Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Reactive, Push-Based Autoscaling in Container Orchestration Frameworks

Author:
Rushil Patel

Supervisor:
Dr. Marios Kogias

Second Marker:
Dr. Lluis Vilanova

January 25, 2024

Abstract

!!!!!

Acknowledgements

!!!!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The developments in containerisation have significantly heightened the relevance and application of container orchestration tools. One of the key features provided by these tools is autoscaling, a dynamic method for adjusting resource allocation for containers in response to fluctuating workloads[4]. A review of novel autoscaling techniques described in recent academic literature and classic autoscaling techniques used in industry-standard tools reveals a limitation: the inability to respond immediately to sudden workload changes due to the reliance on periodic monitoring and centralised aggregation of container utilisation metrics [5][6][7][8], a process we will describe as "pull-based autoscaling". Some container orchestration tools, such as Docker Swarm[9], do not provide autoscaling as a feature at all.

The adherence to Quality of Service (QoS) metrics and Service Level Agreements (SLAs) are crucial for maintaining high service availability. This aspect must be considered when using container orchestration tools. The absence of efficient, reactive autoscaling mechanisms can especially undermine service reliability under varying workloads. The conventional "pull-based autoscaling" method for containers across a distributed network, irrespective of their actual need for autoscaling, also introduces unnecessary latency and operational costs. This highlights the need for a more decentralised approach, where selectively monitoring individual host machines is sufficient when considering a well-functioning load balancer. The current paradigm also relies on user-space applications for metric collection, which lack the granularity and performance efficacy required for real-time monitoring[5].

Through implementing a reactive, "push-based autoscaling" approach for Docker Swarm, where autoscaling decisions are made immediately based on the current container workload without needing to poll all containers in the system periodically, we are presented with the opportunity to explore a novel point in the design space that not only addresses the challenges for "pull-based autoscaling", but also sets a new benchmark for efficiency and responsiveness in container orchestration environments.

1.2 Objectives

This project aims to implement a reactive, "push-based" autoscaler for Docker Swarm. We undertake this project with the following primary objectives:

1. Implement autoscaling capabilities for Docker Swarm.

2. Establish a methodology for autoscaling that monitors metrics from a single host machine, thereby reducing autoscaling complexity inherent in multi-node systems.
3. Create a highly responsive autoscaler capable of instantaneously detecting real-time changes in workload demands.
4. Improve the granularity and efficiency of metric collection by moving beyond the traditional userspace-application solutions currently used.

1.3 Contributions

With this project, we have (hopefully) made the following contributions:

- Implement autoscaling functionality for Docker Swarm, which uses kernel-level structures for monitoring container utilization on a single host.
- Evaluate our implementation of a reactive "push-based" autoscaler for Docker Swarm against another industry standard container orchestration tool.
- Make the project open-source, available on GitHub.

need to link

1.4 Ethical Considerations

While the environmental impact of computation projects is an important consideration, particularly in terms of electricity consumption, this project's scale significantly limits its ecological footprint. The small-scale evaluation setup, not intended for prolonged operation, minimises energy consumption, aligning the research with sustainable practices.

This project is purely computational and technical, focusing solely on developing a novel autoscaling system. It does not involve human or animal participants. Personal or sensitive data is not collected, processed or stored. This project also uses open-source software and libraries freely available via standard package managers. As a result, there are no ethical concerns involving licensing or data processing.

The focus of this project is specifically on the realm of autoscalers within container orchestration tools. While the potential for dual use or misuse in different contexts exists, such applications are beyond this project's intended scope and direct outcome.

Chapter 2

Background

In this chapter, we:

- provide an overview of containerisation and the motivation behind using containers, and introduce some of the constructs used in their implementation in Linux (Section 2.1).
- describe the use of container orchestration tools, and introduce some of the more widely-used frameworks.
- explore the different types of autoscaling used by container orchestration tools.
- provide a review of some novel autoscaling techniques.
- describe the use of virtual filesystems and the eBPF framework for container metric collection.

2.1 Containerisation

Containerisation refers to encapsulating a software system and its dependencies into an isolated package in a specific way, called a container. Reproducible environments can be deployed and scaled reliably across differing computing infrastructure, regardless of underlying operating systems, due to the encapsulation containerisation provides.

2.1.1 Container Implementation

Containers are implemented using an amalgamation of UNIX constructs to fulfil the robust isolation requirements needed for reliability and reproducibility.

pivot_root - a safer chroot

Filesystem isolation is a crucial aspect to consider for the implementation of containers, where processes within a container are restricted from accessing the filesystems of the host system or other containers.

`chroot`, a UNIX system call [10], was initially created to change the apparent root filesystem for a process and its children for restricted access to a filesystem, creating an isolated environment. However, `chroot` was not implemented as a security feature, and malicious processes can exit these isolated environments by exploiting the fundamentals of `chroot` [11]. To mitigate against this, `pivot_root` [12] is used to provide filesystem isolation instead. Instead of just changing the apparent root directory, like `chroot` does, `pivot_root` also moves the old root directory to a different location and then unmounts the old root directory, providing a robust solution to ensure the old root directory cannot be accessed after `pivot_root` is called [13].

Control Groups (cgroups)

Control groups (cgroups) are part of the Linux kernel that allows for the limiting, prioritising and accounting of resources for process groups [14]. Where namespaces are concerned with restricting resource visibility for process groups, cgroups ensure resources these process groups use to stay within acceptable, defined boundaries. Cgroups organise processes into hierarchies, allowing for fine-grained resource distribution and limitation at various levels.

