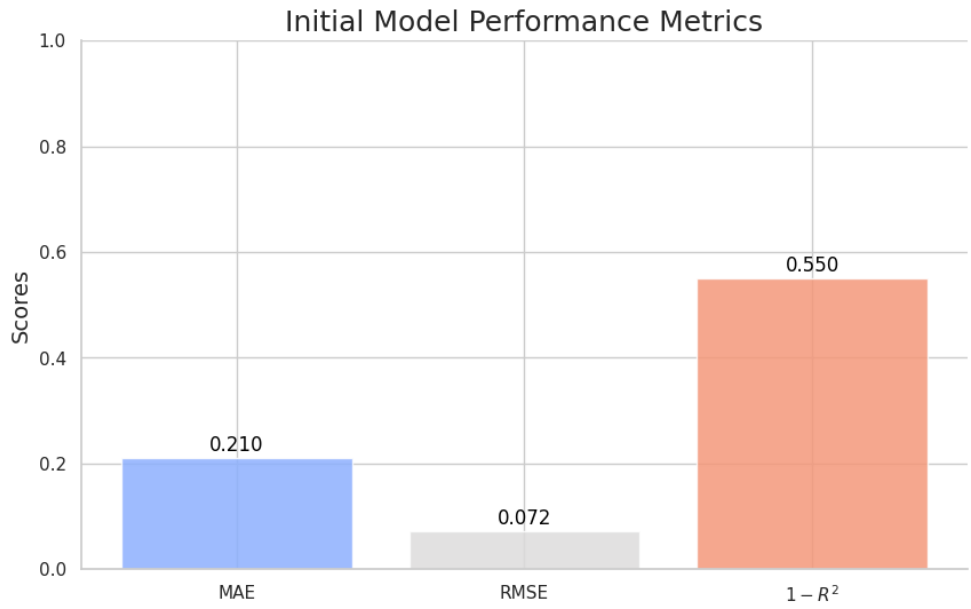

Figure 6.5: Initial model performance metrics.

Figure 6.5 demonstrates that the model is unable to accurately predict the failure rate of the chatbot within its predefined configuration. The Mean Absolute Error (MAE) of 0.210 and the Root Mean Squared Error (RMSE) of 0.170 highlight the model's imprecision. Furthermore, the value of $1 - R^2$, standing at 0.550, indicates that a large portion of the variance in the failure rate is not explained by the model. These metrics demonstrate the necessity of optimizing the model to improve the prediction of the failure rate.

Optimized Model Performance: After cross-validation, we identified the most effective settings for the model. These settings are provided in 5.8. With these adjustments, the model's predictions for SLO2 became more accurate, as shown by better performance numbers.

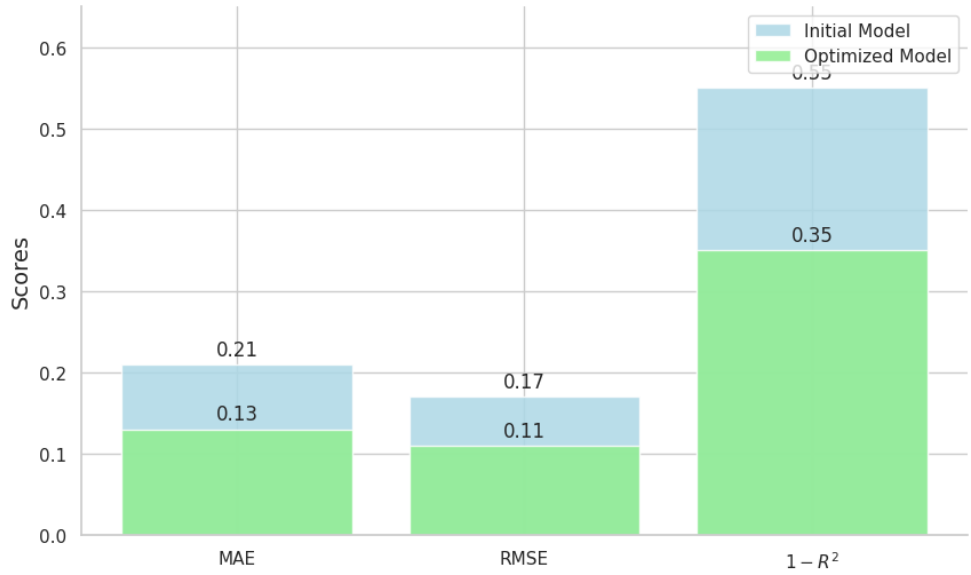

Figure 6.6: Model Performance Metrics Pre- and Post-Optimization for SLO2 Predictions

The bar chart in Figure 6.6 illustrates the performance metrics of our model before and after optimization. Post-optimization, the Mean Absolute Error (MAE) is lower, signifying a closer alignment between the predicted failure rates and the actual observed rates. A reduced Root Mean Square Error (RMSE) indicates fewer large errors in prediction, suggesting a more consistent and reliable model. The improvement in the R^2 value indicates that the optimized model now accounts for a greater proportion of the variability in the failure rates.

Figure 6.7 presents the results of the feature importance analysis. This analysis reveals that the 'Max Number of Replicas' outperforms other features in its predictive power for the SLO2 target value similar to SLO1 target value.

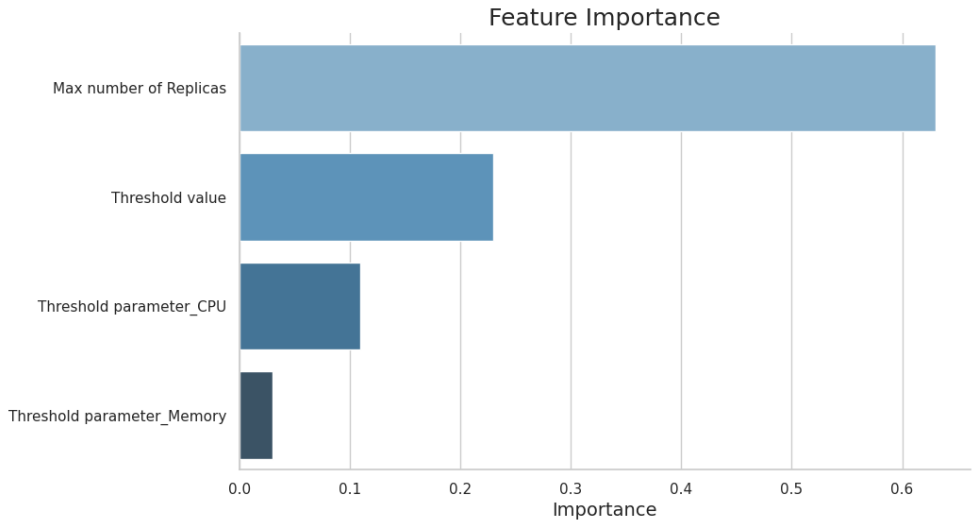

Figure 6.7: Feature Importance for SLO2 Target Value

We want to explore the influence of increasing the 'Max number of Replicas' on meeting the SLO2 target value more effectively.

Now that we know 'Max Number of Replicas' is the most important feature. We would like to know by setting the maximum number of replicas to each number from 2 to 10, how will be the failure rate. The Figure 6.8 illustrates the relationship between 'Max Number of Replicas' and the failure rate value, based on experimental data.

