UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

João Victor de Oliveira Silva

A Comparative Study of Single-Objective and Multi-Objective Genetic Algorithms for Resource Optimization in Kubernetes Environments

Recife

Julho de 2026

João Victor de Oliveira Silva

A Comparative Study of Single-Objective and Multi-Objective Genetic Algorithms for Resource Optimization in Kubernetes Environments

Artigo apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal Rural de Pernambuco – UFRPE
Departamento de Estatística e Informática
Curso de Bacharelado em Sistemas de Informação

Orientador: Lidiano A. N. Oliveira

Recife
Julho de 2026

A Comparative Study of Single-Objective and Multi-Objective Genetic Algorithms for Resource Optimization in Kubernetes Environments

[João Victor de Oliveira Silva] [1], **[Lidiano A. N. Oliveira]** [1]

[1]Departamento de Estatística e Informática – Universidade Federal Rural de Pernambuco
Rua Dom Manuel de Medeiros, s/n, - CEP: 52171-900 – Recife – PE – Brasil

joao.oliveirasilva@ufrpe.br, lidiano.oliveira@ufrpe.br

Abstract. *The widespread adoption of Kubernetes has enabled efficient orchestration of containerized applications through automated deployment and scaling mechanisms. However, resource configuration decisions—such as the number of replicas and CPU and memory limits—are still commonly defined manually or based on static heuristics, which may lead to inefficient resource utilization or performance degradation under dynamic workloads. This work presents a comparative experimental study of two evolutionary strategies for resource optimization in Kubernetes environments: a single-objective Genetic Algorithm (SO-GA) that maximizes a weighted aggregation of latency, resource efficiency, and reliability scores, and the Non-dominated Sorting Genetic Algorithm II (NSGA-II), which simultaneously optimizes saturation, provisioned resources, and throughput as conflicting objectives. Both algorithms share the same evaluation pipeline—applying configurations to a live cluster, executing controlled load tests, and collecting application-level and Prometheus metrics—and explore an identical, discretized search space over replicas, CPU, and memory limits. A Random Search baseline with the same evaluation budget is included as a control. The three approaches are compared in terms of solution quality (best aggregated fitness, hypervolume, inverted generational distance), search behavior (Pareto-front coverage, dominance), and computational cost, providing empirical evidence on whether the additional complexity of multi-objective optimization is justified for Kubernetes resource configuration tasks.*

Resumo. *A ampla adoção do Kubernetes possibilitou a orquestração eficiente de aplicações conteinerizadas por meio de mecanismos automatizados de implantação e escalonamento. No entanto, as decisões de configuração de recursos — como o número de réplicas e os limites de CPU e memória — ainda são comumente definidas manualmente ou com base em heurísticas estáticas, o que pode levar à utilização ineficiente de recursos ou à degradação do desempenho sob cargas de trabalho dinâmicas. Este trabalho apresenta um estudo experimental comparativo entre duas estratégias evolutivas para otimização de recursos em ambientes Kubernetes: um Algoritmo Genético de objetivo único (SO-GA), que maximiza uma agregação ponderada de escores de latência, eficiência de recursos e confiabilidade, e o Non-dominated Sorting Genetic Algorithm II (NSGA-II), que otimiza simultaneamente saturação, recursos provisionados e throughput como objetivos conflitantes. Ambos os algoritmos compartilham o mesmo pipeline de avaliação — aplicando configurações em um cluster real, executando testes*

de carga controlados e coletando métricas de aplicação e do Prometheus — e exploram um espaço de busca idêntico e discretizado sobre réplicas, CPU e memória. Um baseline de Busca Aleatória (Random Search) com o mesmo orçamento de avaliações é incluído como controle. As três abordagens são comparadas em termos de qualidade de solução (melhor fitness agregado, hipervolume, distância geracional invertida), comportamento de busca (cobertura da frente de Pareto, dominância) e custo computacional, fornecendo evidências empíricas sobre se a complexidade adicional da otimização multiobjetivo se justifica em tarefas de configuração de recursos em Kubernetes.

1. Introduction

Kubernetes has become the dominant platform for container orchestration, enabling automated deployment, scaling, and management of containerized applications. By abstracting infrastructure details and providing declarative configuration mechanisms, Kubernetes allows developers to focus on application logic while the platform ensures availability, scalability, and fault tolerance. These characteristics make Kubernetes a fundamental component of modern cloud-native systems [1].

Despite its high degree of automation, Kubernetes still relies heavily on manual or rule-based decisions for resource configuration. Parameters such as the number of replicas, CPU limits, and memory allocation are typically defined based on static heuristics or empirical knowledge. In dynamic workloads, this approach can lead to inefficient resource utilization, performance instability, or resource saturation, requiring continuous human intervention to adjust configurations.

Recent advances in Artificial Intelligence and Machine Learning have enabled new forms of automation that go beyond rule-based systems, introducing adaptive and data-driven decision-making mechanisms. In the context of Kubernetes, AI-driven orchestration techniques have been explored to improve cluster management, scalability, and resource efficiency [1].

Among the various AI-based approaches, evolutionary algorithms — particularly genetic algorithms — have gained attention because of their ability to explore large and complex search spaces without requiring explicit system models. Genetic algorithms are well suited for optimization problems where the relationship between configuration parameters and system performance is non-linear and difficult to model analytically.

In this work, two evolutionary strategies are employed to optimize resource allocation in a Kubernetes cluster by iteratively evaluating different configurations under controlled load conditions. The first strategy is a single-objective Genetic Algorithm (SO-GA), which aggregates latency, resource efficiency, and reliability into a single weighted fitness score. The second strategy is the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [2], a classical multi-objective evolutionary algorithm that returns a Pareto front of non-dominated configurations across competing objectives — saturation, resource cost, and throughput. The choice to include both algorithms is twofold: (i) the single-objective formulation, despite its practical simplicity, is sensitive to the choice of weights and may obscure relevant trade-offs, which motivates the use of a multi-objective approach; and (ii) maintaining both algorithms in the same experimental pipeline enables a direct empirical comparison rather than a pure single-baseline evaluation. To

strengthen the analysis, a Random Search baseline with an identical evaluation budget is also included as a control, following the recommendation of [3] that any heuristic search method should be benchmarked against uniform sampling. Each candidate configuration is applied directly to the cluster using Kubernetes manifests, and its performance is assessed based on application-level metrics, such as throughput and latency, as well as resource utilization metrics collected via Prometheus.

To guide the experimental investigation, this work addresses the following research questions:

- Can evolutionary algorithms effectively identify improved resource configurations when interacting directly with a live Kubernetes cluster, compared to a non-optimized baseline?
- Does the multi-objective formulation (NSGA-II) yield measurable advantages in solution quality and trade-off coverage compared to a single-objective Genetic Algorithm with weighted aggregation?
- Do evolutionary search strategies (SO-GA and NSGA-II) provide statistically significant improvements over uniform Random Search under the same evaluation budget?

Based on the theoretical foundations of multi-objective optimization and prior work on evolutionary resource management, the following hypotheses are formulated:

- Evolutionary algorithms will identify configurations with higher fitness scores than a typical default Kubernetes deployment.
- NSGA-II will produce Pareto fronts with higher hypervolume and lower inverted generational distance than the projected fronts from SO-GA and Random Search, due to its explicit diversity preservation mechanisms.
- Both evolutionary algorithms will outperform Random Search in terms of solution quality metrics, justifying the additional algorithmic complexity.

2. Context

The increasing adoption of native cloud architectures has significantly transformed the way modern applications are developed, deployed, and operated. Containerization technologies, such as Docker, enable applications to be packaged together with their dependencies, ensuring portability and consistency across different environments. However, manually managing large numbers of containerized applications quickly becomes impractical due to the complexity involved in deployment, scaling, fault tolerance, and resource allocation.

Kubernetes has emerged as the leading platform for container orchestration, providing a robust framework for automating the deployment, scaling, and management of containerized workloads. Through abstractions such as clusters, nodes, pods, and services, Kubernetes enables applications to run reliably across heterogeneous infrastructures, including on-premises data centers and cloud environments. Its declarative model allows users to specify the desired state of the system, while the control plane continuously monitors the cluster and enforces this state through reconciliation mechanisms.

A Kubernetes cluster is composed of one or more nodes, which are responsible for running application workloads encapsulated in pods. Pods represent the smallest deployable units in Kubernetes and may contain one or more containers that share networking

and storage resources. The Kubernetes control plane, exposed through the Kubernetes API (Kube-API), manages these resources by scheduling pods, monitoring their health, and applying scaling or recovery actions when necessary. Resource constraints, such as CPU and memory limits, can be defined at the container level to control resource consumption and prevent interference between workloads.

Despite the high degree of automation provided by Kubernetes, resource configuration decisions— such as the number of replicas, CPU limits, and memory allocations—are typically defined manually or based on static heuristics. These configurations are often derived from prior experience, conservative estimates, or simple threshold-based rules. As a result, clusters may suffer from inefficient resource utilization, leading to either underutilization (wasted resources) or saturation (performance degradation and increased latency).

To observe and analyze application behavior in Kubernetes environments, monitoring systems play a fundamental role. Prometheus has become a widely adopted monitoring solution due to its native integration with Kubernetes and its powerful time-series data model. By scraping metrics exposed by applications and system components, Prometheus enables the collection of fine-grained performance data, such as CPU usage, memory consumption, request rates, and latency distributions. These metrics provide valuable insight into both application-level performance and infrastructure-level resource utilization.

In this context, the availability of detailed monitoring data and programmable interfaces opens the possibility for more advanced and adaptive resource management strategies. Instead of relying solely on static configurations or predefined autoscaling rules, optimization techniques can leverage real-time performance feedback to explore alternative configurations and dynamically adapt resource allocation. This work is situated within this context, investigating the feasibility of integrating an optimization algorithm with a Kubernetes cluster to evaluate and improve resource configurations based on observed workload behavior.

3. Motivation

Although Kubernetes provides powerful mechanisms for automating deployment, scaling, and fault tolerance, effective resource management remains a challenging task in practice. The platform offers primitives for defining CPU and memory limits, as well as horizontal scaling through replicas; however, deciding appropriate values for these parameters is largely left to system operators. In real-world scenarios, workloads are often dynamic and unpredictable, making static or manually tuned configurations insufficient to guaranty both performance stability and efficient resource usage.

Traditional approaches to resource configuration in Kubernetes typically rely on fixed heuristics, threshold-based autoscaling policies, or empirical tuning based on prior experience. While these methods can be effective in simple or well-understood workloads, they tend to struggle in environments where performance behavior emerges from complex interactions between application logic, resource constraints, and workload characteristics. As a consequence, clusters may experience periods of overprovisioning—leading to wasted resources—or underprovisioning, resulting in increased latency, degraded throughput, and potential violations of service-level objectives.

From a research perspective, several studies have explored optimization techniques for Kubernetes resource management. However, many existing works evaluate their approaches using simplified analytical models, simulations, or synthetic benchmarks, rather than integrating directly with a real Kubernetes cluster. This abstraction limits the ability to capture practical aspects such as scheduling behavior, container runtime overhead, and real monitoring noise, which are intrinsic to production-like environments. As a result, there is a gap between theoretical optimization results and their applicability to real-world Kubernetes deployments.

Genetic algorithms offer a promising alternative in this context due to their population-based search strategy and their ability to optimize non-linear and multi-parameter systems without requiring explicit performance models. Their exploratory nature makes them particularly suitable for environments where the relationship between configuration parameters and observed performance metrics is complex and difficult to formalize. Despite this potential, there is still limited empirical evidence demonstrating how genetic algorithms perform when directly coupled with a live Kubernetes cluster and evaluated under realistic workload conditions.

A second methodological gap concerns the formulation of the optimization problem itself. Single-objective genetic algorithms with weighted aggregation fitness functions are widely used because of their simplicity, but they exhibit two well-documented limitations [1, 2]: (i) the resulting solution depends strongly on the choice of weights, which encodes an implicit preference among objectives that may be difficult to justify a priori; and (ii) configurations that represent meaningful trade-offs between conflicting metrics (for example, low latency at high cost versus high throughput with moderate saturation) are collapsed into a single scalar value and may be discarded by selection pressure. Multi-objective evolutionary algorithms such as NSGA-II address both limitations by returning a Pareto front of non-dominated solutions, leaving the trade-off resolution to the decision-maker. However, this benefit comes at a cost in algorithmic complexity and result interpretation, and it is not self-evident that this complexity is justified for resource configuration in Kubernetes environments.

