Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Bachelor of Science HES-SO in Engineering
Bd de Pérolles 80
CH-1700 Fribourg

Bachelor of Science HES-SO in Engineering

Department: Computer Science and Communication Systems

Gestion de la mémoire RAM dans les conteneurs

Author:
Martin Roch-Neirey

Under the supervision of:
Sébastien Rumley
Nicolas Schroeter

Written as part of Semester 6 project

Fribourg, HES-SO//Bachelor, May 16, 2024
Version: 1.0

History

Version	Date	Modification
0.0.0	2024.03.02	Document creation
0.0.1	2024.04.15	Work on analysis and start of many parts
0.0.2	2024.04.18	Work on entire document
0.0.3	2024.05.02	Work on Design
0.1.0	2024.05.04	End of design
0.2.0	2024.05.09	End of Analysis, results and interpretation
0.2.1	2024.05.14	Results and interpretation
0.3.0	2024.05.15	End of results and interpretation
1.0	2024.05.16	End of implementation, conclusion, final review

Table 1: Version history

Contents

List of Figures

List of Tables

1 Introduction

This document is the final report of the semester project entitled "Gestion de la mémoire RAM dans les conteneurs", carried out as part of a semester 6 project at the Haute Ecole d'Ingénierie et d'Architecture of Fribourg. This work was produced by Martin Roch-Neirey.

1.1 Context

Modern computer architectures are composed of various complex components that work in conjunction to deliver performance. One of these essential components, the Random Access Memory (RAM), ensures that the computer can efficiently access and manipulate data, providing the necessary speed and flexibility to execute tasks and process information in real time.

On the other hand, applications are becoming increasingly containerized, a trend driven by the demand for more scalable, portable, and resource-efficient application deployment. This shift raises important questions about the optimization of resources, particularly in how RAM is managed to maintain efficient performance and isolation between workloads. While containerization offers significant advantages in terms of scalability and resource management, it also introduces complexities in RAM utilization. Containers share the host operating system's kernel but operate in isolated user spaces, leading to unique challenges in memory allocation and management.

Therefore, efficient RAM management is essential to ensure that each container has sufficient memory to perform optimally without encroaching on the memory required by other containers on the same host. This balance is critical in preventing performance bottlenecks and ensuring that system resources are utilized effectively, thus enabling high performance across all deployed workloads.

This project aims to address numerous questions regarding how RAM is managed in containerized environments, focusing specifically on Docker native environments and Kubernetes clusters.

1.2 Objectives

This project aims to study and understand how both the Docker Engine and Kubernetes handles Random Access Memory management of their workloads, and how the workloads see the context imposed on them, with their RAM usage and the limit they must not exceed. The scope of the project is quite large, with a focus on the allocation, limitations, and monitoring topics. As this is considered an academic project, it should give answers (or partial answers) to many of the questions and hypotheses written below. The following objectives reflect these questions and hypotheses.

Each of the following goals concerns both the Docker Engine and Kubernetes. The main goals of this project are:

- Study the aspect of RAM memory in containers/pods in general.
- Understanding how RAM limits are imposed on containers/pods.
- Answer the following questions:
  - How can you strictly limit the memory resources of a container/pod to ensure that there are no overflows? What happens when this limit is reached and then eventually exceeded?
  - Is it possible for a container to query the Docker daemon to obtain information about memory? Same question from outside the container.
  - As far as security is concerned, is it possible to prevent a container from seeing the memory it uses? Same question from outside the container.
  - How do the Docker engine and Kubernetes behave when there are too many instances running (or being deployed) compared to the allocatable resources?

These questions will be answered using different tools, like Docker, Kubernetes, and monitoring solutions. In addition to these key objectives, there are two secondary objectives:

- Study and list the various "containers as a service" solutions, explaining the aspects of RAM-based billing and how suppliers manage RAM quotas and potential overruns.
- Replicate this study on other containerisation engines (podman, containerd, rkt, etc.) and compare the results.

1.3 Structure and information about this report

This report is organized in 6 chapters, from the introduction to the conclusion:

1. The chapter 2 invites the reader to discover different information on RAM, and how engines like Docker and Kubernetes manages their workloads, limit their RAM usage... This chapter also contains the presentation of different tools used in this project to build the infrastructure and monitor the results. The end of Chapter 2 looks at how public cloud providers manage RAM and charge users for it.

2. Next, the chapter 3 shows the design of the application developed to allocate and deallocate RAM, and the infrastructure deployed to execute the experiments.

3. The chapter 4 explains how the application has been developed and how the infrastructure has been deployed.

4. The penultimate chapter shows the testing strategies and procedures, and the results associated to each one. It also presents some ideas for experiments that were not carried out as part of this project.

5. Finally, chapter 6 presents the overall conclusion of the project.

At the beginning of each chapter, its structure is explained. Some of the technical vocabulary is explained in the glossary at the end of this document. Each word inserted in the glossary is underlined the first time it appears in the document.

Scope of the study

This study focuses solely on the behaviour of the Docker engine and Kubernetes when run on Linux environments. All the machines used in this project to run the workloads are running Ubuntu 20.04 LTS.

This report assumes that the reader already has some knowledge about virtualisation, containerisation, Linux kernel, processes, Docker, Kubernetes and RAM in general.

This report was written by hand with partial assistance from DeepL and ChatGPT (GPT-4 and GPT-4o models). A declaration of honour made by the author is available at the end of the conclusion chapter.

2 | Analysis

This chapter describes the main elements used in this project, from a reminder of the RAM principle to more in-depth explanations of Kubernetes and Docker. It is important to understand from a theoretical point of view the behaviour of Docker and Kubernetes. The testing procedures defined in the design section of this report may be based on official information from both documentation, described as hypotheses that may be verified during the tests. Each testing procedure contains an hypothesis, as described in the chapter 5. Note that during the project, a short analysis has been made on a solution called "parca.dev". This solution has not been implemented, so its analysis in available in the Appendix C.

Contents

2.1 General description of Random Access Memory

Random Access Memory (RAM) is a fundamental component in all modern computing systems, acting as a short-term data storage that a device's processor can access directly and rapidly. Unlike other storage devices such as hard drives, RAM is volatile, which means it loses all stored information when the power is turned off. The term "random access" refers to the ability of the system to access any memory cell directly if the row and column that intersect at that cell are known. This direct access capability enables high-speed data retrieval and storage, which is essential for the efficient performance of modern applications and operating systems

The computer stores information in registers, each one identified by a unique address. To store information, the computer select the address of the desired register, and pushes data to it with specific pins. A control pin (often called "write control signal") is used to read or write data to the selected register. Here is an example of a simple 8*4 bit RAM device.

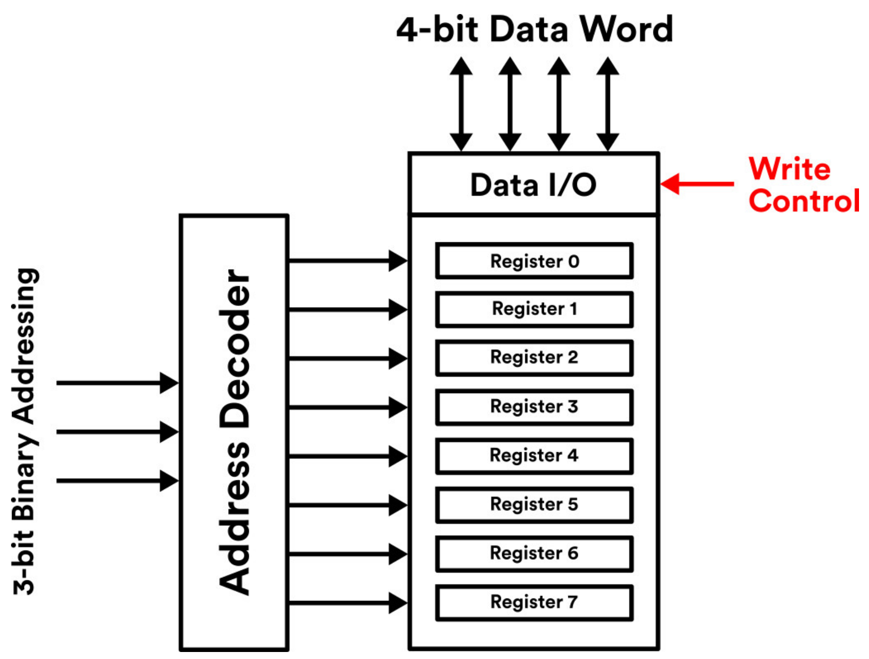


*Figure 2.1: Diagram of an 8*4 bit RAM device (Arith-Matic[1])*

The address decoder converts a 3 bit binary address into 8 different registers. Each register is then accessible using a single address. When the write control signal is enabled, the data from above are pushed in the register selected by the address decoder. When this signal is disabled, the data is read from the register, into the 4-bit data word.