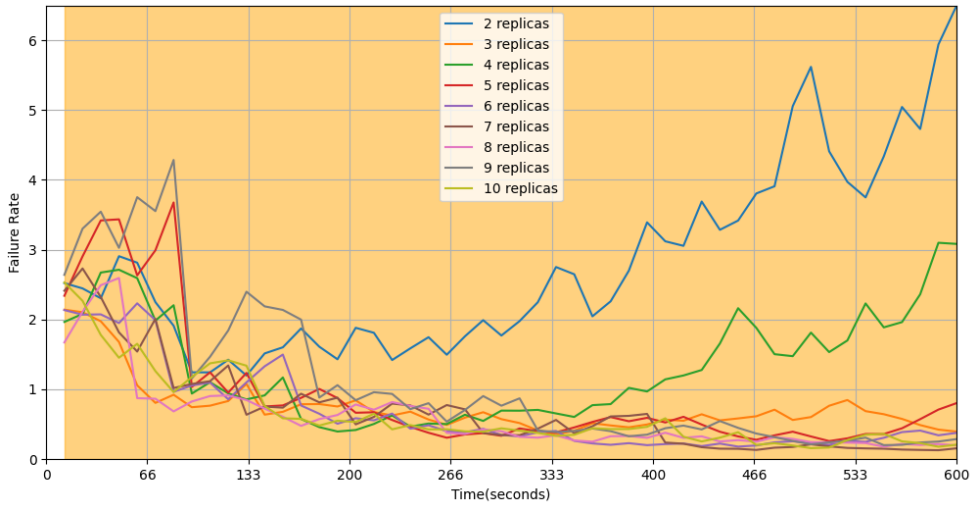

Figure 6.8: Feature Importance for SLO1 Target Value

The Figure 6.8 shows the failure rate over time for systems configured with different maximum numbers of replicas, from 2 to 10. Initially, all configurations perform similarly, but as time progresses, systems with more replicas generally exhibit lower failure rates. The two-replica setup displays frequent high spikes in failure rate, indicating poor performance under load. Mid-range replicas, such as 5 or 6, maintain a lower, steady failure rate for longer, suggesting a more resilient system that uses resources efficiently. Beyond these number, having more replicas does not necessarily reduce the failure rate significantly. In essence, the figure demonstrates a clear trend: adding replicas up to a certain number reduces failure rates Therefore, having more than 6 replicas does not significantly increase system reliability and may just waste resources.

We also examined if CPU or Memory thresholds are more effective at maintaining a low failure rate as we scale up the chatbot. The following is a table that we created by taking

the average failure rate of all the experiments that we have done. We got average for all the experiments with memory threshold and cpu threshold The table below shows that the CPU threshold consistently outperforms the memory threshold at all replica levels.

Max Replicas	**Failure Rate (%) with Memory Threshold**	**Failure Rate (%) with CPU Threshold**
2	3.8%	3.1%
3	3.3%	2.6%
4	2.5%	1.9%
5	2.0%	1.5%
6	1.8%	1.4%
7	1.7%	1.2%
8	1.7%	1.2%
9	1.6%	1.2%
10	1.6%	1.2%

Table 6.2: Comparison of Failure Rates

As indicated, CPU thresholds maintain a lower failure rate compared to memory thresholds. For instance, with two replicas, the failure rate is 3.1% for the CPU threshold compared to 3.8% for the memory threshold. This trend continues with an increasing number of replicas, showing that CPU thresholds are more efficient for scaling the chatbot while keeping failure rates low.

6.2.4 Discussion

We investigated resource allocation for specific traffic volumes to determine achievable Service Level Objectives (SLOs) for a specified number of users given fixed resources (CPU, Memory, and a maximum number of replicas). To address this research question, we

trained a predictive model using experimental data. This model was then validated against manually tested data to assess its accuracy in predicting the same outcomes.

The results showed that our model could reliably predict the SLOs based on the given configuration parameters. Specifically, the number of replicas emerged as the most critical factor in achieving desired response times and maintaining low failure rates. Additionally, CPU thresholds proved to be slightly more effective than memory thresholds in meeting SLO targets.

Overall, our model effectively demonstrated that accurate SLO predictions could be made for specified traffic volumes using the defined resource parameters, thus answering our research question and providing valuable insights into resource allocation for optimal chatbot performance.

Optimized Scaling Strategy in Chatbot Case Study

Key Insight: In chatbot case study, achieving SLOs hinges on precise configurations of replicas and CPU thresholds. Using six replicas and setting the CPU utilization threshold to 7.5% strikes an effective balance between response time and resource efficiency. Exceeding eight replicas shows diminishing returns in performance improvement relative to cost.

Strategic Insights

Key Takeaway: Decision tree scaling offers a data-driven way to manage resources in cloud-native systems. It identifies achievable SLOs for specific user loads and configurations and determines the most effective resource metric (CPU or memory) to trigger scaling. This ensures service quality during peak demand while keeping costs low.

6.3 RQ2: Resource Requirements for Desired SLOs

This section presents the results of evaluating the resource requirements needed to meet our chatbot system's SLOs. We developed an analysis framework that used both summary datasets and detailed raw data files to identify the minimal resource configuration required for the system to operate within the targeted performance thresholds.

To evaluate the resource requirements, we defined two sets of SLOs before starting the experiment. All results in this section are based on these SLOs. We named the sets "Essential" and "Premium."

SLO Category	Response Time Objective	System Reliability Objective
Essential SLOs	Respond to 80% of requests within 20 seconds.	Maintain a failure rate of less than 3%.
Premium SLOs	Respond to 95% of requests within 20 seconds.	Maintain a failure rate of less than 1%.

Table 6.3: Defined SLOs for Evaluating Resource Requirements

In the preprocessing step, we filtered out all records from the summary dataset that did not meet the defined SLO requirements. For the Essential SLOs, 71 rows remained, and for the Premium SLOs, 32 rows remained. These rows indicate configurations where the chatbot system potentially meets or exceeds the SLO1 and SLO2 targets. For these entries, we applied the scoring algorithm as outlined in Algorithm 5.3.

Table 6.4 and Table 6.5 present an example of the combined scores for configurations that met the Essential and Premium SLO requirements, respectively, after preprocessing.

ID	Max Replicas	Threshold Param	Threshold Val	Requests	Fail Rate(%)	Err Rate(%)	Resp $<$ 20s(%)	Combined Scores
0	6	Memory	250	25	11.72	6.354	61.8	[45.16, 45.2, 45.1, 45.05, 45.25]
1	7	Memory	250	50	13.224	5.465	64.568	[36.29, 36.3, 36.25, 36.2, 36.35]
2	6	CPU	5	25	12.398	5.928	69.581	[38.52, 38.5, 38.55, 38.6, 38.45]

Table 6.4: Examples of Combined Scores for Essential SLOs

ID	Max Replicas	Threshold Param	Threshold Val	Requests	Fail Rate(%)	Err Rate(%)	Resp $<$ 20s(%)	Combined Scores
3	8	CPU	10	50	1.72	0.354	91.8	[42.16, 42.2, 42.1, 42.05, 42.25]
4	9	Memory	500	75	0.224	0.465	94.568	[38.16, 40.73, 40.52, 37.32, 40.71]
5	8	CPU	7.5	100	0.398	0.528	95.581	[37.12, 38.3, 39.31, 37.5, 41.38]

Table 6.5: Examples of Combined Scores for Premium SLOs

Now that we filtered configurations that meet the defined SLOs, we want to choose the best option. To do so, we think it is better to see into details of each configuration to ensure that they can provide the same performance in the production. To do so, we want to apply statistical analysis to choose the best configuration among the ones available.

