Vrije Universiteit Amsterdam

Bachelor Thesis

Analyzing and Optimizing RAM Usage of Containers in Kubernetes Environments

---

Author: Hussein Sarrar 2803879

1st supervisor:	Matthijs Jansen
daily supervisor:	Felix Leubken
2nd reader:	Dr. Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 6, 2026

Abstract

Kubernetes memory requests and limits must be declared by operators for each container before runtime. These configurations are vital for scheduling and application capacity, and naively set configurations can either over-reserve memory or risk an OOM kill for reserving too little. Industry data shows that around only 20% of reserved memory is actually used since operators tend to take caution and add safety margins when setting requests and limits. This gap stems back to what current memory monitoring tools report for a container's memory usage, a single aggregate number, hiding the composition of memory and whether it is reclaimable or not. The key scientific question that this thesis addresses is how a cgroup memory decomposition can explain and close the gap between declared and actual container memory in Kubernetes.

We design and implement *MemScope*: a lightweight tool that reads cgroup memory statistics directly from within a pod through `kubectl exec` and reports a breakdown of container memory, requiring no cluster admin privileges or deployment changes. We evaluate MemScope on Ocean, an AI web crawler developed by Capisoft and used in production on a shared AWS EKS cluster. The decomposition MemScope produces reveals a fraction of a container's memory that cannot be let go under pressure, the irreducible floor, equal to the full cgroup-reported memory usage (`memory.current`) excluding all reclaimable components. A limit is then derived using the irreducible floor by adding a 10% safety margin, and then emulated on a real production workload in Ocean. The new limit exhibits a 9% tighter reservation than the Kubernetes-native Vertical Pod Autoscaler. The code and resulting data for this thesis work are openly available at `https://github.com/HusseinSarrar05/memscope.git`.

Contents

1

Introduction

Modern society depends on data centers across all sectors, from commercial to medical, governmental, and scientific; they have become a backbone of modern infrastructure. Along with their growing importance, so too has grown their energy consumption (1). The massive boom in Artificial Intelligence has accelerated this further, with AI-related energy use estimated at 10–50 TWh in 2023, representing 5–15% of global data center energy use and projected to reach 200–400 TWh by 2030, rising to 35–50% of overall data center energy use (2). The scale of this consumption makes efficiency improvements necessary to meet growing computational demand sustainably, and better resource utilization is one of the most impactful tools available.

At the scale of modern data centers, this efficiency challenge increasingly lies in how applications are deployed and managed within them. Containerized deployments have become the dominant model for modern application delivery (3, 4), playing a key role in managing this efficiency. At production scale, a single container is rarely sufficient; hundreds or even thousands are required to reliably host a modern application (5). Coordinating deployments at this scale is managed by Kubernetes, the industry standard container orchestration platform (6, 7). At this scale, small per-container inefficiencies can quickly multiply into significant cluster-wide waste as visualized by Figure 1.1.

Kubernetes depends on user-declared memory configurations to manage resources. Each container declares a memory request, representing a guaranteed minimum allocation, and a memory limit, representing the maximum amount of memory it is allowed to consume. These limits are hard and memory is non-compressible; a container that exceeds its limit is terminated (7), making the stakes of misconfiguration significantly higher. Since declarations must be set before runtime behavior is known, operators always take caution, adding safety margins to avoid unexpected terminations. The result is systematic over-reservation of memory across the cluster (8).

While declared memory could in principle reflect actual runtime usage, it does not match in most scenarios because memory consumption consists of multiple complex components such as application memory, container overhead, and OS-level usage, and other factors

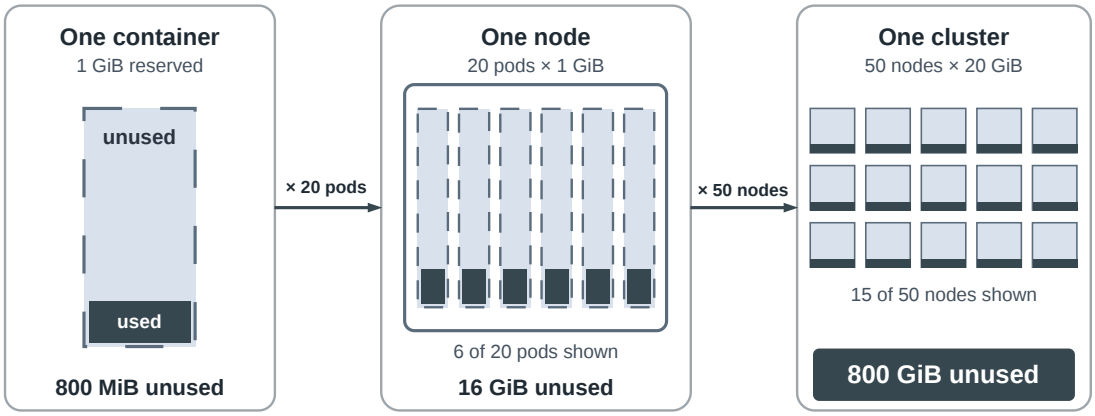

Figure 1.1: Per-container over-reservation compounding across a cluster. The reserved fraction that goes unused is the same at every level, but its size grows with the number of pods and nodes. Figures are illustrative, applying the industry-reported 20% memory utilization rate to a 1 GiB reservation (8).

that change depending on complex runtime effects that are difficult to predict (9). This complexity makes the gap between declared and actual memory hard to characterize; optimization becomes guess-based rather than data-driven. This research is conducted in collaboration with Capisoft, a software company that develops custom software solutions across a wide range of domains, managing and deploying containerized applications for its clients. For a company operating at this scale, memory misconfiguration translates directly into operational cost and technical risk (8). All in all, the gap between what is declared and what is actually consumed remains poorly understood, and closing it requires a clear picture of what container memory actually consists of and where inefficiencies arise. This work investigates exactly that.

1.1 Problem Statement

Container memory is not a single quantity; it spans multiple components: anonymous memory, page cache, kernel memory, slab allocations, and runtime overhead (9). Components are spread across cgroup accounting, the container runtime, and the OS, making it challenging to observe them individually. Operators and monitoring tools typically see only an aggregate figure: `memory.current`, the total memory usage of a pod in a single value. It is not possible to produce a meaningful analysis of how memory is actually consumed using an aggregate value. **We identify a critical lack of understanding of what container memory actually consists of, and where it comes from. (P1)**

The gap in understanding explained in **P1** makes the relationship between declared and actual memory difficult to reason about. Declared configurations are static and predefined, while actual usage is dynamic, flexible and workload-dependent (10). The size and causes of the gap between declared and actual usage vary with workload type, container configuration, and runtime behavior. This leads to complications in scheduling decisions, purely reactive tuning, and in practice, operators tend to reserve significantly more memory than applications actually consume, resulting in poor cluster utilization (8). Prior work focuses on high-level performance and orchestration (11), leaving fine-grained empirical analysis of this gap missing, and optimization guess-based rather than data-driven. **The size and causes of this discrepancy remain poorly characterized. (P2)**

Even with measurements in hand, using findings to create applicable configuration strategies is a difficult task. Strategies must balance two goals: safety (avoid OOM kills) and efficiency (reduce over-reservation). Automated tools like the Vertical Pod Autoscaler (VPA) exist and present a possible strategy to mitigate the issue, but rely on the same aggregate metrics without understanding the complex composition of memory consumption, they operate on an unreliable signal (12). **Practical, data-driven configuration guidelines for memory in Kubernetes-managed containerized workloads do not exist. (P3)**

Investigating the problems outlined above requires a reliable method of measurement and analysis of memory behavior across varying workloads and configurations. No dedicated framework exists for doing so at the level of detail need for this work. **The infrastructure needed to investigate memory behavior in Kubernetes-managed containerized workloads at this level of detail does not exist. (P4)**

1.2 Research Questions

The goal of this work is to answer the **Main Research Question (MRQ)**: *How accurately do Kubernetes memory requests and limits represent actual runtime memory usage, and what factors contribute to observed discrepancies?*.

The scope is limited to RAM resources and focuses on Kubernetes-managed containerized workloads, relying on empirical measurement and analysis. Answering this question requires both a clear breakdown of what runtime memory consists of, and a reliable way to measure it, addressing **P4** as part of this work. The **MRQ** is decomposed into three sub-questions, each addressing one of the problems presented in the previous section.

Research Question 1 (RQ1):

What are the main contributors to container memory usage in Kubernetes-managed environments? As discussed in **P1**, understanding what runtime memory is composed of is a requirement before carrying out any analysis on the discrepancy. Existing studies explore the surface performance and memory usage of containerized workloads (13), but do not present a low-level breakdown of individual memory components. The challenge will lie in decomposing memory across multiple layers: application memory, container runtime overhead, OS-level components; each reported differently by different tools. Obtaining this breakdown provides a principled, decomposed foundation for **RQ2** and **RQ3**.

Research Question 2 (RQ2):

How does memory usage differ between actual runtime consumption and declared configurations, and what are the main contributors to this discrepancy? With a decomposed understanding of memory from **RQ1**, the gap between declared and actual memory usage can be studied and quantified more confidently. Measuring and comparing these values reliably across varying workloads and configurations given the time constraints will present a challenge, which is why a reliable framework is necessary as explained in **P4**. By answering this question, a quantified, attributed picture of the discrepancy between declarations and reality regarding memory usage will be produced, addressing **P2**.

Research Question 3 (RQ3):

Based on the observed discrepancies, what optimization strategies can be suggested to safely reduce unnecessary memory reservations in Kubernetes-managed containerized workloads? A good understanding of a workload's memory decomposition (**RQ1**), and the gap between its declared and actual usage (**RQ2**) is required. However, these insights need to be translated into applicable, generalized strategies that can increase efficiency. Balancing memory efficiency and workload safety across different workload types is a key challenge this question sets out to address. Answering this question will produce concrete, data-driven configuration recommendations.

1.3 Research Methodology

The workload under study will be Ocean: an AI-powered web crawler developed by Capisoft. Ocean helps fight counterfeiting by constantly crawling e-commerce websites

and social media platforms, looking for listings containing counterfeit items and general violations of intellectual property rights (14, 15). We chose Ocean because it nicely represents a real-world workload that continuously runs in production, and gives us a diverse and variable memory trace, making it suitable for analysis. Throughout all experiments, memory configurations are explicitly defined to ensure controlled results.

RQ1 methodology:

Since we need a detailed breakdown of memory usage that standard monitoring tools report only as a single value, we measure and decompose memory at the cgroup v2 level from inside pods: `memory.current` (total) and `memory.stat` (anon, file, kernel, slab). However, we cross-validate using additional tools, such as `kubectl top`, as different tools account for different memory components, and the comparison between them is itself informative. A custom monitoring tool, MemScope, developed as part of this work, is used to capture time-series data continuously, allowing us to analyze component breakdown over time to identify the main contributors to container memory usage.

RQ2 methodology:

To investigate the gap between declared and actual memory usage, we explicitly set and systematically vary requests and limits across experiments. Using MemScope, we compare observed cgroup metrics against declared values to quantify the discrepancy. Along with a varying workload intensity, this allows us to monitor how the discrepancy evolves under different load conditions. We can then attribute the discrepancy to specific memory components identified during the **RQ1** phase.

RQ3 methodology:

We analyze the data collected during **RQ1** and **RQ2** to find patterns of repeated inefficiencies, to then translate these findings into guidelines that are applicable in practice. The guidelines must balance two important aspects: workload safety and memory efficiency. Finally, we evaluate how the guidance holds across different workload configurations to assess its generalizability.

1.4 Thesis Contributions

By answering the research questions, we produce several contributions split into empirical, conceptual, and technical.

Empirical

(i) **C1:** We provide a decomposition of container memory usage in Kubernetes-managed environments across cgroup components. The decomposition is obtained through targeted experiments and measurement of a real-world production workload, providing a foundation for analyzing memory behavior in containerized systems.

(ii) **C2:** We provide a quantitative analysis of the gap between declared memory configurations and observed runtime memory usage. Furthermore, we attribute the gap to specific memory components to get a better grasp of where and why misconfiguration occurs.

Conceptual

(iii) **C3:** We propose practical configuration recommendations for memory requests and limits in Kubernetes-managed containerized deployments. The recommendations are based on the findings and aim to minimize superfluous memory reservations, while ensuring workload safety.

Technical

(i) **C4:** We provide an implementation of MemScope, a lightweight cgroup-level memory monitoring tool for Kubernetes pods. Through direct access to cgroup v2 metrics, MemScope produces structured time-series data that enables continuous, fine-grained memory analysis at the component level.

The tool delivered through C4 is aimed at filling the tooling gap that P4 describes, and which the other three contributions depend on. Chapters 3 and 4 discuss how we deliver it: Chapter 3 designs the tool, and Chapter 4 realizes it as MemScope. Chapter 5 then applies MemScope to Ocean and delivers C1 in Section 5.3, C2 in Section 5.4, and C3 in Section 5.5, each answering a corresponding research question.

1.5 Plagiarism Declaration

I hereby confirm that the contents of this thesis are a product of my own independent work and writing. The work does not contain material copied from any other source (person, internet, or LLM) unless otherwise indicated, and has not been submitted for assessment elsewhere.

1.6 Thesis Structure

Chapter 2 presents the background information needed for the rest of this thesis and the motivation behind our research. To best understand the work, it is recommended to start traversing the thesis there. Chapters 3 and 4 discuss the details of designing and implementing the missing instrument: Chapter 3 designs the experimental pipeline conceptually; Chapter 4 realizes it as MemScope. Chapter 5 applies MemScope to Ocean through a set of experiments and answers all research questions. Chapter 6 discusses related work and positions our thesis relative to it. Chapter 7 concludes by stating the direct answers to all research questions and our contributions, and discusses future work. Finally, Appendix A documents the artifact for reproduction; Appendix B is the self reflection.

For those with little time to spare, each chapter begins and ends with a summary of the most important findings, which should be sufficient to get a high-level overview. For those with even less time to spare, Chapter 7 summarizes the complete work in a few pages.

2

Background & Motivation

This chapter introduces the concepts necessary to understand the research presented in this thesis. We begin with an introduction to containerization and container orchestration, before discussing how Kubernetes manages memory resources. We then examine how memory is enforced and measured at the operating system level, review existing monitoring tools and their limitations, and briefly discuss the Vertical Pod Autoscaler.

2.1 Virtual Machines & Containerization

Deploying applications across different machines was a persistent challenge before the rise of containerization. Applications built and tested in one environment would often fail in another, as they would be reliant on operating system versions, installed libraries, or system configurations. Virtual machines (VMs) allowed multiple isolated environments to run on a single physical host (16), addressing the environment mismatch issue. Each VM runs its own operating system on top of a software layer called a hypervisor, which sits between the physical hardware and the VMs and is responsible for dividing and allocating the host's resources among them (17). The hypervisor allows different VMs on the same host to run entirely different operating systems and software stacks without interfering with each other.

However, virtual machine based deployments were slow to provision, hard to package, and prone to environment mismatch. These issues lead to the increased adoption of containerization (18). Now the de facto standard method of deploying software (3), containerization involves the packaging of software code to create a single lightweight executable, called a container, that runs consistently on any infrastructure (19). Each container can be configured with resource reservations, such as CPU and memory, specifying how much of the host's resources it is guaranteed and how much it is allowed to consume.

Several properties of containers made them a better alternative to virtual machines, contributing to the rise of their adoption. Containers do not run their own OS, they share the host's OS and only package the application and its dependencies, making them lighter

than VMs, faster to start, and more portable (20). This allowed containers to realise the notion of *write once, run anywhere* (19), where an application packaged as a container runs consistently regardless of the underlying infrastructure.

2.2 Container Orchestration with Kubernetes

When containerization first appeared, workloads were smaller and simpler, and therefore small numbers of containers were sufficient for running applications. As internet-scale and AI-driven applications have grown, container adoption has accelerated; survey data shows that 85% of respondents say AI is pushing container adoption, 87% expect container use to rise, and telemetry data shows GPU-based container workloads growing 58% year over year (21, 22). While managing a small number of containers was fairly straightforward, larger deployments brought new challenges: routing traffic, allocating resources, and deciding which tasks each container handles. This led to the development of container orchestration platforms, most notable of which is Kubernetes (6), to assist with solving these management issues.

2.2.1 Introduction to Kubernetes

Kubernetes is a container orchestration platform developed by Google and released in 2014 that provides solutions to the issues brought by modern container scale (6). Kubernetes defines an architecture involving clusters that contain nodes, that contain pods (23). A pod is the smallest deployable unit in Kubernetes and is where the actual application containers live, it usually contains one container but can in practice host more (7). Kubernetes also handles how an application should be deployed across this architecture and uses a scheduler to decide which node each pod is placed on, based on available resources and the pod's requirements (7).

Kubernetes is the layer that enforces what resources each pod gets, and what should happen to a pod when it violates those constraints (24)[1]. It maintains a QoS ranking that allows it to decide which containers get to stay alive when one needs to be terminated due to resource constraints (25).

Kubernetes also handles how many containers are needed to sustain an application's traffic load and how that changes dynamically in real time through horizontal pod autoscaling (26). It provides standard configuration and interfaces like kubectl to allow deployers to

[1]Kubernetes documentation, Resource Management for Pods and Containers: `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/`

communicate with its API, and houses many more features like VPA that provide different solutions to varying deployment problems (7).

2.2.2 Kubernetes Cluster Architecture

To give a clearer picture of the architecture Kubernetes uses, Figure 2.1 illustrates the main components of a Kubernetes cluster. At the highest level is the cluster itself, which consists of a control plane and one or more worker nodes (24)[1].

Within the cluster are nodes, which may be physical or virtual machines and provide the compute resources for running workloads (24)[2]. Each node runs the kubelet, an agent that ensures the containers on that node are running as expected (24).

Each node contains pods, the smallest deployable units in Kubernetes, each containing one or more containers that share storage and network resources (24)[3]. Kubernetes can terminate and replace pods at any time, particularly when they violate resource constraints (24). Pods from various applications can run in parallel on one node, but they would be competing for the same memory and CPU resources. Therefore, for pods to operate in a stable manner, it is crucial to make accurate resource configurations (7).

2.3 Memory Management in Kubernetes

Now that we have a clear picture of the Kubernetes architecture, we can explore further into the central topic of this work: memory management. Memory differs from CPU in that it is a non-compressible resource, meaning that unlike CPU, whose usage can be throttled (slowed down) when a container violates its constraints, memory cannot be gradually reclaimed. The operating system can only kill a container when it surpasses its memory limit. This section covers how Kubernetes manages pod memory through requests and limits, how it classifies pods into Quality of Service (QoS) classes based on their resource configuration, and what happens when a pod violates its memory constraints and is Out-of-Memory (OOM) killed.

2.3.1 Requests and Limits

The method Kubernetes employs to manage a container's memory is simple: each container gets a configuration file which contains, among other things, two main values regarding

[1]Kubernetes documentation, Kubernetes Components: `https://kubernetes.io/docs/concepts/overview/components/`

[2]Kubernetes documentation, Nodes: `https://kubernetes.io/docs/concepts/architecture/nodes/`

[3]Kubernetes documentation, Pods: `https://kubernetes.io/docs/concepts/workloads/pods/`

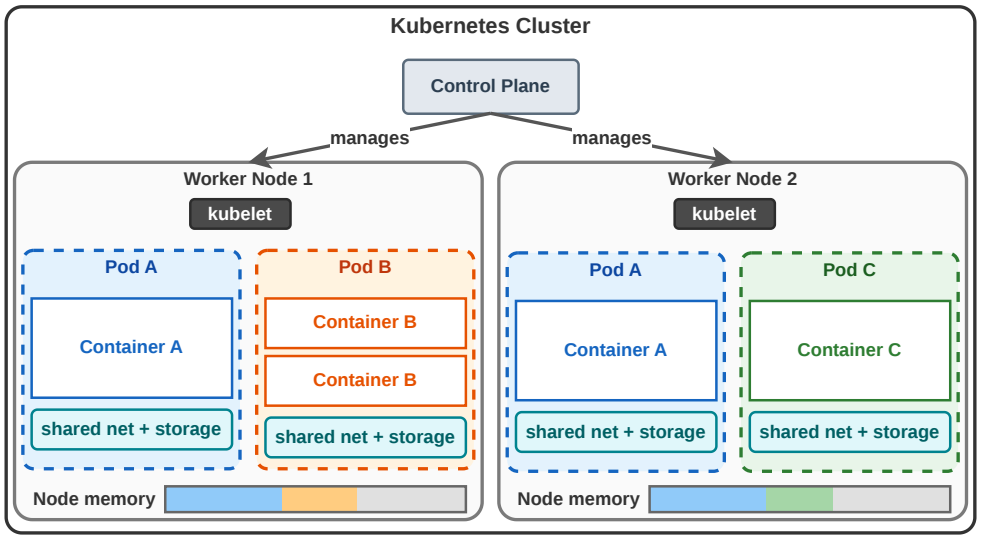

Figure 2.1: Hierarchical view of a Kubernetes cluster. The control plane manages each node through its kubelet; pods group one or more containers that share network and storage; the node memory bar shows the share of node memory consumed by pods, with the remainder available for new pods.

memory; a request, representing the minimum amount of memory that will be allocated to the container, and a limit, representing the maximum amount of memory that the container can use (24). These values are set by operators, and are dependent on the nature of the deployed application among other factors such as safety margins.

