GC AWARE CONTAINERIZED DEPLOYMENT TO ELIMINATE GC LATENCIES

Premarathna E.H.A.D.S.D.

219384L

MSc in Computer Science

Department Of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

07 2025

GC AWARE CONTAINERIZED DEPLOYMENT TO ELIMINATE GC LATENCIES

Premarathna E.H.A.D.S.D.

219384L

Dissertation submitted in partial fulfillment of the requirements for the degree
MSc in Computer Science

Department Of Computer Science and Engineering
Faculty of Engineering

University of Moratuwa
Sri Lanka

07 2025

DECLARATION

I declare that this is my own work and this Dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Shanaka Date: 2025/07/03

The supervisor should certify the Dissertation with the following declaration.

The above candidate has carried out research for the MSc in Computer Science Dissertation under my supervision. I confirm that the declaration made above by the student is true and correct.

Name of Supervisor: Prof. G.I.U.S. Perera

Signature of the Supervisor: Date: 2025/07/04

ABSTRACT

Garbage collection in Java applications continues to be a major source of performance overhead, often causing disruptive pauses in latency sensitive workloads and reducing overall throughput. Although various techniques have been developed ranging from advanced garbage collection algorithms to finely tuned parameters, these strategies do not completely eliminate stop the world interruptions.

This thesis presents a containerized deployment strategy aimed at achieving pause free execution for Java applications, leveraging Kubernetes orchestration and Kubernetes memory monitoring. The proposed methodology employs the Epsilon garbage collector, which inherently disables the garbage collection mechanism, thereby avoiding traditional garbage collection pause events. In place of conventional memory reclamation within a container, the system monitors heap usage and gracefully terminates containers approaching predefined memory thresholds, subsequently launching new containers to handle incoming traffic. This process ensures that application threads do not experience forced suspension for the purpose of heap compaction.

Central to the design is the use of Kubernetes for container lifecycle management, complemented by Kubernetes metrics that detect memory pressure before critical levels are reached. When a container's memory utilization exceeds a configurable limit, traffic is seamlessly redirected to a newly initiated container, allowing the retiring container to shut down without affecting ongoing requests or service availability. By shifting memory reclamation to container restarts, this approach prevents stop the world garbage collection pauses in a manner that reduces operational complexity.

A comprehensive performance evaluation compares the container rotation strategy against the traditional Garbage collection algorithms. Experimental results indicate that the proposed solution consistently delivers lower latency profiles and higher throughput under varying workload conditions. Furthermore, offloading the memory reclamation task to container restarts lessens the need for intricate GC tuning, thus simplifying operational overhead. This thesis demonstrates a viable method of achieving truly pause free Java application execution, with implications for high availability systems and microservices that require minimal disruption from garbage collection.

Keywords: Java, Garbage collection, Containerized deployment, Kubernetes

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Description
CMS	Concurrent Mark Sweep
CPU	Central Processing Unit
CRD	Kubernetes Custom Resource Definition
EJB	Enterprise Java Beans
GC	Garbage Collection
GCI	Garbage Collection Control Interceptor
GUI	Graphical User Interface
I/O	Input/Output
JMX	Java Management Extensions
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MITMEM	Mitigating Memory-induced Delays
NoSQL	NoSQL
NUMA	Non-Uniform Memory Access
OOM	Out of Memory
OS	Operating System
PaaS	PaaS
RBAC	Role-Based Access Control
STW	Stop The World
THP	Transparent Huge Pages
TLB	Translation Lookaside Buffer

CHAPTER 1

INTRODUCTION

Java is a widely adopted programming language that is used in diverse applications, including large-scale data analytics, online transactional systems, and latency sensitive microservices. Its popularity comes from comprehensive libraries, platform independence, and the garbage collection (GC) mechanisms that automate memory management. Although automatic GC relieves developers from manual memory allocations and deallocations, it periodically halts application threads to reclaim unused heap objects. These *GC pauses* can become performance bottlenecks in settings where low latency or high throughput is important.

In containerized environments, such as those orchestrated by Kubernetes, the exchange between GC overhead and strict container resource boundaries magnifies these issues. Small container footprints may frequently invoke GC, while ephemeral and horizontally scaled microservices are easily disrupted by prolonged stop the world GC cycles. Balancing the memory usage within each container and ensuring stable, predictable performance presents a pressing challenge. Traditional GC tuning can reduce pause durations but is often insufficient for applications handling mission-critical workloads requiring near-zero downtime and minimal latencies. Consequently, an alternative approach that overcome major GC overhead by rotating containers before large GC cycles occur emerges as a potentially more effective solution.

1.1 Problem Statement

Although garbage collection offers a simplified memory model, the automatic reclamation can produce extended or unpredictable stop-the-world events. Under moderate loads, conventional collectors deliver acceptable pause times. However, as heap sizes increase or workloads intensify, even advanced algorithms may experience multi-second stalls. In container-based deployments, where memory usage is tightly controlled and pods are expected to start, scale, and shut down gracefully, such interruptions jeopardize both availability and performance.

Moreover, Horizontal Pod Autoscalers (HPAs) and other cloud-native scaling systems typically monitor CPU, memory, or custom metrics for scaling decisions. GC spikes, especially those that appear as short bursts of resource usage might trigger or conflict with these auto-scaling mechanisms, leading to unwanted scaling oscillations. A container that purely relies on conventional GC is more difficult to scale predictably, since memory usage may be abruptly reclaimed or remain near capacity for extended periods prior to a collection event.

A radical solution to avoid GC pauses entirely is to employ a *no-op* garbage col-

lector that allocates memory but but does not reclaims it. This method delegates heap exhaustion handling to a container rotation policy. Once memory usage nears a threshold, a new container is spawned, and the old one is drained. This design eschews GC overhead but relies on stable, preemptive container replacements.

1.2 Research Objectives

The primary objectives of this research are:

- **Identify the domain of applications which can benefit from removing GC pauses.** Establish which types of Java-based systems, particularly those deployed in microservice or containerized environments, experience significant performance degradation due to GC pauses.
- **Evaluate the impact of GC pauses on applications.** Quantify how GC interruptions influence throughput, response time, and resource utilization under diverse workloads, especially in the presence of strict performance or latency constraints.
- **Evaluate the resource utilization by eliminating GC pauses.** Investigate how reducing or removing GC stalls affects overall memory consumption, CPU overhead, and container resource usage, and whether it can lead to more efficient operation in cloud settings.
- **Evaluate the performance gain obtained by eliminating GC pauses.** Determine the improvement in throughput and latency when employing alternative GC algorithms or container lifecycle management that preempts or avoids stop-the-world events.

1.3 Motivation and Context

1.3.1 Challenges of GC in Cloud-Native Applications

Modern software systems are routinely deployed as microservices on container orchestration platforms that automate scaling and resilience. Each microservice is packaged into a container with resource constraints (CPU, RAM) defined in the pod specification. Java applications commonly scale horizontally by adding replicas of the same service, but each replica remains susceptible to GC stalls. As container replicas are ephemeral, properly tuning GC for each environment and load scenario is non-trivial, often resulting in friction or suboptimal usage of memory resources.

Additionally, microservices with real-time streaming or interactive workflows such as financial transactions, e-commerce, must handle surges in traffic with minimal latencies. GC-induced stalls degrade tail latencies, hamper throughput, and reduce the

system's capacity to handle peak loads. Although more modern GC implementations aim to deliver low-latency collections, they remain complex, require specialized tuning, and are not always immune to multi-second pause phenomena under large heap or irregular object lifecycles.

1.3.2 Advantages of Container Rotation Over GC Tuning

Ephemeral containers, a hallmark of cloud-native design, are fundamentally well-suited to frequent or proactive rotation. Operators already perform rolling updates for deployments and can remove or add replicas in response to load or to shift traffic away from unhealthy pods. Adopting the concept of container rotation as a memory management mechanism is a logical extension of existing patterns:

- **Predictable Latencies**: By capping the container lifetime based on memory usage, the system never experiences a full GC. Thus, stop-the-world events are replaced by lightweight container restarts initiated under controlled conditions.
- **Cloud Integration**: Kubernetes can seamlessly handle traffic shifting and graceful shutdown. Memory watchers simply add an additional dimension such as memory threshold into the container lifecycle policy.
- **Reduced Complexity**: Instead of fine-tuning multiple GC parameters (heap layout, concurrency levels, etc.), the operator sets a straightforward threshold. Once memory usage passes it, a fresh container is spawned with an empty heap, while the older container is gracefully shut down.

1.4 Research Scope

The scope of this thesis encompasses:

- **Addressing performance bottlenecks caused by GC pauses** in containerized Java applications, particularly in latency-sensitive or high-throughput domains.
- **Designing and implementing a GC-aware containerized deployment strategy** that can effectively reduce or eliminate GC-induced latencies. This involve specialized container lifecycle logic, integration with Kubernetes and advanced GC configurations.
- **Analyzing performance trade-offs** of different GC algorithms and configurations, then comparing them with more conventional Java deployment strategies.
- **Measuring and monitoring memory usage of containerized applications** via Java Virtual Machine (JVM) metrics and Kubernetes APIs, identifying how memory thresholds and container restarts can mitigate GC overhead.

- **Evaluating the proposed GC-aware approach** using reproducible experiments and performance benchmarks. These tests will be conducted under varied workloads, transactional microservices, and data analytics jobs.

1.5 Contributions

This thesis introduces the concept of a *GC-aware container lifecycle* that moves away from internal Java GC to an external container rotation approach. Key contributions are:

- **Architecture Design**: Presents a cloud-native solution for memory management via ephemeral container restarts. Demonstrates how Epsilon GC in Java can practically operate in production by coupling it with a robust orchestration logic that prevents out-of-memory scenarios.
- **Operator Implementation**: Implements a *MemoryWatcher* resource and *GC-Aware Operator* using Kubebuilder, providing watchers, reconcilers, and custom endpoints that integrate seamlessly with the standard Kubernetes ecosystem.
- **Empirical Evaluation**: Examines the impact of eliminating GC on throughput and latency across multiple load patterns. Illustrates the trade-offs in memory overhead, replacement frequencies, and capacity concerns.

1.6 Structure of the Thesis

This document is structured as follows:

- **Chapter 2: Literature Review** surveys related work in garbage collection techniques, container orchestration, and prior attempts to mitigate GC overhead through memory offloading or ephemeral container strategies.
- **Chapter 3: Proposed Method** discusses the core design principles of no-GC container rotation, focusing on Epsilon GC and the fundamental approach to memory threshold enforcement.
- **Chapter 4: Implementation** explores the operator-based architecture and the creation of a MemoryWatcher CRD, detailing how watchers, reconcilers, and triggers function within Kubernetes.
- **Chapter 5: Results** presents an experimental evaluation using sample Java microservices. Key metrics such as request latency, throughput, container memory usage, and replacement intervals are analyzed.

- **Chapter 6: Discussion** identifies practical challenges and limitations, such as potential interactions with horizontal pod autoscalers or stateful [1] workloads.
- **Chapter 7: Conclusion** summarizes the findings, final observations, and contributions of the study.

Overall, this introductory chapter has laid out the rationale for a GC-aware container approach, articulated the specific research problem, and defined the scope and contributions. The following chapters will delve deeper into the theoretical foundations, system design, implementation details, experimental validation, and concluding insights of this no-GC containerized deployment strategy.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

The entire chapter of literature review emphasizes an analysis of documented sources on the impact and the solutions for GC pauses in applications. Garbage collection (GC) is a vital process in the management of memory resources in modern software applications. GC automates memory management by automatically freeing developers from manually managing memory. The GC algorithm determines whether an object is necessary or not based on its reachability from other objects. Objects that are referenced by other objects are considered reachable and are not collected because they are deemed necessary. Conversely, objects that are not referenced by any other objects are considered unreachable and are collected by the GC as unnecessary.

However, the GC process can cause latency and performance bottlenecks that negatively impact the user experience. In response, latency aware GC processes have emerged as a promising solution to mitigate GC latencies in applications. Furthermore the integration of container orchestration platforms, such as Kubernetes, with GC-aware deployment strategies will be useful for optimizing memory allocation and reducing the frequency and duration of GC pauses. This chapter provides a comprehensive review of the literature on GC, containerized deployment, exploring the benefits, challenges, and best practices associated with this approach.

2.2 Java Virtual Machine

Java virtual machine is an abstract machine that provides the runtime environment in which Java bytecode can be executed. It is an essential component of the Java platform and is used in a wide variety of applications, including web servers, mobile applications, and enterprise software. The JVM is responsible for executing Java programs and converting the bytecode into machine-specific instructions. The JVM also manages the memory allocation and garbage collection of Java applications. Since the JVM provides a platform-independent environment, Java programs can be run on any platform that has a JVM installed.

2.2.1 Memory management

When it comes to the memory management of JVM following are the key components involved [2], [3]. Figure 2.1 ([2]) illustrates a representation of the key memory management components [4], [5], [6], [7].

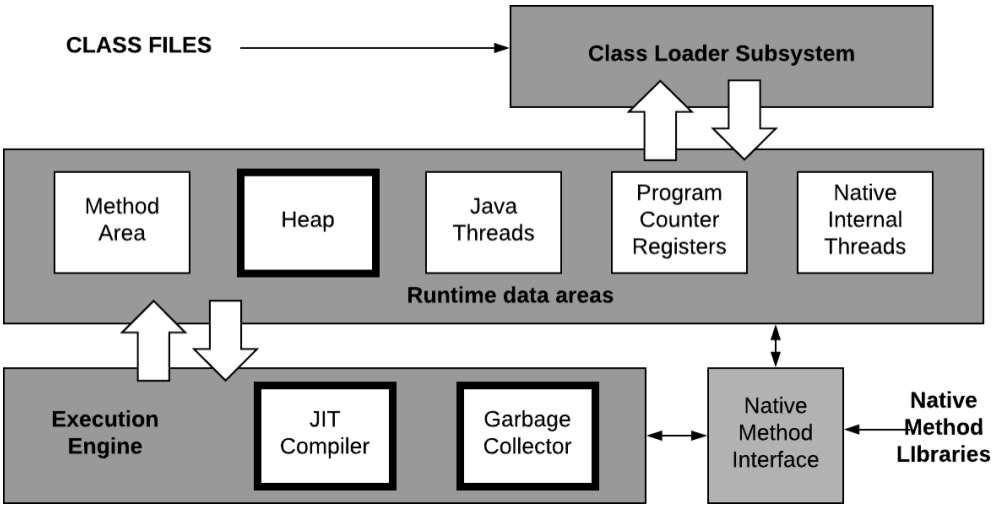

Fig. 2.1: Visual representation of the Java Virtual Machine with key memory components [2]

- **Heap:** In Java, the heap is a region of the memory that is used by the JVM to allocate memory to objects created by a program. Whenever an object is created, the JVM allocates memory on the heap to store the object's data. The size of the heap is determined by the JVM's memory settings.

  All class instances and arrays are allocated on the heap, and once an object is created, it can be accessed by threads running within the JVM. During program execution, the heap can become filled with objects, and the JVM will allocate more memory on the heap as needed until the available memory runs out.

  When the heap becomes full and no more memory can be allocated, the JVM will trigger a garbage collection process to reclaim memory by removing objects that are no longer being used by the program. This process can temporarily pause the program's execution, and its frequency and duration can affect the program's performance.

  The heap can be divided into smaller parts or generations, which include the Young Generation, Old or Tenured Generation, and Permanent Generation (Figure 2.2 [7]).

- **Garbage Collector:** The garbage collector is an essential component of the Java Virtual Machine (JVM) that automatically manages memory allocation and deallocation, freeing the programmer from the burden of manually managing memory. Its primary responsibility is to collect and remove objects that are no longer being used by the program and reclaim the memory space they were occupying. When the GC runs, it halts the execution of application threads, allowing it to inspect all the objects currently stored in the heap. The GC identifies the live objects that are still in use by tracing the references from the application's

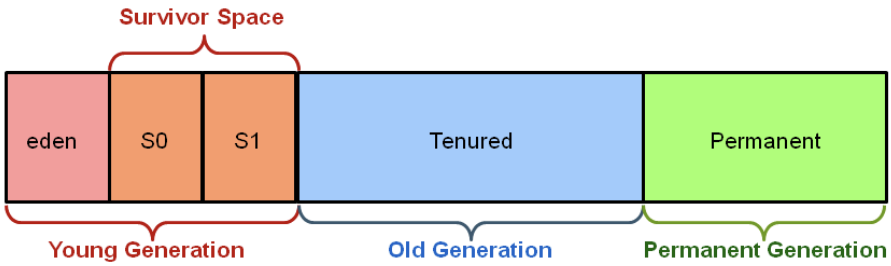

Fig. 2.2: Heap structure [7]

stack frames and any other objects in use. Any objects that are not identified as live are deemed garbage and can be safely collected. This process of identifying live and garbage objects is called marking. Once the marking phase is complete, the GC initiates a process called garbage collection. During this phase, the GC removes the garbage objects, freeing up memory for future use. The GC can employ different algorithms to collect garbage, such as mark and sweep, generational, or concurrent garbage collection. By freeing up memory space occupied by unused objects, the GC prevents memory leaks and reduces memory fragmentation, which can ultimately lead to better application performance and stability.

- **JIT compiler:** The JIT (Just-In-Time) compiler is a component of the JVM that is responsible for dynamically optimizing code at runtime. When the JVM executes Java code, it first converts the bytecode into machine code that can be executed by the CPU. The JIT compiler is responsible for generating this machine code. During the execution of the application, the JIT compiler traces information about the application's behavior and performance. This information is then used to make dynamic optimization decisions that can improve the application's memory usage and execution speed. For example, the JIT compiler may identify a portion of code that is frequently executed and consume a significant amount of memory. It may then optimize this code by reducing its memory usage, which can improve the application's overall performance. In general, the JIT compiler plays a crucial role in improving the performance and memory usage of Java applications by dynamically optimizing code at runtime.

2.3 Garbage Collectors in Java

Garbage Collector is an automated process of reclaiming memory by scanning through the heap and identifying the used and the unused memory. Even though the lower level programming such as C requires the manual allocation of memory, for the Object

Oriented Programming (OOP) languages, GC is an essential component in memory management. The automated process of memory management could be valuable since the manual memory management is prone to memory errors. Therefore, the OOP languages such as JAVA contain garbage collectors as a part of the JVM [2].

This section reviews the major garbage collectors including newer low-pause collectors and discusses improvements that were done over the years [3].

2.3.1 Serial and Parallel GC

The Serial GC is a simple collector, using a single thread to stop the world and collect both young and old generations. It employs a mark-sweep-compact approach and is usually only suitable for small heaps or constrained environments due to potentially long pauses. The Parallel GC (also called "Throughput GC") uses multiple threads during GC events to speed up collection within stop the world phases [8]. Both Serial and Parallel GCs are stop the world generational collectors: minor and major GCs are done with all application threads paused. Parallel GC excels at maximizing throughput on multi core servers but can suffer from long worst case pause times because it pauses the entire application for the full duration of a collection cycle [9], [8]. These traditional collectors are adequate for batch processing or scenarios where occasional long pauses are tolerable, but they struggle to meet low latency requirements.

2.3.2 Concurrent Mark Sweep (CMS)

Concurrent Mark Sweep (CMS) was an earlier low pause collector introduced to mitigate long pauses by performing most GC work concurrently with the application. CMS would mark live objects concurrently (reducing pause time) and then sweep/free dead objects without compacting the heap. It still had short stop the world pauses for initial marking and remark phases. CMS significantly reduced pause times compared to full stop the world collectors, but its concurrent nature could be impacted by fragmentation and it had to fall back to full GC in worst case scenarios. CMS was widely used in Java 5–8 for latency sensitive applications, but it has since been deprecated and removed (starting JDK 14) in favor of newer algorithms like G1 [10].

2.3.3 Garbage First (G1) GC

G1 GC is a region based generational collector that became the default in OpenJDK from Java 9 onward [8]. G1 partitions the heap into many fixed size regions and performs garbage collection in a mostly incremental fashion. It uses concurrent threads to perform global marking of live objects and identifies regions with the most garbage ("garbage first") to evacuate during stop the world phases. G1's design aims to provide controllable pause times, users can set a target max pause time, and G1 will try to

limit collections to fit within that budget by adjusting how many regions it collects in one GC cycle. During a G1 GC, some phases (global marking) happen concurrently, but object relocation (copying survivors and compacting regions) occurs during brief Stop The World (STW) pauses, region by region [8]. G1 thus balances throughput and latency. It does not achieve the very low pauses of fully concurrent collectors, but it dramatically shortens major GC pauses compared to Parallel GC in large heaps. G1 also handles heap fragmentation better than CMS by incremental compaction. Improvements in recent JDKs have enhanced G1's performance. For example, from JDK 8 to JDK 17 and 21, G1 has seen reduced pause times and better memory efficiency due to numerous optimizations [9]. Still, under extreme loads, G1 can experience fallback full GCs [11], so understanding workload characteristics is important.

2.3.4 Shenandoah GC

Shenandoah is a fully concurrent, low pause collector initially developed by Red Hat and integrated as an optional GC in JDK 12 [8]. Shenandoah performs both marking and compaction concurrently with running application threads, aiming to make pause durations independent of heap size. It achieves this by using read and write barriers to track object references and allow the collector to relocate objects in the background. Shenandoah introduced the concept of a Brooks Pointer or forwarding pointer for each object to enable compacting the heap while mutators run. The key benefit is extremely low pause times (typically in the low milliseconds or less, largely constant even as heap grows) [8]. The original Shenandoah was non generational (treating the heap as one region space), which simplified concurrent design at the cost of some throughput because all objects are managed uniformly. Recent developments have focused on adding generational capability to Shenandoah to improve throughput and memory usage. Generational Shenandoah was delivered as an experimental feature in JDK 24 [12], aiming to collect young objects more frequently while still maintaining concurrent collection, thereby reducing CPU overhead and memory footprint relative to the single generation mode. The goals for generational Shenandoah include improving sustainable throughput and avoiding edge cases where concurrent collection can't keep up with allocation spikes [12]. Although Shenandoah's concurrent algorithm imposes additional CPU overhead for barrier processing and more complex synchronizations [11], it is highly effective for applications that require consistent low latency GC behavior.

2.3.5 Z Garbage Collector (ZGC)

ZGC is another ultra low latency GC introduced as experimental in JDK 11 and made production ready by JDK 15 [13]. ZGC is designed to scale to very large heaps (multi terabyte) while keeping GC pauses typically well below 10 ms [8], often sub millisecond [9]. ZGC achieves this through a mostly concurrent, region based algorithm that

uses colored pointer bits and load barriers. When ZGC moves objects, it updates pointers lazily using a read barrier. Any time a reference is loaded, the barrier code checks if the object has been relocated (using metadata encoded in unused bits of 64 bit pointers) and if so, remaps it to the new location [8]. This way, all heavy lifting (marking, relocating, remapping pointers) happens while application threads run, and stop the world pauses are minimal (only needed for very brief root snapshotting or similar phases).

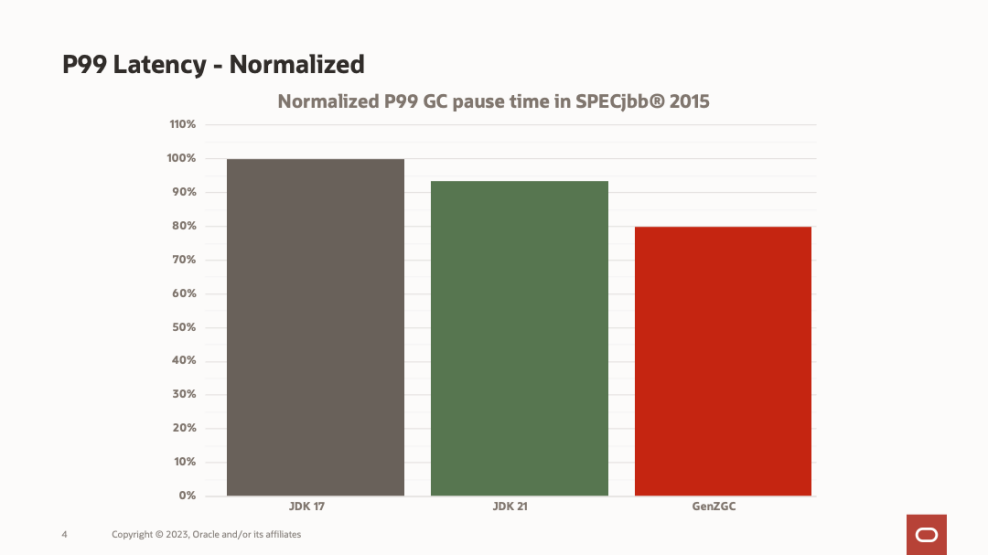

Fig. 2.3: Normalized 99th percentile GC pause time [9]

One consequence of ZGC's design is that it did not initially use generations, every collection considers the whole heap which can waste effort on short lived objects and use more CPU than necessary. In response, Generational ZGC has been developed and was introduced experimentally in JDK 21 [9]. Generational ZGC adds a young generation that is collected more often, allowing ZGC to reclaim garbage more efficiently without scanning the entire heap each time [9]. This enhancement led to notable improvements, where roughly a 10–15% throughput increase with Generational ZGC was observed compared to the single generation mode, while maintaining similarly low pause times [9] . Generational ZGC reduces P99 GC pause time by roughly 10–15% relative to the prior design (Figure 2.3 [13]. All tested modes keep P99 pauses under 1 millisecond. Even as Generational ZGC slightly increases complexity and native memory usage [9], it addresses edge cases like allocation stall scenarios. An allocation stall occurs when allocation outpaces the GC's ability to free memory; generational ZGC significantly postpones or prevents such stalls under high allocation pressure by focusing on young garbage first [13].