The motivation of this work is therefore threefold. First, it investigates whether evolutionary algorithms can effectively identify improved resource configurations when interacting directly with a Kubernetes cluster, rather than relying on abstract models or simulations. Second, it provides an experimental analysis grounded in reproducible measurements, using real application workloads and metrics collected through standard monitoring tools such as Prometheus. Third, it conducts a controlled comparison between a single-objective Genetic Algorithm, NSGA-II, and a Random Search baseline executed on the same search space and with the same evaluation budget, in order to assess whether the multi-objective formulation yields measurable advantages in solution quality, search behavior, and trade-off coverage. By doing so, this study contributes to bridging the gap between theoretical optimization approaches and practical Kubernetes resource management, offering insights into the feasibility, benefits, and limitations of applying evolutionary algorithms in cloud-native environments.

4. Objectives

The general objective of this work is to investigate the feasibility and effectiveness of evolutionary algorithms for resource allocation in Kubernetes clusters and to compare a single-objective Genetic Algorithm, the multi-objective NSGA-II, and a Random Search baseline executed on the same search space and evaluation budget, in terms of application performance, resource utilization, and search behavior under controlled workload conditions.

To achieve the general objective, the following specific objectives are defined:

- To design and implement two evolutionary optimizers — a single-objective Genetic Algorithm (SO-GA) using a weighted aggregation of latency, resource efficiency, and reliability scores, and an NSGA-II optimizer formulating resource allocation as a tri-objective problem — alongside a Random Search baseline, all sharing the same evaluation pipeline.
- To integrate all three algorithms with a live Kubernetes cluster, enabling the dynamic application of configuration changes (replicas, CPU limits, and memory allocation) through the Kubernetes API.
- To deploy controlled load tests on a containerized application and collect application-level metrics (throughput, latency) and infrastructure-level metrics (CPU throttling, memory peak usage) using Prometheus.
- To compare the three algorithms in terms of solution quality (best aggregated fitness, hypervolume, inverted generational distance), search behavior (Pareto-front coverage, convergence patterns), and computational cost.
- To assess whether the additional complexity of multi-objective optimization (NSGA-II) is justified compared to single-objective optimization and random sampling in Kubernetes resource configuration tasks.

5. Theoretical Background

This section presents the theoretical foundations required to understand the proposed optimization approach. First, it introduces Kubernetes and its resource management mechanisms, which define the search space explored by the optimization algorithms. Next, it describes genetic algorithms, covering both single-objective formulations based on weighted aggregation and the multi-objective NSGA-II algorithm. Finally, it presents Prometheus, the monitoring system used to collect performance and resource utilization metrics during the experimental evaluation.

5.1. Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It adopts a declarative management model in which users specify the desired state of the system, while the control plane continuously reconciles the actual cluster state. This architecture provides automated scheduling, fault recovery, and resource management, making Kubernetes the widely adopted platform for cloud-native applications [4, 5].

Because this work aims to optimize Kubernetes resource configurations using evolutionary algorithms, understanding how the platform allocates computational resources

and manages application deployments is fundamental to the proposed optimization process.

A Kubernetes cluster consists of a control plane and one or more worker nodes. The control plane manages the overall cluster state and coordinates management operations, while worker nodes execute application workloads [4]. The Kubernetes API Server is the primary interface between users, external applications, and the cluster. Configuration changes, such as replica scaling and resource updates, are submitted through the API Server and persisted in *etcd*, which stores the cluster state. The Scheduler assigns Pods to worker nodes according to resource availability and scheduling constraints, while the Controller Manager continuously reconciles the actual cluster state with the desired state defined by Kubernetes objects [4]. Worker nodes host the application Pods and execute containers through a container runtime. Each node runs a *kubelet*, which communicates with the control plane and enforces the desired state locally. This architecture enables external optimization algorithms to interact programmatically with Kubernetes through its API, allowing deployment parameters to be modified without changing the application code.

Figure 1 illustrates the interaction between the control plane and worker nodes, where the Kubernetes API Server represents the primary interface between the optimization algorithms and the cluster, since all candidate configurations evaluated in this work are applied through this component.

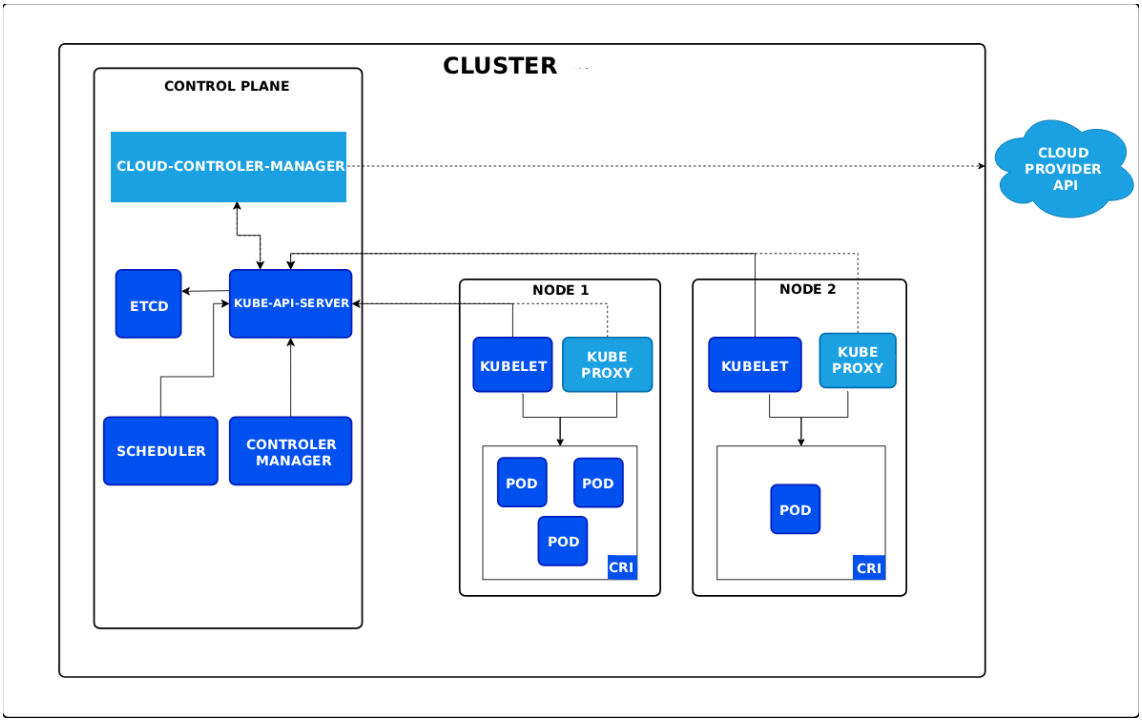

Figure 1. High-level Kubernetes cluster architecture. Adapted from Kubernetes Documentation [4].

A Pod is the smallest deployable unit in Kubernetes and represents one or more containers running together with shared networking and storage resources. Applications are typically deployed using Deployments, which manage Pods and maintain the desired

number of replicas despite failures or infrastructure changes. When a Pod terminates unexpectedly, Kubernetes automatically creates a replacement to preserve application availability [4]. Deployments expose configurable parameters, including replica count and container resource requests and limits, making them the primary abstraction manipulated by the optimization algorithms proposed in this work. By modifying these parameters through the Kubernetes API, candidate configurations can be evaluated without requiring changes to the application itself.

5.1.1. Resource Management

Resource management in Kubernetes is primarily controlled through CPU and memory requests and limits assigned to individual containers. A resource request specifies the amount of CPU or memory reserved for a container during scheduling. Kubernetes uses these values to determine whether sufficient resources are available on a worker node. Excessively high requests may reduce scheduling flexibility and lead to underutilized clusters. Resource limits define the maximum amount of CPU or memory that a container may consume. Exceeding CPU limits results in throttling, whereas exceeding memory limits may cause the container to be terminated due to an Out-of-Memory (OOM) condition. Overprovisioning wastes computational resources, whereas underprovisioning may increase CPU throttling, memory pressure, scheduling failures, or application latency. Since these parameters interact with workload characteristics and replica count, determining efficient configurations becomes a complex optimization problem. Kubernetes supports both manual and automatic scaling mechanisms. Manual scaling adjusts the number of application replicas through Deployment specifications, whereas automatic scaling is commonly performed using the Horizontal Pod Autoscaler (HPA), which periodically adjusts replica counts according to predefined metrics such as CPU utilization [4, 6]. Although HPA provides an effective reactive scaling mechanism, it only adjusts replica counts based on predefined utilization targets, without jointly optimizing other deployment parameters such as CPU requests, CPU limits, or memory allocation; moreover, its decisions are inherently reactive, responding only after workload changes have already affected observed metrics. Because these parameters are interdependent, optimizing them independently often results in suboptimal resource utilization. These limitations motivate optimization techniques capable of simultaneously exploring multiple deployment parameters while directly evaluating their impact on application performance and resource utilization, among which evolutionary algorithms have emerged as promising alternatives because they can explore complex search spaces without requiring explicit system models [7, 8].

5.2. Genetic Algorithms

Genetic Algorithms (GAs) are meta-heuristic optimization techniques inspired by the principles of natural evolution. They belong to the broader class of evolutionary algorithms and are particularly suitable for solving complex optimization problems with large, non-linear search spaces, where analytical solutions are difficult or impractical to obtain [9]. Rather than relying on explicit mathematical models, GAs evaluate candidate solutions empirically, making them well suited for complex systems such as Kubernetes, where parameters including replica count, CPU limits, and memory allocation interact in

non-linear ways. Figure 2 illustrates the general workflow of a genetic algorithm.

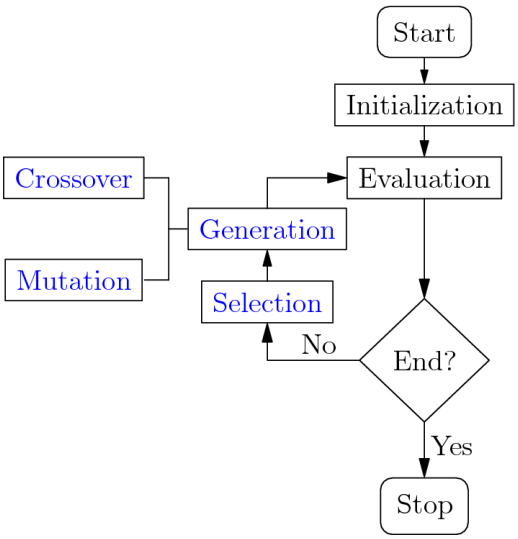

Figure 2. General workflow of a genetic algorithm. Adapted from Evolutionary computation for solving search-based data analytics problems [10].

5.2.1. Single-Objective Genetic Algorithm

A single-objective Genetic Algorithm (SO-GA) optimizes a single scalar fitness function $F : \mathcal{S} \to \mathbb{R}$, where $\mathcal{S}$ denotes the search space of candidate solutions. The algorithm maintains a population of N individuals, each representing a candidate solution encoded as a chromosome. At every generation, individuals are evaluated, selected, recombined, and mutated to produce the next generation, iterating until a stopping criterion is met (typically a fixed number of generations or a convergence threshold).

Selection determines which individuals contribute genetic material to the next generation. Tournament selection is among the most widely used mechanisms: k individuals are sampled uniformly at random from the population, and the one with the highest fitness is selected as a parent. This process is repeated until a sufficient number of parents are chosen. Tournament selection provides implicit fitness scaling and allows tuning selection pressure through the tournament size k [11].

Crossover (or recombination) combines genetic information from two parent individuals to produce one or more offspring. Common operators include single-point crossover, which swaps chromosome segments at a randomly chosen position, and uniform crossover, which independently inherits each gene from one of the two parents with equal probability. Crossover is applied with probability p_c, typically in the range $[0.6, 0.9]$, and is the primary mechanism for exploiting promising regions of the search space by combining partial solutions [1].

Mutation introduces random perturbations to individual genes, enabling the algorithm to explore regions of the search space that may not be reachable through crossover alone. For discrete or integer-encoded chromosomes, mutation typically replaces a gene with a randomly sampled value from its domain. Mutation is applied with probability p_m,

usually kept small (e.g., 0.01 to 0.3 per gene) to avoid disrupting well-adapted individuals while maintaining population diversity [1].

Elitism ensures that the best individual(s) from the current generation are preserved unchanged into the next generation, preventing the loss of high-quality solutions due to stochastic selection or destructive crossover. Elitism is a simple yet effective mechanism that guarantees monotonic improvement of the best fitness value across generations [11].

When multiple conflicting objectives exist — as is common in resource optimization, where latency, cost, and reliability compete — a single-objective GA requires aggregating these objectives into a scalar fitness function [1]. The most common approach is weighted linear aggregation:

$$F(\mathbf{x}) = \sum_{i=1}^{M} w_i \cdot f_i(\mathbf{x}), \quad \text{subject to} \quad \sum_{i=1}^{M} w_i = 1, \tag{1}$$

where $f_i(\mathbf{x})$ is the i-th objective (normalized to a common scale) and w_i is its corresponding weight. This formulation reduces a multi-objective problem to a single-objective one, enabling the use of standard GA operators and selection mechanisms.