The subsystems, or resource controllers, cgroups utilise are as follows:

- CPU - used to set CPU limits or allocate specific CPU time to different process groups.
- Memory - used to limit the amount of memory a group of processes can use, which is crucial for preventing memory starvation on the host system.
- Block I/O (blkio) - controls access to and usage of block devices to limit I/O bandwidth or prioritise I/O requests.
- Network - controls network bandwidth used by particular process groups.

Cgroups are exposed through a virtual filesystem, typically mounted under `/sys/fs/cgroup`. This virtual filesystem provides a way to create, modify and delete cgroups through file operations and configure resource limits and priorities through configuration files [15]. Administrators and monitoring tools can use the information in this virtual filesystem for resource accounting at a per-cgroup level.

Namespaces

Linux namespaces allow for the partitioning of kernel resources so that groups of processes are allocated a set of resources, creating the illusion that each group of processes runs on its own independent system [3].

The namespace API provides three system calls:

- `clone()` - creates a new child process in a new namespace, with control over execution context (e.g., virtual address space) shared between the parent and child process.
- `unshare()` - creates one or more new namespaces and moves the caller into them.
- `setns()` - moves an existing process into another existing namespace.

The eight types of namespaces the Linux kernel currently supports and what they isolate are described below:

Control Group (cgroup)

Cgroup namespaces isolate cgroup hierarchies by providing a restricted view of the cgroup hierarchy to the processes within a namespace [16]. This namespace virtualises the root of the cgroup hierarchy so that processes perceive themselves as at the top of their cgroup hierarchy. This allows containers to have their own set of cgroup policies for resource management, independent of other containers or the host system.

Inter-Process Communication (IPC)

The IPC namespace provides isolation for communication resources between different groups of processes [17], such as message queues, shared memory segments and semaphore sets [18]. This isolation allows processes to communicate using IPC mechanisms as if on a single system and prevents potential IPC-based attacks between containers [19]. The segregation of IPC resources also simplifies resource management and cleanup when a container is destroyed, avoiding resource leaks.

Network

Network namespaces are used for the isolation of the network stack, where each namespace provides a separate network environment, allowing namespaces to have their own virtual network devices, as well as routing tables and network rules for per container traffic control [20]. Containers in different network namespaces can communicate with each other through network bridge interfaces in the host interface [21], providing flexibility for advanced network configurations. By isolating network interfaces and rules, the breach impact of compromised containers can be limited, enhancing security.

Mount

The mount namespace allows for filesystem isolation [22]. When created, it initially shares the same mounts as the parent namespace, usually the host system. Existing mounts are then changed when necessary, and `pivot_root` is called to set up the root file system for the container. Changes to a mount namespace do not affect other mount namespaces.

Process ID (PID)

When processes are created on UNIX-like systems, they are assigned a unique identifier called a process ID (PID). PID namespaces isolate the PID number space, allowing different PID namespaces to have the same PIDs [23]. The isolation of PIDs enables the set of processes within a container to be suspended/resumed and for the container to migrate to a new host while preserving the PIDs. Processes within a PID namespace can only see and interact with processes in the same namespace.

Time

Time namespaces allow for the isolation of clock values across different sets of processes, where system and monotonic clock values can differ between time namespaces [24]. Containers can use this to adjust time settings for debugging or simulation purposes without affecting other containers or the host system.

User

The user namespace allows for the remapping of user IDs (UID) and group IDs (GID) from the host to the namespace [25]. This will enable processes to operate as root within the namespace without the need for privileges on the host system, mitigating security risks if a process escapes the namespace. User namespaces also allow for fine-grained control over process capabilities [26], which can be necessary for access control within the container.

UNIX Time-Sharing (UTS)

The UTS namespace allows hostname and Network Information Service (NIS) isolation [27]. UTS namespaces enable containers to change hostnames and be part of different NIS domains. This proves helpful in scenarios where an application's behaviour depends on these settings without affecting other containers or the host system. Isolated hostnames also help simplify network management and service discovery.

Namespace Type	**Isolates**
cgroup	cgroup root directory
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
Time	Boot and monotonic clocks
User	User and group IDs

Table 2.1: Types of namespaces and what they isolate. [3]

2.1.2 Container Runtimes

A container runtime is software concerned with the creation, execution and deletion of containers on a host operating system. It is responsible for handling the entire lifecycle of a container. Container runtimes provide a reliable and reproducible way to configure the necessary constructs required for running containers [28].

runc

Runc is a lightweight, low-level container runtime used for spawning and running containers in line with the Open Container Initiative (OCI) specification [29]. Runc handles the creation and initial configuration of the namespaces and cgroups needed for a container.

containerd

Containerd is a widely-used container runtime that uses runc as a core component for creating and running containers[30]. As a result, containerd can indirectly interface with cgroups and namespaces. It also offers a broader feature set, including container image management and automated network environment setup. It is the core component in Docker - a popular container management platform [31].

CRI-O

CRI-O is a lightweight, open-source container runtime used as an alternative to containerd [32]. Similarly, CRI-O uses runc as a core component for the same reasons as containerd but is entirely OCI compliant. CRI-O has been developed and tailored for Kubernetes - a popular container orchestration tool.

2.2 Container Orchestration

Container orchestration tools provide a framework that automates container deployment, management, scaling and networking. While container runtimes are concerned with the running and execution of individual containers, container orchestration tools manage the operation of multiple containers and their interactions within a larger system architecture [33].