A RAM strip can be up to 32GB in size, which enables modern computer to address huge quantity of RAM when having multiple stripes. For example, high-performance computers can have multiple terabytes of RAM to compute a lot of information quickly.

2.1.1 Virtual and swap memory

It is possible for a modern computer to virtually extend it's RAM size, by using a portion of it's hard drive usually used to store non-volatile information for longer duration. When the computer's RAM is almost full, it can move some information located in its RAM on the hard drive to free some space. The information stored on the hard disk can later be reinserted into RAM when one of the following two situations occurs::

1. The operating system needs to access it for any reason. It will then swap RAM data with the hard drive data (this operation is called swapping).

2. The initial RAM space has once again free space. The virtual memory space located on the hard drive is then no longer used.

An abusive usage of the virtual memory, or swapping mechanism can result in thrashing[2], significantly degrading computer performance, as hard disks offer much lower performance than RAM.

In addition to the extension capacity of the RAM using a hard drive, the virtual memory is also a key principle seen from a process point of view. As written before, the "random access" term means that the operating system is accessing a memory block using a unique address. Each process executed on a computer has a given RAM block to store and manipulate data, but this block may not be continuous for multiple reasons. For example, as applications are opened, closed, and run over time, they request and release memory. This frequent allocation and deallocation can lead to fragmentation, where free memory is divided into separate blocks (sometimes distributed among multiple processes) that are not physically contiguous.

Virtual memory allows a process to see a contiguous block of address space that is actually composed of scattered fragments of physical memory or even disk space. This is achieved through a process of memory management where the operating system maintains a mapping table to translate virtual addresses to physical addresses. This mechanism helps in efficiently using RAM and allows a computer to compensate for physical memory shortages by temporarily transferring data to disk storage, as said before.

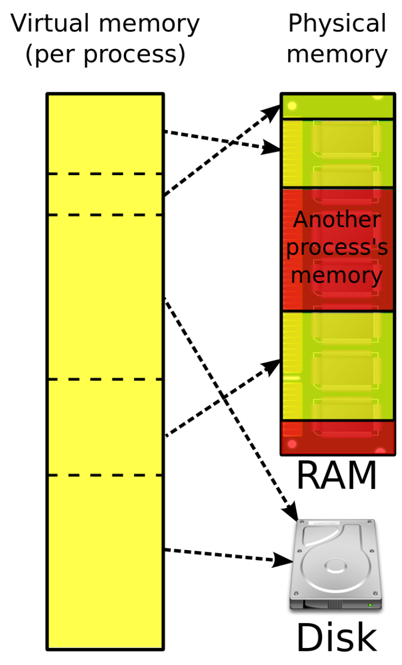


Figure 2.2: Virtual memory combines active RAM and inactive memory on hard drive to form a large range of contiguous addresses (Wikipedia[3])

Even if a portion of a process memory block is located on a hard drive and other portions are distributed in the physical RAM, the process still only sees one memory block with continuous addresses. The operating system is in charge of translating virtual addresses to physical addresses.

2.1.2 RAM issues

One of the main problems with RAM occurs when there is not enough memory available. This issue is called "Out of Memory Exception" (OOME), which occurs when a system or an application tries to claim more RAM than is available or that the system can allocate. This has been a big issue on computers without much RAM, such as IoT devices, or old computers. The swap memory has given a partial answer to this problem, but it is not viable to allocate a lot of memory on a hard drive because of the significantly degrading performance. When an application fails to allocate some memory (for example using the *malloc()* C function), several consequences and behaviors follow, dictated by the program's design and its error handling strategy. One of the biggest issue would be a crash from the program.

When the operating system detects that it is running low on memory and is unable to meet the RAM demands of all running processes, it may initiate the OOMKiller process. It is designed to preserve system stability by forcibly terminating one or more processes that are using a lot of memory. A stable system should never need to run OOMKiller, as it is considered a last resort before the whole system crashes due to lack of memory.

In certain situations, particularly in containerised environments, the OOMKiller may be executed if an application exceeds the RAM limits imposed on it, even if there is memory left on the system. This is a typical point around which this study revolves.

2.2 Docker RAM management

Docker has been the most popular containerisation engine for many years. Most Docker users know how to use it, build images, deploy applications with Docker-Compose... But it's rare to see people really understand the inner workings of the engine. This part of the analysis aims to explain some principles that Docker uses to manage its containers and limit their RAM usage.

By default, a container is not limited in its resource usage and can utilize as much of a resource as the host permits. Administrators must use methods provided by Docker to manage the amount of memory (or CPU) that a container is allowed to use. It can be easily done by setting runtime configuration options in the *docker run* command (or in a Docker Compose file). The resources limits of a container are then set by the kernel cgroups[4] (this is also true for Kubernetes). The official documentation itself says that it is important to restrict the quantity of RAM a container can access:

> *"It's important not to allow a running container to consume too much of the host machine's memory. On Linux hosts, if the kernel detects that there isn't enough memory to perform important system functions, it throws an OOME, or Out Of Memory Exception, and starts killing processes to free up memory. Any process is subject to killing, including Docker and other important applications. This can effectively bring the entire system down if the wrong process is killed."*[5]

Note that to avoid the Docker daemon to be killed by the OOMKiller, it adjusts the OOM priority[1] of itself, so it has less probability to be killed.

To set a RAM limit to a container, the flag *–memory=X* is used, where *X* represents the size of RAM allowed to be used by the container. It doesn't mean that this memory is reserved to it, it only means it can use up to *X* bytes of RAM. For example, the command *docker run –memory=1g hello-world* creates a container based on the *hello-world* image, that can use up to 1GB of RAM. If the container exceeds its limits, it will be terminated, either by a SIGTERM signal or a SIGKILL signal (depending on the CAP_SYS_RAWIO capability of the process).

The Docker engine also provides a flag when starting a container, which is *–oom-kill-disable*. This flag ensures that the container cannot be killed by the OOMKiller, which may be dangerous if the container takes too much RAM from the host. In such case, the OOMKiller would have to start to kill host processes to free memory. This would have been a great experiment case, as this is a project driven in a development environment, but the virtual machines used in this project does not have this kernel

[1]The OOMKiller behaviour and the topic of OOM priority are described on the official kernel documentation[6].

capability. When trying to execute a container with this flag, the following exception was raised: "WARNING: Your kernel does not support OomKillDisable. OomKillDisable discarded."

The full Docker documentation[7] provides other useful information about the configuration of containers, including the swap memory capability.

2.3 Kubernetes RAM management

Kubernetes is a well-known and powerful workloads orchestrator, most of the time using the containerd[8] engine (but not exclusively) to execute containers. The tool is able to automate deploying, scaling and operating multiple containerized applications.

Figure 2.3: Kubernetes logo

By abstracting the hardware infrastructure layer, Kubernetes allows the deployment of various workloads, from stateless to stateful applications, including databases. This versatility, combined with a robust ecosystem, has led to widespread adoption in the IT industry. A significant feature of Kubernetes is its ability to maintain the desired state of applications. For example, if a container fails, Kubernetes can automatically replace it, providing a form of self-healing crucial in production environments.

When the kubelet is launched on a node, it first analyses the node's total capacity and reserves part of the memory for its internal operation and that of the containerisation engine. The rest of the memory is considered allocable to the pods.

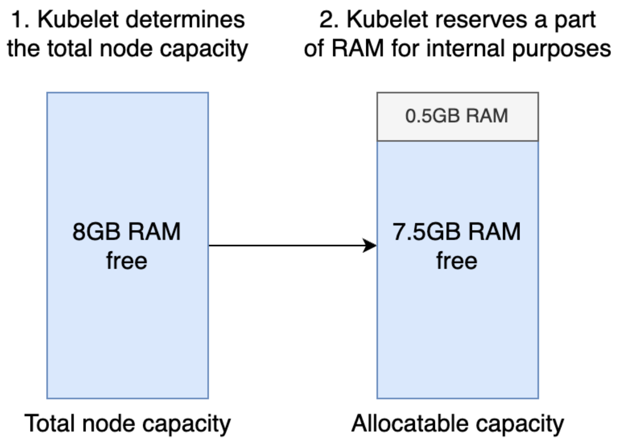

1. Kubelet determines the total node capacity
2. Kubelet reserves a part of RAM for internal purposes
8GB RAM free
0.5GB RAM
7.5GB RAM free
Total node capacity
Allocatable capacity

Figure 2.4: Reservation of a RAM space for internal purposes

In Kubernetes, it is possible to manage resources of pods using 2 definitions: the requests, and limits. The request represents the amount of memory that Kubernetes guarantees to the container. If a container requests a certain amount of memory, Kubernetes tries to ensure that it always has at least that much memory available to it. The limit represents the maximum amount of memory that a container is allowed to use. If a container tries to use more memory than its limit, it can be terminated or restarted by Kubernetes, depending on the restart policy set[2]. It is possible to define both in 2 different places:

- In a deployment file, where the specifications will be applied only to this deployment.
- In a LimitRange configuration file that refers to a namespace, where the specifications will be applied by default to all deployments within this namespace. The default values given by this configuration can still be override by configuring new ones in a deployment file, or a pod configuration file.