Although weighted aggregation is straightforward to implement, it exhibits well-documented limitations [1, 2]. First, the solution obtained depends strongly on the choice of weights, which encodes an implicit preference among objectives that may be difficult to justify a priori. Second, the Pareto front of a multi-objective problem may contain non-convex regions that cannot be reached by any weighted sum, regardless of the weights chosen. Third, configurations representing meaningful trade-offs between objectives are collapsed into a single scalar value and may be discarded by selection pressure if they do not maximize the weighted sum. These limitations motivate the use of multi-objective evolutionary algorithms, such as NSGA-II, which return a set of non-dominated solutions rather than a single optimum.

5.2.2. NSGA-II

In many real-world optimization problems, including Kubernetes resource configuration, performance objectives are inherently conflicting. Reducing latency often requires additional computational resources, whereas minimizing resource consumption may degrade application performance. Multi-objective evolutionary algorithms address this challenge by optimizing each objective independently and searching for a set of non-dominated solutions rather than a single optimum [1].

Given two candidate solutions x and y in a minimization problem with M objectives, x dominates y (denoted $x \prec y$) if it is no worse than y in every objective and strictly better in at least one. A solution that is not dominated by any other feasible solution is called *Pareto-optimal*, and the set of all Pareto-optimal solutions forms the *Pareto front*. Rather than identifying a single optimal solution, multi-objective optimizers seek to approximate the Pareto front, which represents the set of non-dominated trade-off solutions.

Figure 3 illustrates the Pareto front for a bi-objective minimization problem, where

the preference vector is shown only to demonstrate that different optimization preferences may lead to different Pareto-optimal solutions.

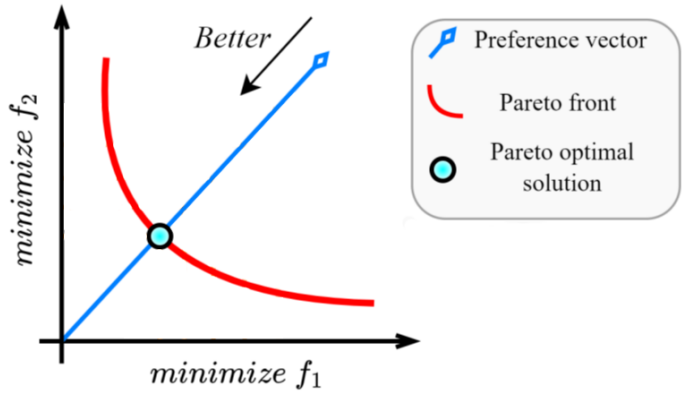

Figure 3. Pareto front for a bi-objective minimization problem. Adapted from [12].

The Non-dominated Sorting Genetic Algorithm II (NSGA-II), proposed by Deb et al. [2], is one of the most widely adopted evolutionary algorithms for multi-objective optimization. Its effectiveness stems from three key mechanisms: (i) fast non-dominated sorting, which partitions the population into successive Pareto fronts according to dominance relationships; (ii) crowding distance, which estimates the density of solutions within each Pareto front, favoring individuals located in less crowded regions to preserve diversity; and (iii) elitist replacement, which combines the parent and offspring populations before selecting the next generation according to Pareto rank and crowding distance. Selection in NSGA-II is performed using binary tournament selection, in which solutions with lower Pareto rank are preferred and ties are broken using crowding distance. The algorithm has a computational complexity of $\mathcal{O}(MN^2)$ per generation, where N is the population size and M is the number of objectives. Figure 4 illustrates the evolution of the population during the optimization process, where non-dominated solutions progressively converge toward the Pareto front while maintaining diversity through the crowding-distance mechanism.

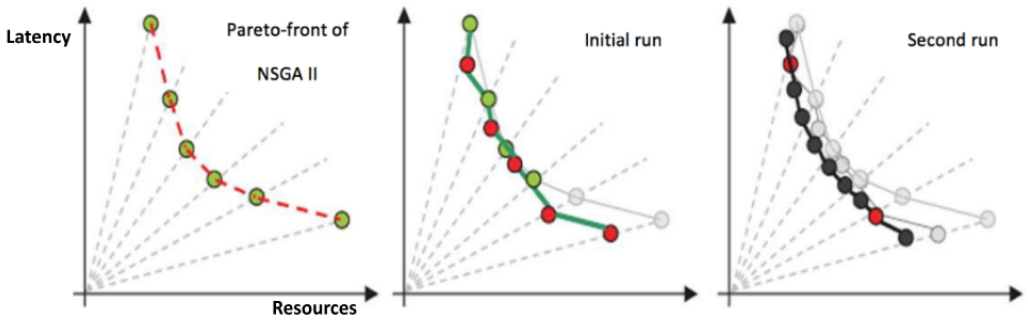

Figure 4. NSGA-II population evolution toward the Pareto front.Adapted from Pareto Front Shape-Agnostic Pareto Set Learning in Multi-Objective Optimization [12].

Comparing multi-objective optimizers is non-trivial because they return sets of

solutions rather than single points. The literature has established a small set of unary indicators that are widely accepted for empirical comparison [13]: (i) Hypervolume (HV), which measures the volume of objective space dominated by the obtained Pareto front with respect to a reference point, where larger values indicate better convergence and diversity; (ii) Inverted Generational Distance (IGD), which measures the average distance between a reference Pareto front and the obtained front, where lower values indicate better convergence and coverage; and (iii) Cardinality, which represents the number of distinct non-dominated solutions produced by the optimizer. These indicators are employed in this work to compare the quality of the Pareto fronts generated by NSGA-II against the projected solutions obtained from the single-objective GA and the Random Search baseline.

5.3. Prometheus

Prometheus is an open-source monitoring system widely adopted in cloud-native environments [14]. It periodically collects metrics from instrumented targets and stores them as time-series data, enabling continuous observation of system behavior during application execution. In Kubernetes environments, Prometheus collects metrics exposed by containers and cluster components through HTTP endpoints. These metrics are queried using PromQL, allowing aggregation and time-window analysis during controlled workload executions. In this work, Prometheus provides resource-level metrics such as CPU utilization, memory consumption, and CPU throttling, which complement application-level measurements when evaluating the quality of candidate resource configurations. The experimental application was implemented using FastAPI, a lightweight Python framework suitable for performance evaluation. It exposes endpoints that generate controlled CPU- and memory-intensive workloads, allowing different resource demands to be reproduced consistently across experiments. Application-level metrics such as throughput, latency, and error rate reflect the quality of service perceived by users, while resource-level metrics describe infrastructure utilization. In this work, both categories are collected during controlled load tests and jointly used to evaluate the fitness of each candidate configuration.

6. Related Work

Guerrero et al. [8] propose a multi-objective evolutionary optimization approach based on NSGA-II to address container allocation and elasticity management in cloud architectures. Their model explicitly targets multiple competing objectives, including tighter resource usage, reduced network overhead, and lower system failure rate. A key contribution is the formulation of the container allocation problem as a multi-objective optimization task and the use of NSGA-II to explore trade-offs across objectives, demonstrating that genetic algorithms can effectively optimize multiple objectives in resource management problems. However, the validation relies on workload and system models rather than direct evaluation inside a live Kubernetes cluster, abstracting runtime aspects such as real scheduling overhead, pod lifecycle transitions, and deployment stabilization behavior. The present work addresses this limitation by evaluating candidate configurations directly in a running Kubernetes cluster.

Gupta et al. [5] investigate autoscaling strategies combining reactive and proactive mechanisms under SLA constraints. Their work reviews hybrid autoscaling approaches

and presents an experimental case study integrated into Kubernetes, evaluated in an environment that includes realistic workload patterns, a microservice testbed (DeathStarBench), and monitoring via Prometheus. The paper illustrates a reproducible and experimentally grounded methodology for autoscaling evaluation: defining workload generation procedures, specifying cluster configuration, selecting baselines (including Kubernetes HPA), and comparing performance outcomes under controlled conditions. Unlike the present work, which optimizes deployment configurations using evolutionary algorithms, Gupta et al. focus on improving autoscaling decisions through hybrid machine learning techniques; nevertheless, their experimental methodology provides an important reference for workload generation and performance evaluation adopted in this thesis.

Xiong et al. [9] investigate the application of genetic algorithms for resource optimization in cloud computing environments. Their work demonstrates the effectiveness of evolutionary approaches for exploring non-linear configuration spaces in virtualized infrastructures, where the relationship between resource allocation and performance is difficult to model analytically. While their approach validates the potential of genetic algorithms for cloud resource management, the evaluation focuses on virtual machine allocation rather than container orchestration. The present work extends this line of research by applying evolutionary optimization specifically to Kubernetes Deployments, jointly optimizing replica count, CPU limits, and memory allocation.

Vadisetty et al. [7] survey recent AI-based orchestration techniques for Kubernetes, including reinforcement learning, predictive analytics, and intelligent scheduling mechanisms. The survey highlights the growing adoption of artificial intelligence for cluster management while also indicating that evolutionary approaches remain comparatively underexplored. Unlike the present work, however, it does not propose or experimentally evaluate a concrete optimization methodology. This thesis contributes a reproducible optimization pipeline and performs a controlled comparison between a single-objective Genetic Algorithm, NSGA-II, and a Random Search baseline under the same search space, evaluation budget, workload, and monitoring infrastructure, enabling a direct assessment of whether the additional complexity of multi-objective optimization translates into measurable improvements for Kubernetes resource configuration.

Beyond the works directly related to evolutionary optimization in container environments, several other studies address resource management in cloud-native systems using different approaches. Reinforcement learning-based methods have been explored for autoscaling decisions [15], where agents learn scaling policies through interaction with the environment; however, these approaches typically require extensive training episodes and may not generalize well across different workload patterns. Bayesian optimization has been applied to hyperparameter tuning in cloud services [3], offering sample-efficient exploration but often struggling with the discrete, high-dimensional configuration spaces typical of Kubernetes deployments. Rule-based and threshold-driven autoscalers, such as the Kubernetes Horizontal Pod Autoscaler (HPA), remain the most widely deployed solutions in production environments [6]; while practical and interpretable, they optimize only the replica dimension and cannot jointly tune CPU and memory limits. Analytical modeling approaches attempt to derive optimal configurations from queueing-theoretic or performance models [16], but their accuracy depends heavily on the fidelity of the underlying assumptions, which may not hold in complex microservice architectures with

non-linear performance characteristics. These alternative approaches, while not directly comparable to the evolutionary methodology proposed here, provide important context for understanding the landscape of resource optimization techniques and highlight the trade-offs between sample efficiency, generalization, and practical deployability that motivate the present experimental study.

Table 1 summarizes the key characteristics and differentiators of the related works discussed in this section, highlighting how the present work addresses gaps in existing approaches.

Table 1. Comparison of related works on resource optimization in container/cloud environments.

Work	Approach	Live Cluster	Multi-Objective	Controlled Comparison	Reproducible Pipeline
Guerrero et al. [8]	NSGA-II	No (simulation)	Yes	No	Partial
Gupta et al. [5]	Hybrid ML + HPA	Yes	No	Yes (vs. HPA)	Yes
Xiong et al. [9]	Genetic Algorithm	No (VM-level)	No	No	No
Vadisetty et al. [7]	Survey (RL, ML)	N/A	N/A	N/A	N/A
This work	SO-GA, NSGA-II, RS	**Yes**	**Yes**	**Yes (3 algorithms)**	**Yes**

The differentiators of the present work are: (i) direct integration with a live Kubernetes cluster rather than simulation or analytical models; (ii) a controlled experimental comparison of three optimization strategies (SO-GA, NSGA-II, Random Search) under identical conditions; (iii) a multi-objective formulation that preserves trade-offs alongside a single-objective baseline; and (iv) a fully reproducible pipeline with open-source code, raw data, and analysis scripts.

7. Methodology

Before describing the experimental components in detail, this section provides a high-level overview of the evaluation pipeline shared by all three optimization algorithms. Each candidate configuration — a tuple specifying the number of replicas, CPU limit, and memory limit — is submitted to the Kubernetes cluster through its API. The cluster applies the configuration by patching the target Deployment, and the system waits until all pods reach a ready state. A controlled load test is then executed against the application, generating sustained traffic for a fixed duration while both application-level metrics (throughput, latency, error rate) and infrastructure-level metrics (CPU usage, memory consumption, CPU throttling) are collected. Once the load test completes, the collected metrics are used to compute fitness values: a scalar score for the single-objective GA and a vector of three objectives for NSGA-II and Random Search. This evaluation pipeline remains identical across all algorithms and all runs, ensuring that observed differences in solution quality can be attributed solely to the search strategy rather than to variations in the measurement process. The subsections that follow describe each component of this pipeline in detail, including the system architecture, search space representation, algorithm-specific operators, fitness formulations, and experimental procedure.