2.3.6 Epsilon GC (No Op Collector)

The Epsilon GC is an experimental GC for the Java Virtual Machine that was introduced as a part of the OpenJDK project [14]. Unlike traditional GC algorithms, Epsilon GC only handles memory allocation and does not have any memory reclamation mechanism implemented. This means that the Epsilon GC does not free up any memory, it simply allocates new memory as needed, but never frees up any of the memory that has been allocated.

The main advantage of using the Epsilon GC is that it eliminates GC pauses, which can result in performance improvements for applications that allocate and de allocate memory frequently. However, the trade off is that the memory usage will eventually increase without any reclamation mechanism, which can lead to OutOfMemoryError exceptions if the application continues to allocate memory without releasing it.

While not intended for production services (except perhaps in very constrained scenarios), Epsilon is useful in research and tuning to understand GC costs by contrast. It confirms the lower bound on pause time. Any practical GC will introduce some pauses or overhead, but Epsilon shows what performance looks like with no GC at all.

The proposed framework is to identify memory usage within JVM processes. When the memory exceeds the predefined threshold, a new container should be scheduled using a container orchestration platform such as Kubernetes. Therefore, the proposed solution will eliminate the GC pauses in the application.

2.3.7 Ongoing Trends

The trend in JVM GC development is toward concurrent, pauseless collection to meet modern application demands (microservices, interactive applications, low latency trading systems, etc.), but with new techniques to reclaim memory more efficiently. At the same time, even the "traditional" collectors (Parallel, G1) have seen improvements. For example, JDK 17 and JDK 21 G1 have shorter pauses and better throughput than their JDK 8 counterpart, showing that GC research benefits all use cases [9]. Selecting an appropriate GC for a given workload is crucial. It should be noted that no single GC is optimal for all situations, and the choice should be guided by whether throughput or latency (or memory footprint) is the top priority [9]. Recent research has even explored automatic GC selection based on application characteristics [8], underscoring the importance of tailoring GC strategy to the use case.

2.4 Memory monitoring

2.4.1 JDK

Memory monitoring in the Java Development Kit (JDK) can be achieved using several built in tools and APIs. Some of the most commonly used tools and APIs for memory monitoring in the JDK can be categorized as follows:

- **JConsole**: JConsole is a Graphical User Interface (GUI) that provides information about Java applications and services. JConsole can be used to monitor the memory usage of a running Java application, including memory usage over time, memory pool usage, and GC (garbage collection) statistics as illustrated in Figure 2.4 [15].

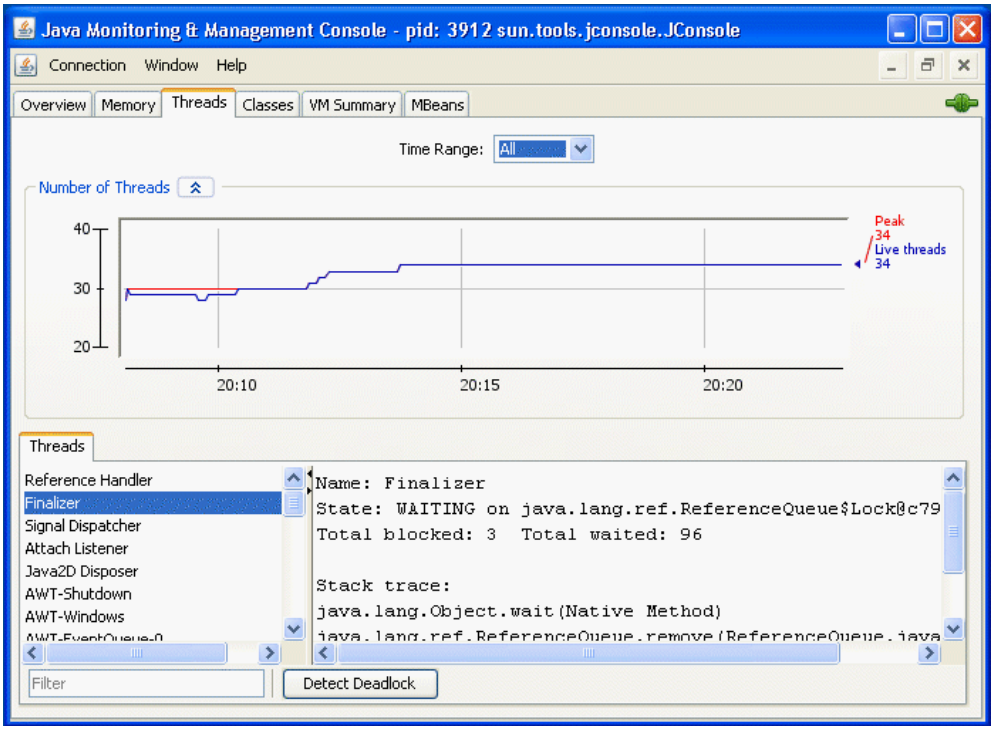

Fig. 2.4: Jconsole [15]

- **VisualVM:** VisualVM is a visual tool that provides detailed information about Java applications and services, including memory usage. With VisualVM, the memory usage of a running Java application can be monitored, including memory usage over time, memory pool usage, and GC statistics as shown in Figure 2.5 [16].

- **jmap:** jmap is a command line tool utilized to dump the memory contents of a running Java application. The jmap provides a platform to analyze the memory usage of a Java application and diagnose memory leaks.

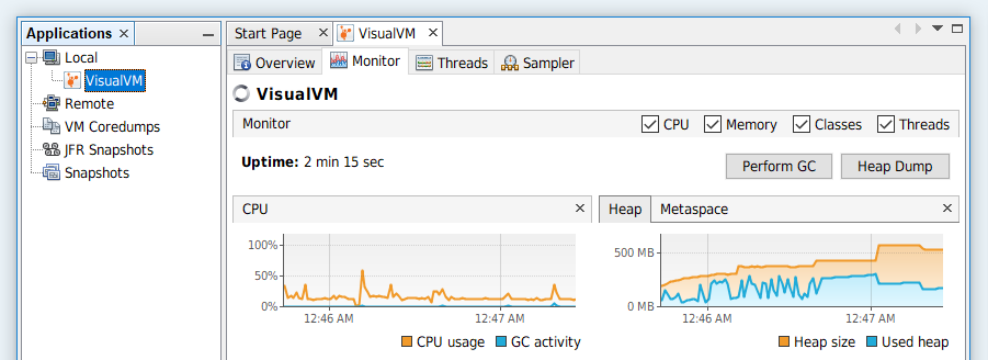

Fig. 2.5: VisualVM [16]

- **jcmd:** jcmd is a command line tool utilized to perform various operations on a running Java application, including memory profiling and heap dump. With jcmd, detailed information on the memory usage of a Java application can be obtained, including memory usage over time, memory pool usage, and GC statistics.

- **JMX API:** The Java Management Extensions (JMX) API is a set of Java interfaces and classes for monitoring and managing Java based applications and services. It provides a way for applications to expose their internal management and monitoring information in a standard format that can be accessed and manipulated by other applications and tools. The JMX API is used to create MBeans that expose information about the state and behavior of the application, and register them with an MBeanServer. Therefore, the JMX API is utilized to access and manipulate the information exposed by the MBeans. The JMX API provides a rich set of functionality for accessing and manipulating MBeans, including methods for querying the MBeanServer for information about registered MBeans, accessing and updating the attributes and operations of MBeans, and subscribing to notifications from MBeans.

- **GC Logging:** The Java Virtual Machine (JVM) can be configured to produce detailed GC logs using flags. GC logs record each collection event, its type (minor/major), duration of pauses, heap occupancy before and after collection, and other relevant details.

  These logs are invaluable for offline analysis of GC performance. Tools such as *GCViewer*, *GCeasy*, or *Splunk* analyzers can parse GC logs to compute statistics such as average pause time, maximum pause duration, frequency of GC events, and overall garbage throughput [17].

2.5 Impact of GC on Applications

In the previous sections, we have observed that over the years, there have been numerous developments to garbage collection (GC) algorithms. Despite years of optimization, GC pauses remain a significant factor affecting the performance and responsiveness of Java applications, particularly for large scale or latency sensitive systems. This section provides an overview of the nature of GC induced performance issues, supported by evidence from recent studies and industry experiences.

2.5.1 Throughput Costs

Garbage collection consumes CPU and can occupy a substantial fraction of total execution time for memory intensive applications. Studies have shown that GC can account for up to one third of an application's execution time in certain workloads [11]. In big data platforms, GC has been reported to consume as much as 50% of total processing time [18]. Every moment spent in GC is time not doing application work; thus, high GC frequency or long GC cycles directly reduce throughput.

Even concurrent GCs that do not pause the world can impact throughput by stealing CPU cycles from application threads (i.e., running GC work in background threads). For example, enabling a low-latency GC like Shenandoah or ZGC often incurs a slight throughput penalty ($\sim 5\%-15\%$) due to barrier overhead and concurrent thread load [11]. This is a classic latency/throughput trade-off: some applications might prefer a parallel stop the world collector that yields higher peak throughput at the expense of occasional pauses, whereas others will sacrifice some throughput to ensure consistent low latency.

2.5.2 Latency and Tail Latency

GC pauses can dramatically affect latency, particularly the tail latency (e.g. 90th or 99th percentile) of request serving applications. Dean and Barroso's seminal work on tail latency in warehouse scale computers highlights GC as a major contributor to long tail delays in managed runtime services [19]. When a stop the world GC occurs, all in flight requests handled by that JVM experience a hiccup. A pause of 100–200 ms (not uncommon for a full GC in a large heap when using a traditional collector) may be barely noticeable in a batch job but can be catastrophic for user experience or service level objective (SLO) compliance in an interactive service.

For instance, LinkedIn and Instagram engineering teams observed severe scalability issues on multi-core systems with large Java heaps due to lengthy GC pauses, which impaired their services' ability to meet latency targets [11]. In those cases, GC was identified as a key bottleneck when scaling up memory for caching: beyond a certain heap size, the GC pause times would spike, causing request timeouts or stalls.

Moreover, even shorter GCs can disrupt tail latency because, in modern microservice architectures, a single user request often fans out to many services. If any one of those services experiences a GC pause at the wrong moment, it adds to the end to end latency, leaving extremely tight budgets for avoiding such stalls [19]. A concrete example is given by Weinstock et al. (2017), who demonstrated that GC was one of the main causes of 99.9% latency outliers in a simple Java HTTP service [19].

2.5.3 Long Pauses and "Stop the World" Effects

A worst case scenario is a full stop the world GC on a very large heap (many gigabytes) using a non concurrent collector. Such events can pause an application for seconds or even minutes. Although modern collectors like G1 aim to avoid such occurrences, they can still happen (for example, if G1's concurrent marking falls behind and the heap is exhausted, a fallback full GC occurs). These long pauses can trigger cascading failures in distributed systems: if one service stalls, upstream services might retry or mark it as unhealthy, potentially causing traffic surges elsewhere or even a spiral of outages (a "metastable failure" scenario). Recent research in distributed systems resilience notes that GC induced stalls have caused real incidents, and ensuring fault tolerance against such runtime pauses is an important challenge [11].

2.5.4 Multi Tenancy and Interference

In cloud and containerized environments, multiple Java applications might share a machine. GC activity in one JVM can interfere with others by contending for CPU, memory bandwidth, or causing OS level scheduling latency. Concurrent GCs will compete for CPU, while stop the world GCs relinquish CPU but might all wake up simultaneously after the pause, causing bursty load.

The study [20] examined latency critical services co-located with batch workloads and found that, even with a low pause collector like ZGC, there was interference that hurt tail latency when resources were shared. They observed that co running applications with GC can inflate tail response times due to CPU contention and cache effects. In such multi tenant setups, coordinating GC or isolating its impact becomes important.

Garbage collection pauses pose a dual challenge: they can reduce overall throughput by consuming CPU, and they can inflate response times by introducing delays. The severity of the impact depends on the choice of GC algorithm, heap size, allocation rate, and workload patterns. Large heaps with high allocation churn are at the highest risk of experiencing problematic GC behavior. Over the last few years, the JVM has made strides in reducing pause times and improving predictability, yet even "pauseless" GCs shift the problem to CPU overhead, which under heavy load can manifest as latency spikes due to CPU starvation. Therefore, modern research and practice not

only improve GC algorithms themselves but also explore application level and system level strategies to cope with GC.

2.5.4.1 Experiment evaluation [2]

An experiment conducted in [2] provides further insight into the impact of GC on applications. The following list briefly describes the benchmarks used, each of which was tested against several common GC algorithms to measure both the total number of garbage collections and the total execution time spent in GC:

- **avrora:** A single client threaded application (internally multithreaded) that simulates a number of programs running on a grid of AVR microcontrollers.
- **fop:** A single threaded application that takes an XSL-FO file, parses it, and creates a PDF file.
- **tomcat:** A multi-threaded application that sends multiple requests to the server for processing.
- **h2:** A multi-threaded application which executes JDBC-like in-memory benchmarks by running transactions against a banking model.
- **pmd:** A single-threaded (internally multithreaded) application that analyzes Java classes for source code issues.
- **xalan:** A multithreaded application that transforms XML documents into HTML.

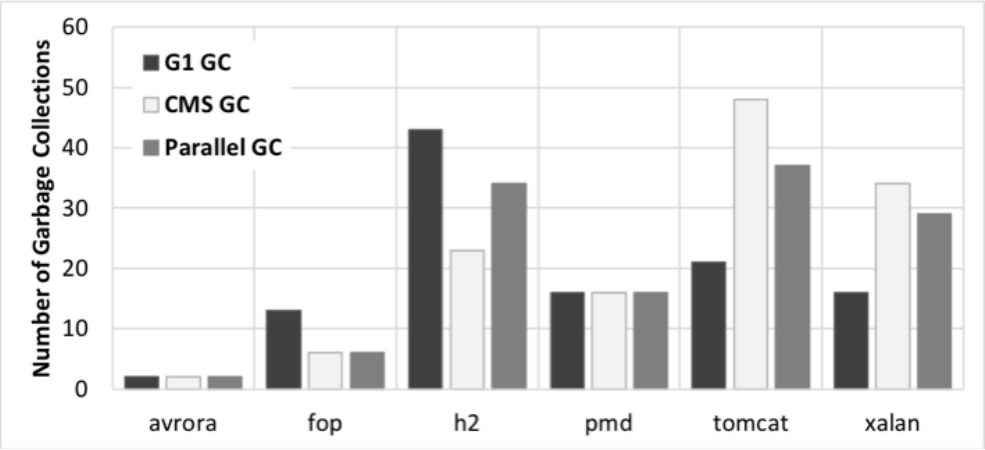

Fig. 2.6: Total numbers of executed garbage collections per application across three GC algorithms [2]

During each GC event, the application freezes all operations. Figures 2.6 and 2.7 illustrate the frequency and total duration of these “stop-the-world” intervals across

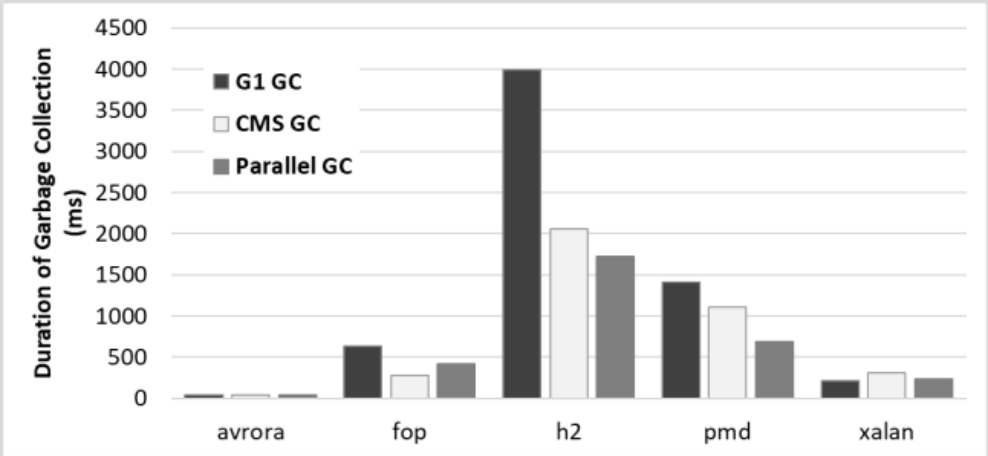

Fig. 2.7: Total execution time of garbage collections per application expressed in milliseconds [2]

different GC algorithms and applications. The time taken for a GC cycle also varies based on the application's characteristics and the collector used. Other experiments have observed that GC can consume up to 50% of the JVM execution time, with pauses as high as 300 seconds in extreme cases [18], wasting a considerable amount of CPU time in garbage collection overhead.

2.6 Improvements to the traditional GC operation

As discussed in the previous section, GC is one of the common causes of performance degradation in Java applications. Also, a best fit GC algorithm to match all the types of application is not available. The GC is especially sensitive to the heap size and variations of heap size would result in a major impact on the GC behaviour [21]. Therefor a variety of approaches have been proposed in the literature and used in practice to reduce the impact of GC pauses on application performance. This section examines several novel approaches used to mitigate the performance impact due to GC.

2.6.1 GC-Aware Load Balancing and Scheduling

With the requirement of high availability, it is common practice to deploy systems in a clustered environment. In a typical cluster, workload is distributed among instances using a load balancer. Therefore, several research efforts have been made to create efficient load balancing algorithms based on various criteria. In [22], a load balancing algorithm was improved by considering the utilization of Java Virtual Machine (JVM) threads, memory, and CPU to determine a proper load distribution strategy. Similarly, [20] focused on balancing the load across the available Enterprise Java Beans (EJB) instances by considering their utilization.

The main focus of [22] is to minimize the impact of major garbage collection using load balancers. As illustrated in Figure 2.2 [7], the Old Generation is a part of the Java heap where objects that have survived a certain number of minor garbage collections are moved to. These objects are referred to as "long-lived" objects, and they are expected to survive for a longer period of time than objects in the Young Generation. When the Old Generation fills up, a major garbage collection is triggered to free up memory. This is also known as a full garbage collection, and it involves stopping all application threads while the garbage collection process is carried out.

In [18], the authors identified that GC can consume up to 50% of the JVM execution time, with pauses as high as 300 seconds, and that major garbage collection represented more than 95% of those pauses under the heaviest workload.

Therefore, [22] suggests a GC-aware load balancing mechanism (shown in Figure 2.8 and Figure 2.9 [18]) wherein an algorithm predicts which application nodes are close to a major garbage collection and routes traffic to other nodes. During testing, the average response time decreased by 22.5% across all tested programs, while the average throughput increased by 60.7% across the tested programs.

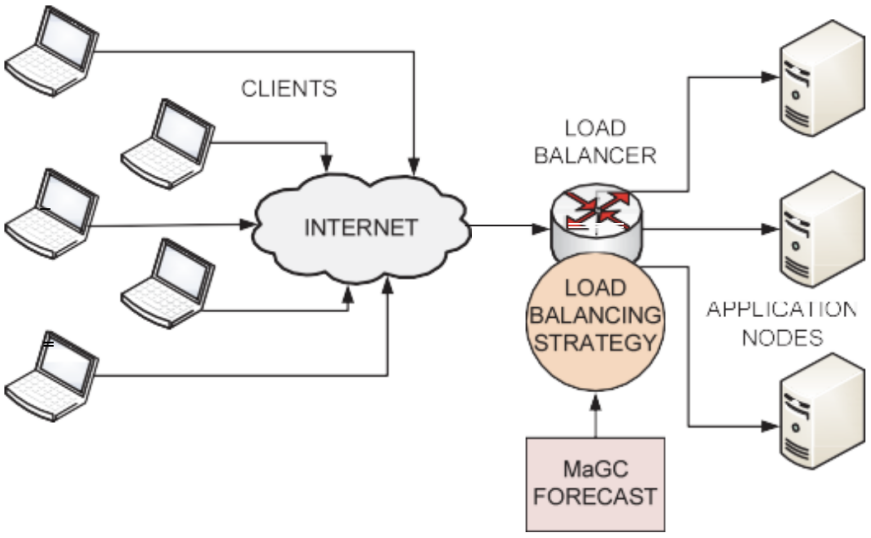

Fig. 2.8: GC-Aware Load Balancer [22]

Proactive GC Prediction. In distributed systems or clustered server environments, one effective strategy is to incorporate GC awareness into load balancing decisions. The goal is to route work away from nodes that are about to experience (or are undergoing) a GC pause, thus shielding users from the pause. The research [23] introduced "TRINI," an adaptive load balancing algorithm that uses GC forecasting to improve cluster performance [23]. TRINI predicts when a JVM will perform a major GC and temporarily reduces or defers assigning new work to that node. By doing so, the cluster avoids sending requests to a server at the exact time it pauses, thereby preventing

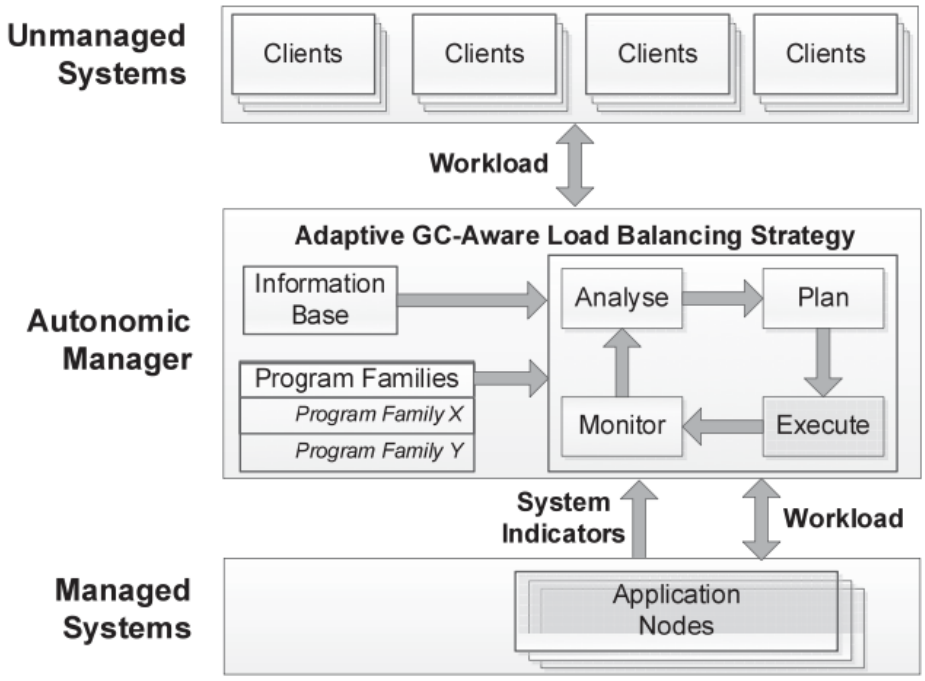

Fig. 2.9: Adaptive GC-Aware Load Balancing Strategy [22]

request latency spikes. In their extended evaluation, TRINI achieved significant performance improvements in a clustered Java system by avoiding GC overlap with request processing, and it scaled well with cluster size [23]. This illustrates the benefit of anticipating GC events. Key to such approaches is an accurate prediction mechanism: TRINI used heuristics based on heap usage and past GC intervals to guess when the next GC would occur. More recent work has explored machine-learning-based predictors for GC timing, but even simple threshold-based triggers can be effective.