Below are shown the 2 way to configure limits and requests, as explained above.

```
# Defining limits and requests using
↪  a pod specifications file
---
apiVersion: v1
kind: Pod
metadata:
  name: ps6-pod
spec:
  containers:
  - name: ps6-container
    image: ps6-image
    resources:
      requests:
        memory: "256Mi"  # Request
        ↪  256 MB of memory
      limits:
        memory: "512Mi"  # Limit to
        ↪  512 MB of memory
```

```
# Defining limits and requests using
↪  a LimitRange configuration
↪  applied to the "ps6" namespace
---
apiVersion: v1
kind: LimitRange
metadata:
  name: ps6
spec:
  limits:
  - default:
      memory: 512Mi # Limit to 512
      ↪  MB of memory
    defaultRequest:
      memory: 256Mi # Request 256 MB
      ↪  of memory
    type: Container
```

A pod can access to the values of these fields using the Downward API[9]. Note that this API does not allow the pod to modify it's requests and limits fields, as these specifications are taken into account when the pod is scheduled by the Kube-Scheduler.

[2]A restart policy in Kubernetes defines the conditions under which a pod should be restarted after it exits.

The use of the request and limit fields can be summarised by the following image:

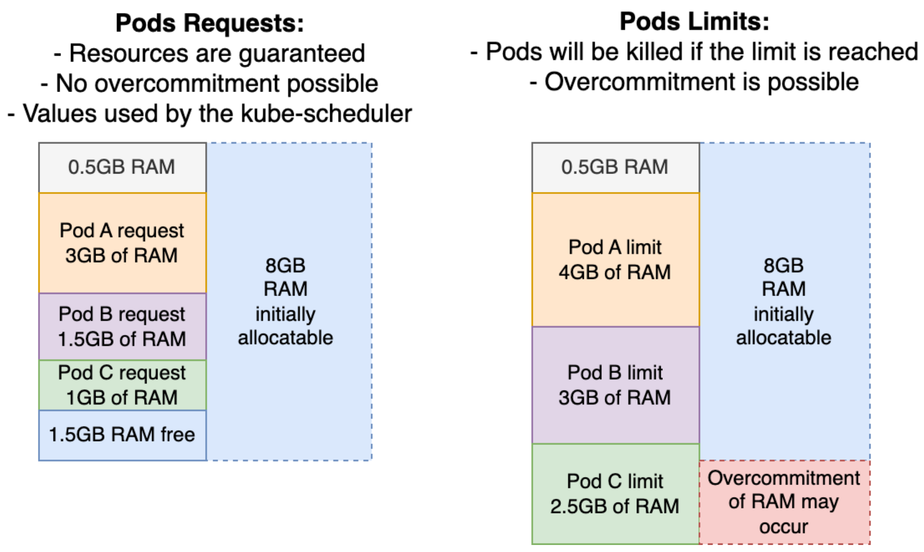

Pods Requests:
- Resources are guaranteed
- No overcommitment possible
- Values used by the kube-scheduler
0.5GB RAM
Pod A request 3GB of RAM
Pod B request 1.5GB of RAM
Pod C request 1GB of RAM
1.5GB RAM free
8GB RAM initially allocatable
Pods Limits:
- Pods will be killed if the limit is reached
- Overcommitment is possible
0.5GB RAM
Pod A limit 4GB of RAM
Pod B limit 3GB of RAM
Pod C limit 2.5GB of RAM
8GB RAM initially allocatable
Overcommitment of RAM may occur

Figure 2.5: Pods requests and limits visual representation

Note that the overcommitment is only theoretical. In this project, the requests and limits values of a pod will be specified in the deployment file of the experiments.

If a pod does not have a specified request/limit field, and neither does the namespace it is in, then it is able to use all the resources available on the node.[10]

When a pod with memory specifications (request/limit) needs to be launched, the Kube-Scheduler looks at the memory it needs (request) and finds a node which satisfies this condition. Once the pod has been scheduled on a node, the node's kubelet will instantiate the pod (with the *docker run* command, for example, or using containerd or any other containerization engine) and pass the memory arguments to the runtime. If a pod exceeds its limit, the kubelet will call the OOMKiller and it will terminate the process requesting memory allocation. If the process's PID is 1, then the container will be killed and restarted according to the pod's restart policy. Limits can be applied in two ways: reactively, where the system takes action after detecting a violation, or through enforcement, where the system ensures the container never surpasses the limit. Various container runtimes may have distinct methods for applying the same constraints.

If a pod needs to be scheduled but no node can meet the pod's RAM demand, then the pod will be in a 'pending' state until a node can execute it.

Note that Kubernetes does not support swap memory in the version used in this project. The command *swapoff -a* must be run on each worker node of a Kubernetes cluster. Consequently, the swap memory is not included in this study.

2.4 Monitoring of both

To study the way Docker and Kubernetes manages containers in different situations, a monitoring infrastructure is needed to gather the metrics and logs. As the monitoring aspect had already been dealt with in certain courses and in other related projects (whether school-related or not), the tools were quickly chosen. The aim of this part of the analysis is not to detail all the tools one by one, but rather to explain how they fit into the overall architecture. If further information is required, a link to the official documentation for each tool will be provided.

2.4.1 Metrics part

To collect metrics and display them on visual dashboards, the following tools have been chosen:

1. **Prometheus**[11], which is a TSDB and monitoring tool used mainly for monitoring purposes. It is one of the most known TSDB used to monitor infrastructures. It represents the tool that scraps the metrics on the monitored servers, by requesting information to specific tools APIs.

2. **cAdvisor**[12], which is an application tool that listens to the container engine API to gather information about running containers. Note that cAdvisor can be installed as a container on a native Docker environment. Such installation is not required on Kubernetes as cAdvisor is already installed in the Kubelet binary (cAdvisor has been developed by Google, like Kubernetes). cAdvisor exposes metrics on a specific port, that Prometheus uses.

3. **Node-Exporter**[13], which is an application tool that gather multipe information about the host. Like cAdvisor, it can be installed as a container. On a Kubernetes installation, a DaemonSet may be preferred, ensuring that all nodes run a copy of a Node-Exporter pod. Node-Exporter exposes metrics on a specific port, that Prometheus uses.

4. **Grafana**[14], which is a web application used to create visual dashboard from Prometheus metrics (and other datasources).

Note that inside containerised environments, commands such as "top" or "htop" cannot be used to provide reliable information as they take their information from the */proc/meminfo* file, which is not namespaced. If one of these two commands is executed inside a container, the results will show information about the entire host, not only the current container.

2.4.2 Logging part

To collect logs from running containers and display them, the following tools have been choosen:

1. **Elasticsearch**[15], which is a database used to store different types of data, including log files.

2. **Filebeat**[16], which is a tool used to export logs of running containers to an Elasticsearch database. It can be ran as a container on a Docker environment, and as a DaemonSet on a Kubernetes cluster. Filebeat sends its data to the Elasticsearch database.

3. **Kibana**[17], which is a web application used to browse logs from multiple sources and environments. Kibana queries Elasticsearch to display specified log files.

This technical stack is often call an "ELK Stack", because those 3 products comes from the same enterprise: Elastic[18].

2.5 Infrastructure automation

The infrastructure deployed in this project is composed of multiple virtual machines, running on an OpenStack[19] cluster at the HEIA-FR. To simplify test execution and reproducibility, it is preferable to automate as many elements of the infrastructure as possible. For this reason, the deployment and configuration of the infrastructure itself are automated, in addition to the launch of the tests. To do so, 2 tools are mainly used: Terraform and Ansible.

2.5.1 Terraform

Terraform[20], an IaC (Infrastructure as Code) tool developed by Hashicorp and first released in 2014, enables users to define infrastructure in documents and deploy configurations to a hypervisor for constructing virtual machines, networks, and more. It is particularly useful for cloud computing projects because it allows users to select virtual machine types, establish virtual networks and routers, incorporate an SSH key for secure remote access, and set security protocols, such as limiting access to just the SSH port.