Now that we have filtered out configurations that meet the defined SLOs, we aim to select the best option. To ensure they deliver consistent performance in a production environment, we will investigate into the details of each configuration.

6.3.1 Statistical Analysis and Configuration Ranking

To determine the best configuration among those that meet the SLOs and can be reliably deployed in production, we performed a statistical test known as Analysis of Variance (ANOVA). This method helps us ascertain whether the differences in performance between various configurations are statistically significant or merely due to random variations. The number of groups in the ANOVA test corresponds to the configurations that meet the SLO and have at least the specified number of requests.

Calculation of Averages The first step in our statistical analysis was to compute the average combined score for each configuration. These averages provide a baseline for comparing the performance levels across configurations, crucial for identifying consistently high-performing setups.

The table below has been updated to include the average combined scores for each configuration in both the Essential and Premium categories:

ID	Configuration	Combined Scores	Average Score
3	...	[42.16, 42.2, 42.1, 42.05, 42.25]	*42.15*
4	...	[38.16, 40.73, 40.52, 37.32, 40.71]	*39.48*
5	...	[37.12, 38.3, 39.31, 37.5, 41.38]	*38.72*

Table 6.6: Example List of Combined Scores for Premium SLOs with Averages

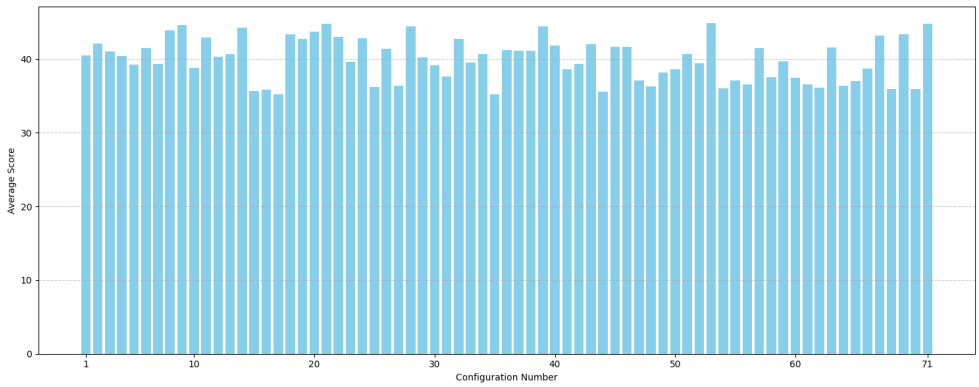

Figure 6.9: Average Performance Scores Across All Configurations for Essential SLOs

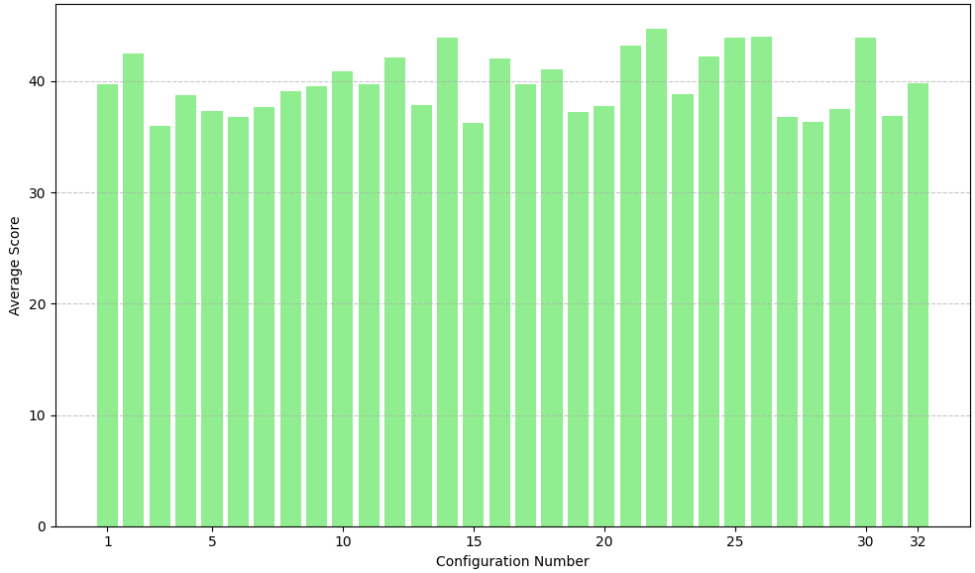

Figure 6.10: Average Performance Scores Across All Configurations for Premium SLOs

We use ANOVA to analyze the performance of different configurations by comparing their combined scores. This method helps determine whether variations in memory allocation, CPU thresholds, and replica limits significantly impact system efficiency.

ANOVA examines the differences in key performance metrics across configurations:

- **Measured Metrics:**
  - Response Time (s): Measures how quickly the system processes requests.
  - Failure Rate (%): Captures the percentage of requests that fail or time out.
  - Combined Score: A weighted metric balancing performance (response time, failure rate) with resource efficiency.
- **Contribution to Analysis:**
  - Response time and failure rate indicate whether configurations meet SLO compliance.
  - The combined score allows ranking of configurations based on both performance and efficiency.
  - ANOVA identifies statistically significant differences among configurations, ensuring that observed improvements are not due to random variations.

By analyzing these measurements with ANOVA, we can determine which configurations provide the best trade-off between response time, failure rate, and resource utilization.

In this study, ANOVA is applied to configurations that have already been passed after filtering so the number of our groups are the number of our configurations

Before conducting ANOVA, we verified that the dataset meets its fundamental assumptions:

- **Independence:** Each experimental run was conducted separately, ensuring that one configuration's outcome does not affect another.
- **Normality:** A Shapiro-Wilk test was applied to confirm that response times and failure rates follow a normal distribution.
- **Homogeneity of Variances:** A Levene's test confirmed that variances across configurations were approximately equal, validating the use of ANOVA.

These steps ensure that ANOVA produces reliable and meaningful comparisons.

The combined score, defined in Equation 5.1, represents a weighted average between performance and resource efficiency. It balances system responsiveness with optimal resource consumption. This metric is particularly suited for ANOVA analysis, as it integrates multiple dimensions into a single scalar value, enabling statistical comparisons across different configurations or experimental conditions.

where Response Time Weight and Resource Efficiency Weight balance performance and computational cost. ANOVA is used to assess whether differences in combined scores across configurations are statistically significant, helping to identify the most effective resource allocations.

Since we compare multiple configurations (27 groups), ANOVA is the appropriate statistical test. A t-test is only suitable for comparing two groups, whereas ANOVA allows us to analyze differences across multiple experimental conditions simultaneously. If significant differences are found, post-hoc tests can further clarify which configurations differ.