Replica Selection and Request Routing. Similarly, replica selection techniques choose which server instance should handle a request based on the replica's internal state (including GC status). For example, if one out of N replicas is detected to be in a GC pause, a smart client or load balancer can avoid that replica temporarily. Modern service meshes or load balancers can use health-check pings or JVM telemetry (if exposed) to infer GC activity. In academic literature, black-box approaches treat increased response time or lack of reply as a signal to route away from a node [11]. However, black-box methods may only react after a pause begins. More advanced techniques share GC state: e.g., a custom load balancer could subscribe to GC notifications (via JMX or a dedicated application-level signal) to know a node is about to pause.

An example of an application-layer solution is described [11], which categorize "replica selection" as a means to pick the fastest server, potentially using prior latency as an indicator. GC-aware replica selection extends this by incorporating signals of

memory pressure. If one replica's heap usage is close to 90% (with a rising allocation rate indicating an imminent GC), a load balancer could proactively send traffic elsewhere until that replica's GC completes and heap pressure is relieved.

Distributed Systems with Failover. In systems with request failover and replication, it is also possible to integrate GC-awareness into the failover logic. The research [11] introduced a JVM modification called Mitigating Memory-induced Delays (MITMEM) , which provides an interface for the JVM to reject incoming requests if it predicts a long GC pause. By immediately returning a rejection, the client can retry on another replica, effectively avoiding timeouts. MITMEM's evaluation on Apache Cassandra (with replication) showed up to a 99% reduction in tail latency by swiftly failing over requests during GC, with negligible throughput impact [11]. This works because distributed systems often have redundancy: by sacrificing one replica during GC, the rest can handle the load without incurring timeouts.

Scheduling GC During Low Load. GC-aware load balancing can also be applied in scheduled or batch scenarios. If a web application has predictable daily traffic patterns, one could programmatically trigger major GCs during known low-load windows such as midnight to minimize impact. Some JVM flags allow applications to request a mostly concurrent GC, reducing the pause time even for explicit GCs. By coordinating such triggers with a load balancer that marks a node as temporarily out of rotation, one can "take turns" pausing nodes and rotating traffic to healthy ones—akin to rolling maintenance. This technique has been used in large-scale systems, where a load balancer stops sending new requests to a server, signals it to perform a GC, and then returns it to service once it finishes.

In summary, GC-aware load balancing requires,

- Detecting or predicting GC events on each node,
- A mechanism to quiesce or drain traffic from that node,
- Logic to redistribute traffic to others.

When implemented, it can almost entirely hide GC pauses from end-users at the expense of allocating more capacity (since effectively one node at a time may be underutilized during GC). As GC algorithms become more concurrent and pauses grow shorter, the necessity to route away might diminish. However, in systems with strict tail-latency requirements, this added complexity

2.6.2 Idle-Time Garbage Collection

A framework to mitigate the impact of GC on distributed applications is proposed in [24], and the authors demonstrate it using an application running on Apache Spark.

A major concern with traditional GC algorithms is that, regardless of their efficiency, they still cause the application to freeze during GC. Therefore, triggering GC cycles during the idle phases of an application can help mitigate the effect of these pauses.

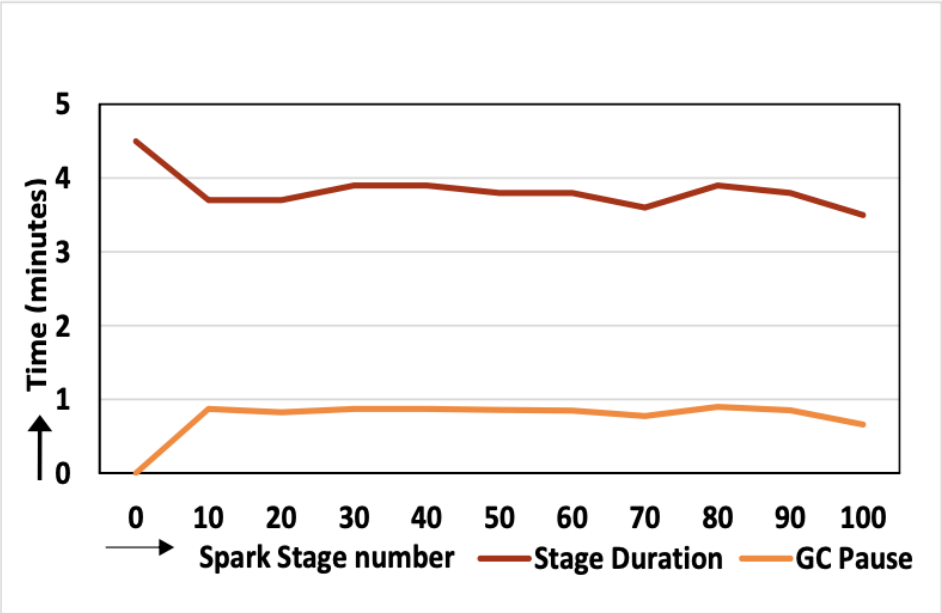

Fig. 2.10: GC impact on Spark PageRank execution [24]

Figure 2.10 (adapted from [24]) plots execution time against Spark application stages. The total time to execute a given stage is shown in brown, while the average GC pause time during that stage is shown in orange. The data come from a PageRank application running 100 iterations on an 8-node cluster, where each executor has 2 GB of Java heap. As illustrated, a significant amount of time (approximately 20–25% of the total) is spent on GC pauses during Spark's execution.

The proposed framework detects idle phases in the Spark application. Once these idle phases are identified, a GC is triggered via the JMX interface whenever the Java heap occupancy exceeds a predefined threshold. Since idle times are common in many workloads, this framework can be used to reduce noticeable GC interruptions.

Figure 2.11 [24] shows a timeline of a Spark stage's execution profile, where multiple parallel tasks are processed by different executor JVMs. The green bars represent the task execution times for a PageRank stage (100 iterations) across five of the eight nodes in the cluster. One executor is a "straggler," taking significantly longer than others to complete its assigned tasks. Meanwhile, other executors finish earlier and remain idle for nearly a minute. Across many stages, this idle time averaged about 30 seconds, offering an opportunity to run GC. Using this framework, GC pauses were reduced by roughly 25–30%, leading to a 3–7% overall speedup in execution time.

Beyond Spark-centric approaches, the broader concept of idle-time garbage collection

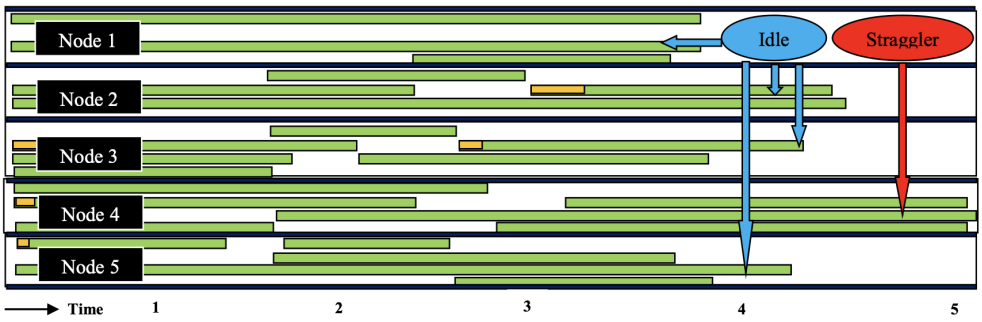

Fig. 2.11: Spark stage execution event timeline [24]

is to exploit periods when an application is idle or under low load to perform garbage collection, thereby reducing interference during peak activity. The general idea is to run GC at times when users are less likely to notice its impact.

Idle-Time in Managed Runtimes. This approach has seen success in other managed runtime environments, such as JavaScript engines. For example, Google's V8 engine uses idle-time GC to proactively clean up memory when it detects that the browser is idle, which significantly reduces memory bloat without affecting user experience. In Chrome's Gmail, this strategy reclaimed approximately 45% of the heap during idle periods [25]. The key is to have a reliable indicator of "idleness." In a GUI or interactive system, the lack of user input or frame rendering is a clear idle signal.

On the server side, true "idle" time may be rare, but many applications have daily or weekly load fluctuations. A trading system, for example, might schedule GC runs just before markets open, ensuring a large amount of free heap and minimal GC activity during the critical trading window.

Explicit GC Triggering. While the JVM does not generally allow scheduling a GC at a precise time, an application can manually invoke or trigger GC through the JMX `MemoryMXBean`. By default, this induces a stop-the-world full GC, which is not ideal under load. However, if done during a maintenance window, it can compact the heap and reduce the chance of an unexpected full GC later.

Heuristic Tuning for Throughput vs. Latency. Current JVM ergonomics can sometimes leverage idle-ish conditions automatically. For example, certain collectors may initiate concurrent GC cycles when CPU availability is high or if the allocation rate slows down. These "heuristics" can inadvertently coincide with idle moments. However, real-world request patterns may not perfectly align with these heuristics, so researchers have explored adaptive GC triggering. One idea is to tie GC to a "throughput slack," running partial collections early if the system is comfortably meeting its latency targets, thus preventing a larger GC pause later [20].

Operating System Idle Detection. Some have proposed using OS-level idle detection—when the CPU is 90% idle, for example—to prompt the JVM to perform GC

(akin to how background tasks run in an OS when the machine is not busy). Currently, HotSpot (as of JDK 17–21) does not offer a built-in feature for OS-driven idle GC triggers, but it could be simulated with tooling. An external script could monitor system load and run whenever the load is below a certain threshold.

Latency-Driven GC Scheduling. A more refined variation can be found in real-time Java systems, where GC occurs in small incremental slices during idle gaps of a few milliseconds (cf. certain real-time Java implementations. Even busy systems typically have micro-idle periods while waiting for I/O, and exploiting these intervals for partial GC can reduce the likelihood of long pauses.

While idle-time GC does not reduce the total amount of garbage collection work, it shifts that work to more convenient times. Ideally, an application enters peak load with a relatively clean heap, having collected garbage during previous lulls. The chief risk is mispredicting idle intervals. If a sudden load spike arrives when the GC is still running, performance can degrade. Consequently, it is often best to combine idle-time GC with other controls, such as concurrency or predictive heuristics. Although not all JVMs provide an "out-of-the-box" idle-GC feature, certain collectors continuously collect and thus naturally utilize available CPU slack, while some APM tools can trigger GC on demand. Overall, leveraging idle time remains a viable strategy for mitigating GC-induced pauses, especially in applications with predictable load cycles.

2.6.3 Load Shedding and Request Management during GC

The impact of garbage collection (GC) on cloud web services was investigated in [26], which highlighted how GC-induced pauses can introduce non-deterministic overhead to service times, particularly in managed languages like Java. Such pauses can significantly inflate tail latencies, undermining the low-latency promises of cloud-deployed services.

2.6.3.1 Garbage Collection Control Interceptor (GCI)

One of the central contributions in [26] is an algorithm known as the Garbage Collection Control Interceptor (GCI) . GCI functions as a decentralized request interceptor that detects when the runtime heap is being cleaned by the GC and sheds incoming load during this interval. By marking the service instance as temporarily unavailable to the load balancer, GCI prevents new requests from queuing behind a GC pause. Instead, the load balancer receives an HTTP 503 "Service Unavailable" response and redirects traffic to another service instance. This effectively avoids subjecting active requests to lengthy wait times. Figure 2.12 illustrates the main GCI components:

- **The Proxy**: A runtime-agnostic intermediary that manages garbage collection signals and load shedding behavior.

- **The Request Processor**: A lightweight in-service layer that checks heap allocation status, invokes garbage collection, and notifies the proxy of ongoing GC activity.

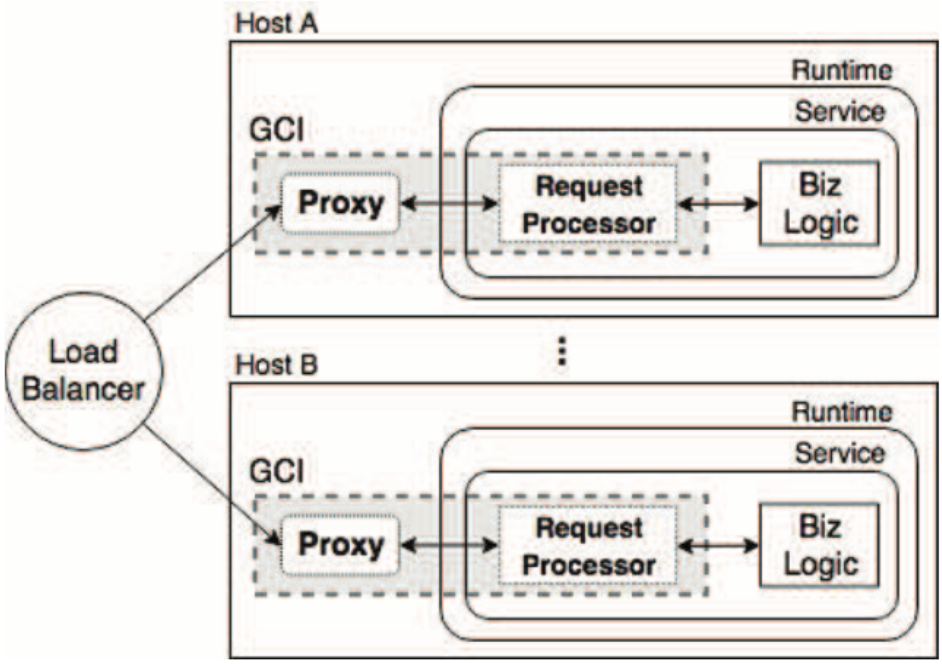

Fig. 2.12: GCI Architecture [26]

Figure 2.13 [26] shows how GCI manages requests in practice. If the GC is running and the service is unavailable, GCI returns a 503 response to inform the load balancer, which redirects the request elsewhere. This design proved effective in a prototype stateful service, eliminating GC-induced tail latency in both small and large deployments with minimal throughput loss.

2.6.3.2 Decentralized Load Shedding Approaches

In parallel work, Weerasinghe et al. (2017) propose a similar method of shedding load during GC [19]. The authors deploy an HTTP interceptor that monitors GC activity and temporarily rejects or defers incoming requests whenever a stop-the-world event is detected. Experimental results indicate up to a 40% reduction in 99.9^{th}-percentile latency with only minor reductions in throughput. By failing fast (via 503 responses), the system prevents requests from experiencing a large queuing delay behind a GC pause. Under typical load-balancing or client retry strategies, these requests are quickly handled by another instance, reducing the user-visible delay.

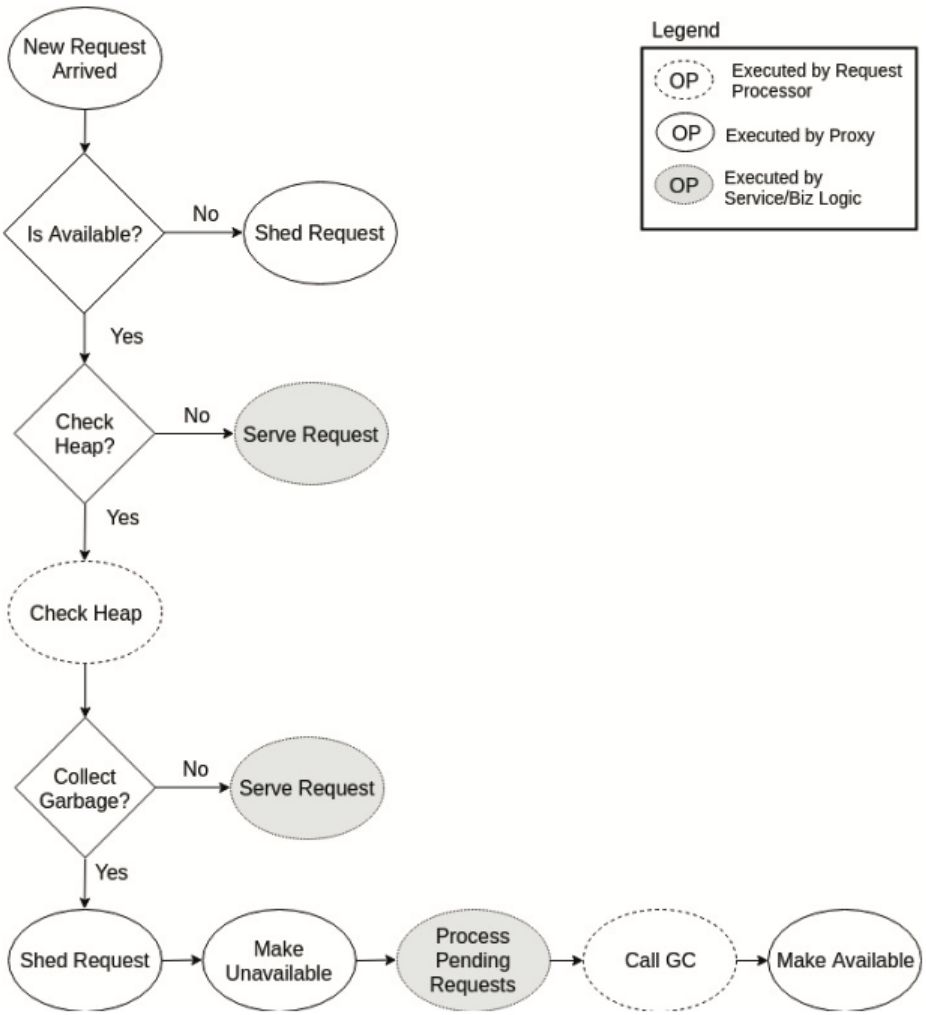

Fig. 2.13: GCI flow [26]

Instead of attempting to reduce GC interruptions inside the JVM, this line of research treats GC pauses as periods of intentional "node unavailability" that appear transiently. The overhead is relatively small because the number of requests shed during a pause is limited, and many systems can tolerate brief 503 responses if retries are quick.

2.6.3.3 Request Cancellation and Fast Failover

A more advanced approach appears in the MITMEM system [11], which places a dedicated thread in the JVM to predict or detect GC events. If a GC pause is imminent or underway, that thread intercepts incoming requests and issues a fast rejection (or "Server Busy" response). This encourages clients to re-route requests to other replicas immediately. MITMEM can optionally queue the requests if the pause is expected to be short, but for potentially long pauses, the system opts to fail fast. Evaluations on

Apache Cassandra show that this strategy greatly lowers tail latencies by ensuring that no request is forced to wait through a full GC.

2.6.3.4 Adaptive Load Shedding Strategies

Load shedding can be applied selectively or with prioritization. For instance, critical requests might still be served (if the GC can complete quickly), while lower-priority requests are shed to minimize latency for premium traffic. Additionally, careful queue management is advisable to avoid an influx of requests immediately after GC completes, which can create bursty CPU usage or trigger further GCs. Many systems thus prefer shedding (dropping) rather than queueing requests during longer pauses [19].

Another consideration is client-side timing. In microservice environments with circuit breakers, a GC pause can trigger the circuit for that instance, causing external components to route around the paused service. Some real-time or high-frequency trading systems coordinate GC cycles among producer-consumer pairs to ensure both are not paused simultaneously, though that level of synchronization is uncommon in general web services.

Overall, shedding or deferring requests is an effective way to mitigate the user-visible impact of unavoidable GC pauses. Although it introduces a small risk of higher failure rates (due to 503 or "Server Busy" responses), well-tuned client retries and load-balancing algorithms typically mask these short disruptions. Literature strongly supports this trade-off in tail-latency-sensitive services [19, 26], as significantly reduced tail latency is often more desirable than processing every request under degraded conditions.

2.6.4 Eliminating OS-Caused Large JVM Pauses

Platform as a Service (PaaS (PaaS)) is a popular cloud paradigm that simplifies application deployment and management for developers. However, operating system (OS) factors can introduce additional overhead into Java Virtual Machine (JVM) processes, including garbage collection (GC). In [27], the authors investigate the effect of OS-level interactions on GC performance within PaaS environments and identify occurrences of large GC pauses related to I/O operations. One notable culprit is GC logging: even though GC logs are written using a buffered mode, the logging calls can still experience significant blocking due to OS-internal mechanisms, such as flushing or lock contention on I/O buffers.

Four approaches were proposed in [27] to address GC pauses caused by GC logging:

- Enhancing the JVM
- Reducing background I/O

- Improving application I/O
- Separating GC logging from other I/O

Among these, the strategy of *separating GC logging from other I/O* yielded more reliable results and was relatively straightforward to implement. Beyond these measures to mitigate GC overhead, other lines of research have explored region-based memory management, moving away from the traditional Java heap to native memory, or devising distributed frameworks that coordinate GC across multiple nodes [24].

2.6.4.1 OS-Level and System-Level GC Pause Mitigation

While Java Virtual Machine (JVM) and application-level adjustments can reduce Garbage Collection (GC) overhead, additional mitigation techniques operate at the Operating System (OS) or hardware level. The overarching objective is to manage system resources so that GC activity imposes minimal disruption to the application's execution and to co-located processes.

2.6.4.1.1 CPU Core Isolation and Scheduling. One effective strategy is to isolate GC threads from application (mutator) threads at the Central Processing Unit (CPU) -core or hyper-thread level. By doing so, GC threads run on dedicated cores, preventing CPU competition and reducing the likelihood of application slowdowns. Zhao et al. [20] developed "iGC," a middleware layer that bridges the JVM and OS scheduler to achieve such core separation. In a machine with multiple cores, for example, GC threads might be pinned to a dedicated core or set of hyper-threads, ensuring that the application's threads are unaffected by GC activity. Experiments on multi-tenant NoSQL (NoSQL) databases demonstrated up to 4× shorter tail latencies when co-located JVMs no longer interfered with each other's GC cycles [20].

Modern OSs already provide partial solutions via tools like `cgroups` (to limit CPU shares), `taskset` (to pin processes to specific cores), and the Realtime scheduling class (which can prioritize critical application threads over GC threads). Lowering the scheduling priority of GC threads guarantees that they only use CPU cycles when the system is otherwise idle, although this risks slower GC progress if the collector cannot keep pace with allocation demands.

2.6.4.1.2 Coordinated Scheduling in Multi-Tenant Systems. When multiple JVMs or diverse tasks share the same hardware, a more sophisticated OS scheduler can further reduce pause impact. For instance, [20] is a research scheduler offering microsecond-scale core allocation and preemption. Although not GC-specific, such a scheduler can detect a sudden drop in runnable threads during a GC pause and quickly reassign the freed CPU resources to other tasks. Conversely, when the GC

completes and the application threads resume, the scheduler can immediately provide those threads with sufficient CPU to catch up. This dynamic reallocation reduces queuing delays and helps maintain quality of service even with multiple "noisy neighbors" on the same machine [20].

2.6.4.1.3 Memory and I/O Isolation. GC can also interfere with memory bandwidth and Input/Output (I/O) . A collector's rapid scanning can evict shared CPU caches used by other threads or processes, leading to increased cache misses once the application resumes. Hardware partitioning of caches if supported or staggering GC start times across multiple JVMs can reduce such thrashing. In cloud environments, providers can schedule when new JVM instances (which often incur heavy GC and just-in-time overhead) come online, preventing simultaneous high-load GC events.

OS-level settings such as Transparent Huge Pages (THP) also influence GC performance, enabling large pages can reduce Translation Lookaside Buffer (TLB) misses. Avoiding swapping is crucial, if GC triggers disk paging, pause durations can balloon significantly. Thus, carefully configured memory limits, or constraints help ensure GC time remains predictable.

2.6.4.1.4 Garbage Collector Optimizations via OS Hooks. Another direction involves direct OS-JVM coordination. A GC might query the OS for idle CPU cores, accelerating its parallel phases when surplus capacity exists, or scaling back when the system is busy. Advanced scheduling on Non-Uniform Memory Access (NUMA) systems can reduce GC overhead for large heaps by localizing threads to the same NUMA node that holds their data. Work by Google on improved GC work-stealing also highlights benefits from carefully pinning GC tasks to specific NUMA domains to avoid unnecessary cross-node traffic.