Terraform can interface with multiple cloud provider APIs, including OpenStack which is the hypervisor used in this project.

2.5.2 Ansible

Ansible[21] is a tool maintained by RedHat and used for configuration automation. It employs the SSH protocol to establish connections with remote machines for configuration purposes. Machines must have a pre-configured SSH key to allow Ansible access, similar to the capability described in the Terraform analysis where virtual machines are set up with a registered SSH key. This setup enables Ansible to configure these machines and, if necessary, add another SSH key for administrator remote access. Ansible's control node can run on any machine equipped with Python. Unlike tools such as Chef or Puppet, Ansible does not require the installation of an agent on each machine; it relies solely on SSH for machine configuration. In this project, Ansible is used to configure the virtual machines and to start an experiment previously configured by the administrator on its laptop.

2.6 How the cloud providers manages RAM

One of the aims of this project is to study the way in which cloud providers monitor and charge users according to the RAM used within their services. Part of the analysis therefore focused on the following cloud providers:

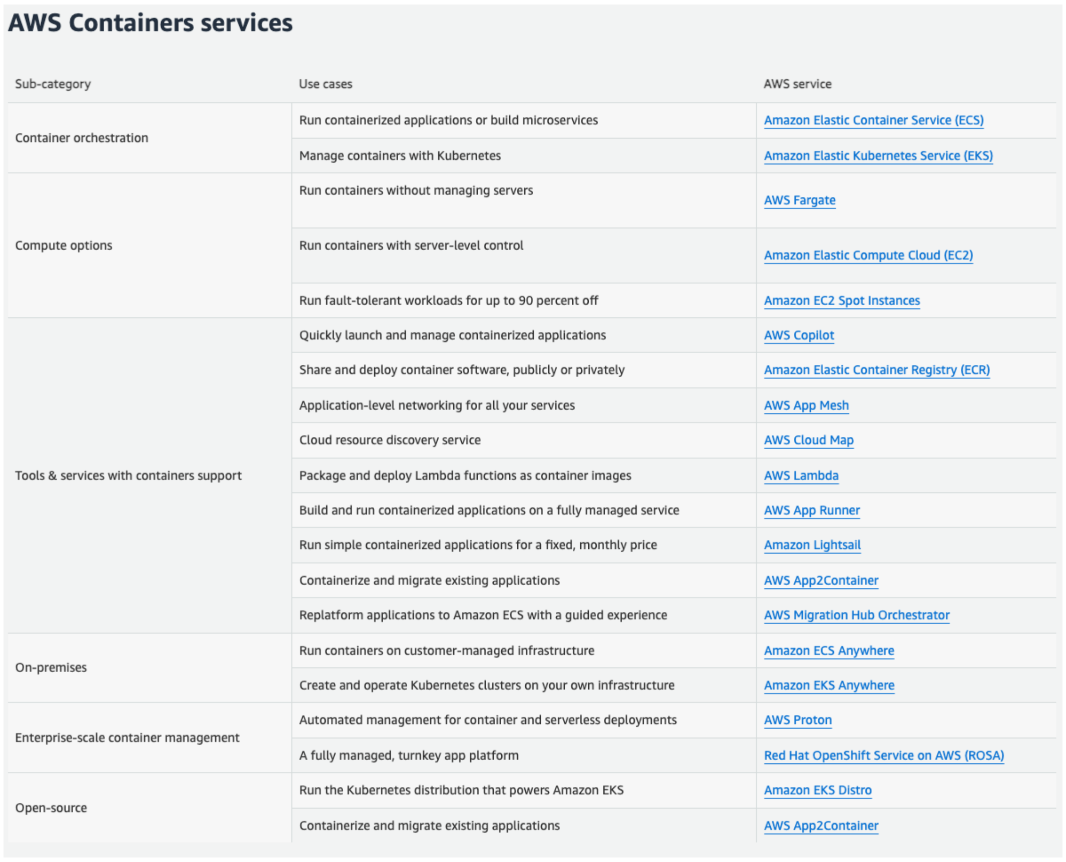
- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure (Azure)

Precisely, this section aims to list multiple "Containers as a Service" solutions, and to explain the aspects of RAM-based billing and how cloud providers manage RAM quotas and potential overruns.

2.6.1 CaaS solutions

Container as a Service solutions have gained widespread adoption across industries and cloud providers due to their ability to enhance application deployment, management, and scalability. The rise of microservices architectures has driven much of this growth, with businesses appreciating the modular approach to managing applications[22]. Each cloud provider offers multiple solutions for deploying scalable containerized applications. The aim of this study is not to examine every solution, but rather to offer an overall view. It is important to note, however, that the solution to be chosen for deploying an application depends on many factors that should be studied before selecting a particular solution.

Here is the example of the solutions offered by AWS that are only used for containerized applications:



AWS Containers services

Sub-category	Use cases	AWS service
Container orchestration	Run containerized applications or build microservices	Amazon Elastic Container Service (ECS)
	Manage containers with Kubernetes	Amazon Elastic Kubernetes Service (EKS)
Compute options	Run containers without managing servers	AWS Fargate
	Run containers with server-level control	Amazon Elastic Compute Cloud (EC2)
	Run fault-tolerant workloads for up to 90 percent off	Amazon EC2 Spot Instances
Tools & services with containers support	Quickly launch and manage containerized applications	AWS Copilot
	Share and deploy container software, publicly or privately	Amazon Elastic Container Registry (ECR)
	Application-level networking for all your services	AWS App Mesh
	Cloud resource discovery service	AWS Cloud Map
	Package and deploy Lambda functions as container images	AWS Lambda
	Build and run containerized applications on a fully managed service	AWS App Runner
	Run simple containerized applications for a fixed, monthly price	Amazon Lightsail
	Containerize and migrate existing applications	AWS App2Container
	Replatform applications to Amazon ECS with a guided experience	AWS Migration Hub Orchestrator
On-premises	Run containers on customer-managed infrastructure	Amazon ECS Anywhere
	Create and operate Kubernetes clusters on your own infrastructure	Amazon EKS Anywhere
Enterprise-scale container management	Automated management for container and serverless deployments	AWS Proton
	A fully managed, turnkey app platform	Red Hat OpenShift Service on AWS (ROSA)
Open-source	Run the Kubernetes distribution that powers Amazon EKS	Amazon EKS Distro
	Containerize and migrate existing applications	AWS App2Container

Figure 2.6: All CaaS solutions offered by AWS (total of 20)

A lot of solutions offered by AWS, GCP and Azure are equivalent. Here are 3 examples:

- **A Kubernetes cluster managed by the cloud provider, which seems to be their highlighted product:**
  - AWS: Amazon Elastic Kubernetes Service - "The most trusted way to start, run, and scale Kubernetes"
  - GCP: Google Kubernetes Engine - "The most scalable and fully automated Kubernetes service"
  - Azure: Azure Kubernetes Service - "Innovate, deploy, and operate Kubernetes seamlessly"

- **A way to deploy serverless containers to avoid managing clusters and infrastructures:**
  - AWS: AWS Fargate - "Serverless compute for containers"
  - GCP: GCP Cloud Run - "Build applications or websites quickly on a fully managed platform"
  - Azure: Azure Container Instances - "Develop apps fast without managing virtual machines or having to learn new tools—it's just your application, in a container, running in the cloud."

- **A way to easily run containerized applications or build microservices:**
  - AWS: Amazon Elastic Container Service - "Run highly secure, reliable, and scalable containers"
  - GCP: Google Kubernetes Engine - "The most scalable and fully automated Kubernetes service" (same as the first example)
  - Azure: Azure Container Apps - "Azure Container Apps is a scalable service that lets you deploy thousands of containers without requiring access to the control plane."

According to several articles[23][24][25], the 3 cloud providers mentioned above offer relatively similar functions, but in general terms:

1. AWS is the most widely used cloud provider in the world, with datacenters all over the world.
2. GCP may be the best cloud provider for Kubernetes-related industries, as Kubernetes has been developed and maintained by Google. These articles are also pointing out that Google may be the best cloud provider for bigdata-related projects, as it offers the most advanced deep learning models and powerful hardware accelerators[25].
3. Azure may be the best cost-effective choice for multiple solutions, especially Microsoft-related solutions.