Another important actor is the scheduler, which decides which node a newly created pod will be deployed on (7). The scheduler uses requests, not limits, to make this decision, if a node does not have enough allocatable memory to satisfy a pod's request, the pod will not be scheduled there. If no node in the cluster can satisfy the request, the pod remains unscheduled until resources become available (24)[1].

Of course, the usage of a container rarely matches the request set in its configuration. Containers may use more memory than indicated in the request. The case where the amount of memory used exceeds the request but remains below the limit is referred to as bursting (24). Kubernetes allows bursting to happen, but it is important to note that it affects other pods on the same node, since the pod is consuming more memory than it was scheduled for, leaving less available to its neighbors.

When no requests or limits are set, the container can consume as much memory as the

[1]Kubernetes documentation, Kubernetes Scheduler: `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/`

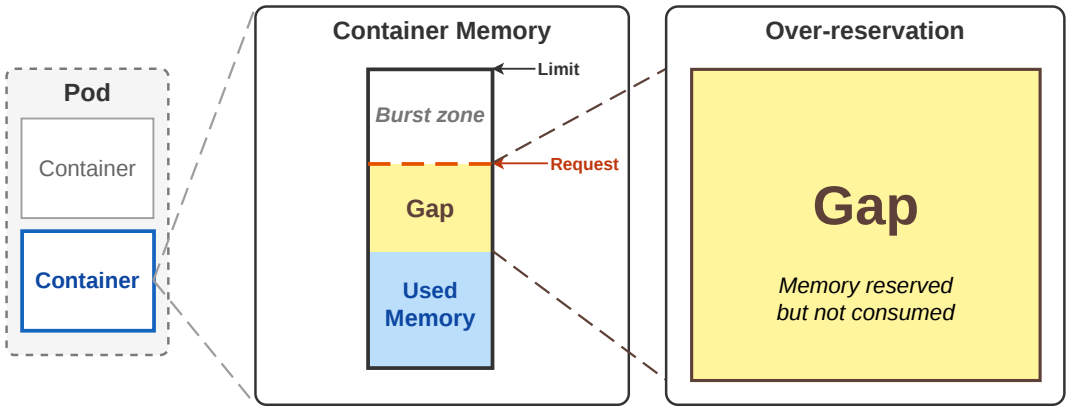

Figure 2.2: Per-container memory allocation. The request reserves a minimum amount of memory for the container while actual usage typically falls below it, leaving a gap of memory that is reserved but never consumed.

node has available, and will be the first to be evicted when the node runs low on memory, as discussed further in Section 2.3.2 (24).

A main problem arises from this, depicted in Figure 2.2. While declared memory could in principle reflect actual runtime usage, it does not match in most scenarios for two reasons. First, memory consumption consists of multiple complex components such as application memory, container overhead, and OS-level usage, that change depending on runtime effects that are difficult to predict (25). Second, because of this complexity, operators typically set declarations conservatively, adding a safety margin to avoid out-of-memory events (8). The result is consistent over-reservation of memory, leaving large amounts of allocated memory unused across the cluster.

2.3.2 Quality of Service and Out-of-Memory Behavior

Kubernetes automatically assigns every pod a Quality of Service (QoS) class based on how its memory requests and limits are configured (24)[1]. These classes are used by Kubernetes to decide which pods to evict first when a node runs low on memory.

There are three QoS classes:

- **Guaranteed**: the pod has both requests and limits set, and they are equal. These pods are treated as highest priority and are the last to be evicted.

[1]Kubernetes documentation, Pod Quality of Service Classes: `https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/`

- **Burstable**: the pod has requests set, but limits are either not set or higher than the request. These pods can consume more memory than their request up to their limit, and are evicted after BestEffort pods.

- **BestEffort**: the pod has no requests or limits set at all. Kubernetes gives these pods whatever resources are left over, and they are the first to be evicted when the node is under memory pressure.

When a container exceeds its memory limit, it is not Kubernetes that terminates it but the Linux kernel, through a mechanism called an Out-of-Memory (OOM) kill (24). Kubernetes then observes that the container has been terminated and decides whether to restart it based on the pod's restart policy. Eviction, on the other hand, is a proactive measure taken by Kubernetes when a node is running low on memory, removing pods before any limit is exceeded, in the order determined by their QoS class (24).

2.4 Linux cgroups and Memory Accounting

Kubernetes describes memory at a high level but does not actually enforce limits or measure memory itself (24). Limits are enforced at the OS layer, in the Linux kernel. The kernel does this through a feature called control groups, or cgroups, that organizes processes into hierarchical groups to limit, account for, and isolate their resource usage, including CPU, disk I/O, network bandwidth, and most notably, memory (27). Figure 2.3 shows this hierarchy for a single container on a node. The feature is relevant for this work because cgroups are the source of memory numbers reported by the monitoring tools covered later. Furthermore, the kernel exposes an individual view of the same memory for each running process at `/proc/<pid>/smaps`, while `cgroup.procs` lists the processes belonging to the container. A page shared by several processes is fully included in the counts, however, the Proportional Set Size (PSS) field divides that page by the number of processes sharing it (28). Taking the sum of the PSS for all processes therefore only counts the shared pages once.

Moreover, cgroups are also an integral part of what containers actually are. Containers are not a primitive in the Linux kernel; they are an abstraction of two kernel features working together: cgroups, which handle resource control and accounting, and namespaces, which isolate global system resources (such as process IDs, network stacks, and mount points) so that different sets of processes are tricked into thinking that they have their own

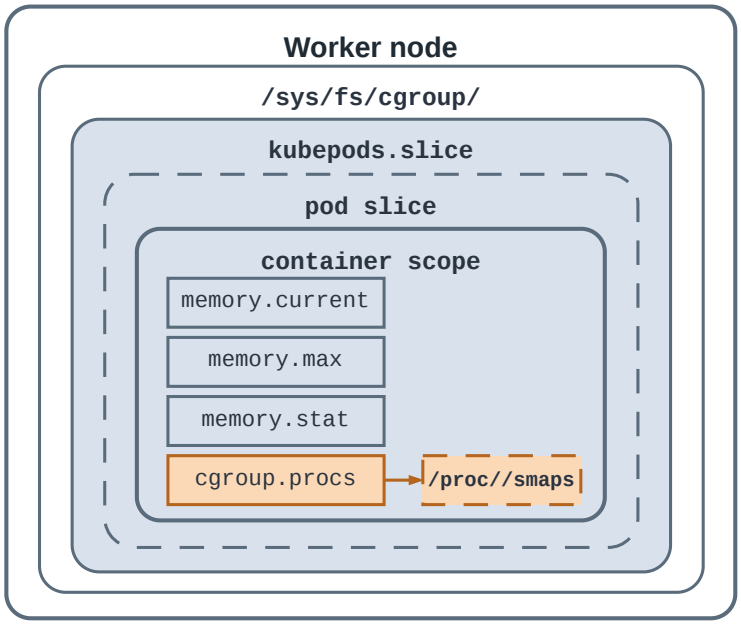

Figure 2.3: The cgroup hierarchy for a single container, starting from the node down to the container scope. The leaf holds the files that expose the container's memory: `memory.current`, `memory.max`, and `memory.stat`, alongside `cgroup.procs`.

independent instances of those resources (29). This work focuses on the cgroups side of the container, since memory is the resource of interest.

2.5 Memory Monitoring in Kubernetes

Several tools are already in place to observe and monitor container memory (7). Operators typically use these tools, as they provide a simpler presentation of memory usage avoiding all of the complexity housed in the cgroup files. A handful of tools are commonly used when monitoring memory in Kubernetes deployments; those include cAdvisor (30), Prometheus (31), Kubernetes Metric Service (32), and Node Exporter (33). Each of these tools report on a different level in the Kubernetes stack, e.g. cAdvisor reports on container level, while Node Exporter reports on the node level. However, they all ultimately read their data from the cgroup files, the differences are in how they expose that data, not where it comes from (25, 27).

Regardless of scope, each tool reports a single aggregated number for memory used, not a breakdown. This raises a limitation: while these tools provide enough detail for an operator's need, they do not really tell us where memory goes exactly. To achieve a

breakdown in such detail, a custom monitoring tool would be required to read and present data from the cgroup files as is.

2.5.1 What the Metrics Actually Measure

The tools mentioned earlier report memory usage in bytes, but bytes are not the unit the kernel itself uses to track memory. It manages memory in fixed-size chunks called pages, typically 4 KiB on x86_64 systems, and the cgroup memory controller accounts for memory at this same level of detail (27). Pages are the natural unit of accounting as they are also the unit at which the kernel allocates physical memory, maps virtual addresses, and reclaims memory under pressure (20, 25). When a process inside a container needs memory, the kernel allocates one or more pages and charges them to that process's cgroup.

The single number that tools like cAdvisor or kubectl top report is the sum of several categories the cgroup controller tracks (27). They are: anonymous memory used by the application heap and stack, file-backed memory cached by the kernel after disk reads, and kernel memory allocated for data structures maintained on behalf of the cgroup. Each category behaves differently under varying workloads, and so is tracked separately. This is the part of the picture this thesis cares about: where the headline number comes from.

2.6 Vertical Pod Autoscaler

The gap between declared requests and limits and actual usage is already targeted by a built-in Kubernetes component, the Vertical Pod Autoscaler (VPA). It monitors each container's memory and CPU usage in real-time and uses it to calculate a suitable request/limit, which it either actively applies to the pod or passively recommends depending on the deployment's configuration (34).

While VPA does pose a potential solution to the main problem this work discusses, it still has its weaknesses. VPA's recommendations are derived from the same kind of aggregated memory metrics discussed in Section 2.5, so it inherits their blind spots. It narrows the gap based on historical usage, but does not address why the gap exists in the first place. Furthermore, it is not widely deployed in practice due to the extra explicit setup required, and its incompatibility with existing systems (22). Understanding the composition behind the discrepancy is exactly the input that VPA depends on but does not itself produce.

3

Design of an Experimental Pipeline for Container Memory Analysis in Kubernetes

Contribution 4.1 (C4.1):
Analysis of the missing infrastructure for fine-grained memory analysis in Kubernetes, and a design process for an experimental pipeline.

Contribution 4.2 (C4.2):
Functional and non-functional requirements for the experimental pipeline.

Contribution 4.3 (C4.3):
A conceptual design for the experimental pipeline based on direct cgroup-level memory measurement.

The analysis this thesis aims to deliver requires a measurement pipeline beyond what existing Kubernetes tools provide. As discussed in previous sections, there is a noticeable gap between declared and actual memory usage when it comes to containers in Kubernetes environments. Existing Kubernetes memory monitoring tools only expose aggregate memory numbers, and no existing infrastructure provides the per-component analysis needed for the rest of this thesis.

This chapter presents the design of an experimental pipeline that fills this gap, which will serve as the methodology that allows us to address all research questions proposed in Section 1.2. Only the conceptual design, along with its motivation, alternative options, and distinguishing factors, will be discussed in this chapter; the realization is presented in Chapter 4. Section 3.1 begins by analyzing the gap and the available solution space in more detail before deriving the requirements.

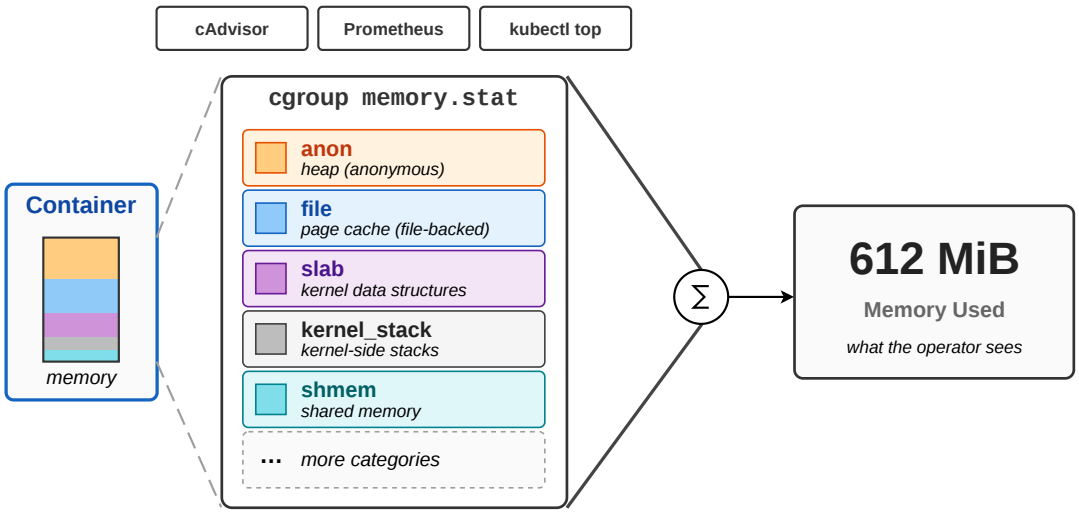

Figure 3.1: Loss of per-category granularity along the Kubernetes monitoring pipeline. The breakdown exposed by cgroup `memory.stat` is summed across categories by cAdvisor, Prometheus, and `kubectl top` into a single value.

3.1 Design Process

We begin by discussing how the gap highlighted by problem statement **P4** along with a survey of exisiting approaches lead to the design idea behind the pipeline. The output of this section is the design direction, which in turn drives the functional and non-functional requirements derived in Section 3.2.

3.1.1 Analysis of the Gap

The main limitation of available Kubernetes memory monitoring tools is the granularity at which they capture memory. As shown in Figure 3.1, all of these tools, whether Kubernetes native or not, report container memory as a single aggregate (30, 31, 32). Container memory is composed of many different categories: anonymous memory (true heap pressure), file-backed memory (page cache, reclaimable), slab and kernel (OS overhead), and many more that without separating, you cannot reason about what the application is actually consuming (9, 27). Moreover, since tools only show an aggregate, the actual state of the container is hidden: much of the memory reported as used may in fact be reclaimable (27). Without the breakdown, request and limit settings are made against the aggregate, which confuses safely reclaimable memory with genuine pressure (7, 24).

The breakdown that Kubernetes monitoring tools do not surface is not missing; the cgroup exposes it directly through the `memory.stat` file (27). All memory monitoring

tools ultimately read this data from cgroup (30), but collapse it into a single aggregate before presenting it. Current container monitoring frameworks do improve on other aspects such as portability and extensibility, yet they still only expose a single value (35). For applications like autoscaling, alerting, and default dashboards this single value is sufficient. A pipeline with the objective of providing a categorical memory measurement does not require new OS-level measurement, only direct access to cgroup accounts.

A point not covered yet is the form of data that monitoring tools produce. Granularity also has a temporal aspect to it, as container memory is not stagnant, but evolves continuously under different factors like load and usage constraints. In order to capture this evolution, we would need detailed time-series data at a high resolution, kept in its raw form rather than aggregated. Existing tools fail to achieve this: they sample at coarse intervals and aggregate over time windows before storage (31). As a consequence, the dynamic behavior most important for understanding memory evolution fades away before it reaches the researcher (36).

Beyond what tools capture, there is a limit placed on what the data can tell us through observation only. That is because memory behavior as seen in production reflects whatever load happens to be present. Without controlling that load, memory patterns can be correlated with certain conditions but not attributed to specific causes. The workload used must be deliberately driven under explicit, reproducible conditions (37). This implies that our pipeline's scope should also include orchestrated load generation, not just memory monitoring.

3.1.2 Analysis of Solutions

The details of the gap analyzed in Section 3.1.1 can be looked at as two coupled parts: what to measure and how to deploy the measurement. We must look at possible measurement approaches that capture the per-component breakdown we need, and a deployment approach that can provide controlled and reproducible workloads without affecting production workloads or requiring cluster-admin privileges. These two are coupled but separable; an option can fit one but fail the other. This section explores candidate approaches against both criteria.

The first approach that comes to mind is to simply extend existing monitoring tools. In principle, cAdvisor or Prometheus exporters could be extended to expose more of `memory.stat` (30, 31). While it would reduce the effort needed to achieve a granular breakdown, this approach requires modifying components built into the kubelet, which requires cluster-admin privileges that are often impossible in shared or managed clusters

like EKS. Moreover, these tools were designed around aggregation; their data path is not built for structured experimental output. This approach would then not be suitable given the constraints.

A second candidate could benefit from the use of a sidecar: a secondary container that runs inside the same pod as the main application, sharing the pod's namespaces (24)[1]. The sidecar could monitor and read the cgroup data from inside the pod and ship it out. Sidecars do provide a continuous monitoring method, with low running effort, but they would require modifying the deployment spec of every pod under observation, which is invasive. Also, sidecars would be impractical for situations involving production workloads that you cannot redeploy. Given that we would like to be as uninvasive as possible, this solution does not suit our case.

Instead of a sidecar, we can similarly use a DaemonSet: a Kubernetes workload that ensures one pod runs on every node, commonly used for node-level services like log collectors and monitoring agents (24)[2]. Like the sidecar approach, a DaemonSet could be used to read the cgroup for every container on its node, without touching the containers themselves. However, similar to a sidecar, deploying a DaemonSet requires cluster-admin privileges and setting up permanent infrastructure, making it too invasive an approach.

The last option would be to directly access the cgroup via `kubectl exec`. This tool allows us to run a command inside an existing pod's container, and is made available through the standard Kubernetes API (38). This option needs to invoke a new process for each sample it collects, resulting in a slightly higher overhead, and would require basic shell utilities in the container. Nevertheless, no cluster-level changes are required, and no privileges beyond what a regular namespace user already has, making it as uninvasive as possible. Furthermore, it would work in any environment that supports kubectl. For our needs, this approach would be the most suitable, since access is limited but the target workload is known.

On the other hand, workload generation has two different needs: it must support isolating individual memory categories for targeted experiments, and use application code paths to mimic real-world usage. These needs are met by using two types of stimuli: synthetic and real. Synthetic stimuli would isolate one cgroup memory category at a time; this is done through large bytearray allocations, page cache via large file reads, kernel memory

[1]Kubernetes documentation, Sidecar Containers: `https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/`

[2]Kubernetes documentation, DaemonSet: `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/`

via socket allocations. Real stimuli would drive pressure along code paths the application is very likely to take in production. Both categories use kubectl exec, the same mechanism chosen for measurement. Each stimulus output timestamped markers that can be used to align memory traces with certain stimuli, so observed memory shifts can be attributed to specific stimuli.

Both memory monitoring and workload generation share the same mechanism of operation: `kubectl exec`. For memory monitoring, the cgroup memory statistics files are read directly from within the pod, providing the raw breakdown as the cgroup reports it. On the other hand, the workload generator pushes stimuli into the same pod, and we can observe the change in behavior through the monitor. Both parts combined, the product is a broken down time-series memory log that was produced under controlled conditions, filling the gap mentioned in Section 3.1.1. Section 3.2 turns the design direction into functional and non-functional requirements.