Each configuration was tested with five independent experimental runs, resulting in

a total dataset of 135 samples. The sufficient number of observations per configuration ensures that ANOVA provides statistically valid results when comparing mean differences across multiple groups.

ANOVA Results: With the averages calculated, we conducted an ANOVA test to compare the means of the combined scores from each configuration to determine if there are significant differences between the configurations. The table below summarizes the key findings from the ANOVA analysis:

Statistic	**Essential SLOs**	**Premium SLOs**
F-statistic	5.23	7.86
P-value	0.0002	0.0001
Degrees of Freedom (Between groups)	31	23
Degrees of Freedom (Within groups)	68	72
Significance Level	0.05	0.05

Table 6.7: Comparative ANOVA Results Summary for Essential and Premium SLO Configurations

Explanation of ANOVA Results:

- **F-statistic:** The F-statistic values (5.23 for Essential and 7.86 for Premium) indicate the degree to which group means vary from each other relative to the variation within each group. Higher values in Premium SLOs suggest more pronounced differences between group means, which could imply that Premium configurations are more sensitive to changes in configuration parameters.

- **P-value:** P-values less than 0.05 (0.0002 for Essential and 0.0001 for Premium) provide strong evidence against the null hypothesis, confirming significant differences

among the groups for both SLO categories. The smaller p-value for Premium SLOs indicates even stronger evidence against the null hypothesis.

- **Degrees of Freedom (Between groups):** Indicates the number of independent categories minus one. Fewer groups in Premium configurations showing more focused testing scenarios.

- **Degrees of Freedom (Within groups):** Represents the total number of observations minus the number of groups. More observations in Premium configurations suggest a more extensive dataset was analyzed, due to more rigorous testing requirements.

- **Significance Level:** Set at 0.05 for both, this threshold indicates the probability level below which the null hypothesis is rejected, consistent across both Essential and Premium configurations.

6.3.2 Configuration Ranking Based on ANOVA Results

To select the most efficient configuration for both Essential and Premium Service Level Objectives (SLOs), we applied a detailed statistical analysis using the Analysis of Variance (ANOVA) method. The following steps outline our approach to rank configurations based on their performance as measured by combined scores, where higher scores indicate better performance.

Ranking Process

The ranking of configurations was carried out through a multi-step process designed to ensure that the selected configurations not only meet the SLO requirements but do so with the highest efficiency:

1. **Compute Average Scores:** For each configuration, we calculated the average of the combined scores from the experimental data. This average score represents the general performance level of each configuration under controlled test conditions.

2. **Statistical Analysis (ANOVA):** We performed an ANOVA test to analyze the variance among the average scores of the configurations. This test helps to determine if the differences in average scores are statistically significant, beyond random chance.

3. **Evaluate Significance:**

    - **F-statistic:** Assesses the variance between group means to the variance within groups. A higher F-statistic indicates a greater disparity among group means.
    - **P-value:** We set a significance level at 0.05. Configurations showing a P-value less than this threshold were considered to have statistically significant differences in performance.

4. **Rank Configurations:** Configurations were ranked based on their average scores in descending order. Those with statistically significant higher scores were given priority in the rankings. This step ensures that we recommend configurations that not only perform well on average but also do so consistently across multiple evaluations.

5. **Select Top Performers:** From the ranked list, we identified the top configurations. These are configurations with the highest average scores and statistically significant differences, indicating robust and efficient performance.

Ranked Results for Essential and Premium SLOs

Based on the outlined process, we compiled the ranked results for configurations under both Essential and Premium SLOs into detailed tables, showing each configuration's key parameters alongside their average scores:

Rank	Max Replicas	Threshold Param	Threshold Val	Average Score
1	9	Memory	750 MB	48.35
2	10	Memory	750 MB	47.30
3	10	CPU	7.5 %	46.75

Table 6.8: Ranked Configurations for Essential SLOs Based on Average Scores

Rank	Max Replicas	Threshold Param	Threshold Val	Average Score
1	10	Memory	750 MB	47.30
2	10	CPU	7.5 %	46.75
3	8	Memory	750 MB	46.00

Table 6.9: Ranked Configurations for Premium SLOs Based on Average Scores

Note: The configuration rankings in Table 6.9 closely similar to those in Table 6.8, with the exception of the highest-ranked configuration in Essential SLOs being excluded from the Premium SLOs. This exclusion was due to a failure rate of 1.2%, which led to its removal during the filtering process for premium SLOs.

6.3.3 Discussion

We developed an analysis framework to identify minimal resource configurations that meet both Essential and Premium SLOs. Comprehensive data preprocessing and a scoring algorithm helped us pinpoint configurations consistently meeting performance targets. Further statistical analysis, including ANOVA, confirmed significant performance differences among configurations. Consequently, we ranked the configurations, ensuring reliable and efficient performance in production environments.

Resource Optimization Strategy in Chatbot Case Study

Key Insight: In the chatbot case study, meeting Essential SLOs (80% responses within 20 seconds, failure rate under 3%) requires at least six replicas with a CPU utilization threshold of 7.5%. Premium SLOs (95% responses within 20 seconds, failure rate under 1%) demand ten replicas and a memory threshold of 750 MB.

Actionable Recommendations:

- For Essential SLOs, configure the system with 9 replicas and a Memory threshold of 750MB.
- For Premium SLOs, scale up to 10 replicas and use a memory threshold of 750 MB to ensure high reliability and low latency.
- Avoid overprovisioning beyond the required thresholds to prevent unnecessary costs.

Strategic Insights

Key Takeaway: Identifying minimum resource configurations ensures that SLOs are met without overprovisioning. This approach optimizes operational costs while maintaining service quality, providing a clear resource baseline for varying customer loads.

6.4 RQ3: Customer Support Capacity with Given Resources and SLOs

This section presents the results of an evaluation of the maximum customer support capacity based on predefined SLOs and system configurations. We developed a decision tree predictive model to estimate the number of users that can be effectively supported.

To evaluate the maximum number of customers, we utilized the same sets of SLOs as those used in the previous section, referred to as Essential and Premium, which are defined in Table 6.3.

First, we applied preprocessing to the summary dataset, which was the same as in the previous section. Table 6.4 and Table 6.5 were created. For similar configurations, the record with the higher number of users is kept, and the others are removed from the dataset.

6.4.1 Optimizing the Model

The decision tree model was optimized using cross-validation. The optimized hyperparameters for both Essential and Premium SLOs are shown in 6.4.1.

Parameter	**Essential SLOs**	**Premium SLOs**
`max_depth`	5	4
`min_samples_leaf`	2	3
`min_samples_split`	15	12

Table 6.10: Optimized hyperparameters for the decision tree model

The baseline models for customer support capacity provided an initial estimate but lacked accuracy, with an MAE exceeding 15 users per configuration. After optimization, the decision tree model, utilizing the refined hyperparameters outlined in Table 6.4.1, is visualized in Figure 6.11 and Figure 6.12., demonstrated significantly lower errors, achieving an MAE of 7.6 for Essential SLOs and 8.2 for Premium SLOs...

6.4.2 Model Performance

For measuring the model performance, test data is randomly selected from the summary dataset. Each test record includes a unique configuration, and the target value is the number of customers for the model to predict. The metrics derived from this testing—R-squared, Mean Absolute Error, and Mean Squared Error—serve as indicators of the model's ability to accurately predict the number of customers.