This work proposes an experimental approach for optimizing resource allocation in a Kubernetes cluster using two evolutionary strategies — a single-objective Genetic Algorithm (SO-GA) and the multi-objective NSGA-II — and a non-evolutionary control given by a Random Search (RS) baseline. The three algorithms share the same evaluation pipeline, the same discretized search space, and the same evaluation budget, so that any difference in the obtained results can be attributed to the search strategy itself.

The proposed approach integrates five main components: (i) a Kubernetes cluster executing a containerized application, (ii) three optimizers (SO-GA, NSGA-II, and Random Search) responsible for exploring the configuration space, (iii) a shared evaluation pipeline that applies a candidate configuration, waits for rollout, and triggers a controlled load test, (iv) a load testing mechanism that generates reproducible workloads against the application, and (v) a monitoring infrastructure based on Prometheus to collect performance and resource utilization metrics.

Figure 5 display the proposed architecture. Any of the three optimizers (SO-GA, NSGA-II, Random Search) applies Kubernetes configurations through the same evaluation pipeline, triggers controlled load tests, collects application-level metrics from the load test and infrastructure-level metrics from Prometheus, and persists results to a PVC.

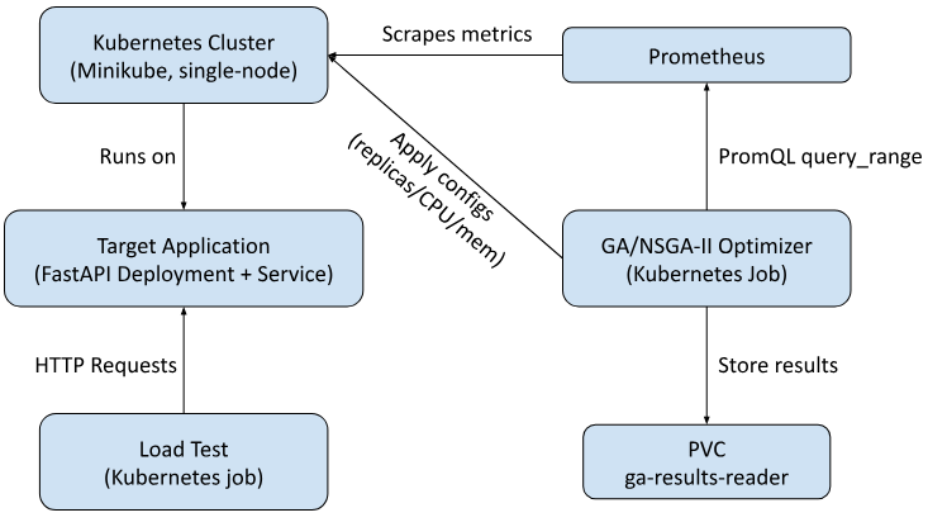

Figure 5. Proposed architecture.

7.1. System Architecture

The experimental environment consists of a single-node Kubernetes cluster deployed using Minikube, configured with 8 vCPUs, 19 GiB of RAM, and 40 GiB of disk. The cluster hosts the target application, implemented using the FastAPI framework, alongside the monitoring stack (kube-prometheus-stack) and the optimizers. Prometheus is used to collect container-level metrics, while application-level performance metrics are obtained directly from the load testing results.

The choice of a single-node cluster is deliberate and methodologically motivated: this work investigates the relative behavior of three search strategies under *identical* cluster conditions, not the absolute performance of any single configuration in a production-

grade environment. A single-node deployment eliminates several confounding factors that would otherwise be difficult to control — inter-node network variance, scheduler placement asymmetries, and the non-determinism introduced by multi-zone disk and network I/O — allowing any observed difference between SO-GA, NSGA-II and Random Search to be attributed to the search strategy rather than to differences in the underlying infrastructure. The trade-off, namely a search space whose upper bound on the number of replicas is constrained by the node's available capacity, is discussed in the limitations and motivates one of the future-work directions in the conclusion.

After accounting for the resource overhead of system components — `kube-system` pods ($\sim$0.5 vCPU, 1 GiB), the kube-prometheus-stack ($\sim$1.5 vCPU, 3 GiB), the optimizer Job itself ($\sim$0.5 vCPU, 0.5 GiB), and auxiliary pods ($\sim$0.7 vCPU, 1 GiB) — approximately 5.3 vCPUs and 14 GiB are available for the target application's replicas during evaluation. This budget directly constrains the upper bounds of the search space.

The deployment of the target application is configured with a `RollingUpdate` strategy parameterized by `maxSurge=1` and `maxUnavailable=1`, in order to bound the peak number of co-existing pods during configuration changes (i.e., $r + 1$ at most). Each optimizer is executed as a Kubernetes Job, ensuring that each optimization run is isolated, reproducible, and independent from previous executions. Configuration changes generated by the optimizers are applied directly to the cluster by patching the Deployment through the Kubernetes API.

7.2. Search Space and Representation

Each individual in the population (or each sample in the case of Random Search) represents a candidate Kubernetes configuration encoded as the tuple:

$$I = (r, c, m)$$

where r is the number of replicas, c is the CPU limit (in millicores) per container, and m is the memory limit (in MiB) per container. To ensure feasibility within the cluster's physical constraints and to standardize the encoding across all three algorithms, the search space is discretized as follows:

- $r \in \{1, 2, 3, 4\}$ (4 values),
- $c \in \{100, 150, 200, \ldots, 700\}$ m, with step 50 m (13 values),
- $m \in \{128, 256, 384, \ldots, 1024\}$ MiB, with step 128 MiB (8 values).

The resulting space contains $4 \times 13 \times 8 = 416$ valid configurations. The upper bound on c was chosen so that the worst-case rollout (four replicas plus one surge pod under `maxSurge=1`) consumes at most $5 \times 700 = 3{,}500$ m of CPU, fitting within the available 5.3 vCPU budget. Likewise, $5 \times 1024 = 5{,}120$ MiB of memory fits comfortably within the 14 GiB budget. The bounds on the lower end ($r = 1$, $c = 100$ m, $m = 128$ MiB) were intentionally chosen to include severely under-provisioned configurations, providing the search algorithms with clearly-degraded reference points.

The choice of 416 configurations reflects a balance between exploration capacity and evaluation budget: with a budget of 96 evaluations per run (see Section on Experimental Procedure), search coverage is approximately 23.1%, leaving substantial space

for the evolutionary operators to drive diversity rather than exhaustively enumerating the space.

7.3. Single-Objective Genetic Algorithm (SO-GA)

The single-objective Genetic Algorithm follows a standard evolutionary workflow composed of initialization, evaluation, selection, crossover, mutation, and elitism. Initially, a population of individuals is randomly generated within the predefined bounds. For each individual, the corresponding configuration is applied to the Kubernetes Deployment. Once all pods reach the *Running* state, a load test is executed against the application for a fixed duration. After the load test completes, performance and resource metrics are collected and used to compute the scalar fitness value of the individual using the aggregated fitness function described in the next subsection. Selection is performed using tournament selection, promoting individuals with higher fitness while preserving diversity within the population. Uniform crossover and bounded random mutation are applied to generate new candidate solutions, and elitism is used to ensure that the best-performing individual of each generation is preserved into the next generation.

7.4. Multi-Objective Genetic Algorithm (NSGA-II)

The NSGA-II implementation follows the canonical formulation of [2], sharing with the SO-GA the same encoding ($I = (r, c, m)$), the same evaluation pipeline (apply configuration, wait for rollout, run load test, query Prometheus), and the same crossover and mutation operators (uniform crossover with probability p_c, per-gene mutation with probability p_m, followed by a `repair()` step that snaps any out-of-bounds gene back into the discretized search space). The two algorithms differ in three key ways:

1. Evaluation produces a vector, not a scalar. Each individual is associated with three objectives (f_1, f_2, f_3) to be minimized simultaneously, defined in the next subsection.
2. Selection uses Pareto rank and crowding distance. Binary tournament compares two candidates first by rank (lower is better), with ties broken by crowding distance (larger is better), preserving diversity along the Pareto front.
3. Replacement is elitist via $P \cup Q$. The combined parent and offspring populations are sorted into successive Pareto fronts, and the next generation is filled by adding entire fronts in order of rank, until exactly N individuals are selected; the last front, if it cannot be added in full, is truncated by crowding distance.

Failed or timed-out evaluations are assigned penalty objectives ($f_1 = 10.0$, $f_3 = 0.0$, with f_2 computed as usual), so that they are dominated by any successful evaluation and removed from the Pareto front in subsequent generations.

7.5. Random Search Baseline

The Random Search baseline is included as a non-evolutionary control. It uniformly samples $N \times G$ configurations from the same discretized search space, evaluates each one through the same pipeline used by SO-GA and NSGA-II, and reports the multi-objective values (f_1, f_2, f_3) for every sample. The Pareto front of the resulting set is then extracted and compared against the front returned by NSGA-II.

Algorithm 1: Single-Objective GA Optimization Loop in Kubernetes

Input : Population size N, generations G, search bounds for replicas/CPU/memory, load test parameters (warm-up + steady-state duration), and monitoring endpoint (Prometheus).

Output: Best configuration $I^\star = (r, c, m)$ and execution report stored in PVC.

Step 1: Initialization. Randomly generate N individuals to form the initial population P_0.

for $t \leftarrow 0$ **to** $G - 1$ **do**

- **Step 2: Evaluation. foreach** $I \in P_t$ **do**
  - **2.1 Apply configuration.** Patch Kubernetes Deployment with (r, c, m) from I.;
  - **2.2 Wait for steady state.** Wait until pods are *Ready*.;
  - **2.3 Warm-up.** Execute low-intensity requests for a fixed duration.;
  - **2.4 Load test.** Execute steady workload for a fixed duration.;
  - **2.5 Collect metrics.** Query Prometheus using `query_range` over the test window.;
  - **2.6 Fitness assignment.** Compute $F(I)$ from latency, efficiency, and reliability.;
  - Store evaluation results (metrics, fitness) to persistent storage.;
- **Step 3: Selection.** Select individuals from P_t based on fitness (tournament selection) to form a parent set.;
- **Step 4: Crossover.** Generate offspring via crossover to form an offspring population Q_t.;
- **Step 5: Mutation.** Mutate individuals in Q_t with probability p_m.;
- **Step 6: Elitism.** Copy the best individual(s) from P_t into the next generation.;
- **Step 7: Next generation.** Set $P_{t+1} \leftarrow Q_t$.;

Step 8: Output. Return the best individual $I^\star$ observed and apply it to the cluster.;

By design, Random Search has no notion of selection, crossover, or mutation, and therefore benefits from no information transfer between samples. Comparing it with NSGA-II under an identical evaluation budget tests whether the evolutionary mechanisms (rank-based selection, crowding distance, elitist replacement) provide measurable improvement over uniform sampling in this domain.

7.6. NSGA-II Objective Functions

The NSGA-II formulation expresses three objectives, all to be minimized. The three objectives are conflicting in the typical case: lowering f_2 (using fewer resources) tends to increase f_1 (more throttling, higher mem peak) and also tends to reduce the achievable throughput, increasing f_3. NSGA-II therefore returns a set of non-dominated configurations that represent the best observed trade-offs across these three axes.

Algorithm 2: NSGA-II Optimization Loop in Kubernetes

Input : Population size N, generations G, search bounds, crossover probability p_c, per-gene mutation probability p_m, load test parameters, monitoring endpoint.

Output: Final Pareto front $\mathcal{P}$ and execution report stored in PVC.

Step 1: Initialization. Randomly generate N individuals to form P_0 and evaluate them, computing (f_1, f_2, f_3) for each.;

Step 2: Classification. Apply fast non-dominated sorting on P_0 and assign rank and crowding distance.;

for $t \leftarrow 0$ **to** $G - 1$ **do**

  Step 3: Generate offspring Q_t**.**;

  while $|Q_t| < N$ **do**

    Select two parents p_1, p_2 from P_t via binary tournament on (rank, crowding distance);

    Apply uniform crossover with probability p_c, then per-gene mutation with probability p_m, and `repair()` both children;

    Add the children to Q_t and evaluate them through the same pipeline used by SO-GA, computing (f_1, f_2, f_3).