By coordinating container deployment across multiple host machines, container orchestration tools can manage container lifecycles in the following ways [34]:

- Scheduling: Allocating containers to target host machines based on defined resource constraints.
- Load Balancing and Service Discovery: Identifying and connecting containers to provide a holistic view of the entire architecture and distributing network traffic among containers to optimise resource usage across multiple hosts.
- Automated Scaling (Autoscaling): Dynamically adjusting the number of container instances based on an application's demands is crucial for handling varying loads.
- High Availability and Fault Tolerance: Distributing containers across different host machines, ensuring availability in the event of container failure.
- Health Monitoring: Monitoring container performance and automatically deploying new containers when one fails.
- Declarative Configuration: Allowing the user to declare the desired state of the application, where the container orchestration tool will take necessary actions to maintain this state.
- Networking: Managing the communication between containers and external networks.

2.2.1 Kubernetes (K8s)

Kubernetes is an open-source container orchestration platform [35]. Initially developed by Google, it is now maintained by the Cloud Native Computing Foundation. It has become the de facto standard for container orchestration due to its large community and ecosystem. It uses CRI-O at its core as its container runtime engine.

2.2.2 Docker Swarm

Docker Swarm is a container orchestration tool part of the Docker ecosystem. Swarm allows users to manage a cluster of Docker containers as a single system [9]. It uses containerd as its container runtime engine. Swarm is more straightforward to configure and use than Kubernetes but lacks features like autoscaling, is less robust and is not as highly available.

2.2.3 HashiCorp Nomad

Nomad, developed by HashiCorp, is a flexible workload orchestration tool that can handle containerised and non-containerised applications [36]. Nomad is tailored to handle a diverse range of workloads by utilising a variety of container runtimes through plugins, such as runc and Docker. Nomad has a reduced feature set compared to Kubernetes but integrates seamlessly with other HashiCorp tools for secret management [37] and networking [38].

2.3 Autoscaling in Container Orchestration Tools

Automated scaled, or autoscaling, is a technique used by container orchestration tools to dynamically adjust both the allocation of resources and the distribution of container instances across a cluster [4]. This scaling is based on various factors, including resource demand, network latency and host capacity. Its primary objective is to enhance the Quality-of-Service (QoS) for users of an application, ensuring optimal performance and efficiency in response to varying workloads.

Autoscaling is crucial for resource optimisation, where scaling containerised applications during demand spikes and low usage periods optimises the overall cost and performance of infrastructure, and resources are used in conjunction with dynamic workloads. Google's Autopilot [6] is an example of an autoscaling implementation for resource management as part of their in-house container orchestration tool Borg [39] for their public and private Cloud systems, where QoS is paramount. Through autoscaling, applications can also be considered highly available and reliable, as various replica instances can be deployed depending on load conditions.

2.3.1 Horizontal Autoscaling

Horizontal autoscaling, or "scale-out" autoscaling, is an autoscaling technique that involves increasing or decreasing the number of container instances in response to workload changes. By changing the number of container instances, incoming network traffic or computational workload can be load-balanced, preventing any single instance from becoming a bottleneck. Horizontal autoscaling can also respond to variable traffic in applications. It can effectively optimise operational costs by scaling the number of container instances to match what the current workload requires.

Container orchestration tools typically implement horizontal autoscaling by querying individual containers for utilisation metrics. By continuously monitoring metrics like CPU, memory or custom metrics relevant to the application's performance, the orchestration can automatically adjust the number of container instances when predefined thresholds are crossed, which are defined in user-defined autoscaling policies [40].

The autoscaler controller implemented within Kubernetes uses the average or raw value for a container-reported metric to produce a ratio to calculate the number of desired container instances, which is defined as follows [41]:

```
desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]
```

Kubernetes uses a central metrics server, which polls containers for metrics every 60 seconds by default [5]. The autoscaling controller then, in turn, polls the Kubernetes metrics server every 15 seconds by default. As a result, autoscaling decisions are not made in real-time when metrics thresholds are crossed for individual containers. To avoid thrashing, where the autoscaler continuously fluctuates between scaling up and scaling down a container, Kubernetes also allows for a stabilisation window to scale gradually over a defined period.

2.3.2 Vertical Autoscaling

Vertical autoscaling, or "scale up" autoscaling, automatically adjusts compute resources, such as CPU and memory, allocated to a container [42]. Unlike horizontal scaling, which changes the number of containers, vertical scaling changes the amount of resources allocated to each container. Google's Kubernetes Engine (GKE) uses Autopilot at its core [6] and offers vertical autoscaling functionality as an extension of the Kubernetes autoscaler. Autopilot can provide recommended values for CPU and memory limits and automatically update the vertical autoscaling policies.

Limitations with Kubernetes mean the only way to modify the resource requests of a running container is to destroy and recreate that container, which can temporarily disrupt the service for the end-user. RUBAS, a product of recent research to avoid this problem, is a system that enhances vertical autoscaling by integrating non-disruptive vertical scaling through container migration [43]. By using Checkpoint/Restore in Userspace (CRIU) for container states, RUBAS presents an opportunity for resource reallocation and vertical autoscaling without restarting the application.

2.4 Novel Autoscaling Approaches

Novel autoscaling techniques referenced in literature today can be broadly classified into proactive and reactive autoscaling.