Metric	**Essential SLOs**	**Premium SLOs**
R-squared (R^2)	0.84	0.79
Mean Absolute Error (MAE)	7.6	8.2
Mean Squared Error (MSE)	9.7	10.5

Table 6.11: Model Performance Metrics for Essential and Premium SLOs

Table 6.4.2 presents the model performance metrics for both Essential and Premium SLOs.

For the Essential SLOs, the high R-squared value indicates that the model explains a significant portion of the variance in customer numbers, suggesting effective pattern capture. The Mean Absolute Error (MAE) shows that, on average, the model's predictions

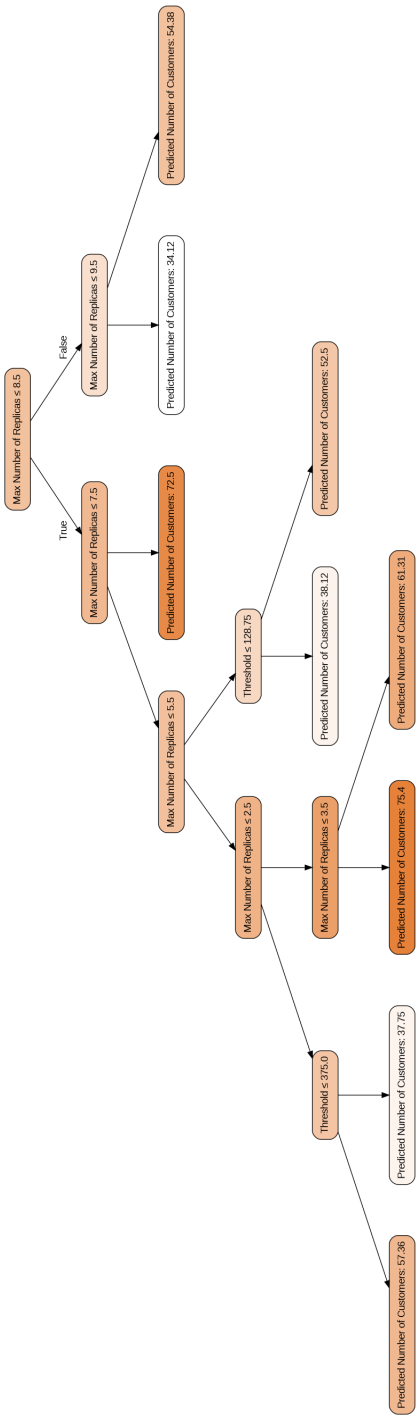

Figure 6.11: Optimized Decision Tree Model for Essential SLOs in RQ3

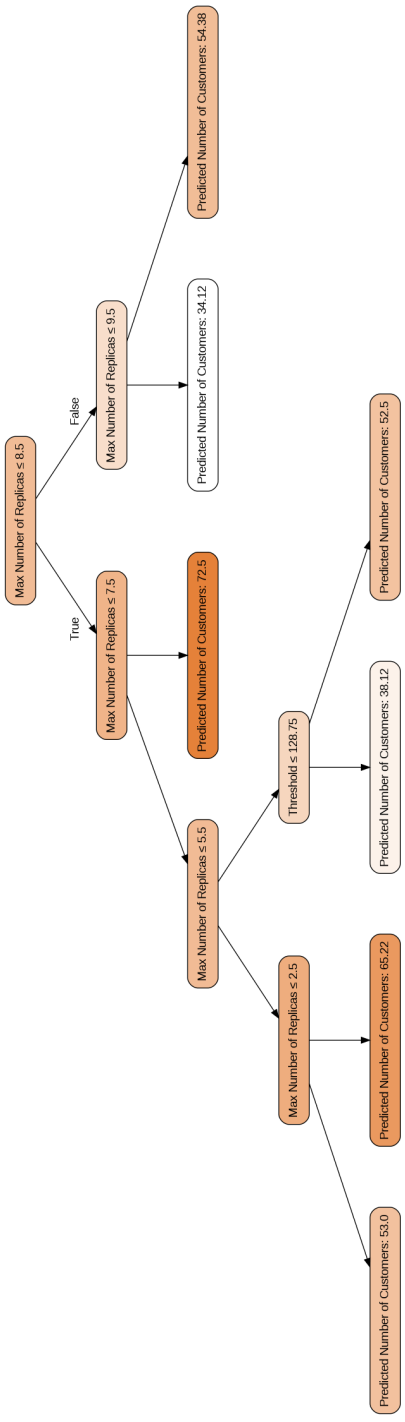

Figure 6.12: Optimized Decision Tree Model for Premium SLOs in RQ3

are close to the actual values. The Mean Squared Error (MSE) further indicates that large prediction errors are infrequent and not severe.

For the Premium SLOs, the model also performs well with a good R-squared value, showing solid explanatory power. The MAE suggests that the predictions are reasonably accurate, though slightly less precise than for Essential SLOs. The MSE indicates that larger errors are still well-controlled but slightly more frequent compared to the Essential SLOs scenario.

6.4.3 Discussion

We developed a decision tree model to evaluate customer support capacity under predefined Essential and Premium SLOs. The Essential SLOs model achieved high predictive accuracy, with strong performance metrics indicating effective pattern capture and low error rates. The Premium SLOs model also demonstrated reliable performance, though with slightly higher error metrics. Overall, both models proved effective in estimating customer support capacity, providing valuable insights for optimizing resource allocation and ensuring reliable service delivery.

Maximum Capacity Estimation Strategy in Chatbot Case Study

Key Insight: The chatbot case study demonstrates that the decision tree model accurately predicts customer support capacity for both Essential and Premium SLOs:

- Essential SLOs: Supports up to 53 users with a high predictive accuracy (R-squared = 0.84, MAE = 7.6).
- Premium SLOs: Supports up to 36 users with reliable predictions (R-squared = 0.79, MAE = 8.2).

Actionable Recommendations:

- Use decision tree models for estimating support capacity under predefined SLOs.
- Optimize the model hyperparameters (`max_depth`, `min_samples_split`, `min_samples_leaf`) based on traffic patterns to improve prediction accuracy.
- Plan scalability limits based on model insights to balance service quality and resource use effectively.

Strategic Insights

Key Takeaway: Decision tree models are effective for predicting maximum customer support capacity under specific SLOs and resource constraints. They provide a way to estimate how many users can be supported without compromising service quality. These insights enable precise scalability planning and resource allocation.

Chapter 7

Conclusion

7.1 Summary of Findings

This thesis aimed to address critical challenges in scaling cloud-native systems, with chatbot as a case study. By conducting an empirical study, we aimed to provide engineers and cloud service providers with predictive tools to optimize resource allocation and ensure system reliability under varying load conditions.

Our research was guided by three primary questions:

1. **Resource Allocation for Specific Traffic Volumes (RQ1):** We developed predictive models using decision trees to determine achievable SLOs based on fixed resources and specified user loads. The models effectively predicted response time and failure rate targets, with the number of replicas identified as the most influential

factor. CPU utilization thresholds were slightly more effective than memory thresholds in meeting SLOs. The optimized models achieved high predictive accuracy, with R^2 values up to 0.85.