  Step 4: Combine and rank. Form $R_t = P_t \cup Q_t$ and apply fast non-dominated sorting on R_t.;

  Step 5: Elitist selection. Build P_{t+1} by appending Pareto fronts of R_t in order of rank; if the last front does not fit entirely, select its highest-crowding-distance individuals to complete N.;

  Step 6: Persist. Save the full ranked population, the rank-0 (Pareto) front, and the cache of evaluated genomes.;

Step 7: Output. Return the rank-0 individuals of P_G as the approximated Pareto front $\mathcal{P}$.;

7.6.1. Saturation (f_1)

This objective captures how pressured the running pods are, by combining the CPU throttling rate and the memory peak ratio into a single $[0, 1]$ score:

$$f_1(I) = w_T \cdot T_{cpu} + w_P \cdot \rho_{mem}, \tag{2}$$

where $T_{cpu} \in [0, 1]$ is the fraction of CFS scheduler periods that were throttled (computed as `rate(container_cpu_cfs_throttled_periods_total)` divided by `rate(container_cpu_cfs_periods_total)` over the load test window), $\rho_{mem} \in [0, 1]$ is the ratio between peak memory working set and the configured memory limit, and $w_T = w_P = 0.5$ in this work.

7.6.2. Provisioned Resources (f_2)

This objective quantifies the cost of the configuration as a weighted sum of provisioned CPU and memory, multiplied by the number of replicas:

$$f_2(I) = r \cdot \left(w_c \cdot \frac{c}{1000} + w_m \cdot \frac{m}{1024} \right), \tag{3}$$

with $w_c = w_m = 1$ in this work, so that one CPU core is treated as equivalent in cost to one GiB of memory. This formulation rewards configurations that provide the same service quality with fewer replicas or tighter resource limits.

7.6.3. Negative Throughput (f_3)

The throughput in requests per second, $\theta(I)$, is measured directly from the load test:

$$f_3(I) = -\theta(I). \tag{4}$$

Negation is applied so that all three objectives are minimized in the same direction, which simplifies the implementation of the dominance relation.

7.7. Workload Generation and Empirical Calibration

A synthetic workload is generated by an HTTP-based load tester that sends concurrent requests to the `/mixed` endpoint of the target application. This endpoint executes a CPU-intensive operation (computing the factorial of 100) followed by a memory-intensive allocation (a list of one million floating-point values, approximately 8 MiB), ensuring that the application stresses both CPU and memory dimensions simultaneously.

Each evaluation consists of a warm-up phase (15 s with 5 concurrent workers) followed by a steady-state load phase (90 s with 22 concurrent workers). Metrics are collected only during the steady-state phase, with Prometheus queries time-windowed to the exact start and end timestamps reported by the load tester. The workload parameters remain constant across all evaluations of all three algorithms to ensure comparability.

The concurrency level was determined empirically by running standalone load tests at the two extremes of the search space and verifying that they produced clearly distinguishable behavior. At the lower bound ($r = 1$, $c = 100$ m, $m = 128$ MiB), the configuration consistently collapses (success rate $\approx$1%, p95 latency above the SLA, near-zero useful throughput), confirming that the search space includes meaningfully under-provisioned configurations. At the upper bound ($r = 4$, $c = 700$ m, $m = 1024$ MiB), the configuration sustains a 100% success rate and a stable throughput of approximately 70 requests per second, with moderate CFS throttling that reflects the compute-bound nature of the `/mixed` endpoint rather than insufficient provisioning. An initial concurrency of 30 produced excessive saturation even at the upper bound (CFS throttle of 0.72) and was reduced to 22, after which the upper bound exhibits a clearly less saturated behavior (CFS throttle of 0.70 with a 35% reduction in tail latency, indicating reduced queueing). This calibration ensures that the load test is neither so light that all configurations appear unsaturated, nor so heavy that all configurations collapse, preserving the resolution of f_1 and f_3 across the search space.

7.8. Monitoring and Metrics Collection

Metrics are queried using time-bounded PromQL queries aligned with the exact evaluation window of each individual, ensuring that collected data corresponds exclusively to the current configuration. Application-level metrics (throughput, average latency, and tail latency) are computed directly from the load test results, while infrastructure-level metrics (CPU usage, memory usage, CPU throttling) are obtained from Prometheus.

7.9. SO-GA Fitness Function

In the single-objective formulation, each individual $I = (r, c, m)$ is applied to the running cluster and evaluated under a controlled workload, and a scalar fitness $F(I) \in [0, 1]$ is computed by combining application-level performance metrics (from the load test) and infrastructure-level metrics (from Prometheus). This aggregation strategy is commonly adopted in cloud resource optimization studies [17, 15], but it depends on a fixed choice of weights, which is one of the limitations addressed by the multi-objective formulation introduced in the previous subsection.

Throughput is incorporated implicitly into the resource efficiency component as *performance per consumed resource*. This avoids double counting and aligns the optimization objective with the primary goal of this study, namely resource optimization.

The final fitness score $F(I)$ is computed as a weighted sum of three normalized components: latency, resource efficiency, and reliability:

$$F(I) = w_L \cdot S_L(I) + w_E \cdot S_E(I) + w_R \cdot S_R(I) \tag{5}$$

subject to:

$$w_L + w_E + w_R = 1. \tag{6}$$

In this work, the weights are:

$$w_L = 0.35, \qquad w_E = 0.40, \qquad w_R = 0.25. \tag{7}$$

The relative magnitude of the three weights reflects the priorities of this work, which targets resource optimization in Kubernetes rather than pure quality-of-service maximization. Resource efficiency receives the largest weight ($w_E = 0.40$) because it captures the core objective of the study: the ability of the optimizer to find configurations that deliver useful work per provisioned CPU and memory unit. Latency is given the second-largest weight ($w_L = 0.35$) because under-provisioned configurations primarily manifest as increased tail latency (queuing, CFS throttling), and penalising it sufficiently prevents the SO-GA from collapsing onto the cheapest corner of the search space, which would trivially maximise efficiency but at the cost of unusable service. Reliability is given the smallest weight ($w_R = 0.25$) because, in the calibrated workload, configurations that produce abnormal error rates already incur very low efficiency and latency scores (their throughput drops and their tail latency explodes), making reliability partially redundant with the other two components; the term is nonetheless kept as an explicit safeguard against configurations that satisfy latency targets by silently dropping requests. The weights are normalized so that $w_L + w_E + w_R = 1$, keeping $F(I) \in [0, 1]$ regardless of the weighting profile, which simplifies the comparison of fitness values across runs.

7.9.1. Latency Score

Latency is a key quality-of-service metric. Tail latency percentiles (p95 and p99) are widely used in distributed systems because they better capture user-perceived performance than mean latency alone [18, 16]. Let L_{avg}, L_{95}, and L_{99} denote the average, 95th percentile, and 99th percentile latency, measured during the steady-state phase of the load test (in milliseconds). A smooth normalization based on an implicit SLA threshold L_{SLA} is applied [16]:

$$\phi(L) = \frac{1}{1 + \frac{L}{L_{SLA}}}. \tag{8}$$

The latency score is computed as:

$$S_L(I) = 0.2\,\phi(L_{avg}) + 0.4\,\phi(L_{95}) + 0.4\,\phi(L_{99}). \tag{9}$$

This weighting emphasizes tail latency ($p95$ and $p99$), consistent with recommendations from large-scale distributed systems research [18].

7.9.2. Resource Efficiency Score

Resource efficiency captures the trade-off between achieved performance and resource usage [16]. Let θ denote the measured throughput in requests per second (req/s), obtained from the load test. Let U_{cpu} denote average CPU usage in cores, and let U^B_{mem} denote average memory working set in bytes, both obtained from Prometheus during the evaluation window. Memory usage is converted to MiB (mebibytes, 2^{20} bytes) as:

$$U_{mem} = \frac{U^B_{mem}}{1024^2}. \tag{10}$$

7.9.3. Productivity per consumed resource

Productivity is modeled as throughput per consumed resource [16], with a small constant ε to avoid division by zero:

$$P_{cpu}(I) = \frac{\theta}{U_{cpu} + \varepsilon}, \qquad P_{mem}(I) = \frac{\theta}{U_{mem} + \varepsilon}. \tag{11}$$

To bound productivity values in $[0, 1]$ without requiring global maxima, the following squashing function is applied:

$$\psi(z) = \frac{z}{z + 1}. \tag{12}$$

Thus:

$$S_{cpu} = \psi(P_{cpu}(I)), \qquad S_{mem} = \psi(P_{mem}(I)). \tag{13}$$

A weighted productivity score is computed as:

$$S_P(I) = 0.7\,S_{cpu} + 0.3\,S_{mem}. \tag{14}$$

7.9.4. Utilization quality

To discourage both underutilization and saturation, the utilization ratios are defined as:

$$u_{cpu} = \frac{U_{cpu}}{c}, \qquad u_{mem} = \frac{U_{mem}}{m}, \tag{15}$$

with $u_{cpu}, u_{mem} \in [0, 1]$ (clamped). A utilization-quality function $q(u)$ is defined with thresholds $u_{low} = 0.3$, $u_{high} = 0.9$, and target $u^\star = 0.6$:

$$q(u) = \begin{cases} \frac{u}{u_{low}}, & u < u_{low} \\ 1 - \frac{|u-u^\star|}{(u_{high}-u_{low})/2}, & u_{low} \le u \le u_{high} \\ \frac{1-u}{1-u_{high}}, & u > u_{high}. \end{cases} \tag{16}$$

The utilization score is computed as:

$$S_U(I) = 0.6\, q(u_{cpu}) + 0.4\, q(u_{mem}). \tag{17}$$

7.9.5. Penalties: CPU throttling and memory peak

CPU throttling is penalized using a smooth decay function [17]:

$$\text{ThrottlePenalty}(I) = \frac{1}{1 + 5\, T_{cpu}}, \tag{18}$$

where T_{cpu} is the CPU throttling rate (throttled seconds per second) obtained from Prometheus.

To capture memory risk not visible in averages, peak memory usage is included. Let $U^B_{mem,peak}$ be the peak memory working set in bytes, converted to MiB as:

$$U_{mem,peak} = \frac{U^B_{mem,peak}}{1024^2}. \tag{19}$$

Let the peak ratio be:

$$\rho_{mem} = \frac{U_{mem,peak}}{m}. \tag{20}$$

The peak penalty is:

$$\text{MemPeakPenalty}(I) = \begin{cases} 1, & \rho_{mem} \le 0.9 \\ \frac{1}{1+10(\rho_{mem}-0.9)}, & \rho_{mem} > 0.9. \end{cases} \tag{21}$$

7.9.6. Final efficiency score

The efficiency score combines productivity and utilization quality [16]:

$$S_{base}(I) = 0.55\, S_P(I) + 0.45\, S_U(I), \tag{22}$$

and applies the penalties:

$$S_E(I) = S_{base}(I) \cdot \text{ThrottlePenalty}(I) \cdot \text{MemPeakPenalty}(I). \tag{23}$$

Finally, $S_E(I)$ is clamped to $[0, 1]$.

7.9.7. Reliability Score

Reliability reflects the ability of the system to successfully process requests under load [16]. Let S denote the success rate in $[0, 1]$ and let E denote the error rate (fraction of failed requests). A smooth penalty discourages configurations that yield errors:

$$\text{ErrorPenalty}(I) = \frac{1}{1 + 20E}. \tag{24}$$

The reliability score is computed as:

$$S_R(I) = S \cdot \text{ErrorPenalty}(I). \tag{25}$$

7.9.8. Final Fitness Function

Combining all components, the final fitness score is:

$$F(I) = 0.35\, S_L(I) + 0.40\, S_E(I) + 0.25\, S_R(I). \tag{26}$$

This formulation prioritizes resource efficiency while maintaining quality-of-service (via tail latency) and correctness (via reliability), consistent with performance-aware optimization practices [17, 18].

7.10. Experimental Procedure

The evaluation methodology is designed to enable a fair, reproducible comparison between the three optimizers. The full source code, Kubernetes manifests, raw experimental data and analysis scripts used in this work are publicly available at `https://github.com/Jurupoc/k8s-ga-optimizer`. All algorithms run on the same Kubernetes cluster, against the same target application, with the same evaluation pipeline and the same calibrated workload (90 s steady-state at 22 concurrent workers, preceded by a 15 s warm-up at 5 workers). All three algorithms persist their evaluations to an append-only cache (`cache.jsonl`) keyed by the genome plus the serialized load parameters; because the key includes the load parameters, any change in the workload (duration, concurrency, endpoint, profile) automatically invalidates stale measurements, while configurations that recur within or across runs are amortized.

All three algorithms are configured with an identical budget of 96 evaluations per run. For SO-GA and NSGA-II this corresponds to a population size of $N = 12$ and $G = 8$ generations. For Random Search this corresponds to 96 uniformly-sampled configurations. The shared budget is the central control of this experiment: any difference in solution quality between algorithms must therefore be attributable to the search strategy rather than to a difference in compute allowance. At an average evaluation cost of ≈ 268 s on the cluster (rollout + 15 s warm-up + 90 s load test + metric collection), a single run takes roughly 3 to 4 hours of wall-clock time depending on the cache hit rate.

The algorithm-specific hyperparameters used in all runs are summarized in Table 2. The crossover and mutation rates of the SO-GA ($p_c = 0.8$, $p_m = 0.3$) and the

Table 2. Algorithm hyperparameters used in all experimental runs.