2.6.4.1.5 Real-Time Java and Pauseless Execution. Research into real-time Java environments, such as IBM's Metronome GC or JamaicaVM, demonstrates that tight integration with a real-time OS can achieve GC pauses on the order of microseconds by interleaving GC tasks with application threads at a fine granularity. These approaches can meet strict pause targets but come at the expense of lower overall throughput and increased complexity.

2.6.4.1.6 Operating System Signals and External Coordination. A practical, if somewhat ad hoc, technique involves using Linux signals. For instance, sending a SIGQUIT (`kill -3`) [13] to a HotSpot JVM triggers a thread dump and can confirm if threads are stalled in a GC safepoint. While primarily for debugging, such signals could help external monitors detect simultaneous GC events across multiple JVMs and attempt to stagger them—though this is not common in production.

In summary, system-level and OS-level mitigation strategies revolve around ensuring that GC is either (a) allocated exclusive or nonintrusive use of resources or (b) scheduled so that it does not unexpectedly preempt critical work. CPU core isolation can render a stop-the-world GC almost invisible to the application's other threads. Smart schedulers or partitioning can further shield co-located workloads. Combined with the application-level strategies discussed previously, these methods constitute a holistic defense against GC pauses, extending from hardware resources up through the software stack to maintain predictable performance.

2.7 Containerized deployment

In the previous sections, an overview of the operation of traditional GC is provided and the use of alternative solutions to eliminate or minimize the impact of GC latencies is discussed. The above analyzed solutions provided in research [24], [22], [27], [26] uses a conservative form of deployments. With the advancement of the cluster management, several sophisticated solutions have been developed with the containerized deployment which provides more advantages to the users. This section describes the utilization of containerized deployments to effectively manage a cluster deployment.

Containerized deployment refers to the practice of packaging applications as containers and deploying them using container orchestration platforms like Kubernetes. Containers provide a consistent and isolated environment for executing applications, simplifying the deployment and management of applications across different environments, including on-premises and in the cloud.

In containerized deployment, each component of an application is packaged into its own container, which provides the ability to run on any host machine with a container engine, such as Docker or Kubernetes. This allows developers to separate the application into smaller components providing more isolation between components, which results in enhanced security.

When using containerized deployment in the context of the Epsilon GC, a new container is scheduled when the memory usage within the JVM process reaches a predefined threshold. This allows the application to continue running without being impacted by GC pauses and memory constraints, as the new container provides a fresh and isolated environment with a new instance of the Epsilon GC.

Containerized deployment provides several benefits for managing and deploying applications, including improved scalability, portability, and security, and is widely used for modern application development and deployment.

2.7.1 Kubernetes

Kubernetes is an open-source platform for automating deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a unified platform for deploying and managing containers, providing a simplified platform to develop, deploy, and run modern applications in a scalable and resilient manner.

Kubernetes offers significant features that makes it an ideal platform for managing containerized applications as discussed in [28], including:

- Automated deployment and scaling: Kubernetes automates the deployment, scaling, and management of containerized applications, facilitating the ability to run applications in a scalable and resilient manner.
- Load balancing and self-healing: Kubernetes provides built-in load balancing and self-healing capabilities, ensuring high availability for applications.
- Storage management: Kubernetes provides a unified storage management system facilitating the ability to manage and access data from containers.
- Service discovery and orchestration: Kubernetes provides service discovery and orchestration capabilities, simplifying the process of management and coordination of communication between different components of an application.
- Kubernetes is widely used for modern application development and deployment and has become one of the most popular platforms for managing containerized applications.

The Kubernetes platform provides a unified platform for managing the deployment of containerized applications, ensuring the implementation of proposed solution to eliminate GC pauses in JAVA applications. With Kubernetes, the monitoring application can focus on identifying when to schedule a new container and eliminate GC pauses, while the platform takes care of managing the deployment and scaling of the containers.

Kubernetes allows to retrieve information about the memory usage of a container through the Kubernetes API. This information can be accessed by making HTTP requests to the Kubernetes API server.

The method for retrieving information about container memory usage through the API depends on the version of Kubernetes, the programming language and tools are being used to make API calls.

2.7.1.0.1 Prometheus Prometheus is an open-source monitoring and alerting system being used to collect metrics from various targets such as applications and system components, store the data, and provide a way to query the data to generate reports and dashboards. Prometheus is commonly used in combination with Kubernetes to monitor the performance and resource usage of applications running in the cluster. The system provides a flexible query language to define custom metrics, alerting rules, and dashboards, making it an ideal solution for monitoring dynamic, cloud-native applications.

2.7.2 Docker

Docker provides a standard format for packaging applications as containers, making it easier to distribute and run applications in different environments. The use of Docker containers in combination with the Kubernetes platform enables organizations to implement a consistent and reliable application development and deployment workflow, making it easier to build, test, and deploy modern applications at scale [29].

Docker provides several benefits for software development and deployment, as follows:

- **Portability:** Docker containers can run on any system with a Docker engine installed, providing a simplified platform to develop, test, and deploy applications across different environments.
- **Isolation:** Docker containers provide a consistent and isolated environment for running applications with enhanced security and stability.
- **Ease of use:** Docker provides a simple and intuitive interface for creating, deploying, and managing containers.
- **Automation:** Docker provides automated tools for building, deploying, and managing containers, making it easier to manage the life-cycle of applications.

Docker has become one of the most popular platforms for containerizing applications and is widely used in modern software development and deployment workflows.

2.7.3 Readiness and liveness probes

Readiness and Liveness probes are two important features in Kubernetes for ensuring the health and availability of containers in a deployment [28].

Readiness probes determine whether a container is ready to receive traffic. When a readiness probe fails, Kubernetes will stop sending traffic to the container, but the container will remain in operation. This ensures that a container is fully started and initialized before receiving traffic.

Liveness probes determine whether a container is still running and healthy. If a liveness probe fails, Kubernetes will restart the container which becomes useful for automatically recovering from failures or crashes, and for ensuring that a container continues to run and remain healthy over time.

By using Readiness and Liveness probes, Kubernetes can provide automatic health checks and self-healing capabilities for containers, ensuring that applications are running smoothly and reliably. In the context of the proposed solution to eliminate GC pauses in JAVA applications, these probes can be used to monitor the health and performance of the containers and to ensure that they continue to run smoothly and without interruption.

2.7.4 Kubernetes Autoscaler Based on Pod Replicas Prediction

The previous sections, the cluster deployments and the associated technologies are described in detail. This section describes the research work conducted in [30] on scaling or shrinking the cluster based on the requirements proposed in this thesis.

The solution proposed in [30] consists of three main components: a monitoring module, a forecasting module, and a scaling module as shown in Figure 2.14 [30]. These components are deployed as services in Kubernetes and communicate with the Kubernetes API server. The monitoring module is responsible for collecting resource usage statistics in Kubernetes and obtaining resource index data through the Prometheus System and the metrics-server component in Kubernetes. The data collected by the monitoring module is utilized by Kubernetes to determine desired Pod replicas. The prediction module estimates the desired Pod replicas based on the historical data and the scaling module uses this information to determine whether to scale up or down as in Figure 2.15 [30].

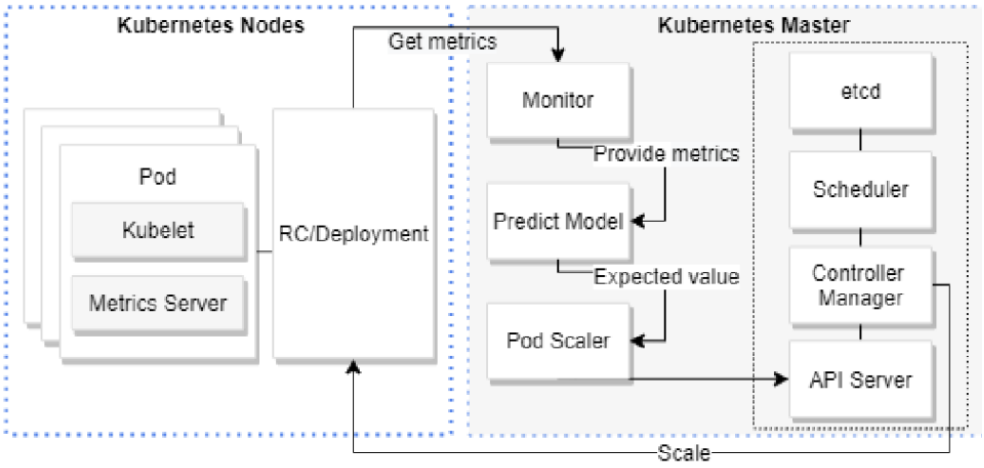

Fig. 2.14: Auto scaler System structure [30]

Begin
Where true do
For(Every Pod in Deployment)
Monitor u_t
Calculate u_a, Num_{target}
Store t, Num_{target}
If $Num_{target} > Num_{current}$
PredictNum_{target}
Invoke Pod Scaling up
End

Fig. 2.15: Pod Expansion [30]

In [30], a solution is proposed to dynamically scale up or down the number of server instances based on the incoming traffic. However, the same principle can be applied to address the issue of GC pauses. By monitoring the heap usage and the frequency of GC pauses, the system can determine whether to scale up or down the number of instances to maintain a steady and acceptable level of latency. This approach facilitates in minimizing the impact of GC pauses on the performance of the service by ensuring that the workload is distributed evenly among the instances and that each instance has adequate resources to handle the incoming requests without experiencing long GC pauses. By implementing an auto scaling mechanism that considers both the server traffic and the GC pauses, the system can achieve a more balanced and efficient allocation of resources and provide a better user experience.

2.8 Literature review conclusion

This chapter concludes that based on the previous research conducted on GC, it has a significant impact on the performance of applications and a variety of approaches have been suggested to mitigate the issue associated with GC. The solutions suggested include improvements to the existing GC algorithms, and radical approaches to address this issue.

Most of the solutions focus on the traditional application deployment while new trending technologies focus on methods such as containerized deployments. However, the use of containerized deployment to eliminate the GC pauses is not yet investigated. With the use of GC aware containerized deployment it there's a possibility to completely eliminate the effect of JVM freezes due to GC cycles.

CHAPTER 3

DESIGN AND METHODOLOGY

This chapter presents a GC-aware containerized deployment strategy aimed at reducing or eliminating garbage collection (GC) pauses in Java applications by proactively rotating containers before memory exhaustion. The proposed approach operates within a Kubernetes environment, leveraging standard capabilities such as the Kubernetes Metrics Server to monitor memory usage. Section 3.1 provides an overview of the architecture, while subsequent sections elaborate on key components and workflows.

3.1 Overview

Figure 3.1 illustrates the high-level workflow of the proposed GC-aware design. The system revolves around monitoring the memory footprint of each container. When container usage approaches a threshold, traffic is drained from that container, and a new container (with fresh heap space) is brought online. This approach effectively bypasses traditional stop-the-world GC cycles by retiring the container before a full collection or an out-of-memory exception would occur.

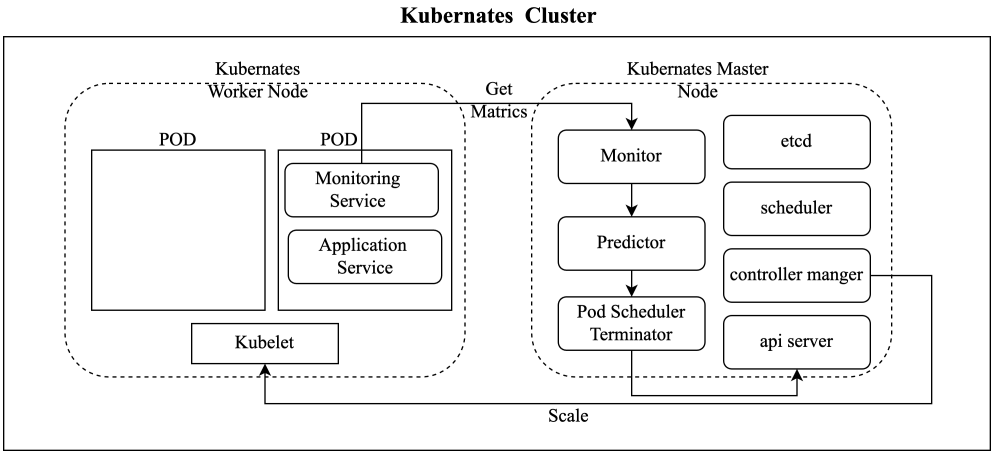

Fig. 3.1: High-Level Overview of GC-Aware Containerized Deployment.

In contrast to typical Java deployments, the container uses a no-op garbage collector (Epsilon GC). Epsilon GC never reclaims memory and will crash with an OutOfMemoryError [31] if usage is left unchecked. Therefore, rotating the container well in advance ensures zero GC-induced latencies. The design includes the following stages [32].

1. **Memory Monitoring.** A Kubernetes controller (sometimes referred to as a *MemoryWatcher*) queries the Metrics Server at regular intervals to obtain container memory usage statistics.

2. **Threshold Comparison.** If a container's memory usage surpasses a configured threshold (e.g., 70% or 700 MB out of 1 GiB), the controller deems it time to replace that container.

3. **Container Replacement.** The controller first ensures at least one additional container is present to handle new requests (by temporarily scaling up the Deployment by one replica [33]). Once the new container is confirmed to be ready, the old container is drained of traffic (made *unready*) and terminated.

4. **Repeat Cycle.** Over time, each container is replaced [34] proactively, thus preventing substantial GC cycles or out-of-memory conditions.

3.2 Core Components

3.2.1 JVM with Epsilon GC

The key principle of this approach is that the underlying Java Virtual Machine should use a *no-op* GC, specifically Epsilon GC (available from Java 11 onward). This collector performs no memory reclamation, ensuring there are no lengthy GC pauses. However, memory usage continuously grows, necessitating timely container rotation to avoid crashes. The container's lifecycle thus parallels the memory demands of the application, and once usage nears a specified threshold, the container is retired.

3.2.2 MemoryWatcher Controller

A custom or extended Kubernetes controller [35], here referred to as a *MemoryWatcher*, oversees the memory usage of Pods in a specified Deployment. The workflow proceeds as follows:

1. **Periodic Metrics Query.** The controller queries */apis/metrics.k8s.io/v1beta1/namespaces/{...}/pods* via the Kubernetes Metrics Server API to retrieve container usage metrics [36].

2. **Threshold Check.** After converting raw usage strings (like `"330Mi"` or `"512Mi"`) to a numeric MB figure, the controller compares usage to an operator-defined threshold.

3. **Replacement Decision.** If usage exceeds the threshold, a container is marked for replacement. In normal circumstances, the controller attempts to increase the Deployment's replica count by one, waits for the new Pod to become *Ready*, and then deletes the old Pod [37].

4. **Graceful Termination.** During removal, liveness probe is toggled to `Unhealthy`, ensuring no new traffic is routed to the old container. A brief grace period allows ongoing requests to complete before the container is forcibly terminated [38].

In practice, this controller can be implemented as a custom resource (e.g., a `MemoryWatcher` CRD) or simply as an off-cluster script that has proper Role-Based Access Control (RBAC) privileges to scale and delete pods.

3.2.3 Kubernetes Metrics Server

The Kubernetes Metrics Server provides on-demand CPU and memory usage statistics for Pods, facilitating near-real-time consumption metrics. Unlike sidecar-based or Prometheus-based approaches that track detailed request parameters, the Metrics Server simply aggregates resource usage [39]. This system is enough for the GC-aware logic since the essential input is *memory usage*. Detailed request or application-level metrics are not strictly required.

3.2.4 Application Pods

The user application runs inside container images [40] that have been built with Epsilon GC and a desired memory cap (e.g., `-Xmx2g`). Kubernetes resource limits can also be used (`limits.memory`) to prevent the container from exceeding, say, 8 GiB. However, the actual threshold for retiring the container might be set lower to avoid sudden `OOMKilled` events.

3.3 GC-Aware Lifecycle Workflow

Figure 3.2 gives a conceptual depiction of how traffic is served to the Java containers within a Deployment. Each container is assigned a readiness probe endpoint, for instance `/health` [41]. Under normal conditions, the endpoint returns an "OK" response, signaling that the container can receive traffic. Once memory usage grows beyond the threshold, the GC-aware controller toggles this readiness to *unhealthy* by invoking, for example, `/toggleHealth?status=false`. Consequently, traffic is shifted away, and the container is subsequently terminated.

3.3.1 Step-by-Step Logic

1. **Monitor Pod Metrics.** The controller polls the Metrics Server every X seconds. It parses memory usage strings and converts them into MB values.

2. **Compare Usage.** For each Pod in the target Deployment, usage is checked:

$$\text{usageMB}/\text{limitMB} \times 100\% \quad \text{vs. threshold}\%.$$

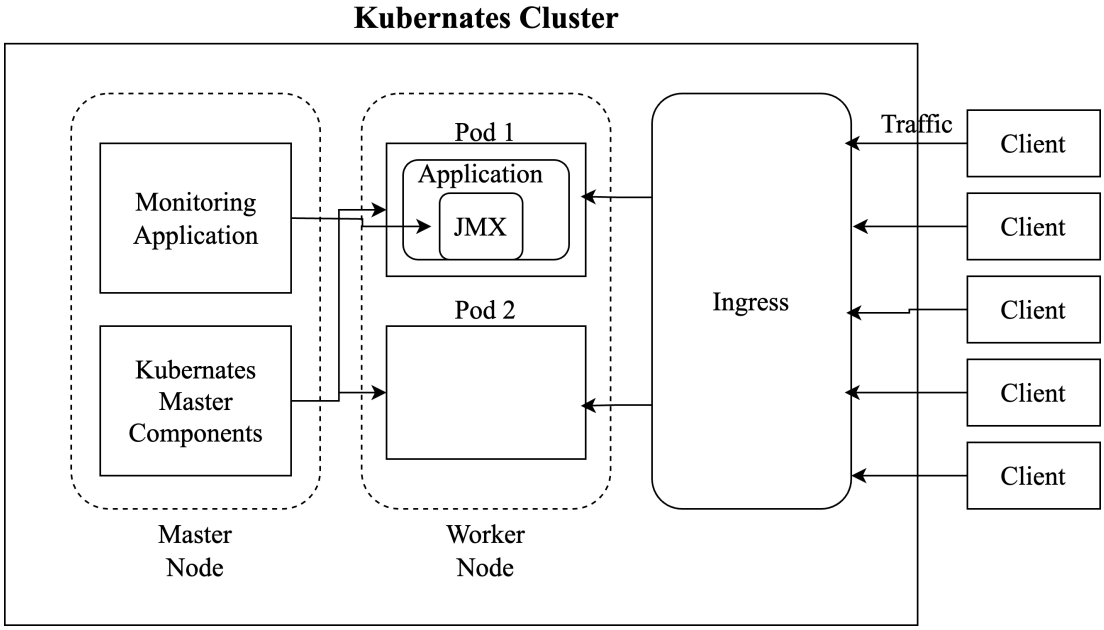

Fig. 3.2: Traffic Flow and Container Lifecycle in GC-Aware Approach

3. **Scale Up (If Over Threshold).** When at least one container is near the threshold, the controller scales the Deployment's replica count by one (e.g., from 2 to 3).

4. **Wait for New Pod Ready.** A brief polling loop checks for the new Pod's `Ready` condition. Once discovered, the system is safe to remove the old container.

5. **Remove Old Container.** The system sets the old container's readiness to `False`, waits for any grace period, and finally deletes the Pod. At this point, the old container is effectively replaced by a new container that starts with fresh memory usage.

6. **Optionally Scale Down.** If the desired stable number of replicas is constant, the system may scale down to its original count.

3.4 Container Graceful termination

Graceful shutdown of a container in Kubernetes with liveness and readiness probes can be achieved by modifying the behavior of the probes. When the container receives a shutdown signal, it should change its state so that the liveness probe fails, indicating that the container is no longer alive, and the readiness probe also fails, indicating that the container is no longer ready to serve requests. This will trigger Kubernetes to start the process of terminating the container (Figure 3.3).

- In the application code a shutdown hook [42] needs to be added, that listens for a shutdown signal. When the signal is received, change the state of the application to indicate that it is no longer ready to serve requests.
- Update the readiness probe to start failing once the shutdown signal is received. This can be done by modifying the command or URL used in the probe to return a non-zero exit code or a non-success HTTP status code.
- Update the liveness probe to start failing once the shutdown signal is received. This can be done in a similar manner to the readiness probe.
- Finally, trigger the shutdown of the container by issuing a SIGTERM signal to the container, which will cause the shutdown hook to run.
- Once the container has shut down, Kubernetes will detect that the liveness and readiness probes have failed, and will terminate the container and create a new one to replace it.

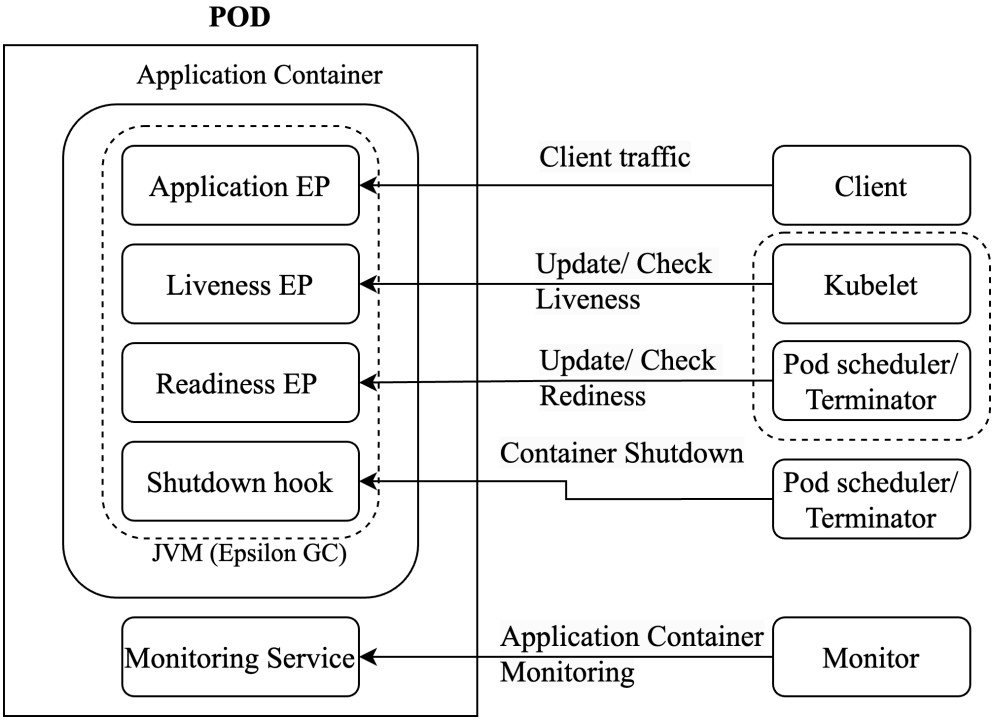

Fig. 3.3: Pod Terminating Flow

It is important to note that the process of shutting down a container in this manner may take some time, as the application must finish processing any ongoing requests and clean up any resources before it can shut down completely. Therefore the predator module needs to collect this information as well in order to determine the number of replicas required and the time at which the termination needs to be initiated.

3.5 Design Summary

The overall design is summarized as follows:

- **Epsilon GC in Java Pods:** Use a specialized garbage collector that never runs major GC, eliminating pause times. Container memory usage simply grows over time.
- **Kubernetes Monitoring:** Rely on the Metrics Server for up-to-date memory usage, polled at a configurable interval (e.g., every 5 s).
- **Controller Logic:** For each Pod surpassing a threshold (MB or percentage), the controller spawns a new Pod to maintain availability, then gracefully shuts down the high-usage Pod.
- **Graceful Removal and Re-scaling:** Once the new Pod is ready, the old Pod's readiness is revoked. After an additional grace period, the old Pod is deleted. The Deployment may optionally revert to the original replica count.

By rotating containers prior to memory exhaustion, the approach prevents both GC overhead and out-of-memory crashes, capitalizing on Kubernetes features (e.g., readiness probes, scaling) to maintain application availability. The next chapter (Implementation) will delve into how this design is realized in practice, describing the resource definitions, code snippets, role-based access control (RBAC), and tested workflows.

CHAPTER 4

IMPLIMENTATION

This chapter discusses the overall implementation of the GC-Aware Containerized Deployment where the collaboration of container orchestration and dynamic monitoring is utilized to eliminate GC-induced pauses in Java applications running within a Kubernetes ecosystem.