2. **Resource Requirements for Desired SLOs (RQ2):** We formulated an analysis framework to identify the minimum resource configurations necessary to meet essential and premium SLOs. By applying statistical methods, including ANOVA, we ranked configurations based on combined performance and resource efficiency scores. The results indicated that for essential SLOs, at least six replicas with a CPU threshold of 7.5% were required, while premium SLOs demanded up to ten replicas with a memory threshold of 750 MB.

3. **Customer Support Capacity with Given Resources and SLOs (RQ3):** We developed decision tree models to predict the maximum number of users that could be supported while maintaining the desired SLOs. The models demonstrated strong predictive capabilities, with R^2 values of 0.84 for essential SLOs and 0.79 for premium SLOs. This allows service providers to estimate support capacity accurately and plan resource allocation accordingly.

7.2 Implications

The findings of this research have significant practical implications.

- **Cost-Efficient Scaling:** Service providers can use our predictive models to allocate resources more efficiently, avoiding overprovisioning while ensuring service quality.

- **Enhanced User Experience:** By maintaining response times within acceptable limits and reducing failure rates, the user experience is improved, leading to higher customer satisfaction.

- **Strategic Planning:** The ability to predict maximum support capacity under given resource constraints and SLOs aids in strategic decision-making and resource planning.

- The predictive resource management approach developed in this study can be extended beyond chatbots to a variety of cloud-native applications, such as intelligent recommendation engines, IoT-based decision systems, and financial risk modeling. By abstracting decision tree-based predictive models, different domains can leverage similar resource optimization strategies to maintain system reliability under dynamic conditions. However, an essential step in applying this methodology to a new domain is redefining the Service Level Objectives (SLOs). Since each system has different operational requirements, what constitutes "acceptable performance" and "resource efficiency" varies significantly.

7.3 Limitations

While the research provides valuable insights, certain limitations should be acknowledged:

- **Scope of SLOs:** The study focused on two specific SLOs—response time and failure rate. Other important metrics like throughput and availability were not considered.

- **Generality of Models:** The predictive models were trained and validated on data from a specific chatbot application using the LLaMA model. While the models are applicable to similar AI-driven chatbots, their generalizability to other types of cloud-native applications or different AI models may require further validation.
- **Static Workload Patterns:** The experiments utilized controlled, uniform request patterns to simulate user load. In real-world scenarios, workload patterns are more dynamic and may include bursty traffic or varying request complexities, which could affect the models' predictive accuracy.
- **Infrastructure Constraints:** The experiments were conducted on a specific hardware configuration. Different infrastructure setups, such as variations in network latency or hardware performance, could influence the applicability of the findings.

7.4 Future Work

Building on this research, future studies could explore the following areas:

- **Incorporating Additional SLOs:** Extending the framework to include other performance metrics like availability, scalability, and security to provide a more comprehensive resource management solution.
- **Cross-Application Validation:** Applying the predictive models to different types of cloud-native applications to test their generalizability and refine them for broader applicability.

Future research should investigate which performance metrics and system parameters from this chatbot study can be abstracted into generalized frameworks applicable to diverse cloud-native workloads. Key metrics such as latency distribution, failure rate thresholds, and adaptive scaling limits should be tested across different AI-driven applications to validate their universality. Furthermore, the **Combined Score formula**, which was designed specifically to balance chatbot performance and resource efficiency, must be adapted to different scenarios. The definition of "performance" and "resource usage" changes across domains—for example, in an AI-driven recommendation system, performance might be defined by prediction accuracy rather than response time, while resource usage could be evaluated in terms of memory bandwidth rather than CPU consumption. Future work should focus on deriving adaptable scoring mechanisms that accurately represent domain-specific trade-offs.

Further investigation into ensemble learning techniques, reinforcement learning, and hybrid regression-classification models could enhance predictive accuracy and adaptivity in varying cloud environments. Additionally, incorporating heuristic-driven scaling policies—such as threshold-based decision rules informed by historical workload patterns—could refine real-time adaptation in production deployments. These techniques could be applied not only to chatbot workloads but also to systems such as **autonomous control frameworks or real-time analytics pipelines**, where predictive resource management is crucial for maintaining both responsiveness and efficiency.

- **Cost-Benefit Analysis:** Integrating economic factors into the predictive models

to optimize not just performance but also operational costs, leading to more cost-effective resource allocation strategies [70, 71].

- **User Behavior Modeling:** Incorporating user behavior analytics to predict workload patterns more accurately. This could involve using time-series analysis or stochastic modeling to simulate real-world usage scenarios.

- **Integration with DevOps Practices:** Embedding the predictive framework into continuous integration and continuous deployment (CI/CD) pipelines to automate resource management and scaling decisions as part of standard DevOps workflows.

7.5 Closing Remarks

This thesis has demonstrated the efficacy of predictive resource management in scaling cloud-native applications, using a chatbot as a case study. By proactively allocating resources based on predictive models, service providers can meet stringent SLOs, ensuring both system reliability and user satisfaction. The integration of decision tree regression models with Kubernetes' autoscaling mechanisms provides a practical and interpretable approach to dynamic resource allocation.

The methodology proposed in this thesis can be adapted beyond chatbot-driven architectures by modifying resource selection strategies and fine-tuning auto-scaling heuristics for different types of services. A key step in this adaptation process is ensuring that **SLO definitions align with the needs of the target application**. The thresholds for acceptable performance and failure rates in a chatbot scenario are not necessarily appli-

cable to an IoT sensor network or a cloud-based financial transaction system. Similarly, the **Combined Score formulation must be re-evaluated to reflect the specific trade-offs between performance and resource efficiency in different domains**. Addressing these domain-specific differences will ensure that the predictive resource management strategies developed in this research can be effectively applied to a broader range of cloud-native environments.

By addressing the challenges of scalability and performance in cloud-native environments, this research moves us closer to realizing fully autonomous, self-optimizing systems capable of meeting the demands of modern applications. It opens avenues for innovation in resource management strategies, ultimately contributing to the development of more sustainable and user-centric cloud services.

Bibliography

[1] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," Journal of Systems and Software, vol. 126, pp. 1–16, 2017.

[2] R. Potla, "Enhancing customer relationship management (crm) through ai-powered chatbots and machine learning," Distributed Learning and Broad Applications in Scientific Research, vol. 9, pp. 364–383, 2023.

[3] J. P. K. S. Nunes, S. Nejati, M. Sabetzadeh, and E. Y. Nakagawa, "Self-adaptive, requirements-driven autoscaling of microservices," in Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2024, Lisbon, Portugal, April 15-16, 2024, L. Baresi, X. Ma, and L. Pasquale, Eds. ACM, 2024, pp. 168–174.

[4] J. Li, B. Moeini, S. Nejati, M. Sabetzadeh, and M. McCallen, "A lean simulation framework for stress testing iot cloud systems," IEEE Trans. Software Eng., vol. 50, no. 7, pp. 1827–1851, 2024.

[5] B. Nachiappan, "Emerging and innovative ai technologies for resource management," in Improving Library Systems with AI: Applications, Approaches, and Bibliometric Insights. IGI Global, 2024, pp. 115–133.