2.4.1 Proactive Autoscaling

Proactive autoscaling refers to a category of autoscaling techniques where resource scaling decisions are made in anticipation of future demand changes rather than reacting to past or current demand. By predicting workload trends using historical data from previous workloads, an autoscaler can scale resources in advance to handle the expected load and minimise latency and performance issues. Proactive autoscaling is especially effective with predictable workload patterns.

Fuzzy Logic

Fuzzy logic extends classic truth values so that the value of variables can be any floating-point number between 0 and 1. In the context of proactive autoscalers, it is well-suited for dealing with specific and ambiguous data, such as slightly elevated CPU usage. The vagueness and variability fuzzy logic brings to metrics allow proactive autoscalers to make more nuanced scaling decisions, where the concept of partial truth for metrics may be more beneficial [44].

Time-Series Analysis

Time-series analysis in proactive autoscaling refers to using historical data analysis to manage the scaling of containers within a system predictively. Through time-series analysis, a proactive autoscaler can identify trends, cyclic patterns and seasonal patterns of the consumption of resources over time from historical data [45]. These predictions help avoid resource under/over-utilisation by allocating the necessary resources before the system becomes overloaded [46].

Various time series forecasting models, such as Holt-Winters exponential smoothing and Long-Short-Term Memory (LSTM) machine learning models, enhance the precision of these predictions [47], enabling more robust scaling decisions when compared to the baseline autoscaling techniques.

Reinforcement Learning (RL)

RL, a subset of machine learning (ML), trains software to make decisions by interacting with an environment [48]. In the context of proactive autoscaling, RL is used to dynamically adjust resources in response to predicted changes in a workload by observing the cluster's state and an autoscaling decision's expected reward.

RL-based autoscalers receive a high reward for optimising resource utilisation, adherence to QoS metrics, and making future autoscaling decisions to maximise the cumulative reward based on past decisions. RL requires a well-defined reward system, which can be complex to design for autoscaling scenarios if Service Level Agreements (SLAs) are not violated. RL algorithms also need sufficient time to converge to an optimally trained point to make autoscaling decisions effectively. This additional complexity adds utilisation overhead when comparing the resource utilisation of two novel RL-based autoscalers, DScaler and A-SARSA [49], against the standard Kubernetes Horizontal Autoscaler.

DScaler [49] and A-SARSA [50] both implement their reward system around minimising SLA violations and are both effective in reducing the SLA violation rate, therefore autoscaling response

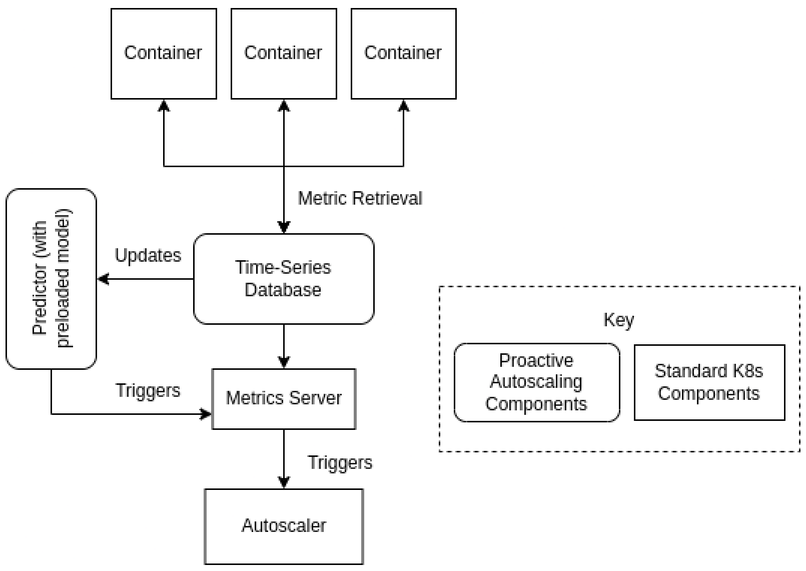


Figure 2.1: The general architecture of proactive autoscalers.

time, when compared against the standard Kubernetes Horizontal Autoscaler.

Queuing Theory

Queueing theory, in the context of proactive autoscaling, is an analytical approach used to analyse and predict the behaviour of workloads and data traffic within a system. By understanding the patterns of requests and the system's processing capabilities, a proactive autoscaler that utilises queuing theory can predict future demand based on historical data and current trends.

Applications of queuing theory use non-homogeneous Poisson processes (NHPP) to describe a series of events occurring randomly over time [51]. An NHPP will obtain historical data by continuously monitoring the system for incoming requests, processing times and other relevant metrics. Stochastically constrained optimisation is then applied by the proactive autoscaler to solve the probabilistically defined optimisation problems generated from NHPP to inform scaling decisions to optimise resource costs and QoS [52].

Experiments conducted when evaluating RobustScaler, a queuing theory-based autoscaler, show that resource adjustment through analytical modelling without over-provisioning individual containers outperforms common baseline autoscaling strategies [53].

2.4.2 Reactive Autoscaling

Reactive autoscaling refers to a category of autoscaling techniques where resource scaling decisions are made in response to real-time workload changes, making adjustments after changes in demand are observed. As a QoS violation needs to occur before autoscaling decisions can be made, reactive autoscalers suffer from a delayed response between a spike in demand and the corresponding scaling action; a challenge proactive autoscalers do not have to solve [54]. As a result, current literature is focused on developing novel techniques with proactive autoscalers.