The chapter is systematically organized into multiple sections, each addressing critical components of the implementation process. The sections contain the runtime workflow, container orchestration logic, core and application-specific modules, API abstractions, and configuration settings respectively. A sample implementations of the GC-aware deployment process is presented to demonstrate practical applications of the framework.

4.1 Overview of GC-Aware Approach

The adopted container-based approach, where the GC overhead is managed at the container level rather than purely at the JVM creates a platform that can scale, replace, or retire application containers before GC becomes restrictive. This approach is supported by,

- **Epsilon GC** which only handles memory allocations but not memory reclaims inside application containers.
- **Monitoring and orchestration** via an external controller that decides events in the container lifecycle.
- **Stateless or near-stateless containers** that can be safely replaced without losing application data or user sessions.

This design ensures that once the memory usage of a container approaches the predefined threshold, the system spawns a fresh container with effectively no GC overhead and gracefully routes traffic away from the old container. After waiting for a termination grace period, the older container is then restarted before it reaches its memory capacity.

By replacing such a microservices instance before the memory limit, each container is effectively kept in a fresh state. This assists in forming a proactive container rotation driven by memory thresholds, rather than purely CPU or request-based auto scaling.

4.2 System Architecture and Primary Components

Figure 4.1 presents the overall architecture of the GC-Aware container orchestration solution. At the center is the GC-Aware Operator, which orchestrates the lifecycle of Java microservices running on Epsilon GC. The operator interacts with the Kubernetes API Server [43], the Metrics Server which reports memory usage, and the underlying container runtime, Docker, to dynamically manage container restarts whenever memory usage approaches the configured threshold.

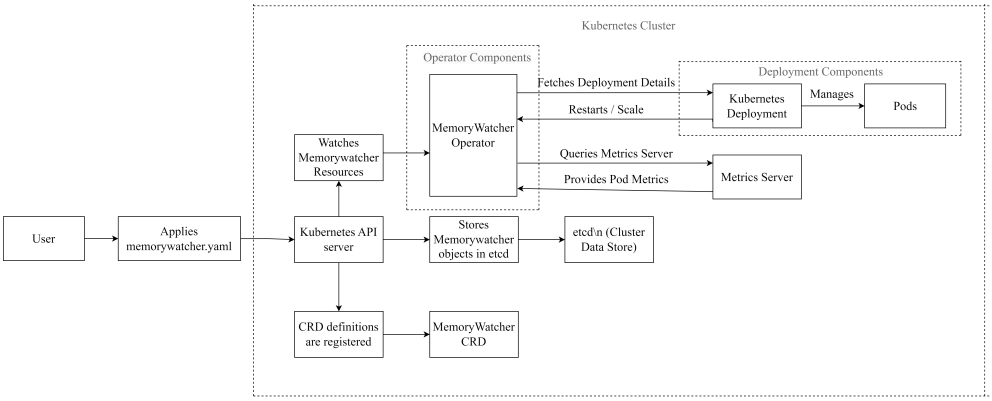

Fig. 4.1: High Level Architecture of the GC Aware container Deployment

Following are the key components of the deployment, which ensures efficient orchestration and seamless operation within the Kubernetes ecosystem.

4.2.1 MemoryWatcher (Kubernetes Custom Resource Definition (CRD))

MemoryWatcher is a custom defined Kubernetes resource [44] , [45] , [46] which provides the operator with instructions regarding the deployment to be monitored and the memory usage threshold that determines when a container should be replaced. The MemoryWatcher resource is created after the application is deployed, storing parameters such as memoryThreshold, deploymentName, and a timestamp indicating the resource's latest reconciliation event. When the MemoryWatcher resource is applied, the GC-Aware Operator is notified to begin monitoring the specified deployment.

4.2.2 GC-Aware Operator (Controller)

The GC-Aware Operator serves as the control logic for interpreting MemoryWatcher resources [47]. It listens to the creation and updates MemoryWatcher objects. When an event is received, the controller identifies the associated application deployment and its specified memory thresholds. This architecture allows flexible deployment of application containers without hard-coding deployment configurations and supports deploying applications that may not require the intervention of the GC-Aware Operator.

Written in Go using Kubebuilder with Kubernetes API support, the operator continuously monitors changes to MemoryWatcher resources and the relevant application deployments. The role of the GC-Aware Operator is as follows:

- **Role:**
  - **Monitor Memory Usage:** Collect memory usage data from the Metrics Server for the monitored containers.
  - **Compare Against Thresholds:** Evaluate the current container memory usage against the thresholds specified in the MemoryWatcher resources.
  - **Enforce Decisions:** If memory usage exceeds the defined threshold, the operator initiates a graceful container replacement procedure to prevent memory exhaustion and eliminate the need for major garbage collection cycles.

4.2.3 Java Microservice Pods

Each Pod includes a JVM launched with Epsilon GC flags (-XX:+UseEpsilonGC), effectively disabling typical GC cycles. Since the sample microservices are stateless, container restarts occur without loosing important session or cached data. As Epsilon GC accumulates memory usage without reclamation, it is vital that the operator intervenes to recycle the container prior to an out-of-memory event.

4.2.4 Kubernetes Metrics Server

Metrics server provides real-time metrics on resource usage, including memory utilization for each container. The GC-Aware Operator periodically queries these metrics at fixed intervals, e.g. every five seconds, to decide if any containers exceed the permissible threshold.

4.2.5 Kubernetes API Server etcd

The API Server acts as a central interface for cluster objects including deployments, pods, and MemoryWatcher resources, while etcd [48] is the backing store. All custom resource definitions, as well as user-supplied objects, reside in API server.

4.3 Implementation of GC-Aware Deployment

The GC-Aware solution is designed to operate seamlessly in two phases:

1. **Deployment of the Operator**: CRDs are installed, and the GC-Aware Operator is deployed.

2. **Runtime Orchestration**: The operator monitors memory usage, orchestrating the creation of fresh containers and the graceful termination of old ones before major a GC occurrence.

During the deployment phase, Custom Resource Definitions (CRDs) are installed, and the GC-Aware Operator is deployed within the cluster, enabling the management of the lifecycle of monitored containers. In the runtime orchestration phase, the operator actively monitors memory usage in Pods, orchestrating the creation of new containers and the graceful termination of existing ones before the occurrance of major garbage collection (GC) cycles. This proactive approach ensures smooth application performance by mitigating GC-induced latencies and maintaining system stability.

4.3.1 Kubernetes and Metrics Server Setup

To enable the GC-Aware Operator to function effectively, a Kubernetes cluster along with a Metrics Server is configured and deployed. The Metrics Server provides critical real-time data, such as container memory usage, essential for decision-making in memory management and container replacement. This setup is structured as follows.

1. **Kubernetes Cluster Deployment**: The Kubernetes environment is initialized using Rancher Desktop [49] [50] for a local deployment of the kubernetes cluster.

2. **Metrics Server Deployment**: The Metrics Server is installed and configured within the cluster [51]. The tool aggregates resource usage metrics, enabling fetch the application metrics by the application. The metrics provide the foundation for monitoring container memory e.g. container_memory_usage_bytes.

3. **Validation**: After deployment, the setup is validated by ensuring that the Metrics Server properly reported resource utilization for all active pods within the cluster. This step confirms that memory metrics can be effectively queried by the GC-Aware Operator for proactive container lifecycle management.

This robust setup ensures the environment is fully equipped to support the subsequent implementation of memory-aware container orchestration mechanisms.

4.3.2 MemoryWatcher (Kubernetes Custom Resource Definition / CRD)

Custom Resource Definitions (CRDs) are extensions of the Kubernetes API that allow users to define and manage custom objects. They are essential for enabling the GC-Aware Operator to perform specialized tasks that extend beyond the default capabilities of Kubernetes.
The MemoryWatcher CRD is designed to monitor and manage the memory usage of

application containers. By defining the MemoryWatcher CRD, Kubernetes gains the ability to handle application-specific memory thresholds and initiate container replacements when required.
The Figure 4.2 shows the interaction of the GC-Aware Operator with the memory watcher resource to manage the user application.

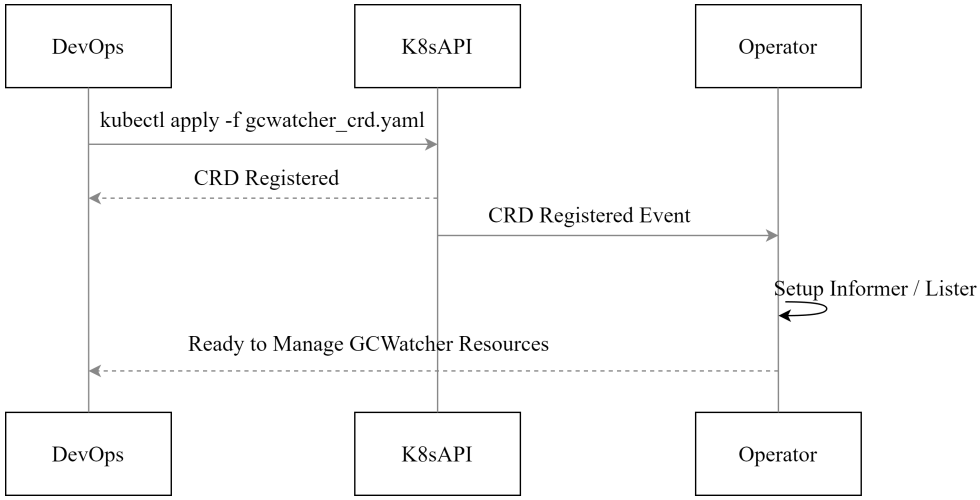

Fig. 4.2: Initialization Workflow for GCWatcher Resource Management

The Figure 4.3 shows the memory watcher resource configuration which is used to deploy in the cluster.

```yaml
apiVersion: monitoring.containergc.com/v1
kind: MemoryWatcher
metadata:
  name: myapp-memwatcher
  namespace: default
spec:
  deploymentName: myapp-deployment
  deploymentNamespace: default
  memoryThreshold: 70
  lastUpdatedTime
```

Fig. 4.3: MemoryWatcher resource definition

Key attributes in the MemoryWatcher CRD include the following. These fields provide the memory operator with critical information about the application container and its associated thresholds. The MemoryWatcher configuration enables dynamic adjustments to threshold values and deployment settings as required. Once the MemoryWatcher resource is created, it triggers an event with this information which is captured by the operator.

- **memoryThreshold**: Specifies the memory usage percentage beyond which a container should be replaced.
- **lastUpdatedTime**: Tracks the last update of the resource, enabling accurate monitoring.
- **deploymentName**: Identifies the associated deployment or Pods being monitored (i.e the java application pods).
- **deploymentNamespace**: Namespace where the application is deployed.

4.3.3 GC-Aware Operator (Controller)

The GC-Aware Operator handles the primary logic for monitoring pod memory usage, assessing thresholds, and orchestrating the container lifecycle. By referencing the event from the MemoryWatcher CRD, the operator identifies which Deployments to be watched and the specific memory thresholds to be enforced. When a container's memory consumption surpasses a configured limit, the operator initiates a graceful replacement procedure. The process first launches a new container and ensures it is ready to serve requests. Once the new container is active, the operator marks the old container's liveness probe as unhealthy, causing Kubernetes to stop sending traffic to it. After a grace period to allow any in-flight requests to complete, the old container is finally restarted or removed.

The Figure 4.4 illustrates the interaction between the operator and the the Metrics Server, the existing container, and Kubernetes to provision a replacement container.

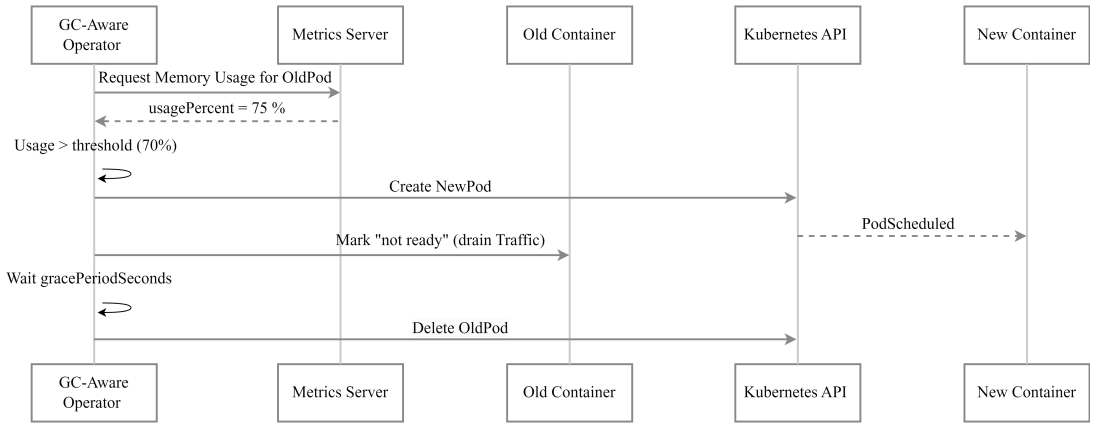

Fig. 4.4: Lifecycle Management Workflow of GC-Aware Operator

4.3.3.1 Example Controller Logic

The Figure 4.5 is a concise Go snippet demonstrating the controller's reconcile method which implements the memory-check logic. It highlights core steps such as retrieving

the custom resource, listing Pods, querying memory usage, and initiating replacement actions [52].

This operator was created using Kubebuilder [53],[54], which supplies a skeleton for defining Kubernetes-native controllers in Go. Kubebuilder simplifies multiple tasks, including:

- **API and CRD Definition**: Generates code to register new resource types e.g., `MemoryWatcher`.
- **Controller Skeleton**: Generates boilerplate code for watchers, reconcilers, and RBAC setups [55], [56].
- **Integration with Kubernetes Patterns**: Uses the standard *client-go* libraries [57] to track resources and handle events.

By utilizing these, the GC-Aware Operator adheres to Kubernetes best practices. It provides comprehensive lifecycle management for container memory usage while minimizing disruptions to typical application operations.

```go
func (r *GCWatcherReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    logger := log.FromContext(ctx)

    // 1. Fetch GCWatcher CR
    var gcWatcher v1alpha1.GCWatcher
    if err := r.Client.Get(ctx, req.NamespacedName, &gcWatcher); err != nil {
        // Handle not-found errors gracefully
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // 2. List all pods that match certain labels/namespaces
    pods := &corev1.PodList{}
    // Example: label "app=my-gc-aware-service"
    if err := r.Client.List(ctx, pods, client.InNamespace(req.Namespace)); err != nil {
        return ctrl.Result{}, err
    }

    // 3. Query memory usage from Metrics or a custom aggregator
    podMemoryUsage := r.QueryMemoryUsage(pods.Items)

    // 4. Decision logic: compare usage to threshold
    for _, p := range pods.Items {
        usagePct := podMemoryUsage[p.Name]
        if usagePct > gcWatcher.Spec.MemoryThreshold {
            // Spawn new container, mark old container as draining, etc.
            r.spawnReplacementPod(ctx, p, &gcWatcher)
        }
    }
}
```

Fig. 4.5: Reconsiliation Logic

- **Fetch GCWatcher**: Retrieves the custom resource that specifies memory thresholds and references the target Pods or Deployments.
- **List Pods**: Locates Pods in the specified namespace that match relevant labels.

- **Query Usage**: Uses the Kubernetes Metrics Server to collect memory usage for each Pod.
- **Decision Logic**: Whenever usage surpasses the threshold, `spawnReplacementPod` is called to handle graceful container rotation.

4.3.3.2 Memory Monitoring

The operator routinely contacts the Metrics Server for every five seconds to retrieve up-to-date container resource metrics, notably memory usage. Whenever additional Pods are added to the target Deployment, the operator automatically begins tracking their memory consumption.

4.3.3.3 Decision Logic

Once usage metrics are obtained, the operator compares each container's consumption level with the threshold recorded in its corresponding `MemoryWatcher` resource. If a container's memory usage surpasses the threshold, for instance, 70%, it is marked for restart. Before traffic is drained from this memory-heavy container, the operator ensures at least one other Pod is running, thus preserving the availability of the application.

4.3.3.4 Lifecycle Management and Graceful Draining

The operator calls an application endpoint that toggles the container's liveness probe to "unhealthy". This action prompts Kubernetes to cease routing traffic to the flagged container. A `terminationGracePeriodSeconds` delay allows any in-flight requests to be completed. After the delay, the old container is restarted. Consequently, the system avoids GC-related pauses by rotating containers well before they reach out-of-memory conditions.

The summary of the operator logic is illustrated in Figure 4.6.

1. **Start Operator**: The operator begins its reconciliation loop, reacting to watch events or periodic timers.
2. **Fetch Pod Metrics**: The Metrics Server is queried for memory usage details.
3. **Calculate Usage %**: The container's memory utilization is computed as $\frac{\texttt{used}}{\texttt{allocated}} \times 100$.
4. **Check Threshold**: The operator evaluates whether the usage surpasses the `memoryThreshol` in the `MemoryWatcher` CRD.

5. **Create a New Pod (If Over Threshold)**: If the threshold is exceeded, the operator spawns a new pod to ensure enough capacity is available for incoming requests.

6. **Verify Readiness**: The operator waits for the newly created pod to transition to a *Ready* state so it can accept traffic.

7. **Stop Traffic to the Old Container**: The old container is marked "unready," causing Kubernetes to stop sending new requests to it.

8. **Graceful Drain**: A grace period (`gracePeriodSeconds`) elapses, permitting any in-flight requests on the old container to complete.

9. **Remove or Restart Old Pod**: After the drain period, the old container is either restarted or removed from service. The cycle then resumes from step 2, maintaining continuous monitoring.

Through these phases, the GC-Aware Operator maintains continuous oversight of each container's memory consumption, proactively replacing them before an out-of-memory scenario. This arrangement significantly mitigates GC-induced latencies by employing a "no-gc" Epsilon approach and relying on Kubernetes' orchestration capabilities.

4.3.3.5 Installing the Operator

After the operator code base is implemented, it is compiled and packaged as a Docker image to run within the cluster. The basic steps for installing the operator includes the following.

- **Build the Operator Image**: Using the dedicated build tool (Makefile Docker build [58]) compiles the operator into a Docker image.

- **Push Image to Registry**: Once built, push the Docker image to a registry accessible by the Kubernetes cluster. In this implementation, the image was pushed to a local Docker registry that the Kubernetes cluster has access to download the image.

- **Deploy Operator Resources**:

  - Apply the generated Custom Resource Definition (CRD) YAML for *MemoryWatcher*.

  - Create the Deployment YAML for the operator, which points to the newly built Docker image.

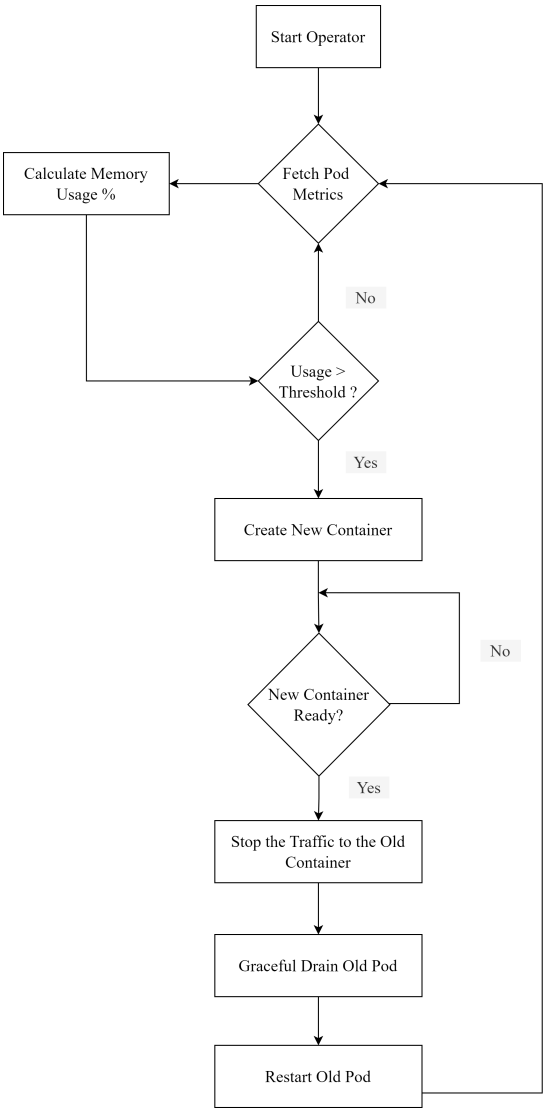

Fig. 4.6: Decision Flow Diagram

- **Set Up RBAC Permissions**:
  - Ensure Role-Based Access Control (RBAC) [56] is configured so the operator can get, list, watch, and update Pods.
  - Provide full access to manage custom resources defined by the `MemoryWatcher` CRD.

These steps ensure that Kubernetes recognizes the new resource type (MemoryWatcher) and that the operator has permission to monitor Pods, gather memory metrics, and perform lifecycle operations.

4.3.4 Container Image with no op GC

The application needs to be developed with Java 11+ since Epsilon GC is available from java 11. The Application requires liveness and readiness probes so that the application can be properly deployed in the kubernetes cluster. The application health check logic requires modification in order to alter the health check state with an api call. Sample reference to this implementation can be observed in the the Figure 4.7. Toggling the health check is performed by the GC operator when retiring the application container. This process ensures proper traffic management and facilitates a smooth transition.

```java
@GetMapping("/health")
public String healthCheck() {
    if (isHealthy.get()) {
        return "Application is healthy";
    } else {
        // HTTP 503 if you want actual liveness probe failure.
        return "Application is NOT healthy";
    }
}

/**
 * Endpoint to toggle health state dynamically:
 * e.g. /toggleHealth?status=false
 */
@GetMapping("/toggleHealth")
public String toggleHealth(@RequestParam boolean status) {
    isHealthy.set(status);
    return "Health set to: " + (status ? "healthy" : "unhealthy");
}
```

Fig. 4.7: Application Health check

An integral part of the solution is the container image. A custom Dockerfile was used (Figure 4.8) to run Java applications with the Epsilon GC. This image can then be deployed in the kubernetes cluster. Any existing java application can be deployed in the cluster with minimum modifications.

```dockerfile
# Use the official OpenJDK image from the Docker Hub as the base image
FROM openjdk:17-jdk-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the Spring Boot JAR file from your local machine to the container
COPY target/springapi-0.0.1.jar springapi.jar

# Expose the port the app will run on (default Spring Boot port is 8080)
EXPOSE 8080

# Run the Spring Boot JAR file
ENTRYPOINT ["java","-XX:+UnlockExperimentalVMOptions", "-XX:+UseEpsilonGC", "-Xms128m", "-Xmx8g", "-jar", "springapi.jar"]
```

Fig. 4.8: Application Docker file

4.4 Practical Challenges and Limitations

Stateful services present one of the primary challenges for a container replacement strategy. When a microservice depends on in-memory caches or session data, simply replacing the container becomes problematic. Potential remedies involve external session stores or using consistent hashing for caches, which prevent data loss during container restarts.

Another concern arises from the possibility of *OutOfMemory Errors*. If either the operator's threshold or the requeue intervals are poorly tuned, pods running the Epsilon GC may exhaust their memory allocation before a successor container comes online. This situation underlines the importance of robust intervals and a safety margin for (e.g., a threshold of around 70% memory usage), ensuring that a new container can be spun up while memory remains available.

A further limitation relates to *cost and overprovisioning*, since a container replacement model can result in short-term overscaling during traffic spikes. Keeping overhead under control often requires restricting the operator to introduce only one additional container per retired container, thus balancing resource demands against performance.

The *multiple GC algorithms* dimension adds complexity as well. Although Epsilon GC offers the simplest demonstration of a no-GC environment, Shenandoah or ZGC reduce pause times more effectively while providing certain degrees of memory reclamation. These advanced GCs, however, demand careful configuration and tuning to achieve the desired latency improvements.

Lastly, there is a concern of *integration with existing autoscalers*. The Kubernetes Horizontal Pod Autoscaler (HPA) [59], [60] may inadvertently conflict or overlap with the GC-aware operator logic if they both attempt to influence the same scale target. Planning a cooperative strategy—such as customizing the operator to communicate with or complement the HPA—is therefore essential for avoiding contradictory or redundant scaling decisions.

CHAPTER 5

EXPERIMENTAL EVALUATION AND RESULTS

This chapter presents an extensive evaluation of the GC-Aware Containerized Deployment framework under two distinct configurations.