2.6.2 RAM-based billing

Each cloud provider provides its users with full documentation on how the services are billed, as well as a price calculator that takes a number of parameters into account. It is often that cloud providers explains the billing process of each solution on the product's web page. These resources are available to the following links:

1. **AWS:**
   (a) Documentation: Example with Amazon Elastic Kubernetes Service: `https://aws.amazon.com/eks/pricing/`
   (b) Calculator: `https://calculator.aws/#/`

2. **GCP:**
   (a) Documentation: Example with Google Kubernetes Engine: `https://cloud.google.com/kubernetes-engine?hl=en#pricing`
   (b) Calculator: `https://cloud.google.com/products/calculator`

3. **Azure:**
   (a) Documentation: Example with Azure Kubernetes Service: `https://azure.microsoft.com/en-us/pricing/details/kubernetes-service/`
   (b) Calculator: `https://azure.microsoft.com/en-gb/pricing/calculator/`

Below are written some explanations on the way AWS, GCP and Azure provides their RAM-based billing for a containerized workload.

1. **AWS:** There are two different charge models for Amazon ECS: the AWS Fargate Launch Type Model and the Amazon EC2 Launch Type Model. For this study, the first one has been choosen. AWS Fargate pricing is calculated based on the vCPU, memory, Operating Systems, CPU Architecture, and storage resources used from the time the user starts to download its container image until the Amazon ECS Task or Amazon EKS Pod terminates, rounded up to the nearest second. The price is defined using the number of vCPU and GB of RAM used by the container. The price of 1 GB of RAM depends on the location of the container (the AWS Region in which the container will be executed). For example, in the region *eu-west-3* which is in Paris, each GB of RAM used costs $0.00159 per hour. There is no overrun possible as AWS considers its resources as infinite. The final cost will then follow the application RAM usage (and CPU).

2. **GCP:** The GKE charging model is based on the Compute Engine billing model. The cost will depend on the Kubernetes nodes specifications. For example, for a single-node cluster with 1GB of RAM and 1 vCPU in the *europe-west4* region based in Netherlands, the cost will be of $104.13 per month. If the single-node has 4GB of RAM and still 1 vCPU, the new cost will be of $114.86 per month. There is no overrun possible as the Kubernetes cluster is executed on a virtual machine with a defined number of RAM available. If a container asks for too many RAM, the behavior will be the same as on a normal Kubernetes cluster.[3]

3. **Azure:** The Azure Container Instance billing model is the same as the AWS Fargate model. For example, in the West Switzerland region, the cost of 1 GB of RAM is $0.00676 per hour. This cost is higher than the AWS one, but Azure offers different saving plans for 1 and 3 years. In such cases, a discount up to 52% can be applied.

To conclude this part of the analysis, each cloud provider offers solutions that are very similar to those of its competitors. Some cloud providers have several billing methodologies, but all agree to bill users according to the number of vCPUs and GB of RAM they use (or reserve in the case of a virtual machine).

[3]The AWS EC2 billing model is similar to the GCP Compute Engine billing model.

3 | Design

This chapter explains how the project has been designed, from the stress tool that has been developed to the infrastructure deployed to study the behavior of the Docker engine and Kubernetes. Many diagrams and images are available in this part of the report to make it easier for the reader to understand the project.

The first part focuses on the C++ stress-tool application, with the list of features, commands and other specificities. The second part explains how the infrastructure is designed and what tools are used to automate its deployment and configuration.

Contents

3.1 C++ application

This section describes the design of the C++ application that is used to allocate and free memory sequentially. Note that this application is inspired by existing stress tools, that can be found on different repositories and download pages[26][27]. This application is deployed in containerisation environments, which is why the chapter on implementation also covers this aspect.

3.1.1 List of features

The objective is to provide a way to the administrator to configure different testing procedures where the stress tool juggles between allocation and deallocation procedures, as defined in a configuration file. The following features have been point out to carry out this objective:

- Allocation of a given number of bytes or RAM, as many times as required by the administrator.
- Each allocation is independent in size and is identified by a unique ID.
- Deallocation procedure of a block identified by its ID can be achieved anytime after the allocation, even if other allocations have been made between.
- The tool has a capability to wait for a given duration.
- Allocation, deallocation and waiting instructions can be defined as many times as wanted by the adminsitrator in a configuration file that the application parses at the beginning of the program.
- In case an allocation instruction is called but no deallocation for the allocated block of memory, this block is deallocated automatically at the end of the program.
- The application logs every action in the STDOUT output. This output is used by Docker to create the log file of the container.

It is also interesting to define aspects of development that are not part of this project, in particular the fact that this is an application developed for testing only. As a result, the parser of the configuration file does not implement error handling, as it is supposed that the administrator knows how to write the list of instructions.

3.1.2 List of commands

Administrators can use 3 different commands to configure their experience. He describes his experiment imperatively and sequentially, and the program will execute each instruction one by one. The commands are written in a text file, called **experiment-config.txt** that will be read by the program. The available commands are the following:

- **Allocation of SIZE bytes of memory under the block ID ID:**
  - Syntax: *allocate [SIZE] [ID]*
  - Example: *allocate 1024 block1*
- **Deallocation of the block ID ID:**
  - Syntax: *allocate [ID]*
  - Example: *deallocate block1*
- **Wait for DURATION second(s):**
  - Syntax: *wait [DURATION]*
  - Example: *wait 10*

If an allocation is made but the administrator forgot to write the deallocation instruction, the program will automatically deallocate this block at the end of the instructions list. Here is an example of an experiment configuration:

```
wait 20
allocate 1024 block1
wait 10
deallocate block1
wait 5
allocate 2048 block2
wait 5
allocate 123456789 block3
wait 40
deallocate block2
deallocate block3
wait 10
```

3.2 Testing infrastructure

The infrastructure designed to study the behavior of Docker and Kubernetes in different situations is shown below:

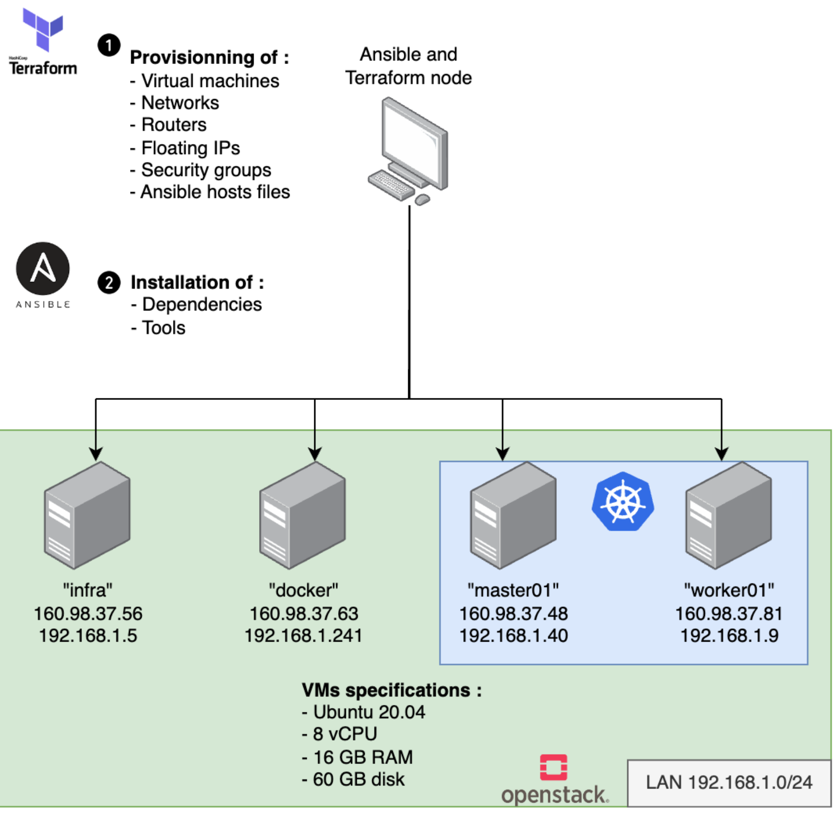


Figure 3.1: Design of the infrastructure automatically deployed

Terraform is used to create virtual machines, networks, and other elements to have the base of the infrastructure. Then, Ansible connects to each node to install softwares such as Docker, Kubernetes, dependencies, monitoring stack... Almost the entire configuration of each machine is automated. Each virtual machine has a specific role, which is explained below:

- The "infra" virtual machine hosts the monitoring tools. This machine is not tested, it is used to provide monitoring dashboards and logs from containers.
- The "docker" virtual machine executes a native Docker engine. This is the first testing environment.
- The "master01" and "worker01" represents a Kubernetes cluster with a single worker node. Pods are only scheduled on "worker01". This is the second testing environment.