2.4.3 Hybrid Approaches

Hybrid autoscaling is an approach that combines both proactive and reactive autoscaling strategies to optimise resource allocation in container orchestration systems. This method utilises proactive autoscaling's ability to anticipate future resource demands through predictive algorithms and historical data analysis, enabling it to scale resources before expected load increases. The reactive component of hybrid autoscaling monitors real-time system metrics and reacts by adjusting resources to match immediate demands when predefined thresholds are exceeded. Hybrid autoscaling, therefore, offers a more adaptable and efficient solution to the challenges of maintaining service quality and operational efficiency.

An example of a novel hybrid autoscaling approach was presented by Pereira et al. [55], where reactive, threshold-based scaling is used to handle immediate workload changes and proactive time-series forecast is used for predicting future demands to optimise system throughput and CPU utilisation. By incorporating reactive components, the hybrid model was able to respond to sudden workload changes, which is a challenge for purely proactive models.

Chameleon, presented by Bauer et al. [8], presents a slightly different approach, where time-series analysis for long-term predictions is used as a primary autoscaling mechanism, and a reactive fallback mechanism is used for immediate scaling decisions. A hybrid approach was efficient in handling sudden surges in demand.

Abdullah et al. introduced a novel hybrid autoscaler which could predict and respond to sudden increases in workload (bursts) [7]. It used a reactive approach for making scale-out decisions and a "regression-based predictive component" for scale-in decisions to adjust the number of containers in the system dynamically. The reactive policy dynamically adds one container instance whenever the 95th percentile of an application's response time exceeds a defined threshold so that autoscaling performance traces can be collected to build the initial resource prediction model, namely the proactive component of the autoscaler.

An important aspect of hybrid autoscalers is their complexity and operational overhead. The proactive component relies on data analysis and predictive algorithms, which can be computationally intensive. Additionally, the operational delay is still a problem with the reactive element for sudden workload changes.

2.4.4 Pull-Based Metric Collection

The explored autoscaling techniques rely on metrics collected at regular intervals from the host's cgroup filesystem for individual container statistics, which help inform autoscaling decisions. These metrics are collected using a "pull-based" approach, where a centralised service periodically polls the cgroup filesystem. This approach introduces a latency issue: extended intervals between successive metric collections can significantly delay the detection and subsequent response to workload fluctuations by reactive or hybrid autoscalers.

The reliance on periodically collecting metrics impedes the system's responsiveness and agility in adapting to dynamic workload conditions, a critical challenge that autoscalers do not currently solve.

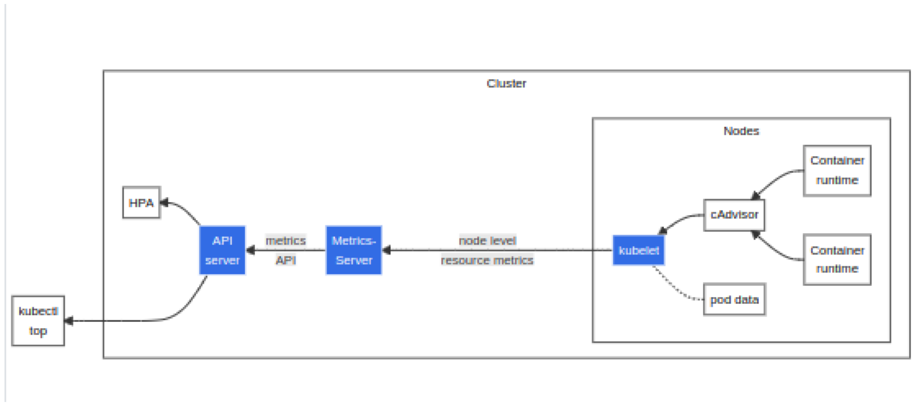


Figure 2.2: Metrics collecting using cAdvisor in K8s [1]

2.5 Monitoring Resource Utilisation

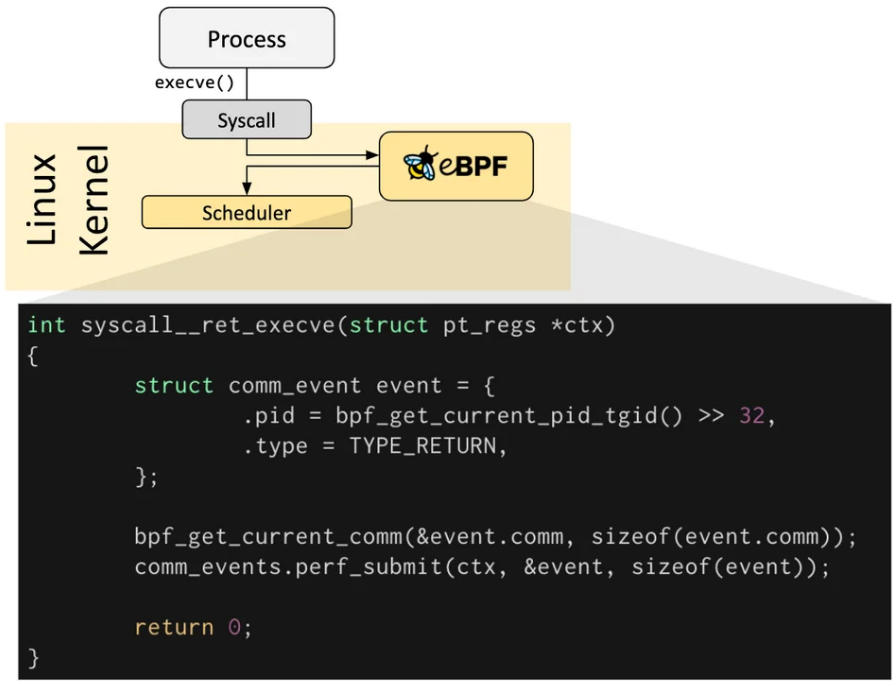
2.5.1 Polling Virtual Filesystems (vfs)

In container orchestration tools, the aggregation and collection of container metrics are carried out by a userspace process part of the container orchestrator's autoscaler. Google's cAdvisor [56] is used as the primary tool for metric collection in Kubernetes. cAdvisor periodically polls the cgroup filesystem from userspace for metrics, which is not suited for immediate autoscaling response. The architecture can be seen in Figure 2.2. Polling the cgroup filesystem takes advantage of the existing cgroup accounting functionality and is information already exposed to userspace [14].

