

LB/TH/41/2025

TH6001

**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

Declaration of the Candidate & Supervisor	i
Abstract	ii
Table of Contents	iii
List of Figures	vii
List of Abbreviations	viii
1 Introduction	1
1.1 Problem Statement	1
1.2 Research Objectives	2
1.3 Motivation and Context	2
1.3.1 Challenges of GC in Cloud-Native Applications	2
1.3.2 Advantages of Container Rotation Over GC Tuning	3
1.4 Research Scope	3
1.5 Contributions	4
1.6 Structure of the Thesis	4
2 Literature review	6
2.1 Introduction	6
2.2 Java Virtual Machine	6
2.2.1 Memory management	6
2.3 Garbage Collectors in Java	8
2.3.1 Serial and Parallel GC	9
2.3.2 Concurrent Mark Sweep (CMS)	9
2.3.3 Garbage First (G1) GC	9
2.3.4 Shenandoah GC	10
2.3.5 Z Garbage Collector (ZGC)	10
2.3.6 Epsilon GC (No Op Collector)	12
2.3.7 Ongoing Trends	12
2.4 Memory monitoring	13

2.4.1	JDK	13
2.5	Impact of GC on Applications	15
2.5.1	Throughput Costs	15
2.5.2	Latency and Tail Latency	15
2.5.3	Long Pauses and “Stop the World” Effects	16
2.5.4	Multi Tenancy and Interference	16
2.6	Improvements to the traditional GC operation	18
2.6.1	GC-Aware Load Balancing and Scheduling	18
2.6.2	Idle-Time Garbage Collection	21
2.6.3	Load Shedding and Request Management during GC	24
2.6.4	Eliminating OS-Caused Large JVM Pauses	27
2.7	Containerized deployment	30
2.7.1	Kubernetes	31
2.7.2	Docker	32
2.7.3	Readiness and liveness probes	32
2.7.4	Kubernetes Autoscaler Based on Pod Replicas Prediction	33
2.8	Literature review conclusion	34
3	Design and Methodology	35
3.1	Overview	35
3.2	Core Components	36
3.2.1	JVM with Epsilon GC	36
3.2.2	MemoryWatcher Controller	36
3.2.3	Kubernetes Metrics Server	37
3.2.4	Application Pods	37
3.3	GC-Aware Lifecycle Workflow	37
3.3.1	Step-by-Step Logic	37
3.4	Container Graceful termination	38
3.5	Design Summary	40
4	Implimentation	41
4.1	Overview of GC-Aware Approach	41
4.2	System Architecture and Primary Components	42

4.2.1	MemoryWatcher (Kubernetes Custom Resource Definition (CRD))	42
4.2.2	GC-Aware Operator (Controller)	42
4.2.3	Java Microservice Pods	43
4.2.4	Kubernetes Metrics Server	43
4.2.5	Kubernetes API Server etcd	43
4.3	Implementation of GC-Aware Deployment	43
4.3.1	Kubernetes and Metrics Server Setup	44
4.3.2	MemoryWatcher (Kubernetes Custom Resource Definition / CRD)	44
4.3.3	GC-Aware Operator (Controller)	46
4.3.4	Container Image with no op GC	51
4.4	Practical Challenges and Limitations	52
5	Experimental Evaluation and Results	53
5.1	Experiment Setup and Methodology	53
5.1.1	Cluster and Hardware	53
5.1.2	Cluster and Hardware	53
5.1.3	Test Scenarios and Metrics Collected	54
5.2	Bulk Data Aggregator Results	55
5.2.1	Memory Usage	55
5.2.2	Latency and Throughput	55
5.2.3	Container Lifetime and GC Observations	55
5.3	Prime Computation Results	57
5.3.1	Memory Usage	57
5.3.2	Latency and Throughput	57
5.4	In-Memory Cache Results	59
5.4.1	Memory Usage	59
5.4.2	Latency and Throughput	59
5.5	Spring Pet Clinic Results	61
5.5.1	Memory Usage	61
5.5.2	Latency and Throughput	62