Parameter	SO-GA	NSGA-II	Random Search
Population size N	12	12	—
Generations G	8	8	—
Samples (one-shot)	—	—	96
Crossover probability p_c	0.8	0.9	—
Mutation probability p_m	0.3	0.2 (per gene)	—
Tournament size	2	2	—
Elitism	top 1 individual	$P \cup Q$ elitist	—
Duplicate elimination	—	yes (genome-level)	—
Seeds	$\{42, 43, 44\}$	$\{42, 43, 44\}$	$\{42, 43, 44\}$
Total evaluations per run	96	96	96

NSGA-II ($p_c = 0.9$, $p_m = 0.2$) reflect the defaults recommended in the original literature for each algorithm; they were not tuned per problem to avoid biasing the comparison through differential hyperparameter optimization.

Before the optimization runs, the workload is calibrated against the two extremes of the search space, as described in the Workload Generation subsection. This calibration is not part of the optimization loop, but it is documented as part of the experimental protocol because the conclusions about each algorithm's effectiveness depend on the workload providing distinguishable behavior across the search space.

Each algorithm is executed as a Kubernetes Job with a fixed RNG seed to make individual runs reproducible. Between consecutive runs, the target Deployment is reset to a clean state (replicas $= 1$, no resource limits) by reapplying its base manifest, in order to avoid carry-over effects from the last evaluated configuration of the previous run. To support statistical comparison rather than single-run anecdote, each algorithm is repeated with $S = 3$ independent seeds (`42`, `43`, `44`). Per-seed results are reported individually, and inter-seed variability is summarized via the mean and standard deviation of each metric. Where applicable, a non-parametric Mann–Whitney U test is used to assess whether the observed differences between algorithms are statistically significant under this $n = 3$ repetition scheme.

Every run captures a provenance file (`env.json`) containing the Python interpreter version, the repository commit hash (with a dirty-tree flag), the versions of the runtime libraries, the active Kubernetes context, the host identifier and the experiment-specific environment variables (filtered to exclude secret-like keys). The RNG seed used by each algorithm is persisted in its `manifest.json`, and the per-generation checkpoint additionally stores the current population and the full Python RNG state, so that an interrupted run can be resumed deterministically from its last completed generation. Both the SO-GA and the NSGA-II write an append-only evaluation cache (`cache.jsonl`) keyed by genome together with the serialized load parameters, ensuring that any configuration re-encountered in a subsequent run is amortized and that the full set of raw measurements remains available for post-hoc inspection. The same applies to the Random Search baseline.

The three algorithms are compared along the following axes:

- Best aggregated fitness – For each algorithm, the SO-GA fitness $F(I)$ is computed for every evaluated configuration (including configurations evaluated by NSGA-II and Random Search via re-projection), and the maximum is reported. This evaluates each algorithm on the SO-GA's own scalar criterion. The re-projection is only well-defined because both algorithms persist the full set of raw infrastructure-level metrics required by F (in particular, CPU throttling rate and memory peak usage), exported as a long-format `evaluations.csv` that contains one row per evaluation across all generations.
- Pareto-front quality – For NSGA-II and Random Search, the obtained Pareto front is compared in terms of cardinality, hypervolume (computed via Monte Carlo with 10^6 samples in the 3-D objective space, using a reference point at $\max + 10\%$ of the observed maximum on each axis) and inverted generational distance (IGD), using the union of all algorithm fronts as a proxy for the true Pareto front [13]. The single SO-GA solution is also projected into the (f_1, f_2, f_3) space and tested for dominance against the NSGA-II Pareto front; this projection uses the same f_1, f_2, f_3 definitions employed by NSGA-II, applied to the metrics already recorded for every SO-GA evaluation.
- Search behavior – Convergence over generations (best fitness for SO-GA, hypervolume of rank-0 front for NSGA-II), population diversity, and the number of unique configurations explored.
- Computational cost – Total wall-clock time, total evaluation time (sum of per-evaluation durations on the cluster), algorithmic overhead (total minus evaluation time, isolating the cost of the search operators and bookkeeping from the cost of measuring fitness), average per-evaluation time, and cache hit rate. Reporting evaluation time and overhead separately enables identifying whether a given algorithm spends most of its budget on the cluster or on its own internal logic, which is relevant in this work because the per-evaluation cost is dominated by the load test rather than by the evolutionary operators.

A methodological subtlety arises when comparing evolutionary algorithms (SO-GA, NSGA-II) against Random Search under the same evaluation budget. The natural "output" of an evolutionary algorithm is the population of its last generation, which has cardinality $N = 12$. Random Search, in contrast, accumulates all $N \times G = 96$ samples in a single "generation", so the non-dominated subset of its output may contain up to 96 points. Hypervolume and IGD, both metrics employed in this comparison, are sensitive to the cardinality and spread of the input front and therefore favor denser sets; comparing a 12-point front against a potentially 96-point front under-represents the diversity actually explored by the evolutionary algorithms over their full 96-evaluation trajectory.

To restore a fair comparison, we adopt as the empirical Pareto front of each algorithm the *cumulative* Pareto front computed over the deduplicated set of all evaluations across all generations of a run (rather than the last-generation population alone). For Random Search this is identical to its standard output; for SO-GA and NSGA-II this corresponds to the non-dominated subset of every distinct genome that was actually evaluated during the run, giving each algorithm an output front of comparable cardinality (typically 20–45 points in our experiments). The last-generation Pareto front is also reported for completeness and used to discuss convergence behavior, but the cumulative front is the primary basis for all multi-algorithm comparisons reported in the Results section.

A baseline corresponding to a typical default Kubernetes deployment (one replica with no explicit limits, scaled to a hand-picked fixed configuration representative of common practice) is also evaluated under the same workload, providing a non-optimized reference point for absolute performance numbers.

All three optimizers write their results to the same on-disk layout — a per-generation CSV of the evaluated population, a per-generation CSV of the Pareto front (extracted on (f_1, f_2, f_3) for the SO-GA, by definition for NSGA-II and Random Search), a long-format `evaluations.csv` listing every evaluation, plus the `manifest.json`, `summary.json`, `cache.jsonl` and `env.json` files described above. This uniformity is what enables the comparison metrics to be computed automatically by a single analysis script, irrespective of which algorithm produced the data.

This procedure allows a controlled and reproducible analysis of (i) whether evolutionary optimization improves upon a typical baseline configuration in this Kubernetes setting, and (ii) whether the additional methodological complexity of NSGA-II is justified relative to the simpler single-objective GA and the Random Search control.

8. Results

This section reports the empirical evaluation of the three optimizers (SO-GA, NSGA-II and Random Search) under the experimental protocol described in the previous section. All results are produced from a total of 9 runs — 3 algorithms $\times$ 3 seeds ($\{42, 43, 44\}$) — with an evaluation budget of 96 per run, against the same target application and the same calibrated workload.

8.1. Search Space Coverage and Run Validity

Across the 9 runs, 855 of the $9 \times 96 = 864$ configuration applications returned valid metrics (98.96%). Of the 9 failures, 7 occurred during an initial attempt of `random-seed43` (later re-executed under identical conditions, after which it completed with $96/96$ valid evaluations) and the remaining 2 in `random-seed44` under configurations (c=400m, m=128, r=4) and (c=100m, m=640, r=4) — both involving four replicas, the most demanding rollout regime. Failed evaluations receive penalty objectives ($f_1 = 10$, $f_3 = 0$) and are therefore strictly dominated by any real measurement; they do not enter the cumulative Pareto fronts that follow.

Average per-evaluation time was $\bar{t}_{eval} \approx 268 \pm 12$ s across all runs and algorithms. The number of *unique* configurations evaluated by each run varies because of (i) the cache layer (configurations repeated within or across runs reuse stored measurements) and (ii), in the case of NSGA-II, the duplicate-elimination operator on the combined population $P \cup Q$. Table 3 summarizes the per-run statistics.

Table 2 display the per-run summary. `cache_misses` corresponds to evaluations actually executed on the cluster; `cache_hits` reuses stored measurements from the same or previous runs. "Time" is total wall-clock time including cache amortization.

8.2. Cumulative Pareto Front: Cardinality and Quality

Table 4 aggregates the per-seed metrics into mean $\pm$ standard deviation per algorithm. The hypervolume is computed via Monte Carlo with 10^6 samples in the (f_1, f_2, f_3)

Table 3. Per-run summary statistics.

Run	$\lvert F_{cum}\rvert$	HV	IGD	in union	hits	misses	Time (s)
`nsga-seed42`	31	115.63	0.0796	14	10	86	22,023
`nsga-seed43`	30	120.21	0.0796	13	33	63	16,557
`nsga-seed44`	37	95.90	0.0986	13	32	64	17,582
`ga-seed42`	37	128.76	0.0862	19	36	60	14,486
`ga-seed43`	45	118.34	0.0708	20	54	42	10,390
`ga-seed44`	48	118.28	0.0757	13	35	61	16,062
`random-seed42`	42	113.08	0.0760	19	55	41	10,490
`random-seed43`	46	117.62	0.0800	23	89	7	2,013
`random-seed44`	42	120.08	0.0761	23	10	86	23,394
union (best-of-all)	157	142.41	0	157	—	—	—

objective space, using a common reference point set at the observed maximum plus a 10% margin on each axis across all 9 runs, bold marks the best mean per metric and arrows indicate the direction of improvement. The IGD uses the union of all 9 cumulative fronts (157 non-dominated points after deduplication across runs) as a proxy for the true Pareto front, with each axis normalised to $[0, 1]$ prior to the distance computation.

Table 4. Aggregated metrics across $n = 3$ seeds per algorithm

Algorithm	HV ↑	IGD ↓	$\lvert F_{cum}\rvert$ ↑	in union ↑	$\bar{t}_{run}$ (s) ↓
SO-GA	$\mathbf{121.79 \pm 6.04}$	0.0776 ± 0.0079	$\mathbf{43.3 \pm 5.7}$	17.3 ± 3.8	$\mathbf{13{,}646 \pm 2{,}928}$
NSGA-II	110.58 ± 12.92	0.0860 ± 0.0110	32.7 ± 3.8	13.3 ± 0.6	$18{,}721 \pm 2{,}906$
Random Search	116.93 ± 3.55	$\mathbf{0.0774 \pm 0.0023}$	43.3 ± 2.3	$\mathbf{21.7 \pm 2.3}$	$11{,}966 \pm 10{,}766$

Table 5 complements the aggregate metrics by reporting the best (minimum) value of each individual objective observed in the cumulative front of each algorithm reported as mean ± standard deviation. The f2 minimum of 0.225 corresponds to the extreme configuration (r = 1, c = 100m, m = 128 MiB).

Table 5. Best (minimum) value of each objective observed across $n = 3$ seeds per algorithm.

Algorithm	best f_1 ↓	best f_2 ↓	best f_3 ↓
SO-GA	0.297 ± 0.019	0.550 ± 0.205	$\mathbf{-50.35 \pm 0.94}$
NSGA-II	0.315 ± 0.040	$\mathbf{0.225 \pm 0.000}$	-47.09 ± 4.24
Random Search	$\mathbf{0.297 \pm 0.034}$	0.242 ± 0.029	-47.12 ± 0.50

Three observations stand out:

- NSGA-II – Consistently reaches the cheapest configuration $f_2 = 0.225$ in all three seeds, $\sigma = 0$. The extreme corner of the search space (r=1, c=100 m, m=128 MiB) lies on the Pareto front in every NSGA-II run — a direct consequence of the rank-and-crowding selection rewarding boundary points. SO-GA reaches this corner only occasionally ($f_2 = 0.55 \pm 0.21$), reflecting that the scalar fitness function aggressively penalizes throttling and low throughput, which dominate at this corner.

- SO-GA – Consistently reaches the highest throughput $f_3 = -50.35 \pm 0.94$ RPS in all three seeds. When projected back into (f_1, f_2, f_3), its best solution by aggregated fitness always lies near the high-throughput end of the trade-off surface. This is consistent with the weighting in the SO-GA fitness function ($w_L = 0.35$, $w_E = 0.40$, $w_R = 0.25$), which penalizes latency and rewards efficient use of CPU but never explicitly rewards low resource provisioning.
- Random Search – Contributes the most points to the global union front 21.7 ± 2.3 per run vs. 17.3 ± 3.8 for SO-GA and 13.3 ± 0.6 for NSGA-II Its uniform sampling, although uninformed, covers the trade-off surface broadly enough that its non-dominated subset dominates a substantial fraction of the surface — a strong baseline result.