The `/proc` filesystem, a virtual filesystem that provides a window into the kernel's view of each process, can also be used for monitoring resource utilisation [57]. Through interaction with files such as `/proc/meminfo` and `/proc/cpuinfo` for memory and CPU metrics, respectively, applications can poll and extract statistics about the system's current state [58]. These metrics are also exposed at a per-process level.

2.5.2 (extended) Berkeley Packet Filter

(extended) Berkeley Packet Filter, denoted as eBPF, is a powerful tool used for system monitoring and performance analysis in Linux [2]. eBPF allows for the dynamic insertion of custom bytecode into the Linux kernel without needing to change the kernel source code or load kernel modules, therefore offering a highly efficient and secure mechanism for real-time, low-overhead resource utilisation monitoring. eBPF programs can be triggered on kernel events, such as system call execution (as seen in Figure 2.3), which can be used to approximate utilisation metrics such as CPU or memory utilisation in a non-intrusive manner [59].



```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Figure 2.3: eBPF Architecture for hooking onto system call execution.[2]

Chapter 3

Project Plan

Table 3.1 details the project plan.

If kernel level metric collection proves too difficult, a userspace application that polls metrics and makes autoscaling decisions based on our load-balancing assumption still presents a novel contribution.

If there is time to extend the functionality of the autoscaler, we could:

- Implement scaling to 0 when containers are not being used through cgroup freezers.
- Look at implementing another metric for autoscaling, such as network utilisation.
- Improve usability and write detailed documentation for usage.

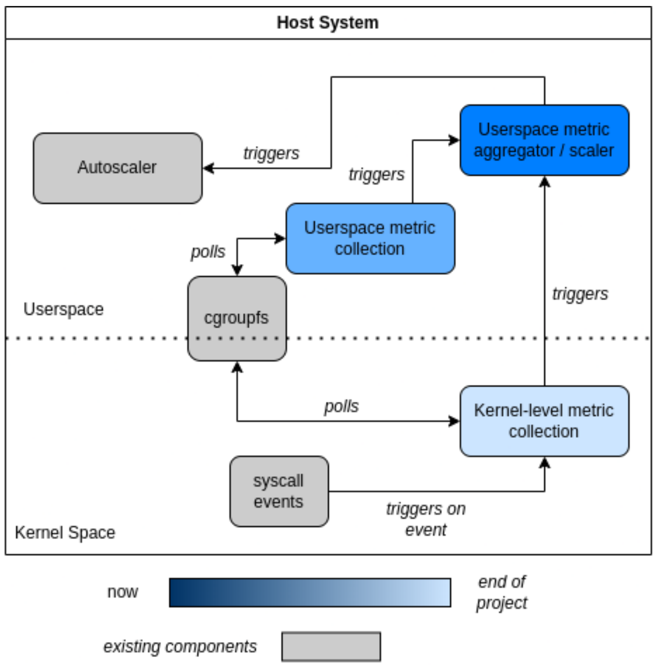


Figure 3.1: Project Architecture Plan

Date Range	**Objectives**
10 Nov - 30 Nov	Explore the Docker Swarm ecosystem and conduct experiments to determine the accuracy of the in-built load balancer. Show load is distributed to a reasonable degree so that our load-balancing assumption for single-node metric collection can be used.
1 Dec - 15 Dec	Survey the existing autoscaling landscape and determine whether "push-based" autoscaling is a novel approach. Understand the different types of autoscaling used in industry and presented in the literature.
16 Dec - 31 Dec	Research kernel-level metric collection, particularly through observing the cgroup filesystem or through counting syscalls. Explore the feasibility of using eBPF to measure CPU and memory utilisation.
1 Jan - 30 Jan	Collate background research, formalise introduction and outline evaluation and project plans for the interim report.
1 Feb - 15 Feb	Implement a userspace application that's able to scale a container based on an arbitrary event (syscall/file creation).
16 Feb - 28 Feb	Implement a decoupled metric collection component in userspace, which periodically polls the cgroup filesystem or counts syscalls for container utilisation metrics.
1 Mar - 31 Mar	Continue with feasibility research on using eBPF for CPU / memory metrics. If feasible, start to detail a plan on how this can be done. If not, potentially fall back to implementing a loadable kernel module (LKM).
1 Apr - 20 Apr	Implement the kernel level metric collector and integrate it into the existing autoscaler.
21 Apr - 17 May	Exam season - put the project on hold. Spare time in this period is helpful in fixing implementation issues.
18 May - 30 May	Develop and utilise the necessary benchmarks needed for the evaluation of the autoscaler.
1 Jun - 20 Jun	Write up findings for the final report and create the project presentation.

Table 3.1: Project Timeline and Objectives

Chapter 4

Evaluation Plan

Assuming we have managed to implement a "push-based" autoscaler for Docker Swarm, we will have extended the state-of-the-art (SotA), as Docker Swarm does not have autoscaling functionality.

We will also introduce a novel autoscaling architecture that does not use centralised metric aggregation across a network and does not poll at predefined intervals. Therefore, we should be able to make an autoscaling decision almost immediately when a container's utilisation threshold (such as a 100MB memory limit) is crossed.

If time allows, we can also implement features not available in classic autoscalers used in container orchestration tools, such as scaling to zero when container usage is idle.

We will use the DeathStarBench[60] benchmarking suite, tailored for benchmarking microservices. The functionality to demonstrate and the experiments undertaken are as follows:

Functionality to Demonstrate	**Experiments Undertaken**	**Successful Outcome**