[6] N. Rane, S. Choudhary, and J. Rane, "Artificial intelligence (ai), internet of things (iot), and blockchain-powered chatbots for improved customer satisfaction, experience, and loyalty," Internet of Things (IoT), and blockchain-powered chatbots for improved customer satisfaction, experience, and loyalty (May 29, 2024), 2024.

[7] "Sedna lab, school of electrical engineering and computer science (eecs), university of ottawa," https://www.uoiot.ca/.

[8] D. Chaudhary, S. L. Vadlamani, D. Thomas, S. Nejati, and M. Sabetzadeh, "Developing a llama-based chatbot for ci/cd question answering: A case study at ericsson," arXiv preprint arXiv:2408.09277, 2024, accepted at the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME 2024). [Online]. Available: https://doi.org/10.48550/arXiv.2408.09277

[9] W. Zhang, S. Kosta, and P. Mogensen, "Multi-cloud containerized service scheduling optimizing computation and communication," pp. 186–193, 2024.

[10] M. Repetto, "Service templates to emulate network attacks in cloud-native 5g infrastructures," in 2023 IEEE 9th International Conference on Network Softwarization (NetSoft). IEEE, 2023, pp. 498–503.

[11] W. Zhu, Q. Meng, Q. Wang, B. Lin, Z. Huang, and C. Shi, "A generic framework for cloud-native network function," in International Conference on Cloud Computing,

Performance Computing, and Deep Learning (CCPCDL 2023), vol. 12712. SPIE, 2023, pp. 85–94.

[12] Z. Kang, Z. Min, S. Zhou, Y. D. Barve, and A. Gokhale, "Dataset placement and data loading optimizations for cloud-native deep learning workloads," in 2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2023, pp. 107–116.

[13] D. J. Milroy, C. Misale, G. Georgakoudis, T. Elengikal, A. Sarkar, M. Drocco, T. Patki, J.-S. Yeom, C. E. A. Gutierrez, D. H. Ahn et al., "One step closer to converged computing: Achieving scalability with cloud-native hpc," in 2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC). IEEE, 2022, pp. 57–70.

[14] K. Cheng, Z. Wang, W. Hu, T. Yang, J. Li, and S. Zhang, "Towards slo-optimized llm serving via automatic inference engine tuning," arXiv preprint arXiv:2408.04323, 2024.

[15] M. Mekki, N. Toumi, and A. Ksentini, "Microservices configurations and the impact on the performance in cloud native environments," in 2022 IEEE 47th Conference on Local Computer Networks (LCN). IEEE, 2022, pp. 239–244.

[16] M. G. Khan, J. Taheri, M. A. Khoshkholghi, A. Kassler, C. Cartwright, M. Darula, and S. Deng, "A performance modelling approach for sla-aware resource recommendation in cloud native network functions," in 2020 6th IEEE Conference on Network Softwarization (NetSoft). IEEE, 2020, pp. 292–300.

[17] A. Kumar, "Ai-driven innovations in modern cloud computing," arXiv preprint arXiv:2410.15960, 2024.

[18] T. Theodoropoulos, L. Rosa, C. Benzaid, P. Gray, E. Marin, A. Makris, L. Cordeiro, F. Diego, P. Sorokin, M. D. Girolamo et al., "Security in cloud-native services: A survey," Journal of Cybersecurity and Privacy, vol. 3, no. 4, pp. 758–793, 2023.

[19] S. K. Khanday, "Optimizing performance and cost efficiency in ai-driven cloud infrastructures: A multi-objective approach," pp. 648–654, 2025.

[20] S. Joshi, B. Hasan, and R. Brindha, "Optimal declarative orchestration of full lifecycle of machine learning models for cloud native," in 2024 3rd International Conference on Applied Artificial Intelligence and Computing (ICAAIC). IEEE, 2024, pp. 578–582.

[21] P. Lertpongrujikorn, H. D. Nguyen, and M. A. Salehi, "Streamlining cloud-native application development and deployment with robust encapsulation," in Proceedings of the 2024 ACM Symposium on Cloud Computing, 2024, pp. 847–865.

[22] D. Chen, A. Youssef, R. Pendse, A. Schleife, B. K. Clark, H. Hamann, J. He, T. Laino, L. Varshney, Y. Wang et al., "Transforming the hybrid cloud for emerging ai workloads," arXiv preprint arXiv:2411.13239, 2024.

[23] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," Proceedings of the IEEE, vol. 112, no. 1, pp. 12–46, 2024.

[24] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," Future Generation Computer Systems, vol. 72, pp. 165–179, 2017.

[25] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros, "Maintaining slos of cloud-native applications via self-adaptive resource sharing," in 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE, 2019, pp. 72–81.

[26] A. Rubak and J. Taheri, "Machine learning for predictive resource scaling of microservices on kubernetes platforms," International Conference on Utility and Cloud Computing, 2023.

[27] S. Ponnusamy and M. Khoje, "Optimizing cloud costs with machine learning: Predictive resource scaling strategies," 2024 5th International Conference on Innovative Trends in Information Technology (ICITIIT), 2024.

[28] V. Flunkert, Q. Rebjock, J. Castellon, L. Callot, and T. Januschowski, "A simple and effective predictive resource scaling heuristic for large-scale cloud applications," ArXiv, 2020.

[29] C. Milroy, D. Patterson, M. Weiss et al., "Kubernetes-based scaling in hpc environments," Journal of High Performance Computing, 2022.

[30] S. Bobba, "Ai-powered predictive scaling in cloud computing: Enhancing efficiency through real-time workload forecasting," IRE Journals, 2021. [Online]. Available: https://www.irejournals.com/formatedpaper/17029432.pdf

[31] S. S. Pallapuneedi and V. K. R. Kondapalli, "Recent advancements in artificial intelligence driven cloud computing," 2024.

[32] N. N. Moyo, "Ai-driven optimization of cloud networking for large language model applications," Journal of Innovative Technologies, vol. 7, no. 1, 2024.

[33] G. K. Shyam and I. Chandrakar, "Resource allocation in cloud computing using optimization techniques," in Cloud Computing for Optimization: Foundations, Applications, and Challenges, ser. Studies in Big Data, B. S. P. Mishra, H. Das, S. Dehuri, and A. K. Jagadev, Eds. Springer, Cham, 2018, vol. 39, pp. 27–50.

[34] N. KODAKANDLA, "Serverless architectures: A comparative study of performance, scalability, and cost in cloud-native applications," Iconic Research And Engineering Journals, vol. 5, no. 2, pp. 136–150, 2021.

[35] N. Syed, A. Anwar, Z. Baig, and S. Zeadally, "Artificial intelligence as a service (aiaas) for cloud, fog and the edge: State-of-the-art practices," ACM Computing Surveys, 2025.

[36] S. Montagna, S. Ferretti, L. Klopfenstein, A. Florio, and M. Pengo, "Data decentralisation of llm-based chatbot systems in chronic disease self-management," Proceedings of the 2023 ACM Conference on Information Technology for Social Good, 2023.

[37] S. Bhattacharyya, Cloud Innovation: Scaling with Vectors and LLMs. Libertatem Media Private Limited, 2024.