5.6	Tomcat Server Results	64
5.6.1	Memory Usage	64
5.6.2	Latency and Throughput	64
5.7	Analysis and Discussion	65
5.7.1	Memory Usage and Container Lifetimes	65
5.7.2	Throughput vs. Latency Trade-offs	67
5.7.3	Suitability for Different Workloads	67
6	Discussion	68
6.1	Impact of No-GC Containers on Performance	68
6.2	Memory Thresholds and Container Churn	68
6.3	Trade-offs in Latency Patterns	69
6.4	Influence of Workload Profiles	69
6.5	Operational Considerations	70
6.5.1	Integration with Autoscaling	70
6.5.2	Cost Implications	70
6.5.3	Data Persistence and Stateful Workloads	70
6.5.4	Rolling Deployments and Graceful Shutdown	71
6.6	Comparison to Existing Literature and Approaches	71
6.7	Summary of Discussion	72
7	Conclusion	73
7.1	Contributions and Key Insights	73
7.2	Limitations and Challenges	74
7.3	Implications for Microservice Architectures	74
7.4	Future Work	75
7.5	Final Remarks	76
	References	77

LIST OF FIGURES

Figure	Description	Page
Figure 2.1	Visual representation of the Java Virtual Machine with key memory components [2]	7
Figure 2.2	Heap structure [7]	8
Figure 2.3	Normalized 99th percentile GC pause time [9]	11
Figure 2.4	Jconsole [15]	13
Figure 2.5	VisualVM [16]	14
Figure 2.6	Total numbers of executed garbage collections per application across three GC algorithms [2]	17
Figure 2.7	Total execution time of garbage collections per application expressed in milliseconds [2]	18
Figure 2.8	GC-Aware Load Balancer [22]	19
Figure 2.9	Adaptive GC-Aware Load Balancing Strategy [22]	20
Figure 2.10	GC impact on Spark PageRank execution [24]	22
Figure 2.11	Spark stage execution event timeline [24]	23
Figure 2.12	GCI Architecture [26]	25
Figure 2.13	GCI flow [26]	26
Figure 2.14	Auto scaler System structure [30]	33
Figure 2.15	Pod Expansion [30]	34
Figure 3.1	High-Level Overview of GC-Aware Containerized Deployment.	35
Figure 3.2	Traffic Flow and Container Lifecycle in GC-Aware Approach	38
Figure 3.3	Pod Terminating Flow	39
Figure 4.1	High Level Architecture of the GC Aware container Deployment	42
Figure 4.2	Initialization Workflow for GCWatcher Resource Management	45
Figure 4.3	MemoryWatcher resource definition	45
Figure 4.4	Lifecycle Management Workflow of GC-Aware Operator	46
Figure 4.5	Reconsiliation Logic	47
Figure 4.6	Decision Flow Diagram	50
Figure 4.7	Application Health check	51
Figure 4.8	Application Docker file	51
Figure 5.1	Memory usage for Aggregator under Epsilon GC.	56
Figure 5.2	Response time distribution under standard GC	56
Figure 5.3	Response time distribution under Epsilon GC	57
Figure 5.4	Prime memory usage under Epsilon GC.	58
Figure 5.5	Response time distribution under Epsilon GC for prime-finding.	58

Figure 5.6	Response time distribution under Standard GC for prime-finding.	59
Figure 5.7	Cache memory usage with Epsilon GC. One container restart occurs upon exceeding ~5.6 GB.	60
Figure 5.8	Response time distribution under Epsilon GC for the cache service.	60
Figure 5.9	Response time distribution under standard GC for the cache service.	61
Figure 5.10	Spring Pet Clinic memory usage under Epsilon GC.	62
Figure 5.11	Response time distribution under Epsilon GC for Spring Pet Clinic.	63
Figure 5.12	Response time distribution under G1 GC for Spring Pet Clinic.	63
Figure 5.13	Tomcat memory usage under Epsilon GC.	64
Figure 5.14	Response time distribution under Epsilon GC for Tomcat.	65
Figure 5.15	Response time distribution under Standard GC for Tomcat.	66

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