"Push-based" Autoscaling Response Time	Set up a Docker Swarm cluster with our autoscaler implementation and a K8s cluster with the standard horizontal autoscaler. Ensure they have the same number of nodes on the same host machines with the same number of replicas deployed. Simulate load to a replica and measure how long it takes for another replica to be deployed once the threshold is crossed.	Docker Swarm deploys the required replica faster than K8s.
Kernel Level Metric Gathering Performance	Similar setup to the above. When a scaling decision is made, compare the resource utilisation of both autoscaling processes and the time taken to execute the autoscaling itself.	Kernel level metric gathering shows lower latency and uses fewer resources.

Table 4.1: Evaluation Plan

Chapter 5

Conclusion

Appendix A

First Appendix

Bibliography

[1] Resource metrics pipeline;. Available from: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/.

[2] What is eBPF? An Introduction and Deep Dive into the eBPF Technology;. Available from: https://ebpf.io/what-is-ebpf/.

[3] namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[4] Datadog. What is Auto-scaling? How it Works & Use Cases; 2022. Available from: https://www.datadoghq.com/knowledge-center/auto-scaling/.

[5] metrics-server/FAQ.md at 69073c08a94ff25ce26e230d38ecb7d9fe257603 · kubernetes-sigs/metrics-server;. Available from: https://github.com/kubernetes-sigs/metrics-server/blob/69073c08a94ff25ce26e230d38ecb7d9fe257603/FAQ.md.

[6] Rzadca K, Findeisen P, Swiderski J, Zych P, Broniek P, Kusmierek J, et al. Autopilot: workload autoscaling at Google. In: Proceedings of the Fifteenth European Conference on Computer Systems. Heraklion Greece: ACM; 2020. p. 1-16. Available from: https://dl.acm.org/doi/10.1145/3342195.3387524.

[7] Abdullah M, Iqbal W, Berral JL, Polo J, Carrera D. Burst-Aware Predictive Autoscaling for Containerized Microservices. IEEE Transactions on Services Computing. 2022 May;15(3):1448-60. Available from: https://ieeexplore.ieee.org/document/9097467.

[8] Bauer A, Herbst N, Spinner S, Ali-Eldin A, Kounev S. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. IEEE Transactions on Parallel and Distributed Systems. 2019 Apr;30(4):800-13. Available from: https://ieeexplore.ieee.org/document/8465991.

[9] Swarm mode overview; 2023. Available from: https://docs.docker.com/engine/swarm/.

[10] chroot(1) - Linux man page;. Available from: https://linux.die.net/man/1/chroot.

[11] Garfinkel T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. 2003 Jan.

[12] pivot_root(8): change root file system - Linux man page;. Available from: https://linux.die.net/man/8/pivot_root.

[13] runc/libcontainer/rootfs_linux.go at cdff09ab875159d004035990c0d45e8bdf20ed35 · opencontainers/runc;. Available from: https://github.com/opencontainers/runc/blob/cdff09ab875159d004035990c0d45e8bdf20ed35/libcontainer/rootfs_linux.go#L1041.

[14] Jain SM. Cgroups. In: Jain SM, editor. Linux Containers and Virtualization: Utilizing Rust for Linux Containers. Berkeley, CA: Apress; 2023. p. 47-81. Available from: https://doi.org/10.1007/978-1-4842-9768-1_4.

[15] cgroups.txt;. Available from: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[16] cgroup_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html.

[17] ipc_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html.

[18] sysvipc(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/sysvipc.7.html.

[19] Vulnerabilities in synchronous IPC designs | IEEE Conference Publication | IEEE Xplore;. Available from: https://ieeexplore.ieee.org/document/1199341.

[20] network_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[21] bridge(8) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man8/bridge.8.html.

[22] mount_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.

[23] pid_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[24] time_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/time_namespaces.7.html.

[25] user_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[26] capabilities(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[27] uts_namespaces(7) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man7/uts_namespaces.7.html.

[28] Demystifying containers - part II: container runtimes; 2019. Available from: https://www.cncf.io/blog/2019/07/15/demystifying-containers-part-ii-container-runtimes/.

[29] opencontainers/runc: CLI tool for spawning and running containers according to the OCI specification;. Available from: https://github.com/opencontainers/runc/.

[30] containerd;. Available from: https://containerd.io/.

[31] What is containerd ? | Docker;. Available from: https://www.docker.com/blog/what-is-containerd-runtime/.

[32] cri-o;. Available from: https://cri-o.io/.

[33] What Is container orchestration;. Available from: https://cloud.google.com/discover/what-is-container-orchestration.

[34] Demystifying Container Orchestration: A Beginner's Guide | SUSE Communities;. Available from: https://www.suse.com/c/rancher_blog/demystifying-container-orchestration-a-beginners-guide/.