3.2 Requirements

Following from Section 3.1, we will now elicit explicit requirements that the system should meet in order to address the gap identified. The requirements will be split into two categories:

- **Functional requirements** (FR): the concrete capabilities the pipeline must provide.
- **Non-functional requirements** (NFR): the qualitative properties the pipeline must exhibit, such as low overhead, low intrusiveness, and reproducibility.

Requirements are labeled **FR-N** for functional and **NFR-N** for non-functional, where N is a numeric index. Each requirement is stated in a few sentences and traced back to the design process from Section 3.1.

3.2.1 Functional Requirements

The functional requirements describe the specific operations the pipeline must perform. Each follows from the analysis in Section 3.1.

- **FR-1: Memory metric capture.** The pipeline must capture, for each monitored container, the aggregate memory usage (`memory.current`), the configured memory limit (`memory.max`), the categorical breakdown exposed by the cgroup's `memory.stat`, including anonymous memory, file-backed memory, slab, kernel memory, and the

kernel's cumulative counters, and for the dominant component, a process-level attribution that counts pages shared between processes correctly. This follows from the granularity argument in Section 3.1.1: aggregate values alone are not sufficient for the per-component analysis this thesis aims to deliver, and the limit must be captured alongside the breakdown so that declared and actual usage can be compared.

- **FR-2: Raw time-series output at configurable resolution.** The metrics from FR-1 must be sampled at an interval decided by the operator, and every sample recorded to a structured output file in its raw format. This follows from the temporal-granularity argument in Section 3.1.1: existing tools sample at high intervals and aggregate over time, hiding the behavior needed for the analysis.

- **FR-3: Controlled stimulus generation.** The pipeline must support driving controlled memory stimuli into target pods, covering both synthetic operations that isolate single cgroup memory categories (for validation of the decomposition logic) and application-driven operations that exercise realistic code paths in the workload under study. This follows from the experimental dimension of the gap analysis in Section 3.1.1 and from the workload generation analysis in Section 3.1.2.

- **FR-4: Stimulus-measurement alignment.** The pipeline must record the start and end of each memory stimulus to ensure that it can be matched against the corresponding memory log. This would allow every change observed through the monitor to be correlated to certain stimuli throughout experiments. Otherwise, the output would simply be memory over time, with no method to place any meaning behind changes in memory usage.

- **FR-5: Per-run metadata capture.** The pipeline must record sufficient metadata for each run to allow reproducing and analyzing it offline. Metadata can include namespace, target deployments, stimulus parameters, sampling interval, and workload configuration. This addresses the experimental angle of the gap analysis in Section 3.1.1. Experiments that cannot be reproduced cannot support the comparative analysis we aim to produce.

3.2.2 Non-functional Requirements

The non-functional requirements describe the qualitative properties the pipeline must exhibit. Each follows from the analysis in Section 3.1.

- **NFR-1: Low intrusion & Lightweight dependencies.** The pipeline must not modify the production workloads it observes, deploy extra cluster-level components, or require privileges beyond a regular namespace user and require minimal dependencies inside the target pods, since the experimenter typically does not own them. Following what was argued in Section 3.1.2 about the invasive-ness of the chosen direction, an unintrusive pipeline would allow for it to be used in most clusters, and by any user. Only basic shell utilities are assumed to be present in the container under study, with no need for any deployment-spec changes or special and unsual permissions.

- **NFR-2: Low overhead.** The measurement and stimulus invocations must add minimal overhead to the system under study, so that the act of measuring does not significantly affect what is being measured. The pipeline uses kubectl exec for both measurement and workload generation, an approach acknowledged in Section 3.1.2 to carry non-trivial per-sample cost. If this cost grows in comparison to the workload's own memory pressure, the integrity of the observations themselves becomes questionable, and the pipeline is less of a reliable instrument.

- **NFR-3: Reproducibility.** The same experimental conditions must produce comparable results across runs, so findings can be confirmed and built on. Findings from individual runs only become reliable when they can be confirmed by repeated runs under the same conditions. Without reproducibility, the pipeline produces snapshots rather than evidence, and claims about made memory behavior cannot be subjected to the kind of analysis the research questions require.

- **NFR-4: Portability.** The pipeline must run in any environment that supports kubectl, without being tied to a specific cluster distribution (EKS, GKE, on-prem) or configuration. The chosen approach in Section 3.1.2 uses only what the standard Kubernetes API exposes, so the pipeline does not assume any particular cluster distribution. Keeping this property explicit in the requirements protects against future implementation choices that would tie the tooling to any environment, and supports reuse of the methodology.

3.3 Design Overview

At the highest level, the pipeline is composed of three main components: a memory monitor, a workload driver, and an analysis layer. Figure 3.2 visualizes these main components.

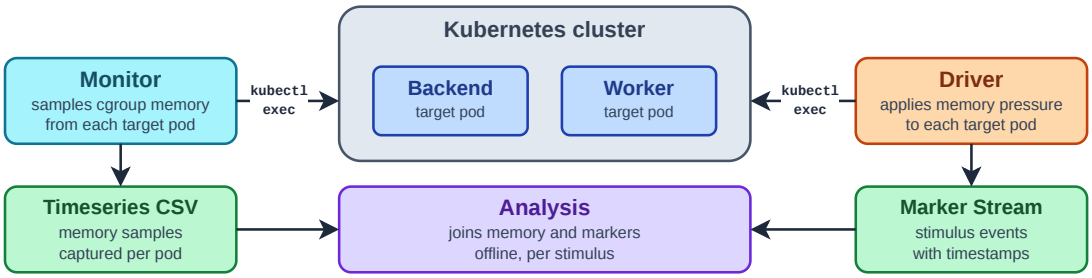

Figure 3.2: High-level overview of the experimental pipeline.

Both the monitor and driver interact with the same target pods via `kubectl exec`, and run in parallel during experimental runs. Timestamped outputs are produced by both; the monitor outputs a CSV time-series of memory metrics, and the driver outputs markers flagging stimulus start and end. The analysis layer joins the two offline through Jupyter notebooks designed to produce visuals from the CSV files.

The three components visualized in Figure 3.2 all perform coupled yet separate tasks (39). The memory monitor periodically samples memory metrics from each target pod via `kubectl exec`, recording the full `memory.stat` breakdown, as well as `memory.current`, `memory.max`, and the processes' weighted anonymous memory breakdown. The driver dispatches stimuli into the target pods, using both synthetic operations to allow for category isolation, and real application operations for realistic code paths. Finally, the analysis layer reads the CSV and marker timestamps, slices the time-series data into per-stimulus blocks, and produces detailed summaries against an idle baseline.

Connecting back to Section 3.2, we can see how this design realizes the mentioned requirements. The memory monitor satisfies FR-1 (memory metric capture) and FR-2 (raw time-series output), as it captures memory metrics and gives us the time-series output we are looking for. The driver satisfies FR-3 (controlled stimuli) and FR-4 (stimulus timing). The kubectl exec mechanism is what gives the design NFR-1 (low intrusion and lightweight dependencies) and NFR-4 (portability). The analysis layer and the structured output achieve FR-5 (per-run metadata capture) and support NFR-3 (reproducibility). NFR-2 (low overhead) is addressed by choosing a low and configurable sampling interval rather than high frequency polling.

Section 3.4 covers each component of the design in more detail.

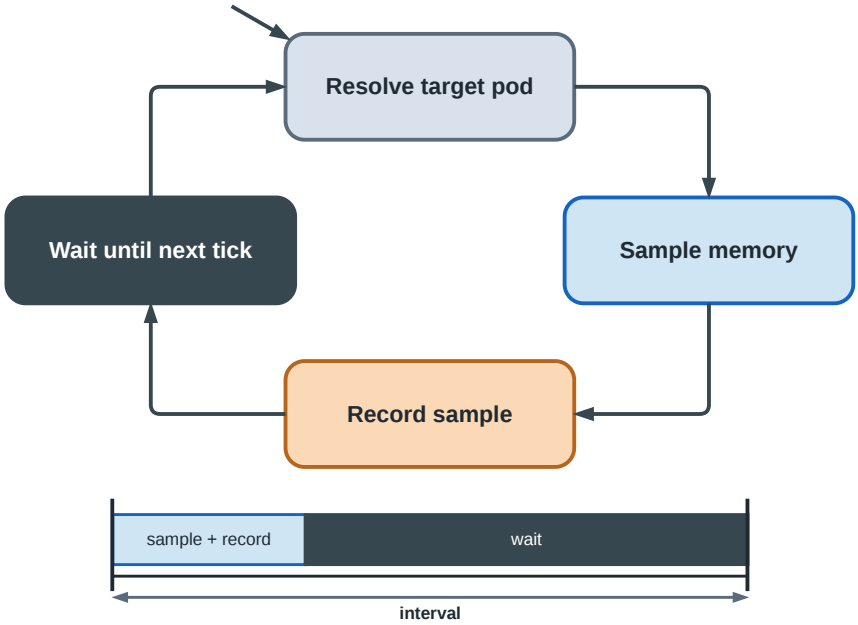

Figure 3.3: The design of the memory monitor as a fixed-period sampling loop. Each cycle resolves the target pod, samples its cgroup memory, and records one row. The wait then fills the remainder of the interval, so a longer sample shortens the wait and the period between samples stays constant.

3.4 Design Details

Following the overview in Section 3.3, this section dives deeper into each of the three components: the memory monitor (Section 3.4.1), the workload driver (Section 3.4.2), and the analysis flow (Section 3.4.3). Each subsection details the component's design choices and traces them back to the requirements from Section 3.2.

3.4.1 Memory Measurement via Direct cgroup Access

The main component of the design is the memory monitor, whose mechanism is to simply call `kubectl exec` periodically and execute a small shell script inside the target pod (38). The script would read `memory.max`, `memory.current`, and `memory.stat` from inside the pod's cgroup files (27), parse, and then print them to stdout. The script also iterates through the processes in the container's cgroup and reads each one's memory mapping file, weighing each mmap by its proportional share so pages shared between processes are only counted once. On the operator side, the output would be converted to one record per sample. This realizes the design direction set in Section 3.1.2 and satisfies FR-1

The operator would choose the interval at which the sampling of memory statistics would run, with fixed-period scheduling that would account for the varying response times of `kubectl` calls. Each cycle subtracts the time the `kubectl` calls took from the sleep, such that frequency stays constant. Figure 3.3 shows this loop. A single CSV row is created for each sample, carrying a timestamp, sample index, and other metadata. The output CSV file is saved after appending every row, so that an unexpected interruption would not cause the loss of the entire log.

Before every iteration, the pods being monitored are re-resolved using the deployment name's prefix to safeguard against restarts, which would change a pod's name, preventing the monitor from being able to reach them. Furthermore, `kubectl` failures (timeouts, missing pods, parse errors) are absorbed into the output as an error sample rather than causing the monitor to abort, so that long runs are not thrown away if one `kubectl` call fails. The shell scripts sent to run in the target pod use minimal shell tools, minimizing the chance that a tool is missing from minimal container images, ensuring NFR-1 is satisfied.

3.4.2 Controlled Load Generation

The memory driver assists in our experiments by dispatching stimuli, reusing the same `kubectl exec` mechanism as the monitor. The driver dispatches two categories of stimuli: synthetic and application-driven. For synthetic stimuli, the command is a small Python or shell snippet executed inside the pod, and targets a specific component of memory. For application-driven stimuli, the command invokes existing application code paths that are likely to also be used in real production environments. Only one stimulus can be run at a time, allowing individual memory shifts to be cleanly attributed and satisfying FR-3.

The way that stimuli are made discoverable in the monitor's time-series is as follows. Each stimulus emits start and end marker lines with timestamps around its execution. These markers carry enough metadata to identify the stimulus (test name, parameters, target pod). The driver and monitor write to separate streams, but their timestamps share the same wall clock, so alignment between the two is simple. This satisfies FR-4, and is how observed memory behavior can later be related to specific stimuli.

3.4.3 Telemetry Collection and Analysis Flow

The analysis layer is what ties everything together, making meaning of whatever data the driver and monitor produce. It takes two inputs: the monitor's time-series and the driver's markers. Using the timestamps described in Section 3.4.2, samples are matched

and sliced into per-stimulus blocks. Each block is compared against an idle baseline drawn from the same run, captured before any stimulus is dispatched. The comparison results in categorical memory differences (anonymous, file-backed, slab, kernel), which is what makes it possible to say what each stimulus actually moved.

Alongside the time-series and marker outputs, each run is registered with a metadata record capturing the parameters that defined it: namespace, target deployments, stimulus parameters, sampling interval, workload configuration, and start/end timestamps. This metadata gives each run the context needed to interpret it correctly, and turns scattered runs into a comparable set, satisfying FR-5. Since the raw outputs and the metadata remain the source of truth, any run can be re-analyzed or repeated under the same conditions, reinforcing NFR-3.

3.5 Summary

This chapter began by explaining the limitations of current Kubernetes monitoring tool, prompting the need for a custom tool, as well as a controlled load generator. We then proceeded by surveying different approaches, weighing out their advantages and disadvantages, and choosing a suitable design direction. From this direction, functional and non-functional requirements were derived that our pipeline must satisfy. All of this leads to our final 3-component design consisting of: a memory monitor, a controlled load generator, and an analysis layer. The realization of this design is presented in Chapter 4.

3. DESIGN OF AN EXPERIMENTAL PIPELINE FOR CONTAINER MEMORY ANALYSIS IN KUBERNETES

4

Realization of the Experimental Pipeline (MemScope)

Contribution 4.4 (C4.4):
A working realization of the three-component pipeline introduced in Chapter 3.

Contribution 4.5 (C4.5):
Realization-level design decisions (scheduling, failure handling, marker alignment, analysis structure) that turn the conceptual design into a reliable measurement instrument.

Contribution 4.6 (C4.6):
`MemScope`, a portable Python codebase that operates under the constraints derived in Chapter 3 (no cluster-admin privileges, no deployment-spec changes).

The previous chapter established the conceptual pipeline we intend to work with as a design idea. A conceptual design alone is not sufficient to run experiments and produce measurements; we need software running against a real cluster. This chapter presents the realization of our conceptual design, named *MemScope*.

A number of choices were left open after Chapter 3, such as sampling cadence, failure handling, marker alignment, or output format; these are implementation-level decisions that will be discussed further in this chapter. The decisions taken will affect the reliability of the measurements produced by our pipeline, and therefore need to be adequately studied. The main functional requirement for this chapter is to realize the Chapter 3 design.

Section 4.1 analyzes the realization problem and the available options, Section 4.2 derives the realization-level requirements, Sections 4.3 and 4.4 present the overview and details of MemScope, and Section 4.5 closes the chapter and bridges into the evaluation.

4.1 Realization Process

This section discusses the choices left open by Chapter 3 and leads to a chosen implementation direction. Subsection 4.1.1 identifies which aspects of the design are too vague for

a working artifact, while Subsection 4.1.2 surveys different implementation options and selects one. After this section, we will have a clear implementation direction, which will drive the implementation-level requirements presented in Section 4.2.

4.1.1 Analysis of the Realization Problem

Chapter 3 produced a conceptual three-component design. This design tells us what to build, but does not cover *how* it should be built reliably in a real-world environment.

Firstly, Chapter 3 mentioned our memory monitor should run at a configurable interval. While that sounds simple, we need to take into account that invoking `kubectl exec` itself might take various amounts of time, each time it is invoked. This leaves us with the following problem: at what frequency to sample, and how to keep that frequency steady. The time-axis is the foundation of every analysis carried out on our data later on; a blurry time-axis hides dynamic behavior we want to capture and undermines NFR-2 (low overhead) if our frequency drifts under load.

Secondly, in Chapter 3 it was acknowledged that `kubectl exec` invocations can indeed fail, and that we do not want failures to render long-running experimental runs useless. What needs to be decided as part of our implementation is: which failures to absorb silently, which to surface, and how to record them in a way that does not corrupt the time-series. This matters because experimental runs pay back our setup cost; losing a long run is a much larger cost than the error that caused the loss itself.

Thirdly, the design mentions that the monitor and driver both share the same wall clock and that joining their outputs is therefore simple. During implementation, it is necessary to decide on which clock to use, what tolerance for skew is suitable, and how to encode markers so the join is indeed easy. Without reliable alignment, we would not be able to attribute any memory change to a specific stimulus, collapsing the experimental design back to a simple observation, and breaking FR-4.

Lastly, the design also discussed that a structured output is necessary for analysis. However, the exact shape of this output was never specified, leaving the decision up to our implementation. We need to explicitly finalize what fields to capture per row, how to handle missing values, and how to keep the format stable across runs so analysis code does not need to know which version produced a file. Our output file is the only artifact we have after an experimental run; whatever we do not store then is unrecoverable.

4.1.2 Analysis of Realization Options

The gaps discussed in Section 4.1.1 leave us with four independent decisions to make. Above all of them sits one main decision regarding the language and runtime chosen for implementation. This subsection discusses each, weighs possible options, and ends with a chosen direction.

The first and most general decision concerns language and runtime. A more basic option would be implementing our components as shell scripts (`bash`) since they would be very lightweight; it would however be difficult to parse `kubectl` output, and unreliable for steady scheduling, because keeping a steady frequency under changing `kubectl` latency requires subtracting elapsed time from the sleep budget, which `bash` does not give us easy solutions for. Go is another strong option, with the upside that it produces a single static binary that runs anywhere `kubectl` runs with no runtime dependency on the operator's environment, but it pulls in a compiled build toolchain and deployment incompatibility that would not be justified. We then arrive at a middle ground, Python: it is readable, has a rich library ecosystem, and integrates flawlessly with Jupyter notebooks, which we will use for analysis later on. Python is slower than Go on per-call overhead and process startup, but neither matters here, since the monitor is a single long-running process whose per-sample cost is dominated by `kubectl` latency rather than by Python itself.

Another undecided yet crucial problem is our sampling frequency. We can choose a sub-second frequency, which would give us very rich data but would result in high per-sample overhead from `kubectl`. On the contrary, a very high interval in the tens of seconds would lose us the dynamic behavior we hope to capture. Therefore, a moderate single-digit seconds interval is most suitable as it balances the trade-off. We default the interval to 3 seconds, but leave it configurable via the CLI. For managing the varying `kubectl` response times, we make our monitor sleep until the next tick using a monotonic clock, which would absorb the latency into the sleep budget.

For failure handling, we have three directions: abort on any error, silently swallow, absorb-and-record. The only option that fulfills all of our design requirements would be absorb-and-record, in the form of an extra error column that would flag corrupted samples. This produces two implementation-level details: unique error strings for the three failure modes (`kubectl exec` fail/timeout, JSON parse fail, no running pod) so the analyst can tell them apart; and the CSV schema is deferred until the first good sample so the stat column set reflects what `memory.stat` actually exposed. Through this, we achieve our goal of maintaining long experiment runs regardless of any faults.

We also have two options for how to relate memory behavior to stimuli. The first option would be our driver writing markers straight into the monitor's output, or at a higher level, simply isolating each experimental run. The latter option promotes decoupling of the monitor and driver, and reduces overall complexity, making it more suitable. We can simply start the monitor, dispatch one stimulus, stop the monitor, and finally register the resulting CSV against the stimulus that produced it.

The last decision to be made is our output format. Options include SQLite, JSON, or CSV. The most suitable option would be CSV, as it is simple, inspectable, portable, and natively consumed by pandas. Along with the memory behavior file, a second CSV, the run registry, holds one row per monitor run with workload, stimulus, request and limit configuration, and free-text notes for findings worth recording alongside the data.

In conclusion, our implementation proceeds in the following direction: Python with stdlib-only for the in-cluster-facing tools and pandas/matplotlib for offline analysis; fixed-period scheduling at a 3-second default; absorb-and-record failure handling with deferred schema; per-run isolation with a registry rather than in-stream markers; CSV time-series with stable schema plus a separate metadata registry. The requirements derived from this stack are presented in Section 4.2.