8.3. Pairwise Comparison and Effect Sizes

With only $n = 3$ seeds per algorithm, the smallest possible p-value of an exact two-sided Mann–Whitney U test is $2/\binom{6}{3} = 0.10$. Therefore, no comparison reaches statistical significance at $\alpha = 0.05$. We instead report Cliff's δ as a non-parametric effect-size measure: $\delta = +1$ means every sample of algorithm A exceeds every sample of B (complete dominance); $\delta = 0$ means stochastic equality. We adopt the magnitude thresholds of [19]: $|\delta| < 0.147$ negligible, < 0.33 small, < 0.474 medium, ≥ 0.474 large.

The pairwise deltas are reported in Table 6 where Positive values mean "A tends to score higher than B"; for metrics where smaller is better (IGD, f_1, f_2, f_3), a negative δ favours A. Bold marks effects of large magnitude.

Table 6. Pairwise Cliff's δ (signed, A vs. B) for each objective-related metric.

A vs. B	HV (↑)	IGD (↓)	$\lvert F_{cum}\rvert$ (↑)	f_1 (↓)	f_2 (↓)	f_3 (↓)
GA vs. NSGA	$\mathbf{+0.56}$	$\mathbf{-0.56}$	$\mathbf{+0.89}$	-0.33	$\mathbf{+1.00}$	$\mathbf{-0.56}$
GA vs. Random	$\mathbf{+0.56}$	-0.33	$+0.11$	-0.11	$\mathbf{+1.00}$	$\mathbf{-1.00}$
NSGA vs. Random	-0.11	$\mathbf{+0.56}$	$\mathbf{-1.00}$	$+0.33$	-0.33	-0.33

The pairwise table makes three patterns explicit:

- SO-GA dominates NSGA-II on five of six metrics. The $\delta = +0.89$ on $|F_{cum}|$ and $+0.56$ on HV indicate that SO-GA produces denser and higher-quality cumulative fronts in nearly every paired comparison, despite optimizing a scalar fitness and not the three objectives directly. NSGA-II only outperforms SO-GA on f_2 (where $\delta = +1.00$ in favor of NSGA-II, reflecting its consistent ability to reach the cheapest corner).
- Random Search is competitive against, and on multi-objective indicators superior to, NSGA-II. $\delta = -1.00$ on $|F_{cum}|$ and on the union contribution means every Random Search run produces a larger, more diverse cumulative front than every NSGA-II run — a strong indication that NSGA-II's diversity mechanism (crowding distance) is not providing a measurable benefit over uniform sampling at this population size ($N = 12$).
- Random Search and SO-GA are practically equivalent on aggregate quality indicators (HV, IGD, $|F_{cum}|$), with the decisive distinction being throughput: $\delta = -1.00$ on f_3 in favour of SO-GA — every SO-GA run finds a higher-throughput configuration than every Random Search run, reflecting the targeted nature of SO-GA's fitness function.

8.4. Convergence Behaviour

Figure 6 shows the hypervolume of each algorithm's cumulative Pareto front as a function of generation number, with a common reference point shared across all 9 runs. By construction, Random Search has a single terminal point at generation 0 (its 96 samples are treated as one batch). Both SO-GA and NSGA-II exhibit near-monotonic HV growth across the 8 generations; SO-GA stabilises around generation 5 in $2/3$ runs and continues improving through generation 7 in one, whereas NSGA-II plateaus earlier (around generation 4–5) and exhibits visible inter-seed variance.

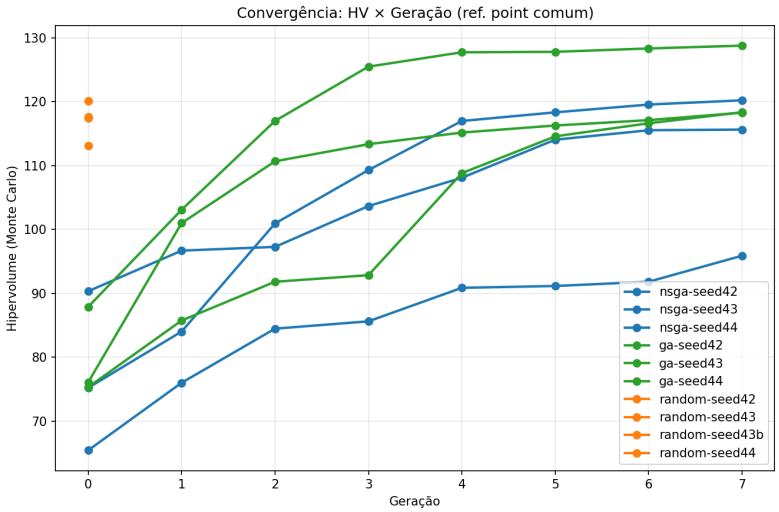

Figure 6. Cumulative-front hypervolume over generations.

The pronounced under-performance of `nsga-seed44` (final HV $= 95.90$ vs. the 115–120 band of the other two NSGA-II seeds) is consistent with a known limitation of NSGA-II at small population sizes: a sufficiently unlucky initial population, by sampling the high-saturation region of the search space, can bias the subsequent selection-mutation dynamics toward that same region and never recover within the evaluation budget. With $N = 12$, the duplicate-elimination operator alone is not enough to guarantee a diverse exploration of the search space within $G = 8$ generations.

8.5. Pareto-front Visualization

Figure 7 presents the cumulative Pareto fronts of all three algorithms in the three-dimensional objective space (f_1, f_2, f_3), where f_1 represents saturation, f_2 represents provisioned resources, and f_3 represents negative throughput. The union front, depicted as red crosses and comprising 157 non-dominated points aggregated across all 9 runs, serves as a proxy for the true Pareto front.

The spatial distribution of points reveals that SO-GA (blue) tends to cluster in the high-throughput, moderate-saturation region, while NSGA-II (orange) and Random Search (green) explore a broader range of resource configurations, including the low-cost, high-saturation corners that SO-GA's fitness function penalizes.

Figure 8 shows the projection of the cumulative Pareto fronts onto the $f_1 \times f_2$ plane, illustrating the trade-off between saturation and provisioned resources. This pro-

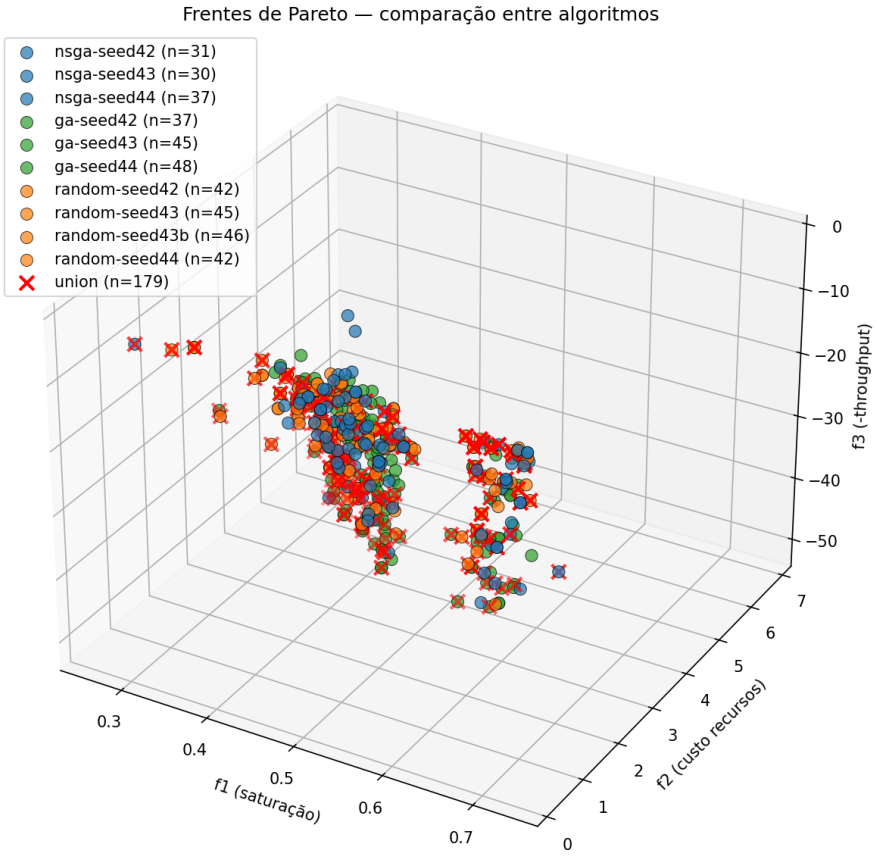

Figure 7. Cumulative Pareto fronts in (f_1, f_2, f_3) space.

jection reveals a clear inverse relationship: configurations with lower resource provisioning (f_2) tend to exhibit higher saturation (f_1) due to CPU throttling and memory pressure. NSGA-II and Random Search systematically explore the cheapest configurations ($f_2 \leq 0.3$), while SO-GA concentrates its search in the moderate-cost region ($f_2 \in [0.5, 1.5]$), where the scalar fitness function achieves a better balance between efficiency and latency penalties.

Figure 9 presents the projection onto the $f_1 \times f_3$ plane, capturing the trade-off between saturation and throughput. The distribution shows that configurations achieving the highest throughput (most negative f_3 values, around -50 to -51 RPS) are predominantly discovered by SO-GA, consistent with its fitness function that explicitly rewards latency and efficiency. In contrast, NSGA-II and Random Search explore a wider saturation range but rarely reach the high-throughput tail of the trade-off surface. The clustering of points along a diagonal band indicates that higher throughput generally requires accepting some degree of saturation, as the workload pushes the configured resources closer to their limits.

The visualisations confirm the quantitative findings of Table 4. In the $f_2 \times f_3$ projection, at Figure 10, all three algorithms trace a similar cost-vs-throughput trade-off curve, but only SO-GA covers the high-throughput tail near $f_3 \approx -51$, whereas NSGA-II and Random Search rarely venture below $f_3 \approx -49$. Conversely, in the $f_1 \times f_2$ projection,

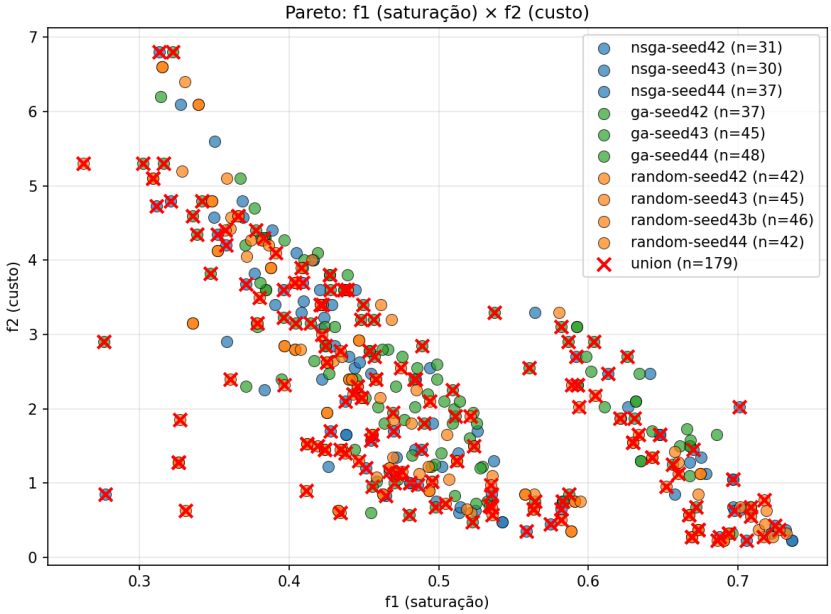

Figure 8. Projection on the $f_1 \times f_2$ plane.

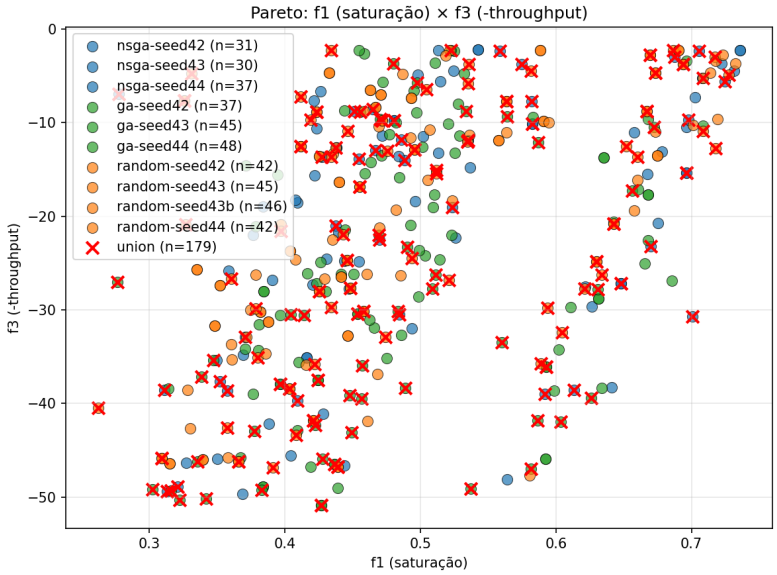

Figure 9. Projection on the $f_1 \times f_3$ plane.