1. *No-GC* mode, where **Epsilon GC** (`-XX:+UseEpsilonGC`) is employed and container memory usage is monitored via an operator that restarts Pods once usage crosses a defined threshold (about 70 % of the 8 GB limit).

2. *Standard GC* mode, wherein a mainstream collector (G1 GC) manages memory reclamation with no container restarts.

Five Java-based microservices (Bulk Data Aggregator, Prime Computation, In-Memory Cache, Spring pet clinic, and tomcat server) were tested for 30 minutes each under both modes, resulting in six tests total. Each test was repeated to ensure reproducibility, yielding twenty total runs. This chapter begins with an overview of the experiment design (5.1) and then reports on memory usage, CPU usage, container lifetime, and request-response metrics across both GC configurations. Special attention is given to tail-latency spikes (either from container restarts or full GC cycles) and overall throughput.

5.1 Experiment Setup and Methodology

5.1.1 Cluster and Hardware

All experiments were conducted on a single-node Kubernetes cluster with the following configurations:

5.1.2 Cluster and Hardware

All experiments were conducted on a single-node Kubernetes cluster with the following configurations:

- **Operating System:**
  - **Distribution**: Fresh installation of Ubuntu 20.04.6 LTS
  - **Mode**: Server mode
  - **Host Specifications**:
    - **CPU**: AMD A8-7600 Radeon R7, 10 Compute Cores 4C+6G

    - **Cores**: 2 cores per socket, 1 socket (4 logical CPUs with hyperthreading)
    - **Base Frequency**: 2.4 GHz (up to 3.1 GHz with boost)
    - **Memory**: 16 GB RAM

- **Kubernetes Cluster:**
  - **Provisioning Tool**: Rancher Desktop
  - **CPU Allocation**: 3 virtual CPUs (vCPUs)
  - **Memory Allocation**: 12 GB
  - **Kubernetes Version**: 1.29.3
- **Container Runtime:**
  - **Docker Version**: v27.2.1
- **Resource Monitoring:**
  - **Tool**: `metrics-server`
  - **Functionality**: Provides container-level CPU and memory usage metrics

5.1.3 Test Scenarios and Metrics Collected

Table 5.1 summarizes each tested application, total number of requests, typical request size, test duration, and memory usage pattern. JMeter [61] was employed to issue requests over a 30-minute interval for each test. The microservices were deployed in either,

- *Epsilon GC* mode, where **Epsilon GC** (mode with an operator threshold at roughly 5.6 GB usage.
- *Standard GC* mode, no operator restarts.

TABLE 5.1: Key parameters for the five microservices under test. Each scenario runs for 30 minutes, repeated twice per GC mode, yielding twelve runs.

Service	**Payload Size**	**Focus**
Bulk Data	250 KB JSON	Memory Intensive
Prime	1–10 KB	CPU Intensive
Cache	2 KB (key-val)	Moderate Memory Intensive
Spring Pet Store	200 KB JSON	Memory Intensive
Tomcat	1 KB	Moderate Memory Intensive

Data collection proceeded as follows. Memory and CPU usage were polled every 5 s from the `metrics-server`. Response time was measured via JMeter. Container restart events were recorded by the operator log (Epsilon runs) or recognized from *zero* in standard GC mode, unless an Out of Memory (OOM) surfaced. In standard GC runs, `-Xlog:gc` logs were captured to examine major GC cycles and pause durations [62].

5.2 Bulk Data Aggregator Results

The aggregator microservice accepts ∼250 KB JavaScript Object Notation (JSON) payloads on each POST, leading to large accumulations of in-memory strings. Over 30 minutes, around 115,000 requests are processed.

5.2.1 Memory Usage

Figure 5.1 depicts memory usage in Epsilon mode. The container's memory grows from near 280 MB up to roughly 5.6 GB in about 2–3 minutes, at which point the operator detects that usage has crossed the threshold (70% of 8 GB) and replaces the container. After a short grace period, the old container is terminated. This "sawtooth" pattern recurs 11–12 times over the 30-minute test.

A comparable standard GC run memory usage climbing toward 3–4 GB, followed by major GC cycles that reclaim large amounts of memory. GC logs reveal multiple major cycles with 200–300 ms pause times.

5.2.2 Latency and Throughput

Table 5.2 summarizes the results. The aggregator's median latencies generally exceed 70–75 ms. Under standard GC, there are fewer large tail spikes (200–300 ms) unless a major GC event coincides with high concurrency. Meanwhile, the Epsilon GC configuration achieves slightly higher throughput (64–67 RPS) compared to standard GC (62–63 RPS).

However, the response time graphs (Figures 5.2 and 5.3) show occasional requests taking as long as 500 ms–2.5 s under standard GC, representing a 500%–2500% variation from the average response time. In contrast, under Epsilon GC, the highest observed latencies range from 200 ms–350 ms, corresponding to a 150%–350% variation—significantly lower than those in the standard GC scenario.

5.2.3 Container Lifetime and GC Observations

In no-GC mode, each container typically survives 2–3 minutes before usage nears 5.6 GB, leading to 11–12 restarts in 30 minutes. By contrast, standard GC logs show

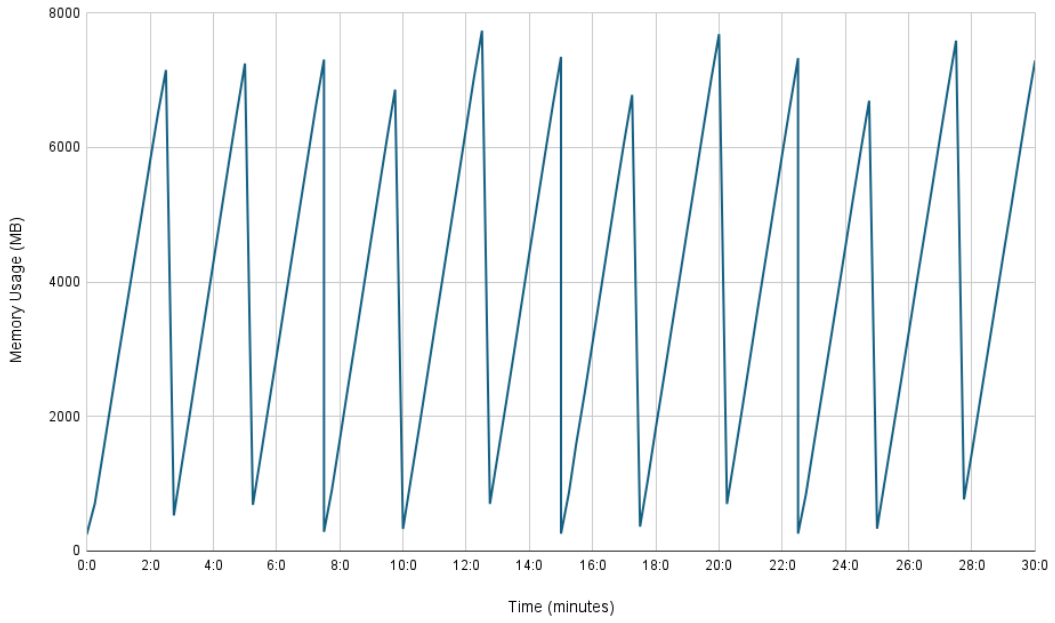

Fig. 5.1: Memory usage for Aggregator under Epsilon GC.

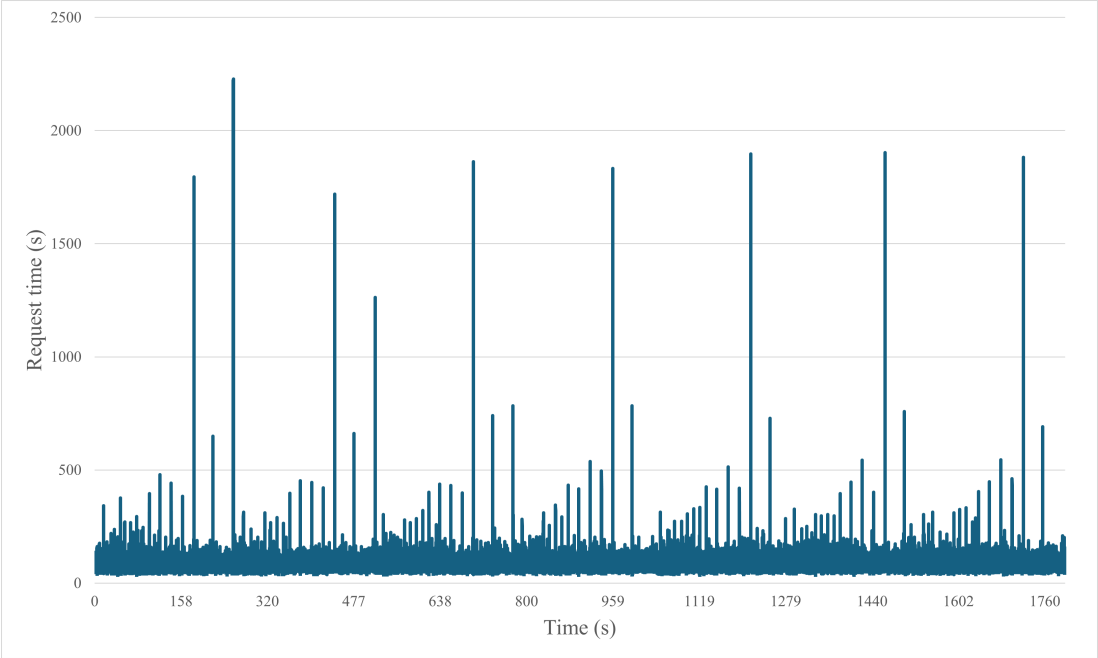

Fig. 5.2: Response time distribution under standard GC

TABLE 5.2: Aggregator latencies (mean values of two test runs).

GC Mode	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (RPS)
Epsilon	69–72	99–102	126–131	64–67
Standard	75–76	106–107	149–151	62–63

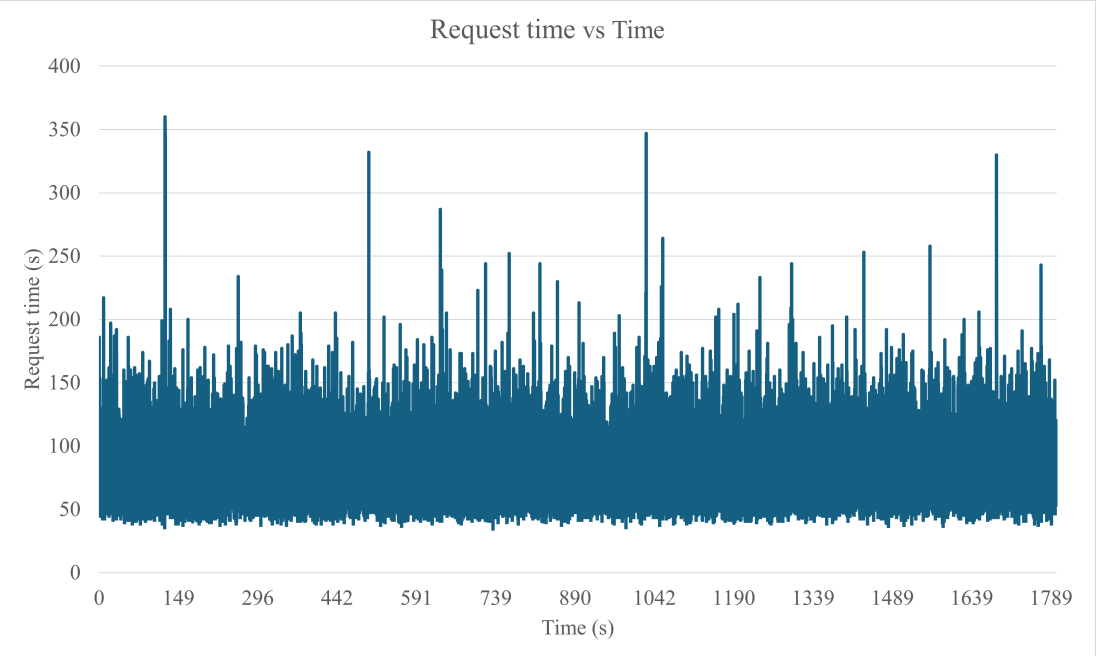

Fig. 5.3: Response time distribution under Epsilon GC

3–5 major collections, each reclaiming 1–2 GB. This arrangement avoids container churn but introduces 150–200 ms stop-the-world events, occasionally up to 500 ms if a *full* GC is triggered at large heap occupancy.

5.3 Prime Computation Results

5.3.1 Memory Usage

Prime computations are CPU-bound, with memory usage driven mostly by a caching mechanism. In Epsilon runs, memory grows from near 270–280 MB to 3.7–3.8 GB over 30 minutes. This application never exceed the threshold over the test period, so no restarts occur. Figure 5.4 shows a scenario with zero restarts. Alternatively, if concurrency is high, usage can cross 70% around minute 20, causing exactly one restart. Standard GC runs show repeated cycles that keep usage around 300–500 MB, with no container restarts.

5.3.2 Latency and Throughput

As shown in Table 5.3, the Epsilon GC configuration achieves slightly lower median latencies (44–45 ms) compared to standard GC (46–47 ms). This pattern persists at higher percentiles, with Epsilon's p99 latencies around 57–58 ms versus 60–61 ms under standard GC. Throughput shows a modest increase as well, rising from 21 RPS (standard GC) to 22 RPS (Epsilon GC).

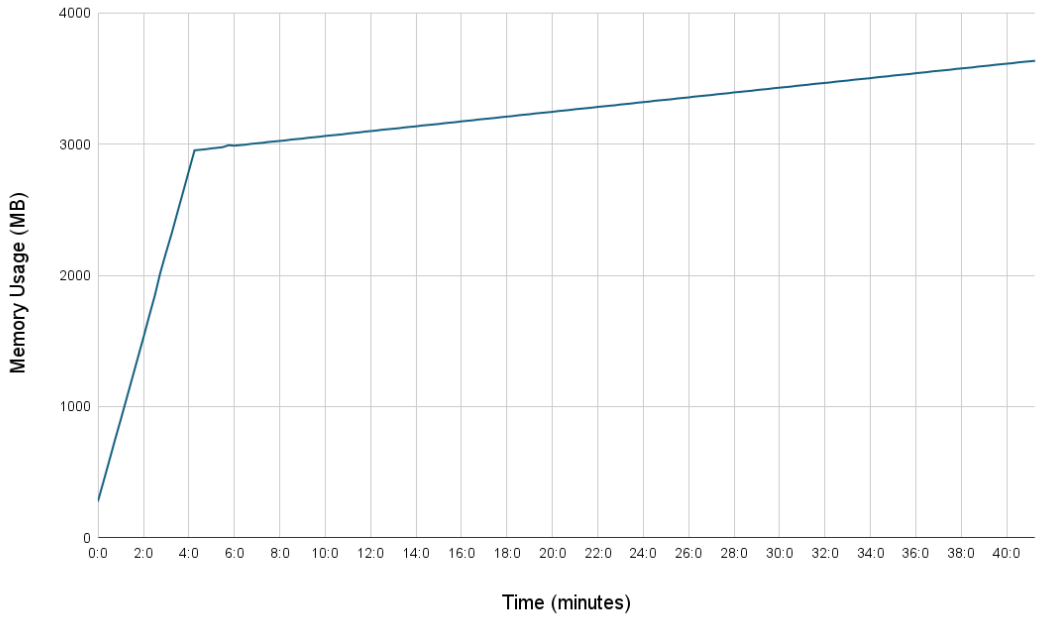

Fig. 5.4: Prime memory usage under Epsilon GC.

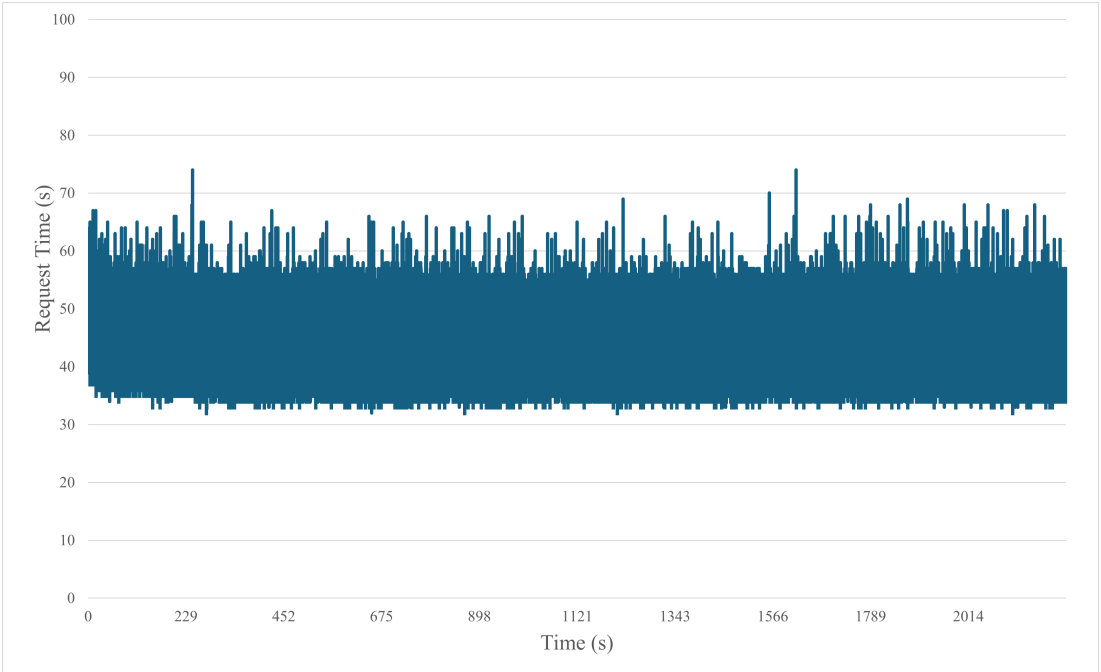

Fig. 5.5: Response time distribution under Epsilon GC for prime-finding.

TABLE 5.3: Prime App latencies (average of repeated runs).

GC Mode	p50 (ms)	p95 (ms)	p99 (ms)	Throughput
Epsilon	44–45	54–55	57–58	22
Standard	46–47	58–59	60–61	21

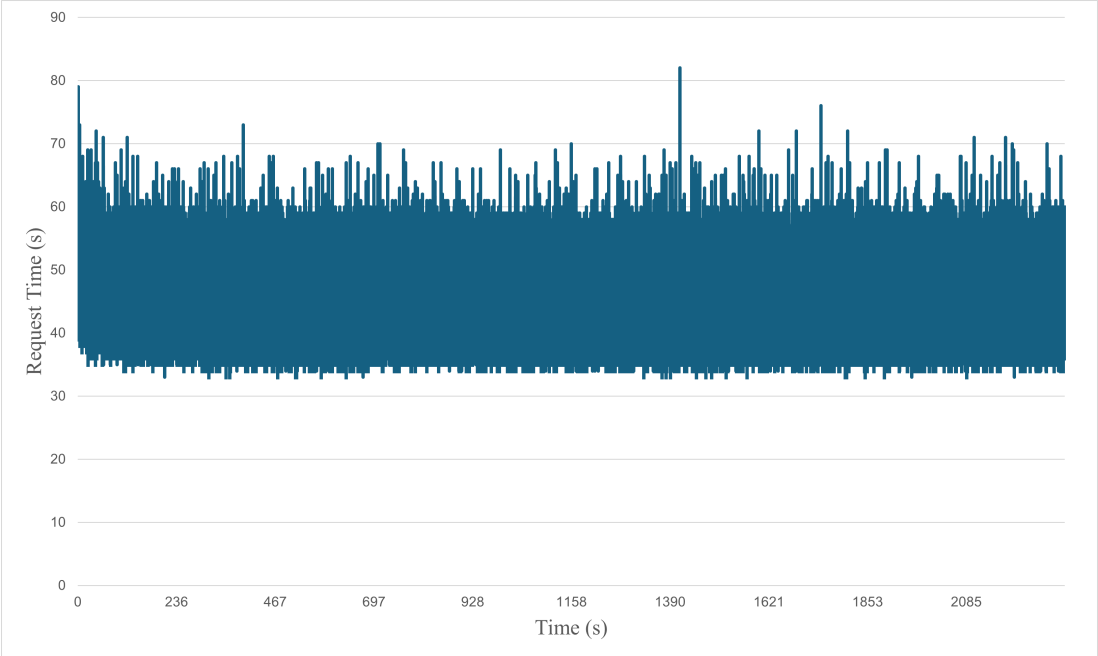

Fig. 5.6: Response time distribution under Standard GC for prime-finding.

In Epsilon mode, container restarts range from zero to one over 30 minutes. Standard GC sees no restarts, typically performing multiple smaller GC cycles. This setup shows that CPU-bound services with moderate memory usage can benefit from minimal GC overhead, provided they do not exceed the memory threshold frequently.

5.4 In-Memory Cache Results

5.4.1 Memory Usage

The cache microservice stores up to 20,000 entries (each about 2 KB), occasionally pushing memory usage above 5.6 GB. As shown in Figure 5.7, an Epsilon GC run typically triggers exactly one container restart when usage exceeds the 5.6 GB threshold (around minute 18). By contrast, standard GC reclaims memory more frequently and thus avoids container restarts under similar loads.

5.4.2 Latency and Throughput

Table 5.4 presents the cache service latencies and throughput under Epsilon and standard GC. Median latencies fall between 73–77 ms. For Epsilon runs, the 99^{th} percentile can reach or exceed 200 ms during heavy contention. Throughput is marginally higher under Epsilon GC, around 133 RPS, compared to 122–123 RPS with standard GC. Container churn remains minimal for this workload, typically limited to one or two restarts when usage surges.

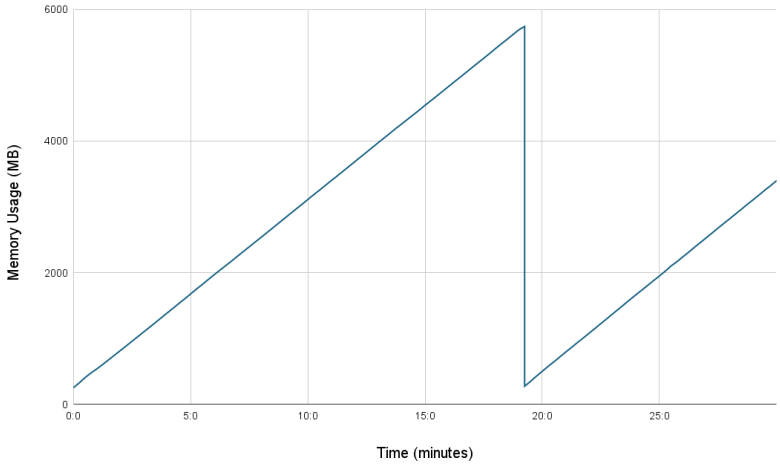

Fig. 5.7: Cache memory usage with Epsilon GC. One container restart occurs upon exceeding ∼5.6 GB.

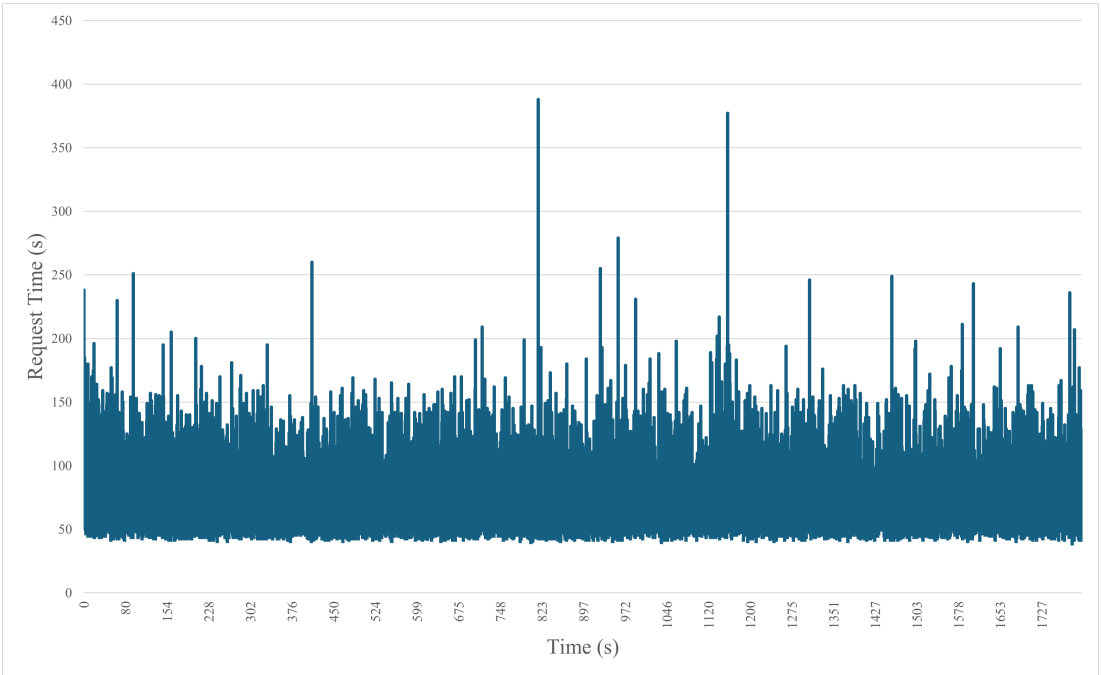

Fig. 5.8: Response time distribution under Epsilon GC for the cache service.