4.2 Requirements

Following from Section 4.1, this section presents the requirements our implementation must meet. The requirements here are an extended set of the ones presented in Section 3.2, adding implementation-specific details. Requirements are also split into two categories: functional and non-functional, exactly like in Section 3.2, but with different labels, **FR-RN** for functional and **NFR-RN** for non-functional, where N is a numeric index.. Each requirement is stated in a few sentences and traces back to a decision in Section 4.1.

4.2.1 Functional Requirements

The functional requirements describe the concrete capabilities the implementation must provide.

- **FR-R1: Realization of the Chapter 3 design.** The implementation shall realize the conceptual design given in Chapter 3 (FR-1 through FR-5 and NFR-1 through NFR-4) as working software. This is the main functional requirement in this chapter:

the conceptual pipeline from Chapter 3 now resurfaces as a functional requirement on the implementation-level.

- **FR-R2: Run-time configurability.** All experiment-affecting parameters (namespace, target deployments, sampling interval, output path, stimulus parameters) shall be configurable at run time via the command-line interface. This follows from the sampling-interval decision in Section 4.1.2, generalized as a principle: every parameter the operator changes between runs should be reachable without having to edit the code.

- **FR-R3: Steady sampling frequency.** The monitor shall preserve a steady frequency at the configured interval by sleeping until the next tick on a monotonic clock, absorbing `kubectl` latency into the sleep budget. This follows from the scheduling decision in Section 4.1.2: a drifting frequency blurs the time axis which our analysis depends on, and partially violates NFR-2.

- **FR-R4: Temporary fault absorption.** The implementation shall absorb temporary `kubectl` failures (timeouts, missing pods, JSON parse errors) without aborting an experimental run, recording each failure as a marked sample with an insightful error string. This follows from the failure-handling decision in Section 4.1.2: a long experimental run amortizes setup cost, and a single fault is much less costly than the run it would destroy.

- **FR-R5: Stable-schema CSV output with separate run registry.** The implementation shall emit per-sample data as a CSV file whose schema is decided on the first successful sample and held stable for the duration of the run. Furthermore, it shall maintain a separate registry CSV with one row per monitor run, capturing workload, stimulus, request and limit configuration, and free-text notes. This follows from the output-format decision in Section 4.1.2: stable schema allows the same analysis code to read any run, and keeping the registry separate keeps per-run metadata out of the sample data.

4.2.2 Non-functional Requirements

The non-functional requirements describe the qualitative properties the implementation must exhibit. Each follows from a decision in Section 4.1.2.

- **NFR-R1: Modular and decoupled code structure.** The implementation shall be made up of independent components (monitor, driver, analysis), each replaceable without having to change the others; within each component, distinct concerns such as data collection, storage, and presentation shall be separated. This follows from the per-run isolation decision in Section 4.1.2, which made the monitor and driver separate independent processes, and the analysis layer should not need to know how data was collected, only what shape it arrives in.

- **NFR-R2: Minimal operator-side toolchain.** The monitor and driver shall require only a standard Python 3 interpreter and a working `kubectl`. The analysis layer may rely on the standard scientific Python stack (pandas, NumPy, matplotlib) (40, 41, 42). This follows from the language decision in Section 4.1.2 and extends Chapter 3's NFR-1 (low intrusion, lightweight dependencies) from the in-pod environment to the operator's environment.

- **NFR-R3: Inspectability of outputs.** All artifacts produced by the implementation (the per-sample CSV, the run registry, and any error markers) shall be readable and editable without specialized tooling. This follows from the output-format decision in Section 4.1.2, which favored CSV over SQLite on ease of readability grounds. The registry is also maintained as a plain CSV so the experimenter can edit rows or add notes by hand.

- **NFR-R4: Extensibility.** Adding a new stimulus, a new memory metric, or a new analysis function shall be a local change confined to one component, without modification to the others. This follows from NFR-R1 and from the per-run isolation decision in Section 4.1.2, which kept the monitor's output shape independent of the stimulus that produced it; new stimuli or metrics can be added without touching the monitor's writer or the analysis layer's loaders.

4.3 Realization Overview

Our memory monitoring tool (MemScope) is implemented as three independent Python components which run on the experimenter's machine and interact with the target pods: the memory monitor (`memory_monitor.py`), the stimulus driver (`memory_stimuli.py`), and the analysis layer (`memmon.py` and `analysis.ipynb`). The components and the way they interact with each other are visualized in Figure 4.1. On the cluster-facing side, the

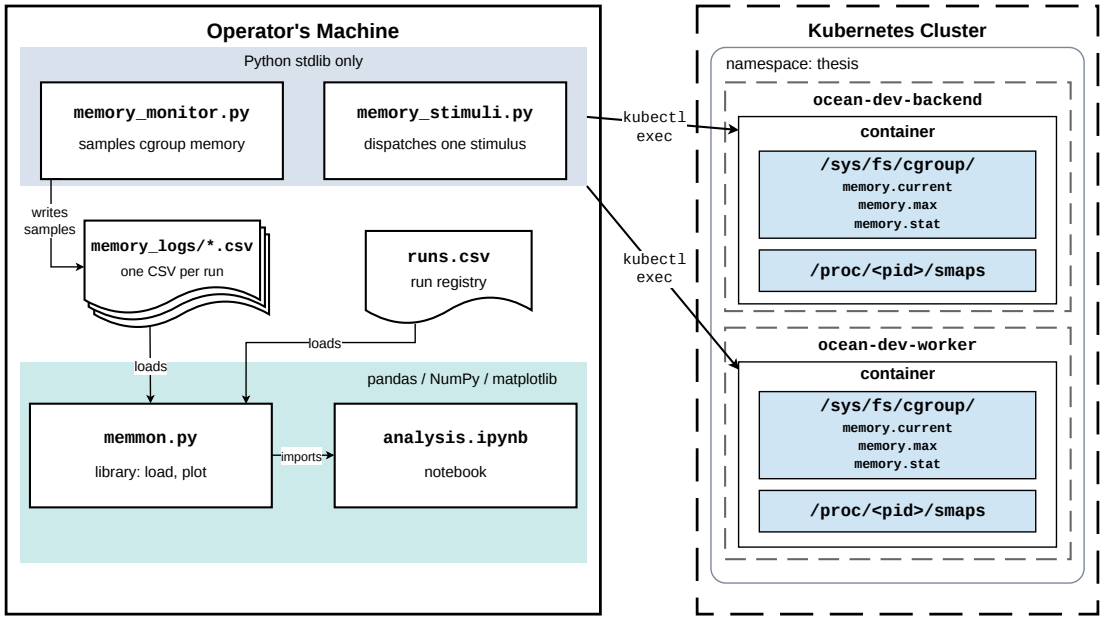

Figure 4.1: Artifact-level overview of MemScope. The monitor and driver are independent Python processes on the operator's machine that interact with target pods through `kubectl exec`; each run produces a per-sample CSV in `memory_logs/`, and a manually maintained registry (`runs.csv`) links each CSV to the stimulus and configuration that produced it. Analysis is offline: the notebook loads the time-series and the registry via `memmon.py`.

pipeline is very dependency-light; the standard scientific Python stack is only used for offline analysis.

The figure has two main sides. On the left lives the experimenter's machine, where all our Python components run, grouped into three lanes by what they do: the cluster-facing tools at the top (`memory_monitor.py` and `memory_stimuli.py`), the output files they produce in the middle (`memory_logs/*.csv` and `runs.csv`), and the analysis layer at the bottom (`memmon.py` and `analysis.ipynb`). On the right is the Kubernetes cluster, containing the two target pods inside the cluster. Between the two sides sits `kubectl exec`, which is the only way our pipeline reaches into the cluster. The monitor and driver use it independently; they do not communicate with each other directly, sharing only the same wall clock and the same target pods.

As a result of our implementation, the pipeline handles one stimulus per experimental run. The operator would start the memory monitor and driver on two different terminals. The experimenter can then dispatch a stimulus from the driver to the chosen target pod, which would cause a detectable change in the behavior of the memory that can be seen in the monitor's output. After the stimulus ends, the experimenter would stop the monitor,

and the whole run gets recorded in a separate CSV. Lastly, the run would be added to the separate run registry (`runs.csv`) describing what was run and recording useful metadata.

Our implementation realizes all requirements mentioned in Section 4.2. MemScope as a whole satisfies FR-R1, which is the main requirement from this chapter. The memory monitor realizes FR-R3 (steady frequency through fixed-period scheduling), FR-R4 (transient fault handling), and the per-sample half of FR-R5 (the stable-schema CSV). Our run registry (`runs.csv`) covers the metadata aspect of FR-R5, and the entire analysis layer realizes the offline-only side of NFR-R2. The CLI, which implements the configurability aspect in our pipeline, realizes FR-R2 (run-time configurability), and the three-file split itself takes care of NFR-R1 (modular and decoupled structure). Our choice of CSV reinforces NFR-R3 (inspectability) since it is easily human-readable, and finally, NFR-R4 (extensibility) follows from the modular structure of the code for both the monitor and driver.

4.4 Realization Details

Following the overview in Section 4.3, this section dives deeper into each module of our realization: the memory monitor (Section 4.4.1), the workload driver (Section 4.4.2), and the analysis flow (Section 4.4.3). Section 4.4.4walks through a full experimental run end-to-end so the reader sees the pieces working together. Each subsection details the component's design choices and traces them back to the requirements from Section 4.2.

4.4.1 Memory Monitor

At the monitor's core, there is a periodic `kubectl exec` invocation against each target pod, running a small shell snippet inside the pod. The shell snippet reads `memory.current`, `memory.max`, and the full `memory.stat` from `/sys/fs/cgroup/`, and uses `awk` to then serialize `memory.stat` into a JSON object. The choice to use `awk` instead of `python` or anything else is because basic shell tools are far more reliably present in minimal container images.

For each process listed in `cgroup.procs`, the shell snippet also reads `/proc/<pid>/smaps` and classifies every mapping as heap, stack, anonymous mmap, or other by its pathname field. The four categories are recorded in their own columns in the CSV output. To account for pages shared across the container only once, each mapping contributes its anonymous bytes scaled by the ratio of its proportional set size (PSS) to its resident set size (RSS). The files are traversed through the same mechanism used for `memory.stat`, only using

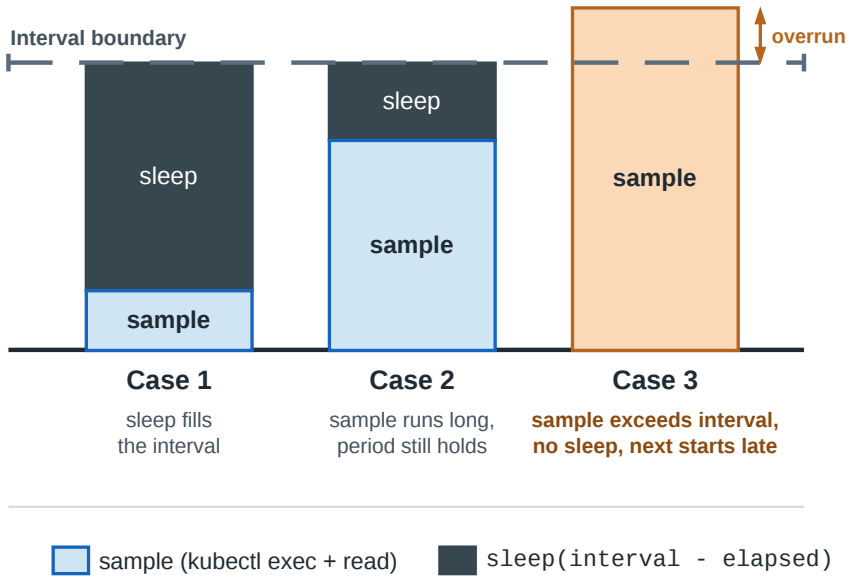

Figure 4.2: Three timing cases for a sampling iteration. The monitor sleeps for the interval minus the time already spent sampling. When the sample is quick the sleep fills the rest of the interval (Case 1); when it runs long the sleep shrinks but the period still holds (Case 2); and when it exceeds the interval there is no sleep left and the next cycle starts late (Case 3).

`awk`. This entire procedure can be optionally reduced to simply reading the **anonymous** category from the `memory.stat` file through the CLI.

Our sampling frequency has a default value of 3 seconds, and is configurable on launch of the monitor from the CLI. Each invocation is scheduled with a monotonic clock: the loop records its start time, sends out the `kubectl` calls, and then sleeps for the remainder of the interval. Figure 4.2 shows how this holds the period across three timing cases. One CSV row is emitted per sample, containing an ISO 8601 UTC timestamp at millisecond precision, a sample index, the deployment and pod name, the memory snapshot fields, and an error field. The CSV is flushed after every write so partial data survives an unexpected interruption. This realizes FR-R2 and FR-R3.

The CSV schema is decided after the first successful sample, based on which categories of memory `memory.stat` actually exposed. Three failure modes are caught with unique errors: `kubectl exec` failure/timeout, JSON parse failure, and no running pod found. Pod names are resolved again on every iteration by deployment-name prefix, following the deployment itself and safeguarding against pod restarts. This realizes FR-R4 and the per-sample part of FR-R5.

The monitor's code is organized into three sections: collection, storage, and presentation,

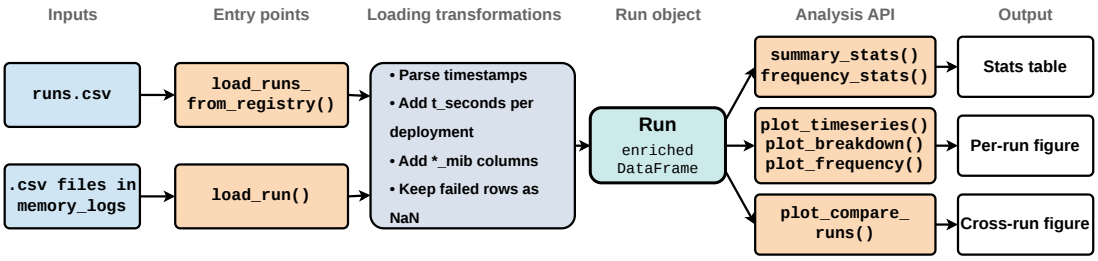

Figure 4.3: Flow of data through the analysis layer. Two entry points in `memmon.py` feed the same loading transformations, producing a `Run` object that branches into three paths: statistical summaries, single-run plotting, and cross-run comparison. Outputs land back in the notebook.

along with the main loop that manages all of them. Collection handles `kubectl` calls and parses cgroup data, storage handles the writing to the CSV, and presentation prints the live terminal table. A SIGINT handler is wired in so that manually terminated runs still produce a clean CSV. This structure contributes to NFR-R1 because of its modular nature.

4.4.2 Workload Driver

The workload driver presents a suite of various memory stimuli actions. In total, the suite contains 14 different stimuli: 9 of which are synthetic and targeted towards one memory category each, and the remaining 5 utilizing real code paths in our deployed workload. All stimuli are dispatched via `kubectl exec` exactly as done in the memory monitor, one at a time. Table 4.1 presents a detailed preview of all 14 stimuli and their purpose. This component of the implementation satisfies the stimulus side of FR-R1.

The driver emits labeled and timestamped markers to the terminal stdout at the start and end of each stimulus, only to contribute to operator visibility during a run. Two `kubectl exec` modes are available: regular argument passing for short payloads, and stdin-streamed for scripts over approximately 4 KB, to help get around the Kubernetes API server's request-header size limit.

An interactive user menu is also provided through the CLI to allow the operator to easily configure any variables such as target pod, hold time, and stimulus parameters. More variables are changeable through CLI flags or environment variables. This satisfies FR-R2 for the driver.

#	Type	Stimulus	Action
1	syn	`anon-bytearray`	Allocate a 200 MB `bytearray`
2	syn	`anon-pyobjects`	Allocate a 10-million-element Python integer list
3	syn	`file-cache`	Write and read a 300 MB file to populate the page cache
4	syn	`slab-manyfiles`	Create and stat 50,000 small files
5	syn	`kernel-sockets`	Open 2000 listening TCP sockets
6	syn	`steady`	Gradual 10 MB/s allocation ramp over 60 s
12	syn	`sustained-hold`	Allocate and hold a bytearray of `-hold-mb` for the duration of the run (default 800 MB)
13	syn	`file-pressure`	Drive `-file-mb` of page-cache demand against the configured limit (default 900 MB)
14	syn	`anon-ramp`	Ramp anonymous memory at `-ramp-mb-s` toward the `-hold-mb` target (default 25 MB/s)
7	app	`celery-image-hashes`	Sustained Celery image-hashing task on the worker
8	app	`phash-readonly`	Compute perceptual hashes for N images, no DB writes
9	app	`debug-sawtooth`	Worker tasks under Django `DEBUG=True` log accumulation
10	app	`http-bulk-create`	`POST /items/bulk_create` write path
11	app	`ai-image-task`	End-to-end crawl: Bedrock + Search + PIL + Vision

Table 4.1: The fourteen stimuli implemented in the workload driver.

4.4.3 Analysis Layer

The analysis layer has two main entry points: `load_run()` to process one CSV, and `load_runs_from_registry()` for the registry-driven set, both of which output neat data frames. On loading, derived columns are added by our custom library `memmon.py`, including per-deployment relative time and MiB conversion of every memory category. Failed samples (marked by error strings) are replaced with NaN values, and hence kept in the analysis instead of silently dropped. The library also provides helpers for summary and frequency statistics, along with a small set of plotting primitives. Figure 4.3 visualizes the flow of data through the layer.

The notebook is what actually performs analyses and inspection on single runs, and also does cross-run comparison. The run registry `runs.csv` contains one row per run, is loaded directly by `load_runs_from_registry()`, and can be edited by hand if necessary. This layer realizes the metadata half of FR-R5, the offline-only side of NFR-R2, and reinforces NFR-R3.

4.4.4 Walkthrough of an Experimental Run

The entire experimental pipeline is run by the operator in the following steps:

1. The operator opens two terminals and launches the memory monitor in one (against the chosen namespace and deployments) and the memory driver in the other (pointed at the same namespace).
2. After waiting a short while to record an idle baseline, the operator picks one stimulus from the driver's menu.
3. The driver prints a labeled start marker, and the monitor's table shows the corresponding memory change live.
4. When the driver outputs the end marker signaling that the run is done, the operator stops the monitor with Ctrl+C, which triggers a flag, saves the CSV after the next sample is read, and then terminates the monitor.
5. A row can then be manually added to `runs.csv` recording the workload, stimulus, configuration, and any notes.
6. Finally, the notebook loads the new run and produces the plots and statistics that feed into the evaluation chapter.

4.5 Summary

This chapter began by explaining the choices left open by the design from Chapter 3, prompting the need to make several decisions before the design could become a working artifact. We then proceeded by surveying different options for each open choice, weighing out their advantages and disadvantages, and choosing a suitable implementation direction. From this direction, functional and non-functional requirements were derived that our implementation must satisfy. All of this leads to our final implementation, MemScope, consisting of: a memory monitor, a workload driver, and an analysis layer. The evaluation of MemScope is presented in Chapter 5.

5

Evaluation

Main Finding 5.1 (MF5.1):
Anonymous memory, dominated by the Python heap, is the main contributor to Ocean's container memory, forming 72 to 90 percent of the total. Addresses C1.

Main Finding 5.2 (MF5.2):
Anonymous memory is required for running the workload and is retained until the process that holds it exits; the kernel cannot reclaim it. Addresses C1.

Main Finding 5.3 (MF5.3):
File cache and a part of slab memory are reclaimable: the kernel can reclaim them when it falls under memory pressure, so they represent temporary demand. Addresses C1.

Main Finding 5.4 (MF5.4):
Different monitoring tools report different values for memory usage because they measure different memory components. The gap between MemScope and `kubectl top` can be attributed to the reclaimable file cache. Addresses C2.

Main Finding 5.5 (MF5.5):
Limits sized based on usage reported by tools always take some form of reclaimable cache into their reservation. Depending on the operator's chosen tools and sizing strategy, limits can over-reserve up to 100% unused memory. Addresses C2.

Main Finding 5.6 (MF5.6):
The boundary that decides when a container gets killed is the irreducible floor: `memory.current` − `file` − `slab reclaimable`, derived from our decomposition. When this value reaches the limit, the container is killed. Addresses C3.