[35] Kubernetes Overview;. Available from: https://kubernetes.io/docs/concepts/overview/.

[36] Introduction | Nomad | HashiCorp Developer;. Available from: https://developer.hashicorp.com/nomad/intro.

[37] Introduction | Vault | HashiCorp Developer;. Available from: https://developer.hashicorp.com/vault/docs/what-is-vault.

[38] What is Consul? | Consul | HashiCorp Developer;. Available from: https://developer.hashicorp.com/consul/docs/intro.

[39] Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-scale cluster management at Google with Borg. In: Proceedings of the Tenth European Conference on Computer Systems. Bordeaux France: ACM; 2015. p. 1-17. Available from: https://dl.acm.org/doi/10.1145/2741948.2741964.

[40] Nguyen TT, Yeom YJ, Kim T, Park DH, Kim S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. Sensors. 2020 Jan;20(16):4621. Available from: https://www.mdpi.com/1424-8220/20/16/4621.

[41] Horizontal Pod Autoscaling;. Available from: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[42] Vertical Pod autoscaling | Google Kubernetes Engine (GKE) | Google Cloud;. Available from: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler.

[43] Rattihalli G, Govindaraju M, Lu H, Tiwari D. Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD); 2019. p. 33-40. ISSN: 2159-6190. Available from: https://ieeexplore.ieee.org/abstract/document/8814504.

[44] Arabnejad H, Jamshidi P, Estrada G, El Ioini N, Pahl C. An Auto-Scaling Cloud Controller Using Fuzzy Q-Learning - Implementation in OpenStack. In: Aiello M, Johnsen EB, Dustdar S, Georgievski I, editors. Service-Oriented and Cloud Computing. Lecture Notes in Computer Science. Cham: Springer International Publishing; 2016. p. 152-67.

[45] Khan MNAH, Liu Y, Alipour H, Singh S. Modeling the Autoscaling Operations in Cloud with Time Series Data. In: 2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW); 2015. p. 7-12. Available from: https://ieeexplore.ieee.org/document/7371434.

[46] ; Microsoft Technology Licensing LLC, assignee. Predictive autoscaling in computing systems. US10476949B2; 2019. Available from: https://patents.google.com/patent/US10476949B2/en.

[47] Wang T, Ferlin S, Chiesa M. Predicting CPU usage for proactive autoscaling. In: Proceedings of the 1st Workshop on Machine Learning and Systems. EuroMLSys '21. New York, NY, USA: Association for Computing Machinery; 2021. p. 31-8. Available from: https://doi.org/10.1145/3437984.3458831.

[48] What is Reinforcement Learning? - Reinforcement Learning Explained - AWS;. Available from: https://aws.amazon.com/what-is/reinforcement-learning/.

[49] Xiao Z, Hu S. DScaler: A Horizontal Autoscaler of Microservice Based on Deep Reinforcement Learning. In: 2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS); 2022. p. 1-6. ISSN: 2576-8565. Available from: https://ieeexplore.ieee.org/document/9919994.

[50] Zhang S, Wu T, Pan M, Zhang C, Yu Y. A-SARSA: A Predictive Container Auto-Scaling Algorithm Based on Reinforcement Learning. In: 2020 IEEE International Conference on Web Services (ICWS); 2020. p. 489-97. Available from: https://ieeexplore.ieee.org/document/9284122.

[51] Nonhomogeneous Poisson Process - an overview | ScienceDirect Topics;. Available from: https://www.sciencedirect.com/topics/mathematics/nonhomogeneous-poisson-process.

[52] Pan Z, Wang Y, Zhang Y, Yang S, Cheng Y, Chen P, et al. MagicScaler: Uncertainty-Aware, Predictive Autoscaling. Proceedings of the VLDB Endowment. 2023 Sep;16:3808-21.

[53] Qian H, Wen Q, Sun L, Gu J, Niu Q, Tang Z. RobustScaler: QoS-Aware Autoscaling for Complex Workloads. arXiv; 2022. ArXiv:2204.07197 [cs]. Available from: http://arxiv.org/abs/2204.07197.

[54] Podolskiy V, Jindal A, Gerndt M. IaaS Reactive Autoscaling Performance Challenges. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD); 2018. p. 954-7. ISSN: 2159-6190. Available from: https://ieeexplore.ieee.org/document/8457912.

[55] Pereira P, Araujo J, Maciel P. A Hybrid Mechanism of Horizontal Auto-scaling Based on Thresholds and Time Series. In: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC); 2019. p. 2065-70. ISSN: 2577-1655. Available from: https://ieeexplore.ieee.org/document/8914522.

[56] google/cadvisor. Google; 2024. Original-date: 2014-06-09T16:36:33Z. Available from: https://github.com/google/cadvisor.

[57] proc(5) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man5/proc.5.html.

[58] top(1) - Linux manual page;. Available from: https://man7.org/linux/man-pages/man1/top.1.html.

[59] bcc/tools/cpudist.py at aa9f9e6f27626743d2681c9b1a58d04224a28adc · iovisor/bcc;. Available from: https://github.com/iovisor/bcc/blob/aa9f9e6f27626743d2681c9b1a58d04224a28adc/tools/cpudist.py.

[60] Gan Y, Zhang Y, Cheng D, Shetty A, Rathi P, Katarki N, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '19. New York, NY, USA: Association for Computing Machinery; 2019. p. 3-18. Available from: https://dl.acm.org/doi/10.1145/3297858.3304013.