The IP addresses shown in this image may be different when re-deploying the infrastructure as the subnet 192.168.1.0/24 has a DHCP server maintained by OpenStack and the subnet 160.98.37.0/24 represents the floating IPs of the OpenStack Green cluster, which are dynamically assigned to virtual machines.

The way the administrator starts an experiment and looks at the monitoring and logging results can be shown as an UML diagram. In this case, a diagram has been created with some components and use case parts.

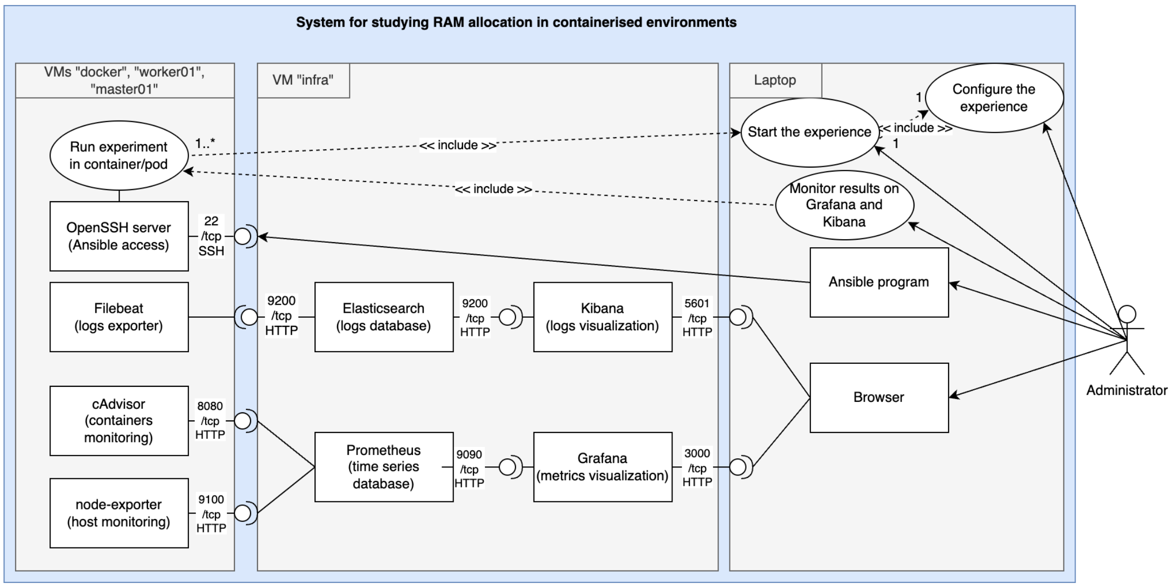

System for studying RAM allocation in containerised environments
VMs "docker", "worker01", "master01"
VM "infra"
Laptop
Run experiment in container/pod
1..*
<< include >>
<< include >>
Start the experience
Configure the experience
1
<< include >>
1
Monitor results on Grafana and Kibana
OpenSSH server (Ansible access)
22 /tcp SSH
Filebeat (logs exporter)
9200 /tcp HTTP
Elasticsearch (logs database)
9200 /tcp HTTP
Kibana (logs visualization)
5601 /tcp HTTP
Ansible program
Browser
Administrator
cAdvisor (containers monitoring)
8080 /tcp HTTP
node-exporter (host monitoring)
9100 /tcp HTTP
Prometheus (time series database)
9090 /tcp HTTP
Grafana (metrics visualization)
3000 /tcp HTTP

Figure 3.2: Components / Use Case diagram of an experiment. This diagram is also available in the Appendix A for a better readability.

The administrator can configure an experiment on it's laptop, by modifying the *experiment-config.txt* file. This file contains all the actions the stress-tool will perform, using commands described in the C++ application design section. Once configured, the experiment can be launched using an Ansible playbook. There is one playbook for the Kubernetes cluster, and one playbook for the native Docker environment. Once started, the administrator can check results on many Grafana dashboards showing RAM consumption of containers and their limits, but also metrics about the host. It is also possible to discover logs of containers on the Kibana web interface. The diagram shows the listening port of each tool, and also the way the experiment is started. From a laptop, the administrator can start an Ansible playbook and the tool will connect to virtual machines to execute commands on them and start the experiment.

4 Implementation

This chapter explains how the project was technically implemented. The first part is about the C++ stress-tool application that has been developed and containerized, and then the focus is made on the deployment and automation of the infrastructure. It has been chosen to automate the infrastructure to facilitate the replication of testing procedures in the same environment.

This chapter does not aim to provide a complete explanation of the C++ source code, either of the containerization process or other technical topics. It outlines the major parts of the implementation, but it is suggested to visit the technical GitLab repository of the project to see the full C++ code, Terraform, Ansible files and other configuration files[28].

Contents

4.1 C++ application

This section explains the key parts of the C++ application. To browse the full source code, follow the technical GitLab repository of the project[28].

4.1.1 Source code

The source code of the application is relatively simple, and is centralized in a single *main.cpp* file. The program reads the configuration file of the experiment, named *config.txt* and does actions depending on the instructions written by the administrator in this file.

```
int main() {
    printWithTimestamp("Starting stress tool.");
    std::string filePath = "config.txt";
    processInstructions(filePath);
    printWithTimestamp("End of program.");
    return 0;
}
```

The 2 key parts of this program are the way the tool allocates and frees memory blocks, and how it can get information on its RAM usage.

Memory allocation and deallocation
The memory allocation is made by the following block of code:

```
# allocations map: std::unordered_map<std::string, char*> allocations;
char* memory = static_cast<char*>(malloc(size));
if (memory != nullptr) {
    allocations[id] = memory;
    printWithTimestamp("Allocation of " + std::to_string(size) + " bytes  (ID =
    ↪  " + id + ").");
    memset(memory, 0, size);
} else {
    printWithTimestamp("An error occured while allocating " +
    ↪  std::to_string(size) + " bytes (ID = " + id + ").");
}
```

It allocates a block of memory of *size* bytes. If the allocation is successful (meaning *memory* is not *nullptr*), it stores the allocated memory pointer in a map allocations with *id* as the key. Note that *id* is chosen by the administrator in its configuration file to identity this block, and tell the program to deallocate it later. Then, the *memset()* function initializes the block to 0. If the allocation fails, it logs an error message with specified information. The deallocation of a block is made by the following block of code:

```
# allocations map: std::unordered_map<std::string, char*> allocations;
auto it = allocations.find(id);
if (it != allocations.end()) {
    free(it->second);
    allocations.erase(it);
    printWithTimestamp("De-allocating memory block with ID = " + id + ".");
} else {
    printWithTimestamp("No memory block found for ID = " + id + ".");
}
```

It first searches for a memory allocation associated with a given id in the allocations map. If found, it deallocates the memory using free and then removes the entry from the map. If not found, it logs an error message with the block ID.

Information about RAM usage
A process can ask the kernel many information on itself using the */proc/self/* directory. It's content is dynamically modified by the operating system depending on the current process running on the logical CPU[29]. Therefore, when the C++ application tries to access this folder, it can browse multiple information on itself. Here is an example[30] of the content of the */proc/self/status* file (truncated):

```
$ cat /proc/$$/status
VmPeak:     131168 kB
VmSize:     131168 kB
VmLck:           0 kB
VmPin:           0 kB
VmHWM:       13484 kB
VmRSS:       13484 kB
RssAnon:     10264 kB
RssFile:      3220 kB
RssShmem:        0 kB
VmData:      10332 kB
VmStk:         136 kB
VmExe:         992 kB
VmLib:        2104 kB
VmPTE:          76 kB
VmPMD:          12 kB
VmSwap:          0 kB
```

There is a lot of information on memory usage of the process, but the interesting ones for this project are the following:

- **VmSize:** Virtual memory size, which represents the global RAM usage of the process.
- **VmRSS:** Resident set size, which represents the portion of physical RAM used. The documentation[30] says that this value may not be accurate.

The C++ application reads this file whenever it needs and prints the value of both fields.

4.1.2 Docker image

When the source code of the application is modified for any reason, the new version is packaged in a Docker image and pushed to a registry (see the "CI/CD pipeline" section). The Dockerfile of the project is the following:

```
FROM debian:bullseye-slim
RUN apt-get update && apt-get install -y \
    build-essential \
    libboost-all-dev \
    && rm -rf /var/lib/apt/lists/*

COPY stress-tool/ /usr/src/stress-tool/
WORKDIR /usr/src/stress-tool
RUN g++ -o stress-tool main.cpp
CMD ["./stress-tool"]
```

This Dockerfile simply installs required packages, setup the environment and build the application. The container then starts with the stress-tool as its only process.