Main Finding 5.7 (MF5.7):
A limit sized based on the irreducible floor is tighter than other limits set by existing strategies (VPA) while remaining safe for the container, resulting in less over-reservation. Addresses C3.

Main Finding 5.8 (MF5.8):
Savings when using the irreducible floor as a basis for the limit would scale with the reclaimable components of memory. Addresses C3.

5.1 Overview

We structure this chapter around the three research questions from Section 1.2, with a section dedicated to each question, presenting the set of experiments done for it. Before going into detail, we restate each research question and its main finding and a pointer to where it is addressed.

- **RQ1: What are the main contributors to container memory usage in Kubernetes-managed environments?** We use MemScope to decompose Ocean's memory usage into several components. MemScope is validated using the synthetic stimuli from our suite before being applied to real workloads. We find three main memory categories: anonymous, file cache, and kernel slab. The dominant memory component is the anonymous Python heap, with 72 to 90 percent of the composition, and we find that it is retained by the kernel for processes with longer lifetimes (Section 5.3).
- **RQ2: How does memory usage differ between actual runtime consumption and declared configurations, and what are the main contributors to this discrepancy?** We compare the gap between the aggregate number reported by memory tools and our decomposition, and quantify the difference between declared and actual memory usage. We find that the gap between our decomposition and other monitoring tools stems back to reclaimable file cache. Limits set based on these aggregate numbers can over-reserve memory by up to 100%, based on the tool an operator uses and the strategy they use to set the limits. (Section 5.4).
- **RQ3: What optimization strategies can be suggested to safely reduce unnecessary memory reservations in Kubernetes-managed containerized workloads?** We derive a new value, the irreducible floor, from our decomposition and run various experiments to show that the floor is what governs when a container gets OOM killed. We then use the floor as a basis for setting requests and limits, and deploy them on a real Ocean workload. We find that the newly derived limits are safe and tighter than other limits set using tools like the Vertical Pod Autoscaler (Section 5.5).

We approach RQ3 in a different manner, since we make a claim that has to be backed, unlike RQ1 and RQ2 which both simply rest on measurements and observations. The claim we make for RQ3 is as follows: a limit set by adding a thin safety margin to the

irreducible floor remains safe, as an OOM kill only occurs when the floor itself reaches the limit. We run four independent experiments, each targeted at a certain aspect of the claim. Section 5.5 lays out this strategy before presenting the four experiments themselves.

5.2 Experimental Setup

Section 5.1 showcased the results for each research question briefly. Before discussing the experiments behind those results, we introduce the workload we studied, the instrument we used, and the conventions we follow in every experiment. This matters because a number resulting from an experimental run means nothing without context. Section 5.2.1 introduces Ocean, the workload the evaluation studies, Section 5.2.2 discusses MemScope, and Section 5.2.3 closes with the followed conventions: the stimulus suite, sampling interval, and repetition count.

5.2.1 System Under Test (Ocean)

The workload being studied is Ocean: an AI web crawler used by Capisoft in production, already introduced in Section 1.3. All of our experiments target the worker deployment running within Ocean, not other deployments like the Django backend or beat scheduler, since the worker is where the memory-heavy Celery tasks run. All deployments exist in an isolated namespace within the shared AWS EKS cluster, and are managed through Rancher (43). Ocean ships with no memory request or limit, placing it in Kubernetes' BestEffort QoS class (24).

5.2.2 Instrument (MemScope)

MemScope, introduced in Chapter 4, samples the cgroup v2 memory controller of Ocean's target pod and writes one row per sample. Each row reports `memory.current` and `memory.max` together with a breakdown into three components: anonymous, file-backed, and kernel/slab memory. Furthermore, each row includes the PSS-weighted breakdown of the anonymous component. These are the components every experiment in this chapter builds on.

The memory monitor produces one CSV file per experimental run, which would record the behavior invoked by one stimuli pushed by the memory driver. In the runs registry, each CSV is manually linked to its corresponding declared requests and limits, the stimuli it monitors, and other metadata. The registry entries are then used by the analysis layer to produce plots, graphs, and structured data that this chapter uses as reference.

5.2.3 Methodology Conventions

Every experiment in this chapter utilizes on the same stimulus library, generated by `memory_stimuli.py`: nine synthetic stimuli, S1 through S6, S13, and S14 that inject one type of memory pressure in isolation, and five stimuli that exercise Ocean's actual application code, including S7, S8, and S9 used throughout this chapter. Unless stated otherwise, `memory_monitor.py` samples every 3 seconds; experiments that need to resolve a sharp transition use a narrower interval instead.

We set the repetition count for each experiment independently, rather than using a fixed count for all experiments. Each result reports its own repetition count. When fewer iterations are run as part of an experiment, it is flagged in Section 5.6. As stated in Section 5.1, for RQ3 expected results were specified before each of the four experiments were run. All runs are recorded in our runs registry (`runs.csv`), along with metadata like targeted deployment, chosen stimulus, and declared memory configurations.

5.3 RQ1: Main Contributors to Container Memory

RQ1 seeks to find the different components of container memory. We approach this question in two parts. Firstly, we validate the fact that MemScope attributes memory to the correct components (Section 5.3.1). Secondly, we use the validated MemScope to monitor Ocean's worker (Section 5.3.2). The decomposition that MemScope produces is only trustworthy if the instrument has been validated first.

Throughout the chapter, we track three main memory components: anonymous memory (the process heap and other mappings not backed by a file), file-backed memory (the page cache built up by reading and writing files), and kernel/slab memory (the kernel's management overhead on behalf of the container, with slab accounted within it). The relevant difference between those components is reclaimability: the kernel can take back clean file cache if it is under pressure, but cannot do the same with a live heap. All our findings are stated in terms of these three components.

5.3.1 Experiment 1: Instrument Validation

The first experiment involves validating the accuracy of the memory driver and monitor implemented as part of MemScope. We do so by injecting a known amount of memory using the synthetic stimuli S1 through S6 from our suite, targeting one memory component in isolation, and observing whether the monitor reports the change in the correct component

Stimulus	Target component	Observed effect	Attribution
Idle baseline	none	reconciles within 0.3 MiB	correct
S1: anonymous mmap	anonymous	+191.9 MiB anonymous	correct
S2: Python integers	anonymous	+384 MiB anonymous	correct
S3: clean file read	file	+300 MiB file cache	correct
S4: 50,000 small files	file, slab	∼196 MiB page cache	correct, see below
S5: 2,000 idle sockets	kernel, slab	+7.8 MiB slab	correct, see below
S6: gradual ramp	all	reconciles beyond 1 GiB	correct

Table 5.1: Validation results per synthetic stimulus. Each injected delta landed in the component it targeted.

and shows the correct delta. We ran each stimulus three times using the default 3 second sampling interval.

In every case, the injected memory landed in the correct component, and the monitor reported the change as expected. Table 5.1 summarizes the results. Moreover, the monitor's PSS-weighted breakdown of anonymous memory into heap, stack, mmap, and other categories agrees with what the kernel reports for anonymous memory within 5 MiB for 98.7% of samples when summed up as shown in Figure 5.1.

When using S4 to write 50,000 small files , we observe a memory increase of 196 MiB even though the files themselves take up much less. That is because the page cache can only allocate 4 KiB pages, and each file gets a full 4 KiB page even if its raw size is smaller. Moreover, when using S5 to open 2,000 idle sockets, we observe an increase of 7.8 MiB in the slab category, not the dedicated socket statistics. That is because idle listeners like what S5 opens hold kernel data structures which would classify as usage under the slab category, while the socket statistics track send and receive buffers.

5.3.2 Experiment 2: Decomposition of Ocean's Memory

We point MemScope at the worker and let Ocean's own code generate memory pressure. We use three stimuli from our suite: S7 drives a sustained image hashing batch through the Celery worker, S8 performs the same hashing inside a short-lived helper process, and a 2.4 hour trace records the worker with no stimulus at all. A fourth run, S9, produced a negative result and is discussed at the end of this section.

The composition of S7 at its peak is shown in Figure 5.2a, drawn from the full run in Figure 5.2b. The total memory usage is 1292 MiB, divided as follows: 72% anonymous

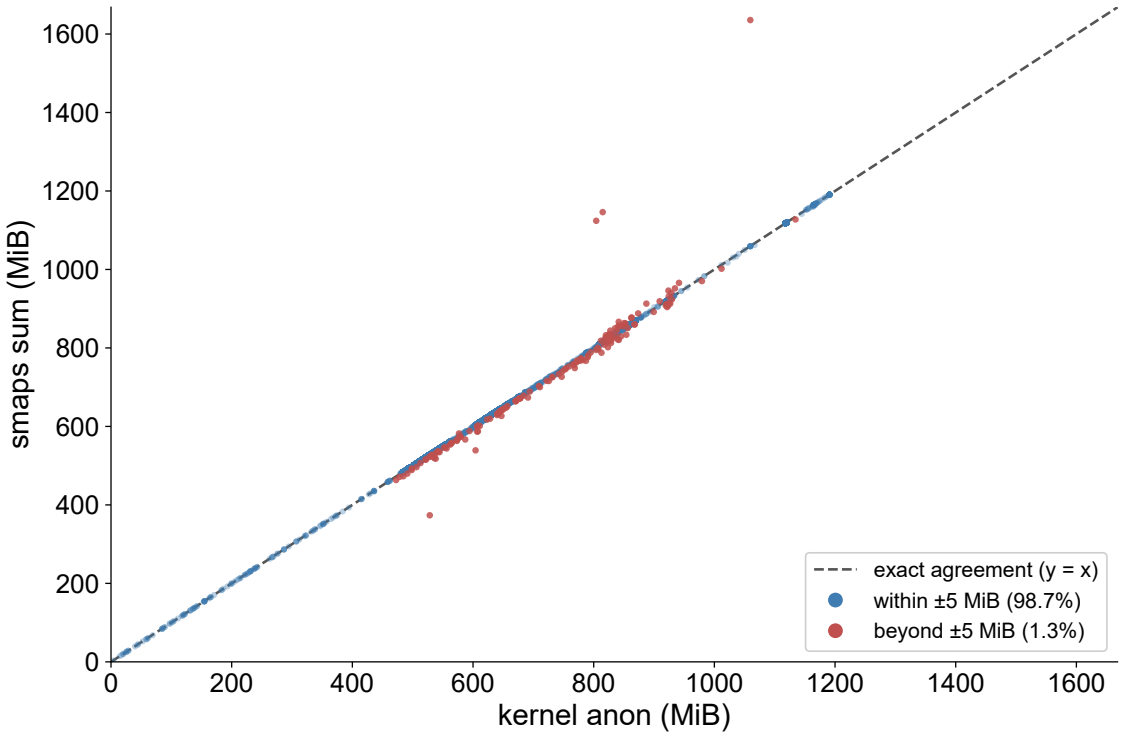

Figure 5.1: Instrument validation across every recorded run: MemScope's summed per-process anonymous breakdown plotted against the kernel's own anonymous accounting, one point per sample. Of 11,947 samples, 98.7% agree within 5 MiB of exact agreement.

memory, 24% file cache, and 3.6% kernel/slab memory, with 3.3% of that being slab specifically. The breakdown is presented in terms of fractions of `memory.current` and does not form a clean partition, because the kernel counts slab inside kernel memory and some categories fall outside the three main components. Across all three runs, anonymous memory takes up 72% to 90% of the total. The heap dominates regardless of what else changes between runs.

Composition alone does not tell us which component to worry about; retention does. During S7 the worker's anonymous heap climbs from 465 to 894 MiB and stays there after the batch completes: the long-lived Celery process holds on to every page its Python heap ever grew to. S8 performs the same hashing work in a process that exits when done, and its roughly 336 MiB of anonymous memory is returned in full (Figure 5.3). The same operation is cheap or expensive depending on the lifetime of the process that runs it.

The 2.4 hour trace backs the same results without needing to inject any workload. The trace shows one pod restart at roughly 51 minutes, shown by a brief drop to 33 MiB. Excluding the pod restart, the memory usage range falls between 550 and 1008 MiB.

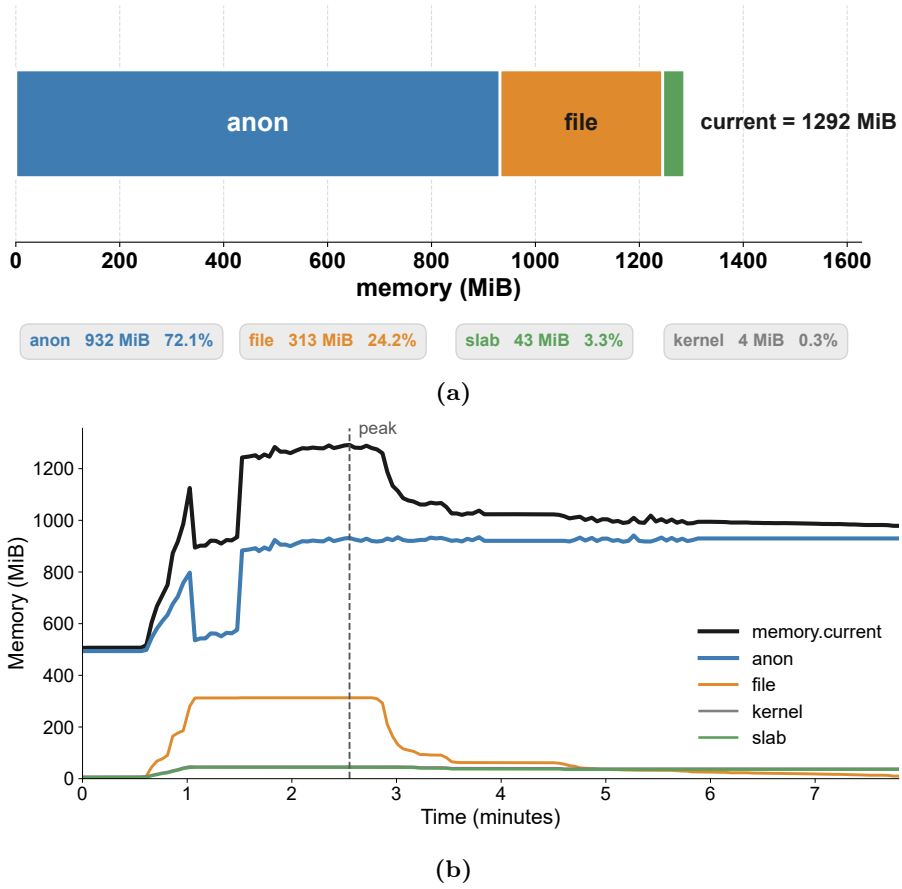

Figure 5.2: Decomposition of the worker's memory under S7: (a) the composition at the peak sample, 1292 MiB split into 72.1% anonymous memory, 24.2% file cache, and 3.6% kernel memory, of which slab accounts for 3.3% of the total; (b) the full run that peak was drawn from, with the peak marked.

Most of the change that occurs in memory usage throughout the entire trace is caused by the file cache moving, since it builds up during crawling activity and gets dropped later (Figure 5.4). The anonymous memory category moves a lot less (191 MiB), in comparison to the full delta (458 MiB). This tells us that the total usage itself is a noisy signal, while the retained heap underneath it is more steady.

S9 attempted to reproduce a known Django failure mode: with `DEBUG=True`, Django records every database query in memory, which should produce a slow sawtooth of anonymous growth. No sawtooth appeared. The explanation is simple: the worker runs with `DEBUG=False`, so there is nothing to record. We report this as a bounded negative result rather than a failed experiment: the stimulus ran and measured cleanly, the effect it was meant to target is not present in our configuration.

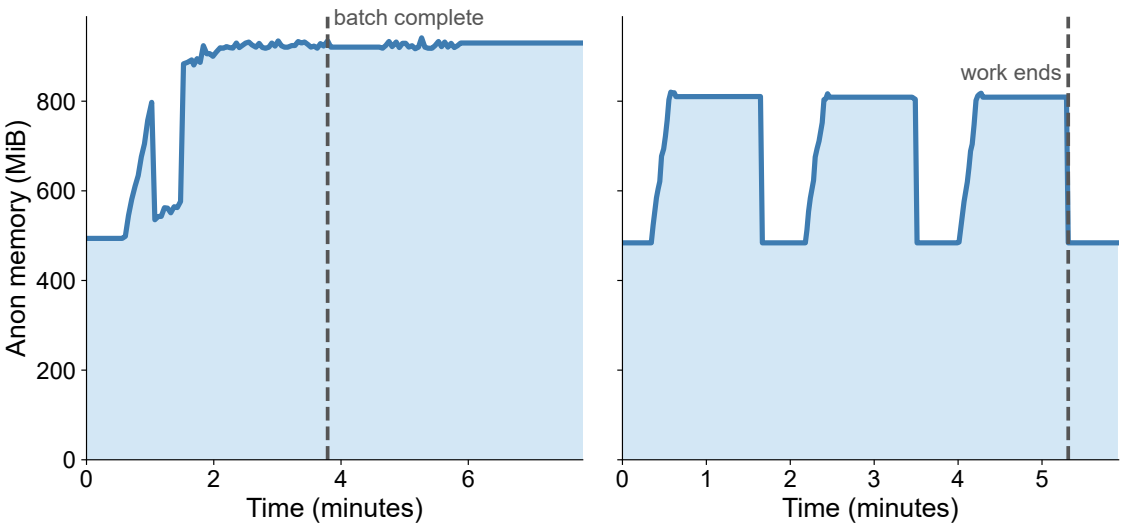

Figure 5.3: Retention contrast between S7 and S8: the long-lived worker holds its heap after the batch completes, the short-lived process returns it in full.

5.3.3 RQ1 Summary

The answer to RQ1 after studying Ocean is as follows: 72 to 90% of a container's memory is composed of the live Python heap, which falls under the anonymous category. It is also the only component that the kernel keeps as long as the process that holds it lives. File cache is the second largest component, but it is temporary. These findings are the result of the decomposition produced by MemScope, and the tool's accuracy was verified in Section 5.3.1. The decomposition, findings, and MemScope deliver contributions C1 and C4, and Section 5.4 goes on to investigate the over-reservation gap.

Looking at what components can be reclaimed, we can gauge how flexible the worker's memory is. The live Python heap dominates the rest of the components, and it cannot be reclaimed by the kernel. Experiment 2 also showed that the worker does not release it, but rather retains it. On the other hand, the file cache, which forms roughly a quarter of the composition, can be reclaimed by the kernel under pressure with no effect on the process. In conclusion, roughly three quarters of what the composition shows is irreducible, while the rest is reclaimable. RQ2 uses this split and builds on it.

5.4 RQ2: Runtime Consumption vs Declared Configuration

RQ2 investigates the difference between what a container's actual memory usage and what was declared for it. Since Ocean does not declare requests and limits, we approach the

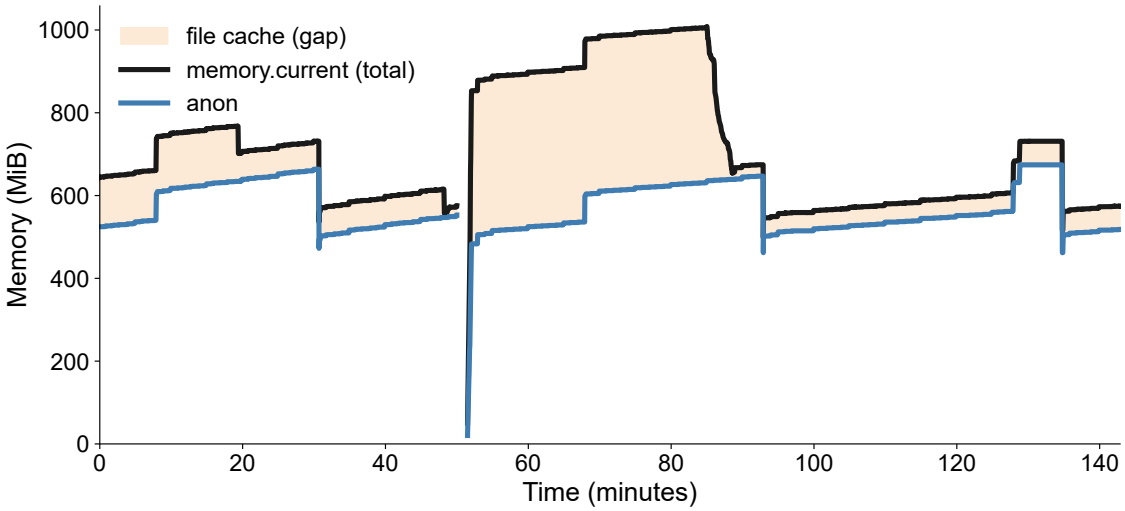

Figure 5.4: 2.4 hour unstimulated trace of the worker: the total oscillates with file cache while the retained anonymous portion moves far less. The brief drop at roughly 51 minutes is a pod restart, excluded from the reported range.

question as follows: First, we compare what standard memory monitoring tools report for container usage against the raw needs of the container in Section 5.4.1. Second, we explore the requests and limits an operator could set and compare them to what a container would use in practice in Section 5.4.2, allowing us to quantify the over-reservation gap.