TABLE 5.4: Cache service latencies (averages of repeated runs).

GC Mode	p50 (ms)	p95 (ms)	p99 (ms)	Throughput (RPS)
Epsilon	73–74	74	96	133
Standard	76–77	76	97–98	122–123

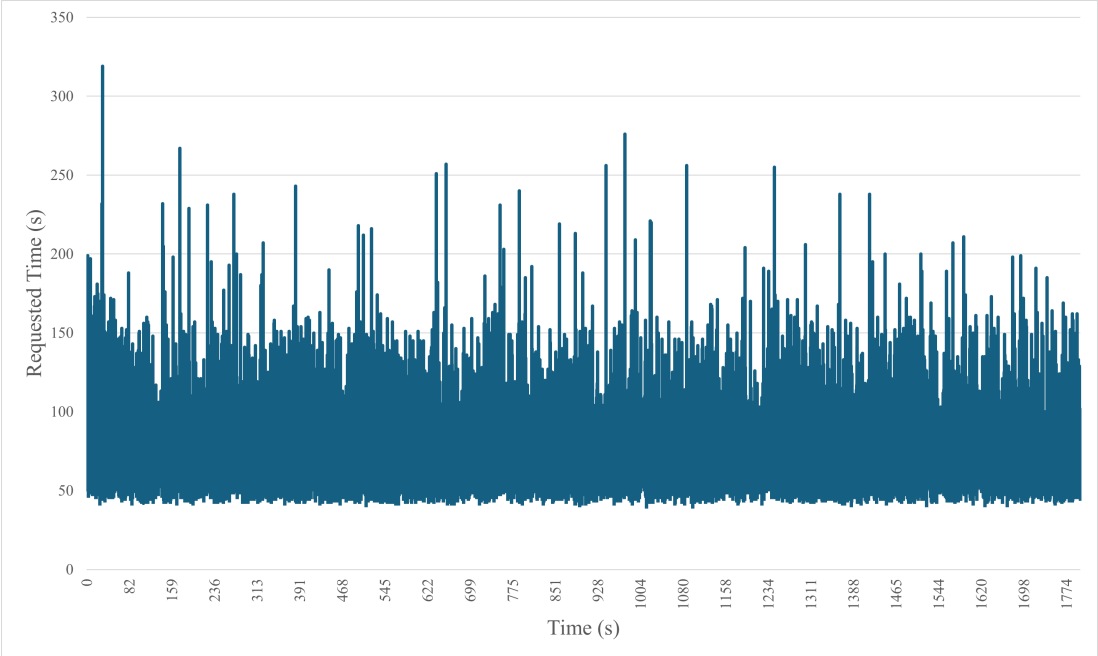

Fig. 5.9: Response time distribution under standard GC for the cache service.

5.5 Spring Pet Clinic Results

5.5.1 Memory Usage

Spring Pet Clinic is a well-known reference application commonly used to benchmark Spring Boot performance and simulate realistic microservice deployments. In this study, it also serves as a representative workload for benchmarking GC behavior under high object churn and memory pressure. Each request carries a 200 KB JSON payload, leading to substantial memory allocation rates.

Under Epsilon GC, memory usage starts at approximately 900 MB and grows rapidly to around 6.5 GB within 3–4 minutes due to the application's allocation-heavy behavior. This triggers 8 container restarts during the 30-minute test period, as memory thresholds are exceeded and containers are restarted gracefully. While the lack of garbage collection leads to predictable memory growth, it also avoids GC-induced pauses, making Epsilon ideal for controlled environments with restart-based memory management.

By contrast, G1 GC constrains heap usage to around 700 MB through periodic garbage collection cycles, preventing restarts but at the cost of occasional latency spikes due to GC activity. Figure 5.10 illustrates the predictable memory ramp-up and restarts seen under Epsilon GC.

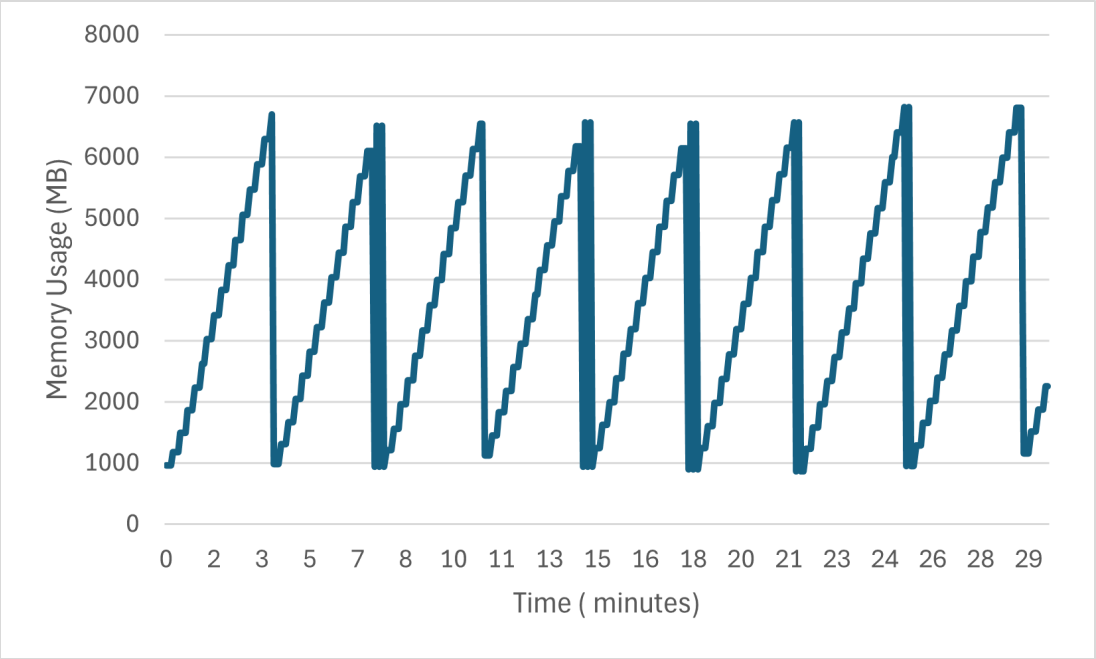

Fig. 5.10: Spring Pet Clinic memory usage under Epsilon GC.

5.5.2 Latency and Throughput

As shown in Table 5.5, Epsilon GC provides consistently lower latency across all measured percentiles. Median response times improve by 3–5 ms compared to G1 GC, and p99 latencies are reduced by up to 6 ms. This is a result of eliminating GC pauses, which can otherwise introduce tail-latency variability in memory-managed applications.

Throughput also improves with Epsilon GC, increasing from 19 RPS (G1) to 21 RPS. Although this comes at the cost of higher memory pressure, the gain in responsiveness and throughput makes Epsilon particularly attractive for performance-critical or latency-sensitive deployments where memory is constrained by design and container restarts are an acceptable tradeoff.

TABLE 5.5: Spring Pet Clinic latencies (average of repeated runs).

GC Mode	p50 (ms)	p95 (ms)	p99 (ms)	Throughput
Epsilon	24.5	26–27	30–32	42
G1	26–26.5	30–31	41	39

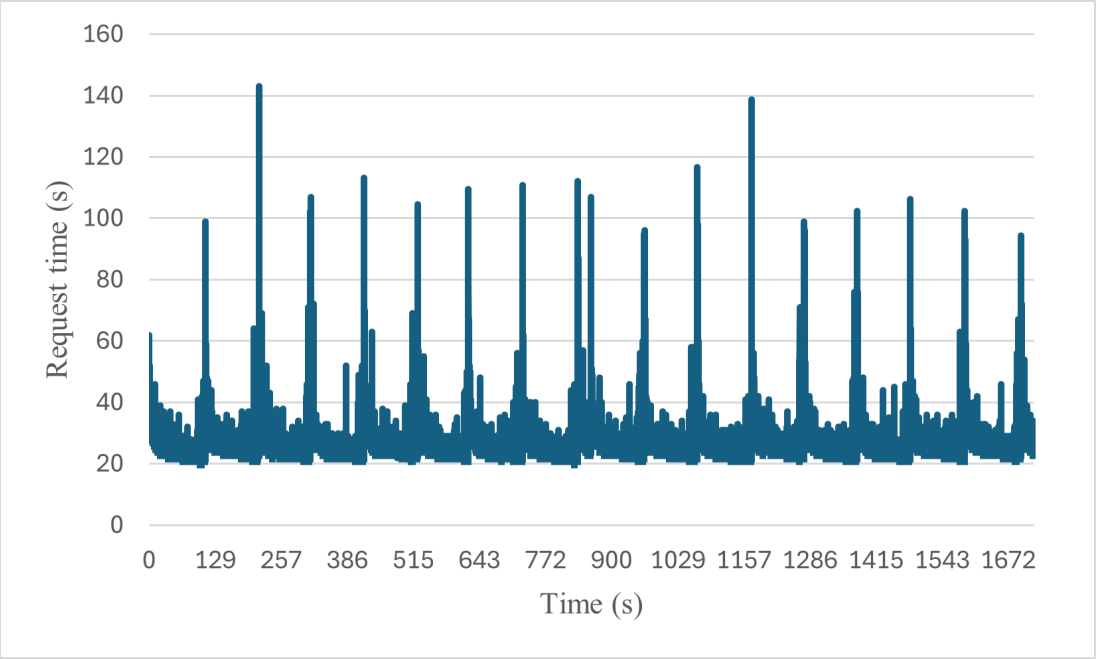

Fig. 5.11: Response time distribution under Epsilon GC for Spring Pet Clinic.

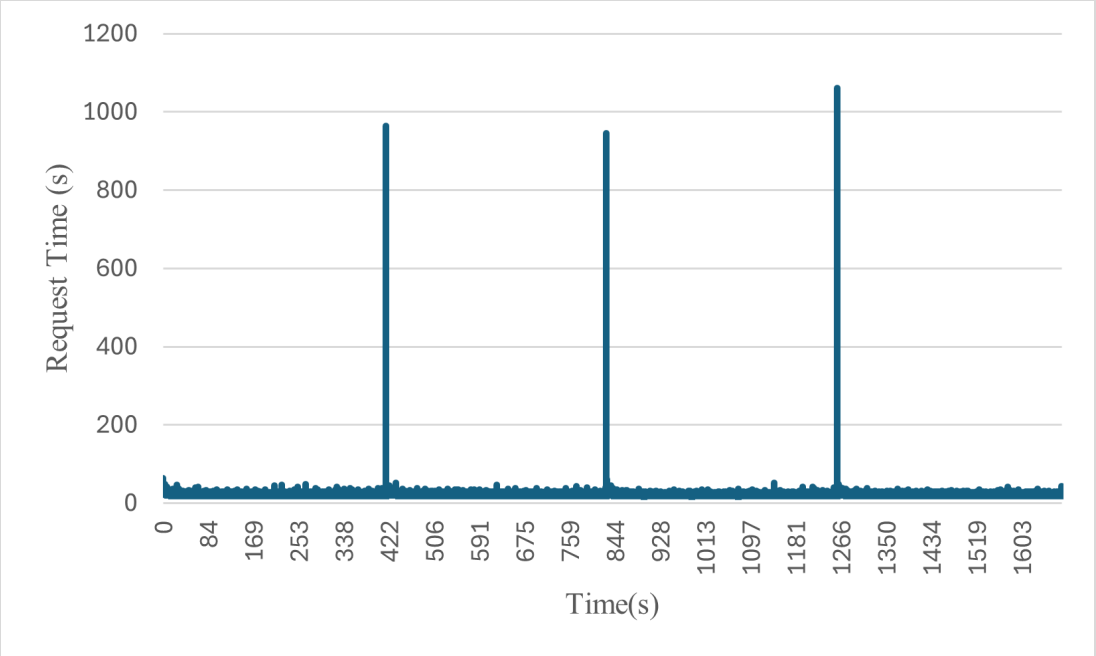

Fig. 5.12: Response time distribution under G1 GC for Spring Pet Clinic.

5.6 Tomcat Server Results

5.6.1 Memory Usage

The Tomcat-based pizza ordering service represents a moderately memory-intensive workload with low payload sizes (1 KB). It is deployed using Apache Tomcat, one of the most common containers in enterprise-grade Java application servers. As such, this service serves twofold, it is a relevant benchmark candidate for evaluating GC performance under real-world conditions and also simulates a production-like deployment environment typical in Java EE stacks.

Under Epsilon GC, heap usage grows from approximately 500–550 MB at startup to over 4.5 GB within 15 20 minutes. Despite this linear memory growth, the container triggers only a single restart throughout the experiment. This demonstrates that Epsilon GC can sustain moderately long workloads with minimal overhead.

In contrast, G1 GC maintains memory usage between 300–420 MB through periodic collection, avoiding restarts altogether.

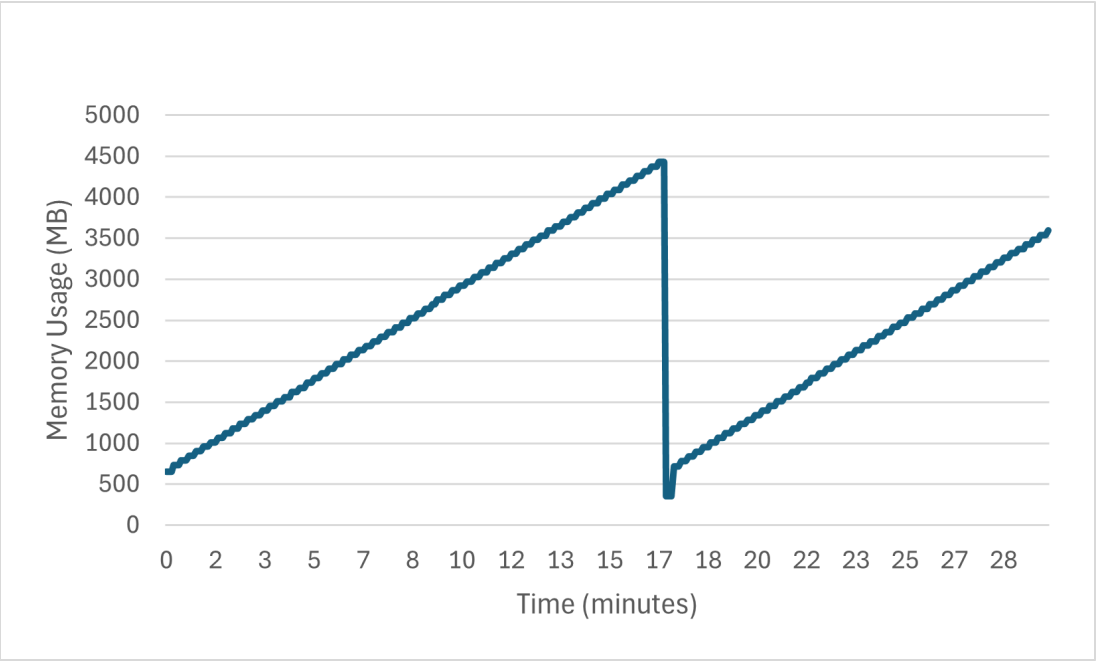

Fig. 5.13: Tomcat memory usage under Epsilon GC.

5.6.2 Latency and Throughput

As seen in Table 5.6, Epsilon GC consistently outperforms G1 in latency and throughput metrics. Median latency improves from 4.6–4.8 ms (G1) to 4.2–4.3 ms under Epsilon. Tail latencies also drop slightly, and throughput increases from 220 RPS to 240 RPS.

These improvements highlight the value of pause-less GC strategies in performance-critical workloads. When bounded runtimes and careful memory management are feasible, Epsilon GC offers compelling gains for production-grade container deployments like this Tomcat-based service.

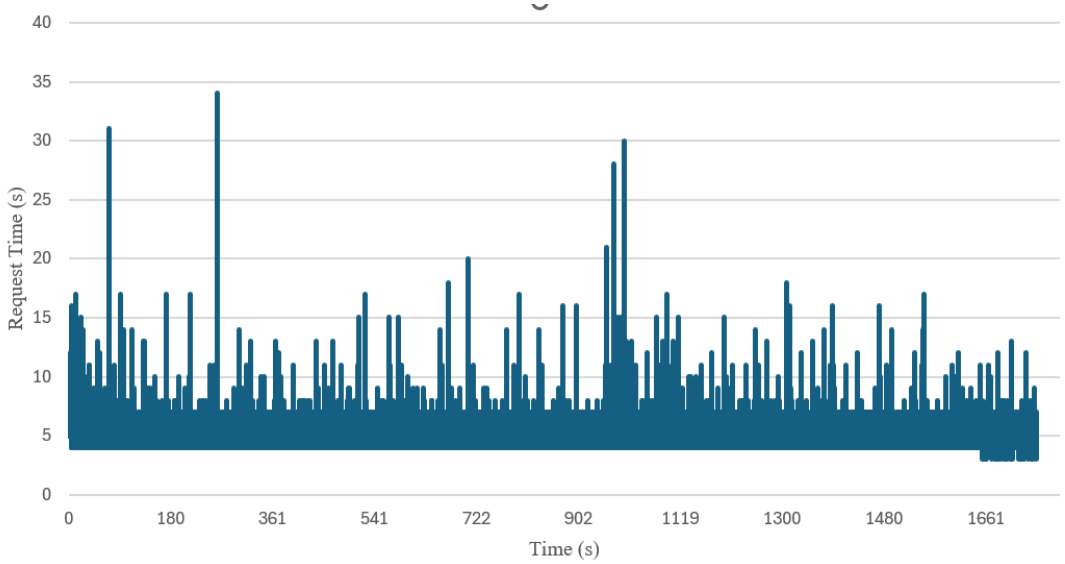

Fig. 5.14: Response time distribution under Epsilon GC for Tomcat.

TABLE 5.6: Tomcat latencies (average of repeated runs).

GC Mode	p50 (ms)	p95 (ms)	p99 (ms)	Throughput
Epsilon	4.2–4.3	4.5	5.5	240
Standard	4.6–4.8	5–5.5	5.8–6	220

5.7 Analysis and Discussion

In this section, lets analyze how the no-GC (Epsilon) mode contrasts with standard GC in terms of memory usage, throughput, latency profiles, and container churn. Table 5.7 provides raw throughput and latency measurements for the aggregator, prime, and cache services, while Table 5.8 highlights the approximate percentage improvements achieved by Epsilon GC relative to standard GC.

5.7.1 Memory Usage and Container Lifetimes

Under Epsilon GC, memory accumulates until it reaches the predefined threshold (about 70% of the limit 8 GB, triggering container restarts. Memory-intensive workloads (such as the aggregator and pet clinic) often encounter restarts every 3–5 minutes. Meanwhile, services with more modest memory footprints (prime or cache) rarely

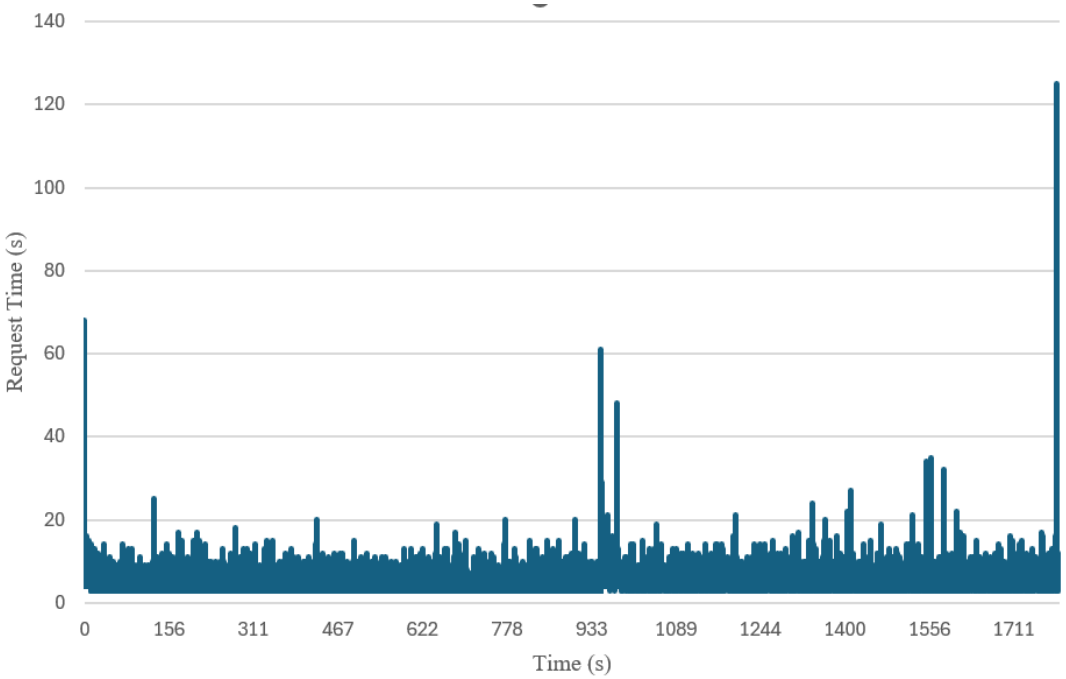

Fig. 5.15: Response time distribution under Standard GC for Tomcat.

TABLE 5.7: Relative Performance of Epsilon GC (No-GC) vs. Standard GC

Service	**Metric**	**Standard GC**	**Epsilon GC**
Aggregator	Throughput (RPS)	62–63	64–67
	Median Latency (ms)	75–76	69–72
	Tail Latency (p99, ms)	149–151	126–131
Prime	Throughput (RPS)	21	22
	Median Latency (ms)	46–47	44–45
	Tail Latency (p99, ms)	60–61	57–58
Cache	Throughput (RPS)	122–123	133
	Median Latency (ms)	76–77	73–74
	Tail Latency (p99, ms)	97–98	96
Pet Clinic	Throughput (RPS)	39	42
	Median Latency (ms)	26–26.5	24.5
	Tail Latency (p99, ms)	41	30–32
Tomcat	Throughput (RPS)	220	240
	Median Latency (ms)	4.6–5	4–4.3
	Tail Latency (p99, ms)	5.8–6	5.5

TABLE 5.8: Approximate Percentage Improvement of Epsilon GC over Standard GC

Service	Throughput Gain	Median Latency Reduction
Aggregator	3–6%	5–7%
Prime	4–5%	3–5%
Cache	8–9%	3–5%
Pet Clinic	7–8%	5–7%
Tomcat	8–9%	9–10%

cross the threshold, resulting in fewer restarts. In the standard GC configuration, container restarts are avoided.

5.7.2 Throughput vs. Latency Trade-offs

Across all three microservices, Epsilon GC typically provides a 5—- 10% increase in throughput, due to the absence of periodic GC pauses. Standard GC, by contrast, relies on garbage collection cycles, which incur pause times of roughly 150–500 ms (occasionally reaching 1–2 s under high load), but avoids the operational overhead of container churn.

5.7.3 Suitability for Different Workloads

For CPU-bound or bursty workloads (such as prime, cache or tomcat server) that infrequently exceed the memory threshold, Epsilon GC offers performance gains with limited container restarts. In contrast, memory-heavy scenarios (e.g. aggregator) may undergo multiple restarts over a given interval, but benefit from eliminating the pause overhead of full GC. Ultimately, choosing between no-GC and standard GC involves balancing container replacement costs against GC pause durations, taking into account each application's memory usage profile and performance requirements.