Note that this is a Debian-based image. This choice has been made to allow administrator to install different packages in the container when it is running, allowing him to debug and test features more easily. For example, a testing procedure of the project requires the installation of the Docker CLI inside of the container, which is made easier on a Debian container than on a GCC-based container. An other good choice would have been to use a lightweight and secure Alpine image.

4.1.3 CI/CD pipeline

The technical GitLab repository contains a *.gitlab-ci.yml* file that automates the creation of a Docker image including the stress-tool application, and pushes this image to the Docker Hub. As this part can be considered as a Continuous Integration pipeline, the second part where the administrator starts an experiment is not fully automated (for voluntary reasons). Anyway, the global pipeline from the development of the application to its deployment on the native Docker environment and the Kubernetes cluster can be seen as following:

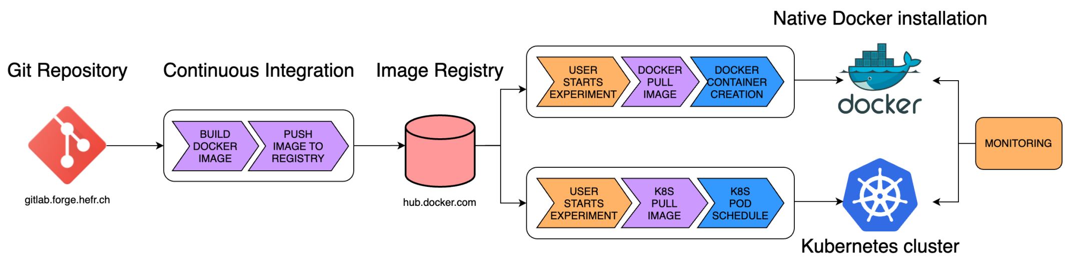


Figure 4.1: Full pipeline process of the project. This diagram is also available in the Appendix B for a better readability.

Note that when the image is pushed to the registry and pulled on both environments, the administrator does not need to commit a new change to start an experiment. To start a new experiment, he just needs to modify the *experiment-config.txt* file on his laptop and start the Ansible playbook related to the environment he wants to deploy the test on. It is also important to know that this pipeline does not show the deployment of the infrastructure, so Terraform is not mentioned on it.

4.2 Infrastructure automation

This section outlines the key parts of the infrastructure deployment and automation using Terraform and Ansible tools.

4.2.1 Deployment using Terraform

This section uses Terraform files already used during other projects, such as "Infrastructures et systèmes virtualisés" course and "Projet de semestre 5" project.

The technical GitLab repository[28] includes a Terraform folder that contains all the code files needed to build the infrastructure:

1. *main.tf*: The main file that uses other files to provision the infrastructure and output results. It assigns a floating IP to each virtual machine.

2. *network.tf*: Defines the virtual network, router, and subnet configuration. It also provides DNS nameservers to virtual machines so they can browse the web like regular computers and access mirror repositories to download packages.

3. *providers.tf*: Specifies the providers used by Terraform, which in this case is only OpenStack.

4. *security_group.tf*: Outlines how the virtual router (acting as a firewall) should filter packets to and from a virtual machine. To avoid issues, and because this project is in a **development environment not accessible from the Internet**, most ports are open.

5. *ssh_key.tf*: Specifies the SSH key automatically added to the virtual machines. A specific key has been created for Ansible to configure the machines directly without administrator intervention.

6. *version.tf*: Defines the minimum required version for the OpenStack provider, which must be 0.14.0 or higher.

7. *variables.tf*: Lists all variables of the infrastructure, including the number of virtual machines, their names, operating systems, flavors, and network attachments.

These are "pure" Terraform files used for provisioning the infrastructure. Additionally, 2 other files are created by the Terraform apply process, which are based on the *ansible-hosts.tpl* and *prometheus-targets.tpl*. These are template files filled by Terraform to create the inventory file used by Ansible to access all virtual machines and the Prometheus targets list used for monitoring purposes.

The Terraform configuration should not be changed as the automation is designed to operate on the 4 machines defined as standard. There is no reason other virtual machines should be added to the infrastructure. It is also not advisable to change the order of the machines in the *variables.tf* file, as this would have the effect of distorting the name of each server (the server named *docker* would actually be the *infra* server, for example).

4.2.2 Configuration and exploitation using Ansible

The *ansible* folder of the project stores all configuration and playbook files used to configure the virtual machines once they have been deployed and execute experiments on them.

Infrastructure configuration
Once deployed with Terraform, the administrator can setup the whole infrastructure using the *setup-everything.yml* playbook file, which is the following:

```
---
# Approximately 12 minutes
- import_playbook: ./setup-infra.yml
- import_playbook: ./setup-docker.yml
- import_playbook: ./setup-k8s-master.yml
- import_playbook: ./setup-k8s-worker.yml
```

All it does is importing other playbooks that configures each virtual machine. Each sub-playbook will not be explained in this report, but they can be browsed on the GitLab project. Keep in mind that these playbooks installs all the dependencies, tools, and the monitoring stack (including pre-configured Grafana dashboards on the *infra* VM) on each machine. Once the playbook is finished, the administrator can start to monitor its infrastructure by logging to the Grafana and Kibana UI, accessible to the following URLs:

- Grafana: [vm-infra-ip]:3000
- Kibana: [vm-infra-ip]:5601

There are 2 pre-configured Grafana dashboard. One is used to display host's information, and the other is used to display containers information. Screens of the dashboards are available in the chapter 5 of this report.

Running an experiment

To automate the deployment of an experiment, there is one playbook for the Docker environment and one playbook for the Kubernetes environment. Here is the playbook for the Docker one:

```
---
- hosts: docker
  become: yes
  tasks:
  - name: copy configuration of experiment
    copy:
      src: ../files/experiment-config.txt
      dest: /home/ubuntu/config.txt
      owner: root
      group: root

  - name: Pull new version of image in case it has been updated
    shell: docker pull leichap/heia-stress-tool:latest

  - name: Pre-config virtual machine
    shell: docker run -name stress-tool -v
    ↪  /home/ubuntu/config.txt:/usr/src/stress-tool/config.txt -d --rm
    ↪  leichap/heia-stress-tool
```

This playbook transfers the configuration of the experiment to the Docker host, pulls the new version of the image and runs the container. It is possible to modify the command if the administrator needs to. For example, once the experiment is finished the container is automatically removed. To avoid this behaviour to perform a "docker inspect" command on the container once it has ended, the administrator can remove the *–rm* flag of the "docker run" command and run the playbook. The Kubernetes-related playbook is the following:

```
---
- hosts: k8sworker
  tasks:
  - name: copy configuration of experiment on worker node
    copy:
      src: ../files/experiment-config.txt
      dest: /home/ubuntu/config.txt
      owner: ubuntu
      group: ubuntu

- hosts: k8smaster
  tasks:
  - name: Remove any old experiment pod
    shell: kubectl delete -f /home/ubuntu/pod.yaml
    ignore_errors: yes

  - name: copy configuration of pod
    copy:
      src: ../files/kubernetes-pod-object.yml
      dest: /home/ubuntu/pod.yaml
      owner: ubuntu
      group: ubuntu
      mode: 0644

  - name: Apply new pod configuration
    shell: kubectl apply -f /home/ubuntu/pod.yaml
```

It first transfers the configuration of the experiment to the worker node, as this file is mounted inside the pod. Then, it makes sure that there is no other experiment currently running, and it finally starts a new experiment using the new version of the Pod configuration.

The technical GitLab repository contains a README file that explains in details how to start an experiment for both Docker and Kubernetes environment, and how to monitor results. It also explains how to modify the Pod configuration of the Kubernetes experiment.