5.4.1 Experiment 3: Runtime vs Aggregate Tools

The source of the values an operator would use to set requests and limits is `kubectl top`. Our experiment aims to investigate the number reported by `kubectl top` for memory usage by running S8: the image hashing stimulus. We ran S8 at 200 images per batch across three different repetitions, while capturing numbers reported by `kubectl top` and MemScope simultaneously.

We observe that the two reported numbers do not match, and that the difference between them is consistent. At peak usage, the cgroup's `memory.current` reached 1060 MiB while `kubectl top` reported 920 MiB. We can attribute the gap between the values to the fact that they report different measures of memory: `kubectl top` reports the working set, which equals `memory.current` (what MemScope reports) minus the inactive file cache. If we recompute the working set from MemScope's decomposition we get 940 MiB, which falls within roughly 2% of what `kubectl top` showed.

The 140 MiB difference between the reported numbers is the cold file cache: memory the container is charged for but that the kernel considers reclaimable and the working set

therefore hides. An operator looking at the 920 MiB reported by `kubectl top` would not be able to tell whether the memory is all live heap, or 700 MiB of heap hidden under a cold file cache. Each of these cases requires different configurations, given that the kernel uses `memory.current` to decide if a container gets OOM-killed for exceeding its limit.

5.4.2 Experiment 4: Over-Reservation Quantification

After establishing that the cgroup charges a different amount of memory to a container than what is reported by monitoring tools, the follow-up question is what this costs when the number is turned into configuration. We use a quantity that was newly exposed by MemScope's decomposition: the floor, `memory.current` minus file cache minus reclaimable slab. The floor represents the portion of a container's memory that the kernel cannot reclaim. Using the same trace as in Experiment 3, we observe that the floor equals 875 MiB at the container's peak usage.

The operator can set limits using a number of different strategies. We compare three different strategies as part of this experiment, all of which use the same basis for setting limits, the `working_set`, because it is the only signal the standard tools expose. Using a memory trace produced by running S8, we note that the `working_set` at peak usage reaches 906 MiB. The first limit applies the Vertical Pod Autoscaler's default 15% safety margin (34) to the `working_set`, giving 1042 MiB. The second uses a 50% safety margin, giving 1359 MiB. The third doubles it, giving 1812 MiB.

Figure 5.5 visualizes the resulting limits, all strategies reserve well more than the `working_set` they are based on. The `working_set` already contains a portion of reclaimable memory that the kernel can take back and reuse under pressure. The operator setting the limits would not be aware of this fact, since the aggregate they see would hide it. This is where the floor would come in, as it strips `memory.current` of all reclaimable portions, making it more suitable as a basis for setting limits.

5.4.3 RQ2 Summary

The gap RQ2 aimed to investigate has two layers. First, standard monitoring tools like `kubectl top` report the `working_set`, which holds a reclaimable cache, and is not what the kernel uses to enforce memory limits. Second, strategies that use the `working_set` as a basis for setting limits can over-reserve by 15 to 100% depending on the safety margin used, and they pad an already inflated number. MemScope's decomposition exposes the floor, which strips what the cgroup reports through `memory.current` as the full charge

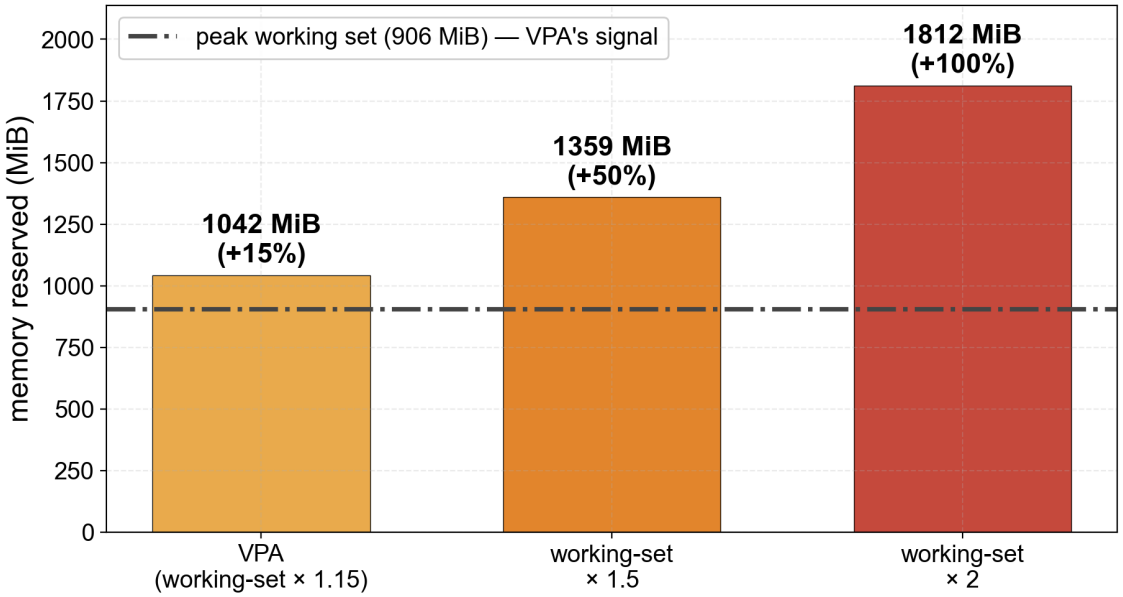

Figure 5.5: Declared limits per strategy against the peak `working_set` of 906 MiB, the signal every strategy pads.

from any reclaimable fractions, and is a better suited signal for setting limits than the `working_set`. This delivers contribution C2, what remains is to show what an operator stands to gain by using the floor as a basis when setting limits.

5.5 RQ3: Optimization Strategies

RQ3 asks if there are any strategies to be followed that would allow the gap from RQ2 to be closed without risking a container being OOM killed. In other words, can we set tighter limits based on the findings we have uncovered. As shown previously, limits are set by operators and existing solutions like the Vertical Pod Autoscaler (VPA) (34) using values reported by tools like `kubectl top`, which reports the `working_set`. However, a limit set using the measured floor plus a thin margin would also be safe, because the kernel reclaims reclaimable memory under pressure instead of killing. This rests on the fact that an OOM kill occurs if and only if the floor exceeds the limit. Not `working_set`, not `memory.current`.

To back our claim we carry out four different experiments, each answering one question: does composition decide outcomes, what mechanism absorbs memory pressure, where exactly is the kill line, and does it hold on the real workload. The expected outcome of each experiment is therefore fixed before running it. One scope note: VPA could not be

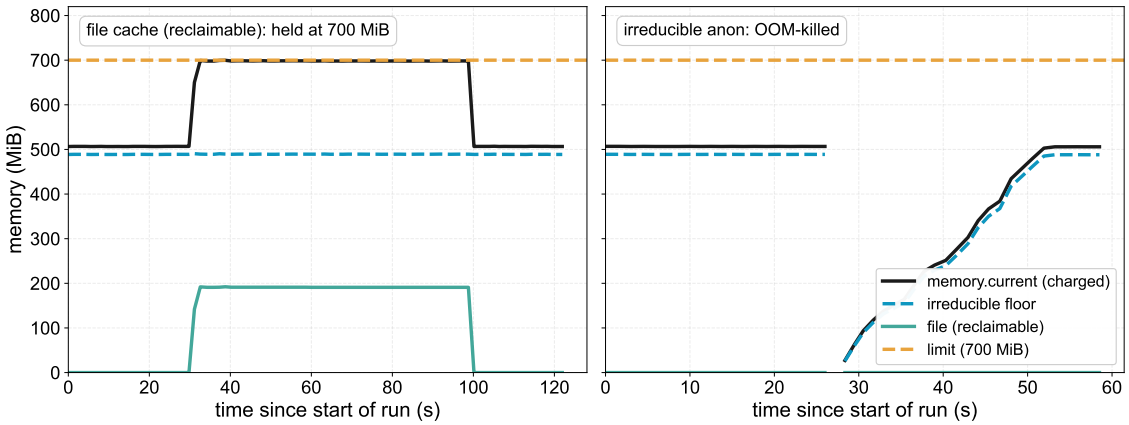

Figure 5.6: Two arms against the same 700 MiB limit: the file arm pins just below the limit and survives, the anonymous arm is OOM killed.

deployed due to difficulties in gaining the required permissions in the Ocean cluster, so it appears as emulated context only.

5.5.1 Experiment 5: Same Limit, Opposite Outcomes (Phenomenon)

This experiment sets the same fixed 700 MiB limit on our worker deployment. We apply memory pressure on the worker using two arms: the first using S3 and filling 1 GiB of reclaimable file cache targeting the file category, and the second allocating 1 GiB of Python objects targeting the anonymous category. Same limit, same demand, different components. Each arm underwent three repetitions to ensure a reliable result.

For the file arm we saw no OOM kills, `memory.current` rises and pins at 699.8 MiB, just short of the 700 MiB limit. On the other hand, we observe OOM kills when using the anonymous arm through all three repetitions. Figure 5.6 shows both outcomes side by side. Even though the limits were identical, the composition of memory decided what containers got OOM-killed.

Having only run three repetitions per arm, the ratio between survives and kills does not rule out chance. However, the evidence sits in the nature of the observed numbers, not the counts. For the file arm, `memory.current` reached and settled at a maximum of exactly 699.8 MiB during every single run, which matches the 700 MiB limit. Three identical observations show us the kernel's reclaim mechanism acting in practice. Therefore, we use the results of this experiment only to show that the kernel does in fact reclaim memory.

The result also raises the question this section answers next: the file arm demanded 1 GiB against a 700 MiB limit, so roughly 300 MiB of excess demand was absorbed by

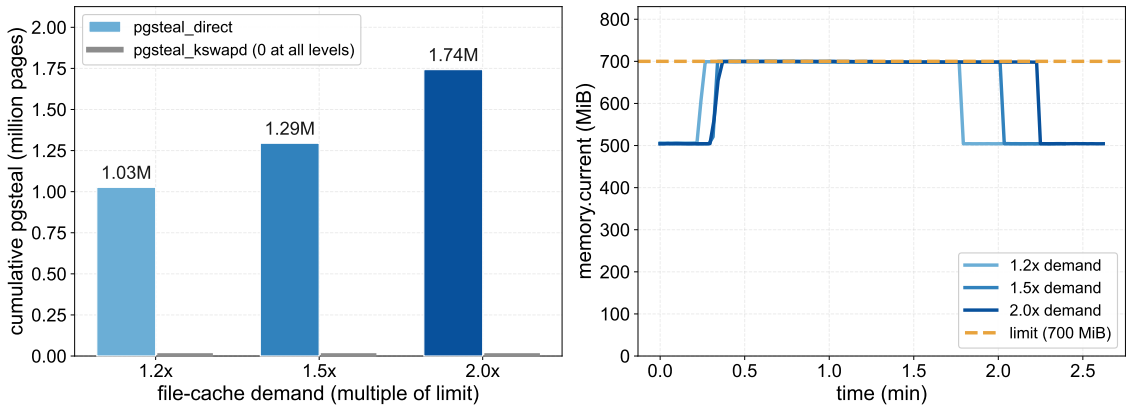

Figure 5.7: Direct reclaim under a fixed 700 MiB limit: `pgsteal_direct` climbs monotonically with file-cache demand while `pgsteal_kswapd` stays at zero and `memory.current` remains pinned at the limit.

something. Section 5.5.2 catches that something in the act.

5.5.2 Experiment 6: Reclaim Caught in the Act (Mechanism)

This experiment also uses a 700 MiB limit. However, the file-cache demand is swept this time not fixed, we allocate 1.2x, 1.5x, and 2x the limit. We carry out one repetition per each level, our signal is a dose response shape across the different levels and not the repetition within each level. We are interested in monitoring two main variables, the reclaim counters in `memory.stat`: `pgsteal_direct` and `pgsteal_kswapd`.

We observe, as expected, that `memory.current` gets pinned at the 700 MiB limit across all 3 levels, the file component caps near 199 MiB. Most notably, `pgsteal_direct` increases monotonically with the demand: 1.03M pages at 1.2x up to 1.74M at 2.0x, while `pgsteal_kswapd` stays at zero throughout and no OOM kills were observed even at double the limit. Figure 5.7 shows all three signals across the sweep.

We can now safely say that the absorption of memory observed in Experiment 5 was a result of direct reclaim. When the cgroup's memory usage reaches its limit, it reclaims its own file cache. We confirm that node pressure played no role in the reclaim from the observation that `pgsteal_kswapd` remained at zero, which is precisely why pressure in the form of reclaimable memory cannot cause an OOM-kill on its own. When the kernel's memory reaches its limit, it will attempt to reclaim file cache and all other reclaimable portions, until there are none left, before deciding to terminate. The question is no longer whether a container dies at its limit, but at what point the part that cannot be reclaimed runs into the limit, which is what Experiment 7 investigates.

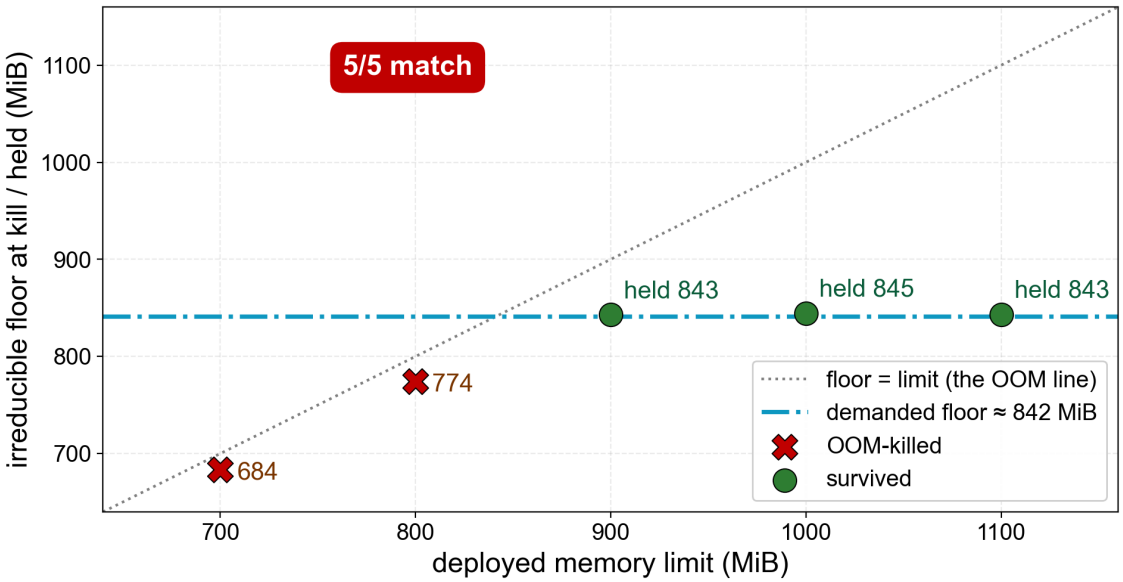

Figure 5.8: Limit sweep against a fixed anonymous ramp demanding a floor of roughly 840 MiB. Limits of 700 and 800 MiB are OOM killed with the floor at death tracking the limit; limits of 900, 1000, and 1100 MiB survive and converge on the same floor.

5.5.3 Experiment 7: The OOM Line Sits at the Floor (Law)

Unlike Experiments 5 and 6, this experiment follows an inverted strategy. The demand inflicted on the worker pod will be fixed, the limit will be varied. The stimulus used to push the demand is S14: a slow anonymous ramp that demands a floor of roughly 840 MiB, targeting the anonymous category because as proven by Experiment 6, anonymous memory is what cannot be reclaimed. The limit is set at the following levels: 700, 800, 900, 1000, and 1100 MiB, with one run per level. We expect to observe OOM kills whenever the limit sits under the demanded floor, whereas every limit set above it survives. For each kill, the floor measured at the moment of death should equal the limit.

We observe, as expected, that both limits set under the demanded floor at 700 and 800 MiB experience an OOM kill. The floor at the last sample before the kill was reported at 684 and 774 MiB respectively, each just under its predefined limit. On the other hand, limits set at 900, 1000, and 1100 MiB all survive. The floor converges to around 845 MiB during every level, regardless of how much extra headroom the limit leaves. All 5 levels showed results that matched our expectations. Figure 5.8 shows the sweep.

From these results, we can tell that the kill/survive boundary falls between 800 and 900 MiB, exactly where the demanded floor that our memory driver pushes (roughly 840 MiB) sits. No other quantity predicts this boundary, `memory.current` and demand exceed every

limit in the sweep, yet only the two runs showed an OOM kill.

Furthermore, we saw that the floor just before the worker pods were killed sat exactly below the limit set for the respective run (684 vs 700, 774 vs 800), not at the expected demand of roughly 840 MiB. The ramp was killed on its way up, precisely when the irreducible fraction hit the limit.

Lastly, we note that during the surviving runs the floor settled at the same value, around 845 MiB, regardless of the limit set. Once the limit sits clearly above the floor extra headroom gets us nothing, the worker will simply use what it needs. All in all, the observations of the kill below the floor, safety above it, and the boundary being at the floor itself, all quantify the claim made at the beginning of this section: an OOM kill occurs if and only if the floor exceeds the limit. The law here was established on a synthetic memory demand ramp, Experiment 8 investigates how it would apply to a real workload.

5.5.4 Experiment 8: Real-Workload Payoff

Now that we know the kill/survive boundary for a worker pod, the limit is no longer an experimental value, it is derived from the data: at every sample, the limit is the floor times a 1.10 margin. This experiment evaluates this limit on the workload pushed by the stimulus S8, which is a real image hashing workload that processes 200 images, repeated five times. At every sample of the recorded trace we compute two candidate limits, ours (floor times 1.10) and a VPA-equivalent (`working_set` times 1.15, applying VPA's default 15% safety margin to the signal the standard tools expose (34)), and compare both against what the workload actually held. We predict that the derived limit stays above `memory.current`'s irreducible fraction at all times, and would therefore result in no OOM kills.

As expected, we observe that the calculated limit (shown as the green line above the current load in Figure 5.9) sits above the irreducible floor always. Across all five repetitions the headroom between the floor and our limit never touches zero: not at the ramp, not at the peak, not during release. The workload ran to completion every time with zero OOM kills, which is exactly what the law predicts when the floor never crosses the limit. The VPA-equivalent line sits visibly higher throughout; the shaded band between them is reservation that was never needed. The derived limit achieved a 9% more efficient reservation strategy than the VPA equivalent, at the same zero kills.