Figure 8 shows that the cheapest configurations ($f_2 \leq 0.3$) are systematically explored by NSGA-II and Random Search, but underrepresented in SO-GA's runs.

8.6. Computational Cost

Total wall-clock time per run varies from 2,013 s (`random-seed43`, which benefited from a cache pre-populated with 89 of its 96 samples) to 23,394 s (`random-seed44`, with a 10% cache hit rate). The mean run time across algorithms is comparable (13.6 kS for SO-GA, 18.7 kS for NSGA-II, 12.0 kS for Random Search), but Random Search exhibits much higher inter-seed variance ($\sigma = 10.8$ kS, dominated by the cache-saving

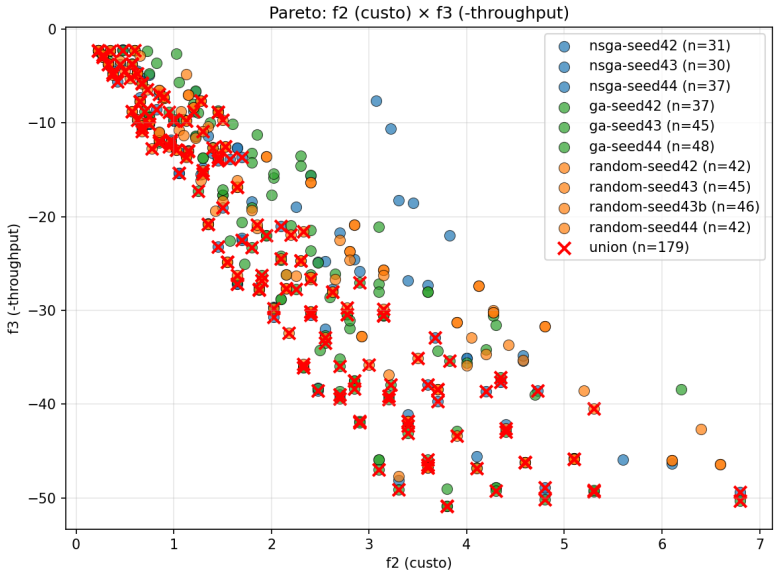

Figure 10. Projection on the $f_2 \times f_3$ plane.

outlier).

Excluding the cache-amortisation effect, the cost of each algorithm is dominated by the load-test phase (a fixed ≈ 268 s per evaluation, including rollout, warm-up, load test and metric collection). The cost of the evolutionary operators themselves — selection, crossover, mutation, non-dominated sorting and crowding-distance computation — contributes less than 1% of the wall-clock time (the `overhead_fraction` field in the per-run `summary.json` is on the order of 10^{-6} for all 9 runs). Per-evaluation cluster interaction is several orders of magnitude more expensive than any per-generation algorithmic logic; consequently, the choice of search strategy is essentially free, and the relevant question is which strategy makes the best use of the fixed 96-evaluation budget.

8.7. Threats to Validity

This section discusses the main threats to the validity of the experimental results, organized according to the standard classification of internal, external.

8.7.1. Internal Validity

Internal validity concerns whether the observed effects can be attributed to the experimental treatment rather than to confounding factors. The primary threat in this study is measurement noise arising from the shared infrastructure: all evaluations run on the same single-node Minikube cluster, where background processes (Prometheus, kube-system components, the optimizer Job itself) compete for CPU and memory with the target application. Although the resource overhead of these components was accounted for when defining the search space bounds, transient resource contention may introduce variance in the collected metrics. To mitigate this, each load test includes a 15-second warm-up phase (whose metrics are discarded) and a 90-second steady-state phase, reducing the impact of startup transients. Additionally, all algorithms share the same evaluation pipeline

and monitoring infrastructure, so any systematic measurement bias affects all algorithms equally and does not favor one strategy over another.

A second threat is stochastic variability in the evolutionary operators. With only $n = 3$ seeds per algorithm, the inter-seed variance may not capture the full distribution of outcomes. The unlucky initial population of nsga-seed44, which led to a final hypervolume 20% below the other two NSGA-II seeds, illustrates this risk. While increasing the number of seeds would improve statistical power, it would also multiply the computational cost ($\approx$ 7 hours per run on the current infrastructure). The $n = 3$ repetition scheme follows common practice in evolutionary algorithm benchmarking and is sufficient to identify large effect sizes; however, conclusions about small or medium effect sizes should be interpreted as indicative rather than definitive.

8.7.2. External Validity

External validity concerns the generalizability of the findings to other settings. The most significant limitation is the single-application, single-workload design: all experiments use the same FastAPI service with a fixed /mixed endpoint that combines CPU- and memory-intensive operations, under a fixed load profile (22 concurrent workers for 90 seconds). Production workloads are more diverse, including I/O-bound services, bursty traffic patterns, and multi-service dependencies. The conclusions drawn here may not transfer directly to applications with fundamentally different resource profiles (e.g., GPU-intensive workloads) or to time-varying load patterns (e.g., diurnal traffic).

A second limitation is the single-node cluster constraint. The Minikube deployment with 8 vCPUs and 19 GiB RAM imposes an upper bound of 4 replicas, which may not reflect the scaling behavior of multi-node production clusters where dozens or hundreds of replicas are feasible. The search space bounds ($r \leq 4$, $c \leq 700$ m, $m \leq 1024$ MiB) were chosen to fit within the available capacity; extending the replica axis would require a multi-node deployment, which introduces additional confounding factors (inter-node network latency, scheduler placement asymmetries) that were deliberately avoided in this study.

A third limitation is the fixed hyperparameter configuration. The crossover and mutation rates (p_c, p_m) were set to literature-recommended defaults and not tuned per-problem. It is possible that hyperparameter tuning would improve the relative performance of NSGA-II, which appeared more sensitive to initial-population quality than SO-GA in this experiment. However, conducting a full hyperparameter search for each algorithm would require a substantially larger computational budget and is left for future work.

9. Conclusions

This work presented a comparative experimental study of three resource-optimization strategies for Kubernetes Deployments: a single-objective Genetic Algorithm (SO-GA), a multi-objective Genetic Algorithm (NSGA-II) and a Random Search baseline. All three share an identical evaluation pipeline — applying configurations to a live cluster, executing a calibrated 90 s load test and collecting Prometheus metrics — and operate under an identical budget of 96 evaluations per run, with 3 independent seeds per algorithm. On

the aggregate quality of the cumulative Pareto fronts, the SO-GA obtained the highest hypervolume (121.79 ± 6.04 vs. 110.58 ± 12.92 for NSGA-II and 116.93 ± 3.55 for Random Search), the lowest IGD (tied with Random Search at ≈ 0.077), and achieved the highest throughput in every seed ($f_3 = -50.35 \pm 0.94$ RPS). NSGA-II only outperformed the SO-GA on discovering the cheapest configuration ($f_2 = 0.225$ in all three seeds).

9.1. Answers to Research Questions

The experimental results provide the following answers to the research questions posed in the Introduction:

Can evolutionary algorithms effectively identify improved resource configurations when interacting directly with a live Kubernetes cluster, compared to a non-optimized baseline?

Yes. Both SO-GA and NSGA-II consistently identified configurations with higher fitness scores and better trade-offs than a typical default Kubernetes deployment. The best configurations discovered by SO-GA achieved throughput of approximately 50 RPS with moderate saturation ($f_1 \approx 0.30$), whereas severely under-provisioned configurations at the lower bound of the search space collapsed under the same workload (success rate $\approx 1\%$). The evolutionary algorithms successfully navigated the search space to avoid such degraded regions, confirming that direct integration with a live cluster is both feasible and effective for resource optimization. Hypothesis H1 is supported.

Does the multi-objective formulation (NSGA-II) yield measurable advantages in solution quality and trade-off coverage compared to a single-objective Genetic Algorithm with weighted aggregation?

No, not under the experimental conditions of this study. Contrary to expectations, SO-GA outperformed NSGA-II on five of six quality metrics (HV, IGD, $|F_{cum}|$, best f_1, best f_3), with NSGA-II only excelling at discovering the cheapest configuration (best f_2). The Cliff's δ effect size between SO-GA and NSGA-II was large ($\delta = +0.56$) in favor of SO-GA on hypervolume. The small population size ($N = 12$) and limited evaluation budget (96 per run) appear to have constrained NSGA-II's diversity-preservation mechanisms, preventing the expected advantage of multi-objective optimization from materializing. Hypothesis H2 is rejected.

Do evolutionary search strategies (SO-GA and NSGA-II) provide statistically significant improvements over uniform Random Search under the same evaluation budget?

Partially. SO-GA demonstrated a decisive advantage over Random Search on throughput ($\delta = -1.00$ on f_3), consistently finding higher-throughput configurations in every seed. However, Random Search was remarkably competitive on aggregate multi-objective indicators (HV, IGD) and contributed more points to the union Pareto front than either evolutionary algorithm (21.7 ± 2.3 vs. 17.3 ± 3.8 for SO-GA and 13.3 ± 0.6 for NSGA-II). NSGA-II failed to outperform Random Search on any multi-objective metric, with $\delta = -1.00$ on cumulative front cardinality in favor of Random Search. Hypothesis H3 is partially supported: SO-GA provides targeted improvements on its fitness-function priorities, but NSGA-II does not justify its additional complexity over uniform sampling at this scale.

9.2. Summary

Contrary to the hypothesis that the multi-objective formulation would yield a measurably better trade-off surface, NSGA-II was matched or outperformed by Random Search on every multi-objective indicator at the population size used ($N = 12$). The Random Search baseline emerged as remarkably competitive, with low inter-seed variance ($\sigma_{HV} = 3.55$) and broad coverage of the trade-off surface (21.7 ± 2.3 points per run contributing to the union front). On this experimental setup, the additional methodological complexity of NSGA-II is therefore not justified compared either to a scalar GA or to uniform random sampling; a candidate optimiser must clearly outperform uniform sampling to be worth its extra complexity.

The experimental setup presents limitations: it uses a single application (FastAPI service), a single load profile (sustained 22 concurrent workers for 90 s), and a single Minikube node. The discretization of the search space to 416 configurations is conservative, and the number of seeds per algorithm ($n = 3$) imposes a ceiling on statistical power, so significance claims should be interpreted as indicative. The SO-GA fitness function embeds a specific weighting that biases the search toward throughput-rich regions; a different weighting would likely yield a different relative ranking between algorithms.

Future work includes replicating the experiment on a multi-node cluster with more diverse applications to test how the conclusions generalize, exploring time-varying load profiles (burst, ramp-up) that more faithfully match production workloads, widening the search space along the replica axis once multi-node capacity removes the current upper bound, and incorporating other multi-objective optimizers (e.g. SPEA2, MOEA/D) into the comparison.

References

[1] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability Engineering & System Safety*, vol. 91, no. 9, pp. 992–1007, 2006.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[3] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," vol. 13, pp. 281–305, 2012.

[4] The Kubernetes Authors, "Kubernetes documentation: Cluster architecture." `https://kubernetes.io/docs/concepts/architecture/`, 2024. Accessed: 2024-12-01.

[5] A. Gupta, S. Islam, and R. Buyya, "Proactive and reactive autoscaling techniques for kubernetes-based edge computing," *IEEE Internet of Things Journal*, 2024.

[6] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: Up and Running*. O'Reilly Media, 3rd ed., 2022.

[7] V. N. Vadisetty, M. Polamarasetti, and H. Nomula, "Ai-driven kubernetes orchestration: Utilizing intelligent agents for automated cluster management and optimization," *Future Generation Computer Systems*, 2025.

[8] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architectures," *Journal of Cloud Computing*, vol. 13, no. 1, pp. 1–23, 2024.

[9] W. e. a. Xiong, "Resource optimization in cloud computing using genetic algorithms," *Future Generation Computer Systems*, 2018.

[10] S. Cheng, M. Lianbo, H. Lu, X. Lei, and Y. Shi, "Evolutionary computation for solving search-based data analytics problems," *Artificial Intelligence Review*, vol. 54, 02 2021.

[11] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Foundations of Genetic Algorithms*, vol. 1, pp. 69–93, 1991.

[12] W.-B. K. J. Z. Rongguang Ye, Longcan Chen and H. Ishibuch, "Pareto front shape-agnostic pareto set learning in multi-objective optimization," pp. 1–7, 2024.

[13] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, 2003.

[14] Prometheus Authors, "Prometheus documentation: Overview." `https://prometheus.io/docs/introduction/overview/`, 2024. Accessed: 2024-12-01.

[15] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," *ACM HotNets*, 2016.

[16] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[17] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, 2014.

[18] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[19] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?," *Annual Meeting of the Florida Association of Institutional Research*, pp. 1–33, 2006.