CHAPTER 6

DISCUSSION

This chapter presents the insights gained from developing and experimentally evaluating the GC-Aware Containerized Deployment framework. Through practical comparisons of Epsilon GC (a no-op collector) and a mainstream collector (G1 GC), it can be appreciated how bypassing major garbage collection events affects performance, resource management, and the operational characteristics of containerized Java services.

6.1 Impact of No-GC Containers on Performance

The evaluation reveals that using Epsilon GC confers a modest yet consistently higher throughput—often in the 5–10% range due to the absence of traditional GC pauses. In many real-world scenarios, such an improvement is nontrivial. Services operating under tight performance constraints, such as financial or interactive web applications, stand to benefit from the steadier rate of request processing afforded by avoiding stop-the-world events.

However, the absence of reclamation mechanisms naturally raises the specter of out-of-memory (OOM) errors, which is where the GC Aware approach intervenes. By closely monitoring container memory usage and rotating Pods before they reach a critical threshold, the framework sidesteps OOM crashes without incurring the overhead of GC. This strategy can effectively mitigate sudden performance drops at the expense of occasionally restarting containers—an overhead that can erode or even negate the gains if restarts are too frequent or improperly orchestrated.

6.2 Memory Thresholds and Container Churn

One of the most noteworthy consequences of adopting a no GC strategy is the increased reliance on container restarts to clear accumulated memory. With memory-intensive workloads, containers may be replaced several times within relatively short intervals. Each replacement event involves:

- **Infrastructure Coordination:** The GC Aware Operator triggers scheduling actions on Kubernetes, spawning a fresh container and winding down the old one.
- **Startup Costs:** Any data cached at startup or warm up routines that load configurations may introduce delays.
- **Potential State Loss:** Even if the application is stateless or semi-stateless, in-flight activity requires careful handover.

Notwithstanding these challenges, many applications may find the trade off acceptable if multi second GC stalls are otherwise unavoidable. In particular, systems that aim to deliver predictable latencies can view container restarts, especially short ones as a better alternative to abrupt, sometimes lengthy pauses. The key lies in balancing the threshold at which the operator triggers replacements, thereby limiting churn while still keeping memory usage from spiraling.

6.3 Trade-offs in Latency Patterns

A particularly striking result is the difference in tail latency distributions. Conventional GC introduces sporadic but occasionally severe spikes. Under peak load, these outliers can extend to hundreds of milliseconds or even seconds if the collector struggles to reclaim a heavily fragmented or large heap. By contrast, the no-GC configuration removes these internal GC stalls, leading to more stable latencies overall.

Nonetheless, container restarts can cause momentary “micro spikes," typically when requests are being drained or rerouted from an old Pod to a new one. While not always negligible, these spikes are often shorter and more predictable, aligning with the basic ethos of many cloud native systems: if a component becomes sluggish or outdated, replace it quickly rather than waiting for it to self-recover. This approach to ephemeral containers takes advantage of the inherent elasticity of Kubernetes deployments, relying on external orchestration rather than on in VM GC.

6.4 Influence of Workload Profiles

The suitability of the GC Aware approach is highly dependent on the nature of the application:

CPU-Bound Services

Services that allocate relatively little memory (e.g., stateless or CPU intensive computations) rarely approach memory thresholds. Here, no GC containers can run without interruption for extended periods, effectively eliminating the overhead of garbage collection.

Moderately Memory-Intensive Services

Applications such as those storing or caching moderate data volumes may trigger a few restarts during the test window. In these cases, the overhead is tolerable, and the improved throughput and shorter GC free latencies outweigh the added complexity.

Heavily Memory Bound Services

Workloads that accumulate large in memory structures will cross the threshold more often, increasing restart frequency. The overhead of repeatedly cycling Pods can be substantial, but the approach can still prove beneficial if large GC cycles would otherwise cause prolonged stop the world pauses. Certain usage patterns may require additional optimizations—like caching reconfiguration or streaming data rather than retaining it.

It follows that the decision to adopt a no GC model revolves around how easily an application tolerates container churn, how critical it is to maintain consistent response times, and whether the architecture can externalize enough state to withstand frequent pod replacements.

6.5 Operational Considerations

6.5.1 Integration with Autoscaling

Since Kubernetes clusters often rely on autoscaling to handle traffic surges, overlaying the GC Aware framework with Horizontal Pod Autoscaler (HPA) rules demands careful tuning. Proactive scaling can dilute load across replicas, avoiding rapid memory accumulation in any single container. Conversely, resource-constrained clusters might see churn events exacerbate scaling oscillations. Operators must therefore align memory thresholds and scaling rules so they complement rather than confound each other.

6.5.2 Cost Implications

Running more Pods or restarting them frequently can raise resource usage and, by extension, operational costs particularly in pay as you go environments. On the flip side, if skipping major GC cycles allows for higher request throughput on fewer instances, overall costs may decrease. Understanding these financial trade offs remains essential for making an informed decision about adopting container rotation for memory management.

6.5.3 Data Persistence and Stateful Workloads

Although the proof of concept focused on microservices with limited in memory state, many enterprise applications cannot so readily discard data. Externalizing caches, sessions, or partial computations to a distributed data store can help, but it adds network overhead and complicates the architecture. Consequently, not all deployments are equally amenable to no GC container lifecycles; highly stateful services must weigh the prospect of losing ephemeral data against incurring GC stalls.

6.5.4 Rolling Deployments and Graceful Shutdown

Effectively rotating Pods before they exhaust memory calls for well structured rolling updates. Graceful shutdown—waiting for in-flight requests to complete ensures minimal user facing disruption. Without such care, restarts introduced by hitting a memory threshold might cause abrupt cutoffs or spikes in error rates.

6.6 Comparison to Existing Literature and Approaches

Recent research into garbage collection (GC) optimization for Java services spans a wide spectrum from low-level JVM tuning to complete runtime reengineering. Many of these require tight coupling with the application's runtime environment, deep GC customization, or even changes to application code and deployment pipelines.

In contrast, the GC-Aware Containerized Deployment framework described in this work offers a significantly more lightweight and infrastructure-centric alternative. It introduces no modifications to the JVM internals, requires no source-level application changes, and maintains compatibility with existing deployment pipelines and standard garbage collectors. Applications run unmodified under Epsilon GC, with the external orchestration layer monitoring container memory pressure and intervening only when necessary. This clean separation of concerns aligns well with modern DevOps and cloud-native philosophies, making the solution easy to adopt in practice.

From a systems design perspective, one of the principal advantages of this approach lies in its non-intrusiveness. Many memory-optimized JVM techniques necessitate expert-level tuning and deep understanding of the memory behavior of each service, often requiring per-application calibration. In contrast, the GC-Aware mechanism applies uniformly to any Java container, stateless or otherwise. Its operator pattern and memory threshold logic are externalized and parameterized, allowing quick deployment across services with minimal engineering effort.

Moreover, performance comparisons from the experimental results suggest that this architecture performs competitively with state-of-the-art GC techniques in many relevant metrics—especially for services that are CPU-bound or moderately memory-intensive. Median and tail latencies are consistently improved under Epsilon GC, and throughput shows gains in the 5–10% range across several benchmarks. These results mirror or exceed performance figures reported for certain finely tuned GC configurations in literature, yet require less operational overhead.

While other techniques such adaptive GC policies may offer more granular control over memory behavior, they often demand invasive instrumentation or custom JVM builds. By offloading lifecycle control to Kubernetes, this framework takes advantage of existing container management infrastructure, thereby avoiding the need to re-engineer the garbage collection process itself. This makes it especially attractive

for teams looking for drop-in performance enhancements without rewriting services or altering deep runtime configurations.

In summary, the proposed framework fills a notable gap in the current body of work: it provides a practical, low-friction path to GC performance optimization by leveraging container-level restarts instead of runtime-level memory reclamation. This inversion of control—from VM to container orchestrator—aligns well with the principles of microservice resilience and operational simplicity. While not suitable for all workloads, especially highly stateful systems, the approach represents a compelling middle ground between performance gains and operational cost, and demonstrates a unique contribution to the evolving discussion on memory management in cloud-native Java applications.

6.7 Summary of Discussion

Overall, the GC Aware Containerized Deployment framework demonstrates how an external rotation mechanism can reliably achieve noticeable throughput gains and tighter latency distributions by circumventing traditional Java GC. This, however, comes with trade offs:

- **Reduced GC Overhead vs. Churn Overhead:** Eliminating GC cycles can enhance performance, but it can also yield frequent container restarts for memory intensive scenarios.
- **Predictable vs. Sporadic Latencies:** Restart induced micro spikes are generally shorter and more predictable than long GC events, yet they still require robust load balancing and orchestration.
- **Stateless vs. Stateful Architectures:** Stateless or lightly stateful applications adapt smoothly to ephemeral containers. Heavily stateful services must consider the cost and complexity of re initializing data or persisting state externally.

These outcomes substantiate the idea that container orchestration can be leveraged as an alternative or complementary strategy to in JVM garbage collection. By offloading memory reclamation to Kubernetes—treating containers as transient units—services benefit from more uniform performance profiles. While not universally applicable, this approach is particularly compelling for services that place a premium on consistent low latency, are designed to handle ephemeral lifecycles, and can manage or externalize state effectively.

CHAPTER 7

CONCLUSION

This chapter brings together the key findings, implications, and potential future directions for the GC Aware Containerized Deployment framework. By interweaving the strengths and limitations of the current implementation with ideas for enhancement, it offers a comprehensive view of how container rotation can serve as an alternative or complement to traditional Java garbage collection in cloud native environments.

7.1 Contributions and Key Insights

The proposed framework adopts a novel tactic for mitigating garbage collection (GC) overhead in Java microservices by shifting memory reclamation from in JVM collectors to container lifecycle events. Instead of relying on periodic GC cycles, the system restarts containers when memory usage approaches a defined threshold. This design effectively harnesses the ephemeral and elastic nature of containers in Kubernetes.

Epsilon Based Design

Leveraging Epsilon GC (a no op collector) allows services to avoid GC pauses entirely, thereby improving throughput and reducing latency spikes. Experimental results indicate that CPU bound or moderately memory intensive workloads often experience a 5–10% boost in throughput compared to using a standard collector.

Operator Driven Orchestration

An essential component is the GC Aware Operator, which monitors pod level memory usage and seamlessly initiates container replacements. This reduces the complexity of manually tracking memory overhead while preserving availability—especially important when dealing with services that scale dynamically.

Empirical Evaluation

Systematic comparisons of Epsilon GC versus a standard collector (G1 GC) were conducted on three representative microservices (Bulk Data Aggregator, Prime Computation, and In Memory Cache). The results consistently highlight a trade off between the predictable performance gains of avoiding GC and the operational overhead of frequent container restarts in more memory intensive scenarios.

Overall, these contributions indicate that no GC configurations offer notable gains in scenarios involving strict latency budgets or high concurrency demands. Applica-

tions with smaller or medium memory footprints reap these benefits with fewer restarts, while highly memory bound services can still benefit if they can tolerate more frequent container churn and state re initialization.

7.2 Limitations and Challenges

Despite the promise of this approach, certain constraints became clear during implementation and testing:

- **Frequent Restarts with Large Heaps:** Memory heavy applications (e.g., the Bulk Data Aggregator) may trigger multiple restarts in short windows if heap usage quickly approaches the set threshold. The extra overhead from frequent container re initializations can partially offset the gains from eliminating GC pauses.
- **Statefulness:** While stateless or semi stateless architectures adapt seamlessly to ephemeral containers, services storing large, essential data in memory face a risk of data loss upon restart. Externalizing session or cache data is viable but adds network overhead and architectural complexity.
- **Threshold Tuning:** Determining a safe yet efficient memory threshold is vital. Setting it too low leads to unnecessary churn, whereas a threshold set too high increases the risk of out of memory incidents if the new container is not fully ready in time.

These factors underscore that the framework's efficacy hinges on an application's capacity to handle restarts gracefully, the complexity of its data lifecycle, and the concurrency and latency requirements of its workload.

7.3 Implications for Microservice Architectures

In modern, cloud native ecosystems, microservices are already designed with ephemerality in mind: container deployments frequently scale out or roll versions with minimal disruption. The GC Aware approach dovetails nicely with such principles:

- **Automated Deployment Pipelines:** Rolling upgrades become more predictable if occasional container restarts are a normal part of application behavior rather than an exception triggered by out of memory errors.
- **Horizontal and Vertical Scalability:** Memory thresholds can work in tandem with autoscaling rules (e.g., the Kubernetes Horizontal Pod Autoscaler, HPA), balancing memory usage across multiple replicas while averting abrupt GC pauses.

- **Reduced Latency Spikes:** By eliminating "stop the world" events, microservices can maintain tighter worst case latencies, which is valuable in use cases such as financial transactions or real time analytics.

Nevertheless, the framework also highlights the operational overhead of container churn and the complexities of scaling stateful data workloads. Each organization must weigh these trade offs against performance gains, maintenance costs, and existing infrastructure design.

7.4 Future Work

Although the GC Aware Containerized Deployment framework has shown encouraging results, several avenues remain for refining and extending it. These focus on broadening its applicability, improving resource efficiency, and deepening our understanding of cost performance trade offs:

- **Adaptive Thresholds and Intelligent Autoscaling:** Rather than relying on a static threshold (e.g. 70% of the container limit), future iterations could use dynamic thresholds that adjust in real time based on observed load, latency metrics, or predictive modeling. Closer integration with the HPA might allow for flexible scaling decisions that factor in CPU load, memory consumption, and response times collectively.

- **Handling Stateful and Persistent Workloads:** Many enterprise systems maintain extensive in memory state that cannot be casually discarded. Moving this data into external caches (e.g. Redis) or partitioning it across multiple pods may minimize restart overhead. Evaluations with more complex stateful services would clarify how best to balance ephemeral container rotations with persistent or semi persistent data requirements.

- **Dynamic GC Toggling and Advanced Collectors:** Epsilon GC is either fully off or on, which can be suboptimal for applications that ramp up memory usage early but require less in steady state. Advanced collectors like Shenandoah or ZGC aim for very low pause times and may provide a middle ground particularly if there is a way to switch between a no GC approach and a low pause GC at runtime. Although not trivial to implement with current JVMs, this concept could further reduce container churn.

- **Performance Modeling and Cost Analysis:** A more formal framework for analyzing operational costs including container spin up, CPU overhead for restarts, and additional cloud expenses would help determine the break even points. This is particularly relevant for large scale applications where even minor increases in container churn translate to substantial resource bills.

Developing these enhancements could bolster the GC Aware Containerized Deployment framework, making it more responsive, adaptable, and robust across a broader spectrum of real world scenarios.

7.5 Final Remarks

In essence, this thesis proposes and validates an alternative strategy for Java memory management under containerized deployments. By deferring garbage collection to the lifecycle of containers themselves rather than the JVM, the GC Aware framework shows tangible improvements in throughput and latency particularly in CPU bound or moderate memory scenarios. The trade off lies in managing container churn, ensuring graceful transitions, and designing the application to tolerate ephemeral containers.

For usecases seeking consistent low latency, the approach can serve as a compelling option, particularly when combined with stateless or externally managed data architectures. As with most optimizations, the choice between a traditional GC based deployment and a no GC strategy will depend on each workload's unique demands, operational tolerance for restarts, and budgetary constraints. Ultimately, the interplay between application design, container orchestration, and advanced garbage collection techniques continues to evolve, presenting fertile ground for ongoing research and innovation in cloud native Java ecosystems.

REFERENCES

[1] "Stateful vs stateless — redhat.com," https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless.

[2] H. Grgic, B. Mihaljević, and A. Radovan, "Comparison of garbage collectors in java programming language," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, pp. 1539–1544.

[3] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2019, pp. 1677–1682.

[4] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[5] "Memory Management in Java - Javatpoint — javatpoint.com," https://www.javatpoint.com/memory-management-in-java.

[6] "Understanding Memory Management — docs.oracle.com," https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html.

[7] "JIT in Java | Java JIT - Javatpoint — javatpoint.com," https://www.javatpoint.com/jit-in-java.

[8] S. Tavakolisomeh, R. Bruno, and P. Ferreira, "BestGC: An Automatic GC Selector," *IEEE Access*, vol. 11, pp. 72 357–72 373, 2023.

[9] Stefan Johansson. (2023) Jdk 21: The gcs keep getting better. Stefan Johansson's Blog. [Online]. Available: https://kstefanj.github.io/2023/12/13/jdk-21-the-gcs-keep-getting-better.html

[10] R. Lakshmanan. (2024) Cms gc algorithm removed from java 14? GC Easy - Universal Java GC Log Analyser. [Online]. Available: https://blog.gceasy.io/cms-gc-algorithm-removed-from-java-14/

[11] D. H. Kurniawan, C. A. Stuardo, R. A. OS, and H. S. Gunawi, "Notification and prediction of heap management pauses in managed languages for latency stable systems," *To appear*, n.d.

[12] "Jep 404: Generational shenandoah (experimental)," https://openjdk.org/jeps/404.

[13] Inside.Java. (2023) Introducing generational zgc. [Online]. Available: https://inside.java/2023/11/28/gen-zgc-explainer/

[14] "JEP 318: Epsilon: A No-Op Garbage Collector (Experimental) — openjdk.java.net," https://openjdk.java.net/jeps/318.

[15] "Using JConsole - Java SE Monitoring and Management Guide — docs.oracle.com," https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html.

[16] "VisualVM: Home — visualvm.github.io," https://visualvm.github.io/.

[17] Oracle. (n.d.) Jrockit to hotspot migration guide. Oracle Help Center. [Online]. Available: https://docs.oracle.com/en/java/javase/11/jrockit-hotspot/logging.html

[18] F. Xian, W. Srisa-an, and H. Jiang, "Garbage collection: Java application servers' achilles heel," *Science of Computer Programming*, vol. 70, no. 2-3, pp. 89–110, 2008.

[19] D. Fireman, R. Lopes, and J. A. B. Monteiro, "Using load shedding to fight tail-latency on runtime-based services," in *Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, 2017, available via Google Scholar: https://scholar.google.com/scholar?as_q=Using+load+shedding+to+fight+tail-latency+on+runtime-based+services&as_occt=title&hl=en&as_sdt=_.

[20] J. Zhao, "Improving performance of garbage collection for data-intensive applications in cloud systems," Ph.D. dissertation, University of Colorado Colorado Springs, 2024.

[21] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 25–36, 2004.

[22] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, "Adaptive gc-aware load balancing strategy for high-assurance java distributed systems," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE, 2015, pp. 68–75.

[23] A. O. Portillo-Dominguez, P. Perry, D. Magoni, M. Wang, and J. Murphy, "Trini: an adaptive load balancing strategy based on garbage collection for clustered java systems," *Software: Practice and Experience*, vol. 46, no. 12, pp. 1705–1733, 2016.

[24] S. Saraswati, S. Chatterjee, and R. Ramachandra, "Steal-a-gc: Framework to trigger gc during idle periods in distributed systems," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 392–400.

[25] N. Veretelnyk. (2021) Garbage collection: V8's orinoco - nikolay veretelnyk - medium. [Online]. Available: https://medium.com/@nikolay.veretelnik/garbage-collection-v8s-orinoco-452b70761f0c

[26] D. Fireman, J. Brunet, R. Lopes, D. Quaresma, and T. E. Pereira, "Improving tail latency of stateful cloud services via gc control and load shedding," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pp. 121–128.

[27] Z. Zhuang, C. Tran, H. Ramachandra, and B. Sridharan, "Eliminating os-caused large jvm pauses for latency-sensitive java-based cloud platforms," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 694–701.

[28] "Kubernetes Documentation — kubernetes.io," https://kubernetes.io/docs/home/.

[29] "Docker overview — docs.docker.com," https://docs.docker.com/get-started/overview/.

[30] T. Hu and Y. Wang, "A kubernetes autoscaler based on pod replicas prediction," in *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. IEEE, 2021, pp. 238–241.

[31] "Understand the OutOfMemoryError Exception — docs.oracle.com," https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html, [Accessed 10-12-2024].

[32] "GitHub - kubernetes/client-go: Go client for Kubernetes. — github.com," https://github.com/kubernetes/client-go.

[33] "ReplicaSet — kubernetes.io," https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, [Accessed 12-12-2025].

[34] C. Stenberg, "Frameworks for lifecycle management of stateful applications on top of kubernetes: Testing and evaluation," Dissertation, 2022.

[35] M. Sebrechts, T. Ramlot, S. Borny, T. Goethals, B. Volckaert, and F. De Turck, "Adapting kubernetes controllers to the edge: on-demand control planes using wasm and wasi," in *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*, 2022, pp. 195–202.

[36] "Kubernetes Metrics (v1beta1) — kubernetes.io," https://kubernetes.io/docs/reference/external-api/metrics.v1beta1/.

[37] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3539606

[38] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 970–973.

[39] O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in kubernetes cluster: Automatic anomalies detection using prometheus," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 565–570.

[40] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *IJCSNS International Journal of Computer Science and Network Security*, vol. 17, no. 3, March 2017, manuscript received March 5, 2017, revised March 20, 2017.

[41] "Configure Liveness, Readiness and Startup Probes — kubernetes.io," https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.

[42] "Design of Shutdown Hooks API — docs.oracle.com," https://docs.oracle.com/javase/8/docs/technotes/guides/lang/hook-design.html.

[43] "kube-apiserver — kubernetes.io," https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/.

[44] A. Muppeda, " A Hands-On Guide to Kubernetes Custom Resource Definitions (CRDs) With a Practical Example — muppedaanvesh," https://medium.com/@muppedaanvesh/a-hand-on-guide-to-kubernetes-custom-resource-definitions-crds-with-a-practical-example-%EF%B8%8F-84094861e90b.

[45] M. Devops, "CRDs in Kubernetes — minimaldevops.com," https://minimaldevops.com/crds-in-kubernetes-c38037315548.

[46] "Extend the Kubernetes API with CustomResourceDefinitions — kubernetes.io," https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/.

[47] "GitHub - anveshmuppeda/kubernetes: Kubernetes Complete Hands-On Guides — github.com," https://github.com/anveshmuppeda/kubernetes.

[48] "Operating etcd clusters for Kubernetes — kubernetes.io," https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/.

[49] "Installation | Rancher Desktop Docs — docs.rancherdesktop.io," https://docs.rancherdesktop.io/getting-started/installation/.

[50] "Working with Containers | Rancher Desktop Docs — docs.rancherdesktop.io," https://docs.rancherdesktop.io/tutorials/working-with-containers.

[51] "GitHub - kubernetes-sigs/metrics-server: Scalable and efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines. — github.com," https://github.com/kubernetes-sigs/metrics-server.

[52] "pkg package - sigs.k8s.io/controller-runtime/pkg - Go Packages — pkg.go.dev," https://pkg.go.dev/sigs.k8s.io/controller-runtime/pkg.

[53] "Releases · kubernetes-sigs/kubebuilder — github.com," https://github.com/kubernetes-sigs/kubebuilder/releases.

[54] "- YouTube — youtube.com," https://www.youtube.com/watch?v=rfXk6svglrA.

[55] "Using RBAC Authorization — kubernetes.io," https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[56] G. Rostami, "Role-based access control (rbac) authorization in kubernetes," *Journal of ICT Standardization*, vol. 11, no. 3, pp. 237–260, 2023.

[57] "Introducing Kubebuilder: an SDK for building Kubernetes APIs using CRDs — kubernetes.io," https://kubernetes.io/blog/2018/08/10/introducing-kubebuilder-an-sdk-for-building-kubernetes-apis-using-crds/.

[58] "Containerizing Test Tooling: Creating your Dockerfile and Makefile — docker.com," https://www.docker.com/blog/containerizing-test-tooling-creating-your-dockerfile-and-makefile/.

[59] K. Aykurt, R.-M. Ursu, J. Zerwas, P. Krämer, N. Asadi, L. Wong, and W. Kellerer, "Hypa: Hybrid horizontal pod autoscaling with automated model updates," in *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2023, pp. 8–14.

[60] "Horizontal Pod Autoscaling — kubernetes.io," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[61] V. Tiwari, S. Upadhyay, J. K. Goswami, and S. Agrawal, "Analytical evaluation of web performance testing tools: Apache jmeter and soapui," in *2023 IEEE 12th International Conference on Communication Systems and Network Technologies (CSNT)*, 2023, pp. 519–523.

[62] Q. Cooper, D. Krishnamurthy, and Y. Amannejad, "Budget aware performance test selection for microservices," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024, pp. 376–385.