We can now apply our newly derived limit to a real workload trace dynamically, over every sample. The floor-based limit should be safe by construction, since it simply follows the floor around wherever it goes. Our floor-based limit also leads us to an explanation behind VPA's extra reservation, since VPA watches the `working_set` which still contains

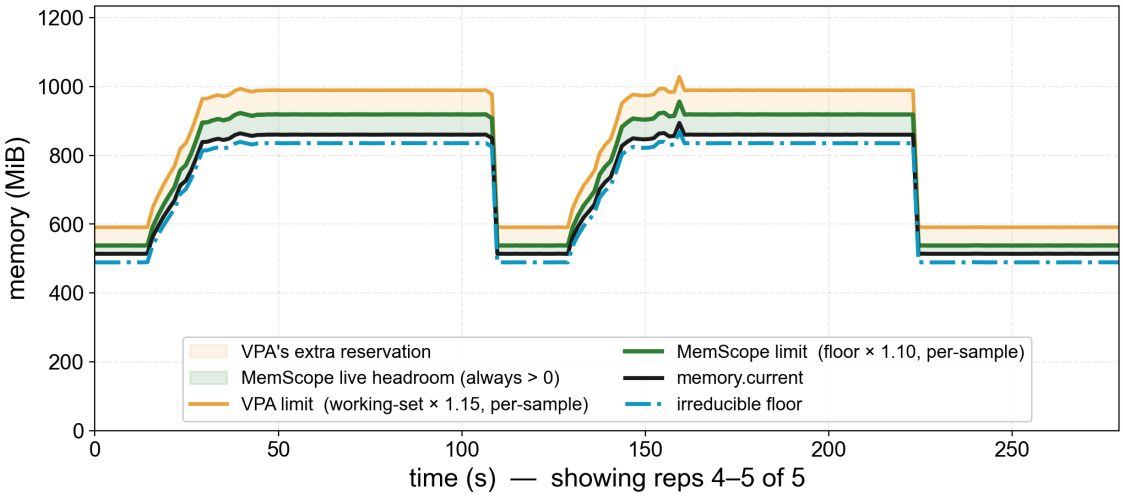

Figure 5.9: The real S8 workload against two per-sample derived limits: ours (floor times 1.10) and the VPA-equivalent (`working_set` times 1.15). The shaded band between them is reservation the workload never needed.

reclaimable memory, so it pads memory the kernel was already going to reclaim. Since the floor has no reclaimable portions, the new limit rids the operator of this problem, and provides better utilization than VPA. This experiment completes the chain we have used to back our claim that answers RQ3.

5.5.5 Configuration Recommendation: Request vs Limit

Our findings until now translate into a concrete configuration recommendation for the worker: a request sized from observed `working_set`, and a limit sized from the floor plus a thin margin. The decision to set the limit based on our derived floor is backed by the experiments carried out regarding RQ3, as they show that the limit is what governs the kernel's OOM decision. For the request, the justification is outside the scope of the limit: the request determines the pod's QoS class, and QoS class decides eviction order when the node itself comes under memory pressure (24). In other words, a pod can die in two different ways, and each way is decided by its own number: the limit is what protects the pod from the kernel, and the request is what protects it from the scheduler.

Ocean declares no requests and limits in production, and so is placed in the **BestEffort** QoS class by Kubernetes. If the scheduler were to need to evict a pod, those classified as **BestEffort** would be first in line. During our experiments, we observed this live by seeing an Ocean pod get evicted when the node fell under pressure, with a message stating that the request declared for that pod was zero. If an operator declares a request and a limit

based on our sizing strategy, the pod would be reclassified as **Burstable**. This prevents the pod from being the first victim in case of node pressure. If an operator would only set the limit and leave the request, the pod will be safe from OOM kills, but will remain threatened by eviction through the scheduler, which is why we recommend setting both.

5.5.6 Scope: Savings Scale with Reclaimable Fraction

The savings that result from our configuration recommendation are as big as the reclaimable fraction that the memory decomposition exposes. The gap between what monitoring tools report and the irreducible floor is what we end up saving. Ocean is very heavy on anonymous memory, and so the 9% savings shown in Experiment 8 are quite modest. These savings can grow when we are dealing with cache heavy workloads, which were shown synthetically in Experiments 5 and 6 but not measured on a production deployment. The percentage is therefore a property of the workload and its nature, not of the chosen strategy. When savings are small the value is diagnostic: the decomposition identified the heap as the holder and the long-lived worker as the reason (RQ1), pointing the real fix at the application, which no configuration tuning delivers. This is what the decomposition adds beyond a tighter number: it does not just size the container, it tells us where the memory went and who is keeping it. Chapter 7 picks this up when framing our third contribution.

5.5.7 RQ3 Summary

We found that an operator can set tighter limits while still keeping the container safe from an OOM kill, given that the limit is calculated using the right values. We claimed that an OOM kill was decided by the irreducible floor, and we carried four experiments to support that claim. Experiment 5 showed that equal magnitudes of memory pressure behave differently based on their composition. Experiment 6 demonstrated the reclaim mechanism in real time. Experiment 7 demonstrated that the kill boundary was approximately at the floor. We applied the new rule to a real workload as part of Experiment 8, and the limit calculated using the irreducible floor remained safe through the emulation and reserved 9% less than the VPA-equivalent.

These findings translate into a recommendation with two numbers, not one. The limit, sized at the floor plus a thin margin, closes the OOM exposure; the request, sized from the observed `working_set`, closes the eviction exposure by lifting the worker out of the

BestEffort class. How much the tighter limit saves depends on the workload: on anon-heavy Ocean the configuration gain is modest, and the decisive value is diagnostic, the decomposition points the real fix at the application rather than the config. Together, this answers RQ3 and delivers our third contribution. What remains is an honest account of where these results are limited, which Section 5.6 provides.

5.6 Limitations

The results of this chapter hold, but they hold within bounds. This section states those bounds explicitly.

When in some experiments runs are expensive, our repetition counts are smaller. For example, Experiments 6 and 7 run only 1 repetition per level. For those experiments, resulting values usually get their strength from their nature rather than their repetition counts. To elaborate, we saw a linear increase in the reclaim values proportional to the applied memory pressure in Experiment 6, while `memory.current` converged to the same values throughout all levels in Experiment 7. The repetition count won't increase the validity of these facts. In the future, running at least 5 repetitions for each relevant Experiment would let the contrast carry conventional statistics on its own (44).

All of our measurements target the same workload, Ocean's worker, which is an anon-heavy application. The large-reclaimable regime where the savings are more significant than the shown 9% was only shown with synthetic stimuli, never on a production cache-heavy service. This means the 9% figure does not generalize to other workloads of a different nature than Ocean's worker.

In Experiment 7, at a 1 second sampling against a ramp of roughly 20 MB per second, the last sample before a kill sits up to one interval below the true kill point; this is why the floor reads at about 20 MiB under the limit before the container gets killed (684 vs 700, 774 vs 800). While this does not threaten the law's direction, it sets a bound on how accurately we can report a floor-at-kill value.

The VPA comparison in this chapter is emulated, not deployed. Installing the VPA requires cluster-admin privileges, which the shared cluster does not grant us. We instead computed its recommendation from our own traces using its default sizing parameters: a high-percentile usage target plus the default 15% safety margin (34). This is why VPA appears as context throughout the chapter, our proof never depends on it.

Lastly, all experiments were run on a shared EKS cluster hosting production deployments, and faced real and unexpected disturbances: node failures, a namespace-wide scale

to zero, transient `kubectl exec` outages, and one live eviction. We also could not monitor node-level memory due to the lack of permissions. These disturbances are taken into account as part of our run registry, runs that happened at the same time as a node event were excluded and the one pod restart that survived into a reported trace is disclosed in Section 5.3.2

5.7 Summary

This chapter set out to answer the three research questions posed in Chapter 1, and each received an answer. For RQ1, the memory of Ocean's worker is dominated by anonymous memory, the Python heap, at 72 to 90% of the total, and it is the only component that is retained rather than recycled: the long-lived Celery process holds every page its heap ever grew to, while file cache comes and goes with crawling activity. These findings rest on an instrument we validated first, injecting known amounts into one component at a time and confirming that every change landed where it should. The decomposition is therefore evidence, and it delivers contributions C1 and C4.

For RQ2, the discrepancy between what is reported and what is actually held has two layers. The visibility layer: tools like `kubectl top` report the `working_set`, which hides reclaimable cache, while the kernel enforces limits against the full charge. The configuration layer: strategies built on the visible number over-reserve relative to the measured floor, because they pad a value that is already inflated. The decomposition separates the reclaimable from the irreducible and bases the configuration off the latter, which is contribution C2.

For RQ3, we showed that the gap can be reduced safely. We claimed that an OOM kill occurs if and only if the floor exceeds the limit, fixed the expected outcomes in advance, and backed the claim four independent ways in Experiments 5 through 8: composition decides survival at an identical limit, direct reclaim is the mechanism that absorbs the pressure, the kill boundary sits exactly at the floor, and a floor-derived limit follows the real workload with zero kills at 9% less reservation than the VPA-equivalent. The resulting recommendation carries two numbers, a limit against the kernel and a request against the scheduler, and delivers contribution C3. On anon-heavy Ocean the configuration gain is modest and the diagnostic value is the larger prize: the decomposition tells us where the memory went and who is keeping it. Chapter 7 returns to these answers and places them next to our contributions.

6

Related Work

In Chapter 3 we presented a design for the pipeline, Chapter 4 realized the pipeline as MemScope, and Chapter 5 evaluated and answered our research questions using MemScope. These results establish what MemScope does, but not where it sits relative to existing work. This chapter discusses the position of this thesis against the state of the art across the areas it touches, and states what we do differently for each area.

Section 6.1 surveys memory monitoring and characterization, Section 6.2 discusses resource right-sizing and over-reservation, and Section 6.3 explores workload characterization. The chapter concludes with a summary in Section 6.4.

6.1 Memory Monitoring and Characterization

Existing tools for monitoring container memory include cAdvisor, Prometheus, node exporter, metrics-server, `kubectl top`, and many more. As stated before, these tools all read cgroup files but expose one aggregate memory figure per container, resulting in a low resolution. Frameworks like Monintainer have improved portability and extensibility across orchestrators, but keep the same single-value granularity (35). Detailed or multi-layer container monitoring is a known gap in the survey literature, without an existing solution (45). Studies on the kernel level and others regarding container overhead treat memory as an aggregate or examine it on benchmarks. Some works do provide a characterization of memory past a single number. Google's far-memory system separates cold from hot memory across their fleet and reclaims the cold portion by compressing it (46). This is similar to the difference we found between reclaimable and irreducible portions of memory, with the distinction that it operates at the fleet and OS level, leaving an operator with no breakdown of container memory. The question of how an operator should size a single container's requests and limits remains open.

None of the previously stated works break down the single container-level reported value into its underlying cgroup constituents; the composition remains hidden. MemScope adversely reads `memory.stat` directly and provides a component-level decomposition (anony-

mous, file, slab, kernel) in the form of time series data, at seconds resolution. This decomposition leads to the approach of separating the reclaimable part from the irreducible floor, which a reported aggregate could not have led to. Moreover, MemScope achieves this without needing cluster-admin privileges and without in-pod dependencies, unlike other monitoring frameworks. In plain terms, no surveyed tool provides the per-component view this thesis relies on.

6.2 Resource Right-Sizing and Over-Reservation

The most notable existing right-sizing solution is Kubernetes' own Vertical Pod Autoscaler (VPA), which recommends memory requests from a working-set histogram over time and adds a safety margin on top of it (34). Another approach is seen with Google Autopilot, which uses the same shape at a production level: exponentially-decayed usage samples, a high percentile, a safety margin (47). The motivation behind Autopilot is the same as MemScope, operators over-reserve out of caution, which is wasteful at a large scale. Microsoft's Resource Central takes the same route at the provider level, predicting future usage from historical telemetry to reclaim over-reserved capacity (48). All three size the limit from an aggregate.

The field is extended into another area with research autoscalers that share the same aggregate signal. EVMMv2 reacts to memory pressure by suspending and checkpointing containers in priority order rather than setting a static limit, but its priority and scaling math are still based on aggregate usage ratios (49). Another approach from Gwydion changes container replica count through reinforcement learning and deliberately avoids vertical memory resizing, yet also still observes aggregate memory per deployment (50). This tells us that the blindness resulting from the lack of memory composition is not specific to static recommenders; reactive and learning-based approaches operate on it too.

Industry data suggests very high memory over-reservation at the production scale (8, 22). Academic traces also support this statement, with the average cluster memory usage being near 50%, allocations above 80% (51, 52, 53). The over-reservation gap exists, but is not explained at the component level. We decompose container memory and separate reclaimable file cache from the irreducible floor, allowing us to derive tighter limits based on our decomposition.

6.3 Workload Memory Characterization and Working-Set Estimation

Previous work investigates and characterizes how real cloud workloads reserve and actually use resources like memory, CPU, and disk, profiling declared versus actual usage (54). A different field of study investigates an approach that estimates a VM's working-set size so that only unused memory is reclaimed, with the goal of pinpointing the reclaimable boundary of the virtual machine (55). A third field looks at cgroup memory behavior directly, studying the performance pitfalls that arise from how cgroups account for and reclaim memory (9).

All of the aforementioned approaches target either the aggregate level (per-resource usage statistics) or a single-boundary level (one working-set number for the whole VM). None of them decomposes container memory into its underlying components like what MemScope does. While working-set estimation does yield one reclaimable figure, MemScope allows us to attribute this figure to cgroup components, resulting in the derivation of the irreducible floor, which tells us more than a singular boundary does. The component-level decomposition of container memory is the gap this thesis fills.

6.4 Summary of Related Work

We surveyed other works across areas like monitoring, characterization, right-sizing, and utilization. All works share the one flaw: dealing with container memory as a single aggregate figure. Without a clearer picture and a decomposition of container memory, the reservation gap can be measured but not explained or tightened. This aggregate is broken down by MemScope, which by doing so makes over-reservation go from a fact to an actionable target.

7

Conclusion

We have known throughout this thesis that Kubernetes exposes a container's memory as a single aggregate number, and that operators therefore lack visibility when it comes to what that number consists of. Due to this knowledge gap, operators usually set requests and limits through guesswork with safety margins that are meant to prevent a container from getting killed, which results in the over-reservation of memory resources. Nobody has measured how far declared configurations are from actual usage, and explored what the cause of that gap is. Furthermore, no data-driven container resource sizing guidelines exist, and no instrument has been made to specifically investigate this issue. These gaps all brought up the research questions in Section 1.2.

This thesis aimed to answer these research questions, and did so in the following manner: Chapter 3 designed the missing memory monitoring pipeline, Chapter 4 implemented it as MemScope, and Chapter 5 used it to run the experiments on the deployment under study (Ocean). Our experiments studied a real production workload, on a shared EKS cluster, and our tool required no deployment changes or cluster admin permissions.

7.1 Answering Research Questions

This section states the answers to each research question in the order they were posed. Answers are stated plainly and concretely, with pointers to evidence chapters and numbers where the questions are quantitative. Full experimental details were stated in Chapter 5.

RQ1: What are the main contributors to container memory usage in Kubernetes-managed environments?

To address **RQ1**, we use MemScope to monitor and analyze the memory of Ocean. We observe three main components after decomposing container memory: anonymous memory, file cache, and kernel/slab memory. Anonymous memory is composed of the running program's live data (heap, stack, `mmap`), file cache represents any pages the program read from the disk and stores for faster access, while kernel/slab memory houses the kernel's

management overhead (socket structs, kernel stacks, etc.). The dominant contributor in the case of Ocean is the Python heap, taking up 72 to 90 percent of the container's memory. The main variable component is the file cache, and the kernel and slab are minor and stable. This decomposition allows us to know whether a fix for decreasing memory use belongs in the application itself or the configuration.

Main Finding 5.1 (MF5.1): Anonymous memory, dominated by the Python heap, is the main contributor to Ocean's container memory, accounting for 72 to 90 percent of the total.

Main Finding 5.2 (MF5.2): Anonymous memory is required for running the workload and is retained until the process that holds it exits, the kernel cannot reclaim it.

Main Finding 5.3 (MF5.3): File cache and a part of slab memory are reclaimable: the kernel can reclaim them when it falls under memory pressure, so they represent temporary demand.

RQ2: How does memory usage differ between actual runtime consumption and declared configurations, and what are the main contributors to this gap?

To address RQ2, we investigate the gap between the reported memory usage of MemScope and other monitoring tools on the same workload, and use the result to attribute the over-reservation gap to certain components. After comparing both values, we find that the difference between the memory usage reported by MemScope and `kubectl top` equals the file cache. A limit set by an operator based on what current tools report can over-reserve up to 100%, depending on the safety margin and monitoring tool that the operator uses. We further observe that monitoring tools disagree because they report varying components; any limit sized based on the working set or `memory.current` includes some form of reclaimable memory in its reservation.

Main Finding 5.4 (MF5.4): Different monitoring tools report different values for memory usage because they measure different memory components. The gap between MemScope and `kubectl top` can be attributed to the reclaimable file cache.

Main Finding 5.5 (MF5.5): Limits sized based on usage reported by tools always take some form of reclaimable cache into their reservation. Depending on the operator's chosen tools and sizing strategy, limits can over-reserve up to 100% unused memory.

RQ3: What optimization strategies can be suggested to safely reduce unnecessary memory reservations in Kubernetes-managed containerized workloads?

To address RQ3, we derive a new value using our memory decomposition, the irreducible floor, with the formula: `memory.current` − file − slab reclaimable. The irreducible floor is what actually governs the kernel's decision to kill a container; we use it to derive tighter limits. Multiple experiments targeted towards finding the real boundary that decides when a container would be OOM killed showed us that it is when the irreducible floor reaches the container's limit that it is killed. The irreducible floor is also consistently lower than the working set, as it takes away all reclaimable components from `memory.current`. Emulating a limit sized for a real Ocean workload based on the newly derived value with a 10% safety margin proved a 9% saving when compared to what the Vertical Pod Autoscaler reserves (working set + 15%), while still being safe. The savings are modest on Ocean since it is an anonymous memory heavy workload, but the method replaces guesswork with measurement, and savings scale with the reclaimable fraction.

Main Finding 5.6 (MF5.6): The boundary that decides when a container gets OOM killed is the irreducible floor: `memory.current` − file − slab reclaimable, derived from our decomposition. When this value reaches the limit, the container is killed.

Main Finding 5.7 (MF5.7): A limit sized based on the irreducible floor is tighter than other limits set by existing strategies (VPA) while remaining safe for the container, resulting in less over-reservation.

Main Finding 5.8 (MF5.8): Savings when using the irreducible floor as a basis for the limit would scale with the reclaimable components of memory.

7.2 Main Contributions

Through answering the research questions, we deliver four contributions as stated in Section 1.4:

- **C1 (Empirical):** We delivered a decomposition of the underlying components of container memory through direct cgroup access, evaluated on a real production workload.

- **C2 (Empirical):** We quantified the existing over-reservation gap between declared and actual memory usage, and attributed it to specific memory components, mainly the reclaimable file cache.
- **C3 (Conceptual):** We suggest a sizing method for requests and limits based on a newly derived value, the irreducible floor. The method allows for tighter limits that remain safe from OOM kills when deployed. This contribution was held open in the introduction, but is now concrete.
- **C4 (Technical):** We implemented MemScope, a lightweight cgroup-level memory monitoring tool that provides a time-series breakdown of memory components and requires no cluster-admin privileges or deployment changes to run.

Together, the conceptual and technical contributions make limit sizing data-driven rather than guess-based.

7.3 Summary and Future Work

Previously, setting requests and limits for Kubernetes containers was guess-based and based on a single aggregate number. This resulted in a systematic over-reservation of memory across Kubernetes clusters, which is very wasteful of resources and has environmental, performance, and cost effects.

We built MemScope to monitor memory and provide a time-series decomposition by reading directly from the cgroup files inside a pod. The decomposition on Ocean showed us that the anonymous Python heap dominated the other components, and that file cache and a portion of slab memory are both reclaimable under memory pressure. We used this decomposition to look further into the known gap between declared memory and actual usage; we found that different monitoring tools report different values for usage since they measure different components, and attributed the gap to this difference which mainly consists of reclaimable file cache. Setting limits based on these reported values always caused over-reservation since it took into account these reclaimable components. Therefore, we derive the irreducible floor, a value describing memory clear of any reclaimable components. We have deduced that using the irreducible floor to set requests and limits gives us safe and tight results that reduce over-reservation by a quantity dependent on the nature of the running application. The conditions and limitations bounding our results are discussed in Section 5.6.