[38] G. Sanodia, "Revolutionizing cloud modernization through ai integration," Turkish Journal of Computer and Mathematics Education, vol. 15, no. 2, pp. 266–283, 2024.

[39] L. Patan, "Leveraging cloud-native architecture for scalable and resilient enterprise applications: A comprehensive analysis," INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING AND TECHNOLOGY (IJCET), vol. 15, no. 5, pp. 583–591, 2024.

[40] B. Sharma and D. Nadig, "ebpf-enhanced complete observability solution for cloud-native microservices," in ICC 2024-IEEE International Conference on Communications. IEEE, 2024, pp. 1980–1985.

[41] S. Agrawal and D. Singh, "Study containerization technologies like docker and kubernetes and their role in modern cloud deployments," in 2024 IEEE 9th International Conference for Convergence in Technology (I2CT). IEEE, 2024, pp. 1–5.

[42] Y. X. Chia, C. K. Seow, K. Chen, and Q. Cao, "Exploring resource prediction models based on custom kubernetes auto-scaling metrics," in 2024 9th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA). IEEE, 2024, pp. 47–52.

[43] V. Salunkhe, D. Pakanati, H. Cherukuri, S. Khan, and D. A. Jain, "The impact of cloud native technologies on healthcare application scalability and compliance," Available at SSRN 4984999, 2021.

[44] J. Garrison and K. Nova, Cloud native infrastructure: patterns for scalable infrastructure and applications in a dynamic environment. " O'Reilly Media, Inc.", 2017.

[45] B. Nascimento, R. Santos, J. Henriques, M. V. Bernardo, and F. Caldeira, "Availability, scalability, and security in the migration from container-based to cloud-native applications," Computers, vol. 13, no. 8, p. 192, 2024.

[46] T. Bomström, "Case study: cloud computing model for vehicular data processing," Master's thesis, T. Bomström, 2024.

[47] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, "Sla guarantees for cloud services," Future Generation Computer Systems, vol. 54, pp. 233–246, 2016.

[48] E. Kapassa, M. Touloupou, P. Stavrianos, G. Xylouris, and D. Kyriazis, "Managing and optimizing quality of service in 5g environments across the complete sla lifecycle," Advances in Science, Technology and Engineering Systems Journal, vol. 4, no. 1, pp. 329–342, 2019.

[49] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "A novel middleware for efficiently implementing complex cloud-native slos," in 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). IEEE, 2021, pp. 410–420.

[50] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," IEEE Internet Computing, vol. 24, no. 3, pp. 39–50, 2020.

[51] P. Kochovski, V. Stankovski, S. Gec, F. Faticanti, M. Savi, D. Siracusa, and S. Kum, "Smart contracts for service-level agreements in edge-to-cloud computing," Journal of Grid Computing, vol. 18, pp. 673–690, 2020.

[52] C. Lekkala, "Ai-driven dynamic resource allocation in cloud computing: Predictive models and real-time optimization," J Artif Intell Mach Learn & Data Sci, vol. 2, no. 2, pp. 450–456, 2024.

[53] M. Hoyer, K. Schröder, D. Schlitt, and W. Nebel, "Proactive dynamic resource management in virtualized data centers," in Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking, 2011, pp. 11–20.

[54] N. Alapati and V. Valleru, "Ai-driven optimization techniques for dynamic resource allocation in cloud networks," MZ Computing Journal, vol. 4, no. 1, 2023.

[55] K. Sathupadi, "An ai-driven framework for dynamic resource allocation in software-defined networking to optimize cloud infrastructure performance," Int J Intell Appl Comput, 2023. [Online]. Available: https://research.tensorgate.org/index.php/IJIAC/article/download/132/125

[56] Y. Jiang, "Navigating the digital evolution: I 4.0 technologies and their role in reconfiguring dynamic resource management networks," International Journal of Industrial Engineering: Theory, Applications and Practice, vol. 31, no. 5, 2024.

[57] W.-C. Chien, C.-F. Lai, and H.-C. Chao, "Dynamic resource prediction and allocation in c-ran with edge artificial intelligence," IEEE Transactions on Industrial Informatics, vol. 15, no. 7, pp. 4306–4314, 2019.

[58] M. Abouelyazid, "Machine learning algorithms for dynamic resource allocation in cloud computing: Optimization techniques and real-world applications," J. AI-Assist. Sci. Discov, vol. 1, pp. 1–58, 2021.

[59] A. Asheralieva and D. Niyato, "Distributed dynamic resource management and pricing in the iot systems with blockchain-as-a-service and uav-enabled mobile edge computing," IEEE Internet of Things Journal, vol. 7, no. 3, pp. 1974–1993, 2019.

[60] S. Ahmad and A. u. Asar, "Reliability enhancement of electric distribution network using optimal placement of distributed generation," Sustainability, vol. 13, no. 20, p. 11407, 2021.

[61] S. Barja-Martinez, M. Aragüés-Peñalba, Í. Munné-Collado, P. Lloret-Gallego, E. Bullich-Massagué, and R. Villafafila-Robles, "Artificial intelligence techniques for enabling big data services in distribution networks: A review," Renewable and Sustainable Energy Reviews, vol. 150, p. 111459, 2021.

[62] H. Gadde, "Ai-based data consistency models for distributed ledger technologies," Revista de Inteligencia Artificial en Medicina, 2023. [Online]. Available: https://redcrevistas.com/index.php/Revista/article/download/182/205

[63] V. Manduva, "A comprehensive framework for ensuring data reliability in distributed ai systems," Revista de Inteligencia Artificial en Medicina, 2021. [Online]. Available: http://redcrevistas.com/index.php/Revista/article/download/94/86

[64] Y. Hong, J. Lian, L. Xu, J. Min, Y. Wang, L. J. Freeman, and X. Deng, "Statistical perspectives on reliability of artificial intelligence systems," Quality Engineering, vol. 35, no. 1, pp. 56–78, 2023.

[65] R. Chen, F. B. Bastani, and T.-W. Tsao, "On the reliability of ai planning software in real-time applications," IEEE Transactions on Knowledge and Data Engineering, vol. 7, no. 1, pp. 4–13, 1995.

[66] T. Laszewski, K. Arora, E. Farr, and P. Zonooz, Cloud Native Architectures: Design high-availability and cost-effective applications for the cloud. Packt Publishing Ltd, 2018.

[67] L. Ying et al., "Decision tree methods: applications for classification and prediction," Shanghai archives of psychiatry, vol. 27, no. 2, p. 130, 2015.

[68] H. Drucker and C. Cortes, "Boosting decision trees," Advances in neural information processing systems, vol. 8, 1995.

[69] A. Agarwal, Y. S. Tan, O. Ronen, C. Singh, and B. Yu, "Hierarchical shrinkage: Improving the accuracy and interpretability of tree-based models." in International Conference on Machine Learning. PMLR, 2022, pp. 111–135.

[70] S. Nejati, S. Di Alesio, M. Sabetzadeh, and L. C. Briand, "Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing," in Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings, ser.

Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 759–775.

[71] S. Di Alesio, L. C. Briand, S. Nejati, and A. Gotlieb, "Combining genetic algorithms and constraint programming to support stress testing of task deadlines," ACM Trans. Softw. Eng. Methodol., vol. 25, no. 1, pp. 4:1–4:37, 2015.