Our results leave room for several discussions. Other kinds of cache-heavy and mixed workloads would behave differently using our limits, and would harness varying amounts of savings; experiments need to be run using these workloads that might suggest alteration to the guides we have set. Moreover, emulating this limit proved that it would be safe, yet a natural continuation would be making a recommender that would feed the newly derived limits into a VPA-style controller, turning a manual analysis into an operational observation. Longer windows of observation on running workloads would also be interesting, as we would be able to record and catch long-horizon or seasonal behavior that we were not able to see with our Ocean deployments. Lastly, the environment we ran our experiments in along with many existing environments did not have swap enabled, which is the reason behind anonymous memory not being reclaimable. Running more experiments where there is a reclaim path for anonymous memory would require revisiting the definition of our irreducible floor.

We turned one aggregate number into its constituent parts, and setting requests and limits became a measurement not a guess.

References

[1] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. **Recalibrating global data center energy-use estimates**. *Science*, **367**(6481):984–986, 2020. 1

[2] George Kamiya and Vlad C. Coroamă. **Data Centre Energy Use: Critical Review of Models and Results**. Technical report, EDNA – IEA 4E TCP (Technology Collaboration Programme on Energy Efficient End-Use Equipment), March 2025. 1

[3] Cloud Native Computing Foundation. **CNCF 2023 Annual Survey**, April 2024. Accessed 2026-05-23. 1, 9

[4] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. **Cloud Container Technologies: A State-of-the-Art Review**. *IEEE Transactions on Cloud Computing*, **7**(3):677–692, 2019. 1

[5] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. **Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis**. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, pages 412–426. Association for Computing Machinery, 2021. 1

[6] Carmen Carrión. **Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis**. *J. Grid Comput.*, **20**(4), December 2022. 1, 10

[7] John Arundel and Justin Domingus. *Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*. O'Reilly Media, 2019. 1, 10, 11, 12, 15, 18

[8] CAST AI. **CAST AI Analysis Finds Only 13 Percent of Provisioned CPUs and 20 Percent of Memory is Utilized**. `https://cast.ai/press-release/cast-ai-analysis-finds-only-13-percent-of-provisioned-cpus-and-20-percent-of-memory-is-utilized/`, February 2024. Based on the 2024 Kubernetes Cost Benchmark Report. 1, 2, 3, 13, 62

[9] Zhenyun Zhuang, Cuong Tran, Jerry Weng, Haricharan Ramachandra, and Badri Sridharan. **Taming memory related performance pitfalls in linux Cgroups**. pages 531–535, 01 2017. 2, 18, 63

[10] Maria Carla Calzarossa, Marco L. Della Vedova, Luisa Massari, Dana Petcu, Momin I. M. Tabash, and Daniele Tessera. *Workloads in the Clouds*, pages 525–550. Springer International Publishing, Cham, 2016. 3

[11] Daniel E. Ukene. *Assessing Performance Optimization Strategies in Cloud-Native Environments Through Containerization and Orchestration Analysis*. Master's thesis, Georgia Southern University, 2024. 3

[12] Miguel De Lima, Luan Teylo, and Lucia Drummond. **Towards a Novel Vertical Scaling Approach for Bursty Workloads in Kubernetes**. In *Proceedings of the 18th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC '25, New York, NY, USA, 2026. Association for Computing Machinery. 3

[13] Arnaldo Ferreira and Richard Sinnott. **A Performance Evaluation of Containers Running on Managed Kubernetes Services**. pages 199–208, 12 2019. 4

[14] Capisoft. **Project Ocean**. KVK Innovatie Top 100, 2024. 5

[15] React.org. **Ocean: Your trusted platform for anti-counterfeiting**. Official website, 2024. 5

[16] Robert P. Goldberg. **Survey of virtual machine research**. *Computer*, **7**:34–45, 1974. 9

[17] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. **Hypervisor: A Survey on Concepts and Taxonomy**. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, **2**(3):222–225, February 2013. Retrieval Number: C0453022313/2013©BEIESP. 9

[18] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. **Containers and Virtual Machines at Scale: A Comparative Study**. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery. 9

[19] Stephanie Susnjara and Ian Smalley. **What is Containerization?**, 2024. IBM Think. Accessed: 2026-07-27. 9, 10

[20] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. **An updated performance comparison of virtual machines and Linux containers**. **00**, pages 171–172, 03 2015. 10, 16

[21] Nutanix. **AI Driving Container Adoption as Shadow IT Raises Risks**, March 2026. Reported survey findings summarized in a news article. 10

[22] Datadog. **10 Insights on Real-World Container Use**, November 2023. Telemetry-based container usage report. 10, 16, 62

[23] Anthony E. Nocentino and Ben Weissman. *Kubernetes Architecture*, pages 53–70. Apress, Berkeley, CA, 2021. 10

[24] The Kubernetes Authors. **Kubernetes Documentation**. `https://kubernetes.io/docs/`, 2026. Accessed: 2026-07-05. 10, 11, 12, 13, 14, 18, 20, 43, 56

[25] Pieter-Jan Maenhaut, Bruno Volckaert, Veerle Ongenae, and Filip De Turck. **Resource Management in a Containerized Cloud: Status and Challenges**. *Journal of Network and Systems Management*, **28**(2):197–246, Apr 2020. 10, 13, 15, 16

[26] Thanh-Tung Nguyen, Yu-Jin Yeom, Taehong Kim, Dae-Heon Park, and Sehan Kim. **Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration**. *Sensors*, **20**(16), 2020. 10

[27] Tejun Heo. **Control Group v2**. Linux Kernel Documentation, 2015. `https://docs.kernel.org/admin-guide/cgroup-v2.html`. 14, 15, 16, 18, 25

[28] Linux Kernel Documentation. **The /proc Filesystem**. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`, 2019. Section on /proc/<pid>/smaps and Proportional Set Size (PSS). 14

[29] David Bernstein. **Containers and Cloud: From LXC to Docker to Kubernetes**. *IEEE Cloud Computing*, **1**(3):81–84, 2014. 15

[30] Google. **cAdvisor: Container Advisor**. `https://github.com/google/cadvisor`, 2014. 15, 18, 19

[31] Prometheus Authors. **Prometheus: Monitoring system and time series database**. https://prometheus.io, 2012. 15, 18, 19

[32] The Kubernetes Authors. **Kubernetes Metrics Server**. https://github.com/kubernetes-sigs/metrics-server, 2018. 15, 18

[33] Prometheus Authors. **Node Exporter: Exporter for Machine Metrics**. https://github.com/prometheus/node_exporter, 2013. 15

[34] The Kubernetes Authors. **Vertical Pod Autoscaler**. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, 2018. 16, 50, 51, 55, 58, 62

[35] Miguel Correia, Wellington Oliveira, and José Cecílio. **Monintainer: An orchestration-independent extensible container-based monitoring solution for large clusters**. *Journal of Systems Architecture*, **145**:103035, 2023. 19, 61

[36] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. **Cloud monitoring: A survey**. *Computer Networks*, **57**(9):2093–2115, 2013. 19

[37] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley.* 04 1991. 19

[38] The Kubernetes Authors. **kubectl exec**. https://kubernetes.io/docs/reference/kubectl/generated/kubectl_exec/, 2026. 20, 25

[39] D. L. Parnas. **On the criteria to be used in decomposing systems into modules**. *Commun. ACM*, **15**(12):1053–1058, December 1972. 24

[40] Wes McKinney. **Data Structures for Statistical Computing in Python**. *SciPy 2010*, 2010. 34

[41] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre

Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. **Array programming with NumPy**. *Nature*, **585**(7825):357–362, Sep 2020. 34

[42] John D. Hunter. **Matplotlib: A 2D Graphics Environment**. *Computing in Science & Engineering*, **9**(3):90–95, 2007. 34

[43] SUSE Rancher. **Enterprise Kubernetes Management Platform & Software | Rancher**. `https://www.rancher.com/`, 2026. Accessed: 2026-07-05. 43

[44] Andy Georges, Dries Buytaert, and Lieven Eeckhout. **Statistically rigorous java performance evaluation**. *SIGPLAN Not.*, **42**(10):57–76, October 2007. 58

[45] Emiliano Casalicchio and Stefano Iannucci. **The state-of-the-art in container technologies: Application, orchestration and security**. *Concurrency and Computation: Practice and Experience*, **32**(17):e5668, 2020. e5668 cpe.5668. 61

[46] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. **Software-Defined Far Memory in Warehouse-Scale Computers**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery. 61

[47] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. **Autopilot: workload autoscaling at Google**. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. 62

[48] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. **Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms**. In *Proceedings of the 26th Symposium on Operating Systems Principles*,

SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. 62

[49] Taeshin Kang, Minwoo Kang, and Heonchang Yu. **Vertical auto-scaling mechanism for elastic memory management of containerized applications in Kubernetes**. *Future Generation Computer Systems*, **180**:108407, 2026. 62

[50] José Santos, Efstratios Reppas, Tim Wauters, Bruno Volckaert, and Filip De Turck. **Gwydion: Efficient auto-scaling for complex containerized applications in Kubernetes through Reinforcement Learning**. *Journal of Network and Computer Applications*, **234**:104067, 2025. 62

[51] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. **Heterogeneity and dynamicity of clouds at scale: Google trace analysis**. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery. 62

[52] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. **Imbalance in the cloud: An analysis on Alibaba cluster trace**. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017. 62

[53] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-scale cluster management at Google with Borg**. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. 62

[54] Siqi Shen, Vincent Van Beek, and Alexandru Iosup. **Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters**. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474, 2015. 63

[55] Vlad Nitu, Aram Kocharyan, Hannas Yaya, Alain Tchana, Daniel Hagimont, and Hrachya Astsatryan. **Working Set Size Estimation Techniques in Virtualized Environments: One Size Does not Fit All**. *Proc. ACM Meas. Anal. Comput. Syst.*, **2**(1), April 2018. 63

Appendix A

Reproducibility

A.1 Abstract

MemScope bundles four artifacts together: a cgroup-level memory monitor, a stimulus driver, an analysis library, and the recorded memory logs behind all figures from Chapter 5. The memory monitor can be used on any Kubernetes cluster without any deployment changes or cluster-admin privileges, and the recorded traces can be used to reproduce the full analysis. However, experimental runs specific to Ocean code paths cannot be reproduced externally using other workloads.

The memory monitor reads four inputs: `memory.current`, `memory.max`, the full `memory.stat` fields, and a breakdown of the anonymous memory component weighted using the Proportional Set Size (PSS). These inputs give us the ability to differentiate between what the kernel can reclaim and what it cannot when it is under pressure. What cannot be reclaimed, the irreducible floor, is what actually governs when a container gets OOM-killed. The recommender calculates requests and limits for a given memory trace to check that the irreducible floor never crosses the limit.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Derivation of memory requests and limits from cgroup memory decomposition, with offline trace deployment to verify OOM-kill safety.
- **Program:** *MemScope*: a cgroup memory monitor, a stimulus driver, an analysis library, and a recommender. Python 3.
- **Data set:** 40 recorded monitor sessions, 12,666 samples, 80 columns per sample. Registry in `runs.csv`.
- **Run-time environment:** On the operator side, Linux or WSL2, Python 3.10 or newer, `kubectl` with exec permission. On the pod side: a POSIX shell and `awk`. Nodes must expose cgroup v2.
- **Hardware:** No special hardware. Analysis runs on any machine. Live monitoring requires a Kubernetes cluster with cgroup v2 nodes; the recorded sessions were taken on AWS EKS.

- **Execution:** Periodic `kubectl exec` sampling at a configurable interval, 3 s by default, one stimulus per monitor session.
- **Metrics:** `memory.current`, `memory.max`, the full `memory.stat` field set, and a PSS-weighted breakdown of the anonymous component from `/proc/<pid>/smaps`.
- **Output:** Timestamped CSV memory logs, the thesis figure set, and a recommendation giving observed peaks, composition, and the derived request and limit.
- **Experiments:** Monitoring instrument validation, memory composition categorization, cross-tool comparison against `kubectl top` and the Vertical Pod Autoscaler, over-reservation quantification, OOM-boundary determination, and deployment of a floor-derived limit.
- **How much disk space required (approximately)?:** 11 MB for the repository, of which 5 MB is recorded trace data.
- **How much time is needed to prepare workflow (approximately)?:** Under two minutes; clone and install four Python packages.
- **How much time is needed to complete experiments (approximately)?:** Under one minute to rebuild every figure from the recorded traces. Roughly ten minutes per live stimulus session.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access

The artifact can be found in one public repository at `https://github.com/HusseinSarrar05/memscope`. Everything this appendix describes is in there: the monitor, the stimulus driver, the analysis library, the recommender, and every memory log recorded during the experiments. The tag `v1.0-thesis` marks the repository at the time of submission. Later commits may not match what the appendix describes, so we recommend checking out that tag rather than working from the default branch.

A.3.2 Software dependencies

Almost all of the dependencies sit on the operator side, with few having to be fulfilled on the pod side. The operator machine would need Python 3.10 or newer, a `kubectl` binary configured against the target cluster (using the correct kubeconfig for the cluster) with permission to run `kubectl exec`, and the four packages listed in `requirements.txt`. The monitor itself only uses the standard library; `pandas`, `matplotlib`, `numpy`, and `jupyter` are there for the analysis notebook. Inside the pod we need a POSIX shell and `awk`, both

of which should already be present in any image that can run a shell. Nothing is installed into the container, and no sidecar or agent runs next to the workload.

A.3.3 Data sets

The directory `memory_logs/` contains all of our memory logs, each in a separate CSV file named after the time of the session. Every row represents a single sample and has 80 columns that we divide into 5 categories: identity and timing, the raw `memory.current` and `memory.max` readings, an error field, the full `memory.stat` field set prefixed with `stat_`, and the anonymous memory breakdown prefixed with `smaps_`. The schema is consistent across all sessions, so any trace in the directory loads with the same reader. The specific sessions used in our evaluation are noted in the registry `runs.csv`: it holds one row per run with its label, the memory request and limit used during the run, the workload, the stimulus that was pushed, and notes. Some CSVs in the directory are exploratory or aborted sessions and are left out of the registry.

A.4 Installation

The artifact can be installed in three simple steps: clone the repository, check out the `v1.0-thesis` tag that pins the repository to its state at submission, and install the packages in `requirements.txt`. Running the analysis again needs nothing beyond this. To monitor a specific cluster, download the kubeconfig YAML file from the cluster dashboard, and set the `KUBECONFIG` environment variable to the configuration file. You can then use the memory monitor and stimulus driver to run experiments against the target cluster.

```
1  # 1. Get the code at the submission tag
2  git clone https://github.com/HusseinSarrar05/memscope.git memscope
3  cd memscope
4  git checkout v1.0-thesis
5
6  # 2. Install the analysis packages
7  pip install -r requirements.txt
8
9  # 3. Only needed for live measurement: point kubectl at the cluster
10 export KUBECONFIG=~/cluster-kubeconfig.yaml
11 kubectl get pods -n <NAMESPACE>
```

Listing A.1: Installing MemScope and pointing it at a cluster.

A.5 Experiment workflow

As already explained in Chapter 4, each monitoring session should involve one stimulus. The operator runs the monitor against a deployment of their choice on one terminal, and triggers the stimulus driver from a second terminal. When the stimulus is done, the operator stops the monitor, and the resulting CSV will be saved. The operator can then append a row to `runs.csv` describing the run. The monitor's namespace and deployment defaults are set for Ocean, so the operator has to pass their own.

The synthetic stimuli in the stimulus driver can be run against any cluster, while the Ocean stimuli need Capisoft's workload and cluster, and cannot be run externally. The analysis can be produced from the recorded traces in either case.

A.6 Evaluation and expected results

All figures used throughout the thesis can be regenerated directly from the available traces. The build prints a line for each figure reporting the headline values, so results can be compared with ease. This would allow the evaluation to be fully reproduced. The recommender can be used to produce the request, limit, and safety verdict against any given trace. Our own numbers come from `memory_20260619_203125.csv`, the VPA-comparison run: replaying that trace reproduces the floor-derived request and limit we report for RQ3, along with the verdict that the irreducible floor never crosses the limit we derived.

To verify the instrument, an operator can use the stimulus driver to fire one stimulus against their cluster. The triggered stimulus is expected to land in the targeted memory component, and the behavior would be observed in the memory monitor's terminal output and later the recorded trace. For example, if the operator uses S3, which writes a 300 MB file inside the pod and then reads it back, they should expect to see the file-backed memory category rise by roughly 290 MiB and stay elevated after the read completes. The absolute numbers might differ across different workloads, but the behavior should remain consistent.

A.7 Experiment customization

Most of the stimulus sizes are tunable from the command line: `-hold-mb` for the sustained hold, `-file-mb` for the file-cache pressure, `-ramp-mb-s` for the ramp toward OOM, and `-phash-n` for the number of images hashed. The monitor's sampling frequency is set with `-interval`. The parameters that actually matter for our results are the three margins

the recommender takes: `-vpa-margin`, `-req-margin`, and `-limit-margin`. The floor we measure is a property of the workload, but the headroom placed on top of it is a choice, and anyone who disagrees with that choice can rerun the recommender with different margins to see what it does to the savings we report.

A.8 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

Appendix B

Self Reflection

When I chose this topic I knew nothing about Kubernetes. That was the point. Cloud was a field I always wanted to learn more about, and Kubernetes was a word I had heard all around without ever understanding the meaning behind it. Picking a thesis topic that forced me into it was the most direct way I could think of to fix that. It is a lot less intimidating to me now, and that alone would have made the last few months worth it.

The hardest technical problem was the first one: how to watch a container's memory from the inside without changing the container. I tried several approaches before settling on the periodic cgroup reads the tool uses today, and my lack of experience did not help. I always had the feeling that there was a better way to do what I was doing, but had no idea where to look for it. Getting a reliable result out of the memory monitor was a milestone that took far longer than the finished code makes it look.

The result I am proudest of almost did not come about. It came out of the synthetic file-cache stimulus, the one I built to isolate page cache from everything else. Watching that category behave differently from the rest is what first made me keep it in the back of my head. I did not trust the finding at first, so I went back over run after run to make sure I was not fooling myself, and only once it held every time did I use it for my end results. Turning that suspicion into a direct comparison against the Vertical Pod Autoscaler's weaknesses is what gave the thesis its final shape, and it is the part of the work I would defend hardest.

Doing this inside a company like Capisoft was the best part. I got to see how the real world actually operates, and to feel that what I was building could translate into something with real effects and real savings later. It did bear a cost however, working in a live production cluster, deploying my own namespace and workloads off production images, meant being consistently afraid I would break something at every turn, and waiting on Felix for access or a fix more often than I would have liked. It was inconvenient in the moment. In hindsight it taught me more than a sandbox ever could have, and those are lessons I will carry into whatever I do next.

B. SELF REFLECTION

I owe a lot to my daily supervisors Matthijs and Felix, and to the Ocean developers I bothered along the way. Both of my supervisors were available at every turn, and I never felt unsupported. What no one could have prepared me for was the nature of the work itself. I knew the final goal, but most of the time, after I broke the goal into pieces, I had no idea where any single piece was taking me. The exploratory nature of the thesis meant I was developing the problem and its solution at the same time, swimming in an endless ocean with no sense of direction. That, more than any tool or bug, was the real challenge.

The size of what I was reaching for scared me from the start. I never wanted to just optimize some code; anyone can do that, and it is not special. I wanted something general, and given that Kubernetes is already heavily studied, I kept questioning if my approach even made sense or lead to any concrete result. Early on, while I was building the monitor and the driver and picking through Ocean's code for what I could reuse, I was making good progress and still dreading the step I knew was coming: looking at my own results and finding a real solution. Some days I did not believe I could.

If I could do one thing differently, it would be to not think too far ahead, and take all the help I can get. Solve one problem at a time, and stop being afraid of the end goal.

The BSc has been genuinely eye-opening. More than any single topic, it taught me how to learn, and where to go to chase the things I want. Capisoft gave me a real taste of the industry, and I would like to stay close to Kubernetes, though at that level rather than this low and purely technical one. One problem at a time, still.