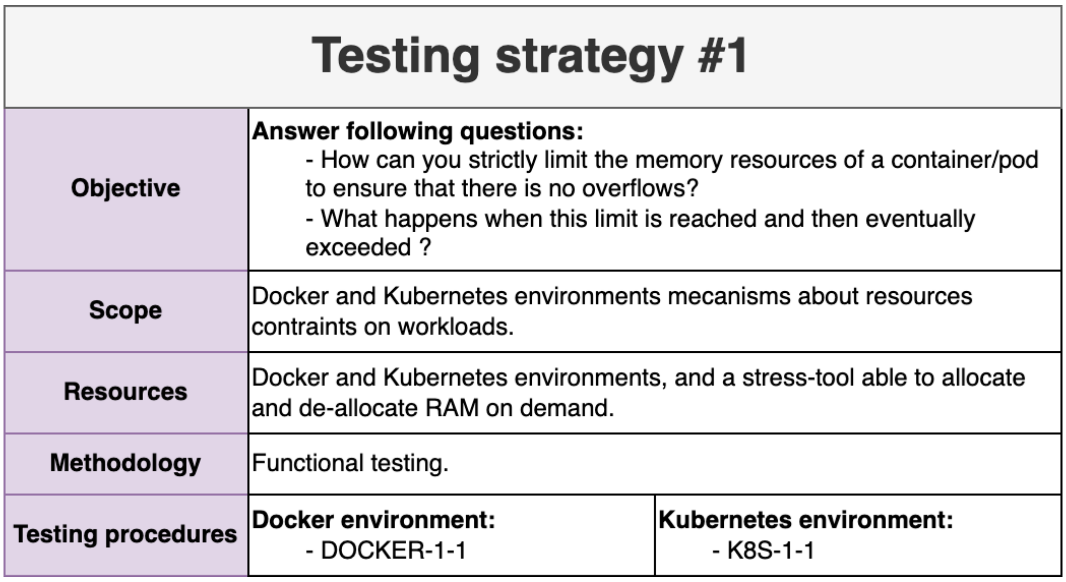
5 | Results and interpretation

This chapter presents all the test strategies and procedures defined as part of the project. For each strategy, several procedures have been defined. For each procedure, a detailed explanation is provided in a summary table. A brief overview of the results is also available in each table. The results are then explained in detail. A summary of all the tests is given at the end of the chapter with ideas of other experiments and side results.

Contents

5.1 Testing strategy 1

The first testing strategy of the project is described below:



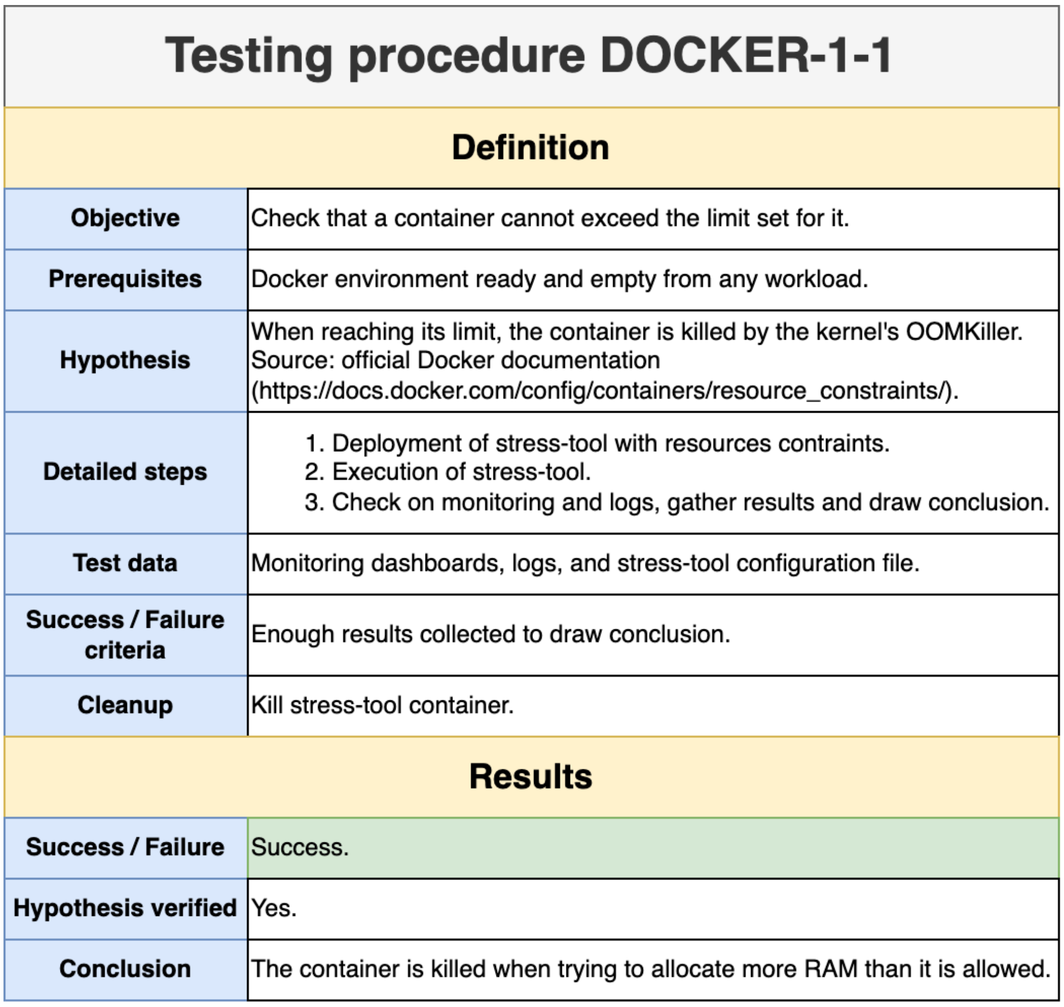
Testing strategy #1		
Objective	**Answer following questions:** - How can you strictly limit the memory resources of a container/pod to ensure that there is no overflows? - What happens when this limit is reached and then eventually exceeded ?	
Scope	Docker and Kubernetes environments mecanisms about resources contraints on workloads.	
Resources	Docker and Kubernetes environments, and a stress-tool able to allocate and de-allocate RAM on demand.	
Methodology	Functional testing.	
Testing procedures	**Docker environment:** - DOCKER-1-1	**Kubernetes environment:** - K8S-1-1

Figure 5.1: Testing strategy 1

Its purpose is to answer the first questions asked in the objective section of this report.

5.1.1 Testing procedure DOCKER-1-1

The testing procedure DOCKER-1-1 is described below with associated results:



Testing procedure DOCKER-1-1	
Definition	
Objective	Check that a container cannot exceed the limit set for it.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	When reaching its limit, the container is killed by the kernel's OOMKiller. Source: official Docker documentation (https://docs.docker.com/config/containers/resource_constraints/).
Detailed steps	1. Deployment of stress-tool with resources contraints. 2. Execution of stress-tool. 3. Check on monitoring and logs, gather results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The container is killed when trying to allocate more RAM than it is allowed.

Figure 5.2: Testing procedure DOCKER-1-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1 #100MB
wait 20
allocate 104857600 id2
wait 20
allocate 52428800 id3 #50MB
wait 20
allocate 52428800 id4 # ------ this instruction must kill the container
wait 20
allocate 52428800 id5
wait 20
```

Note that the container has been launched with a RAM limit set to 300MB. To confirm the hypothesis, it must be OOMKilled before the 4th allocation instruction. The associated Grafana dashboard is available here:

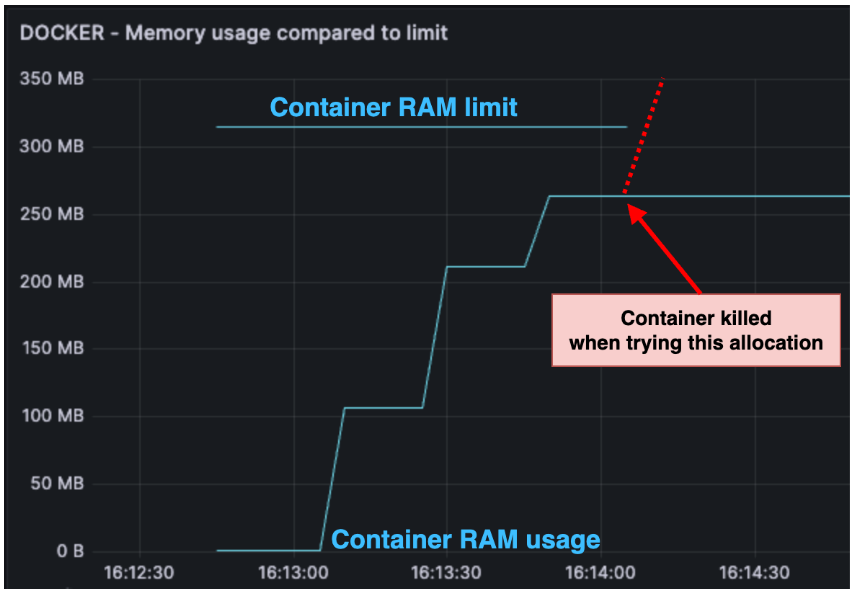


Figure 5.3: DOCKER-1-1: Grafana dashboard

The horizontal line at the top represents the limit set for the container. Note that the 2 allocations of 100MB and the 3rd allocation of 50MB run without any problem, but that the 4th allocation never runs. This is because the container was killed trying to make this allocation. The container cannot therefore exceed the limit imposed on it. To ensure that the container has been killed by the Docker engine for having exceeded its limit, it is possible to check its state using the "docker inspect" command (output truncated):

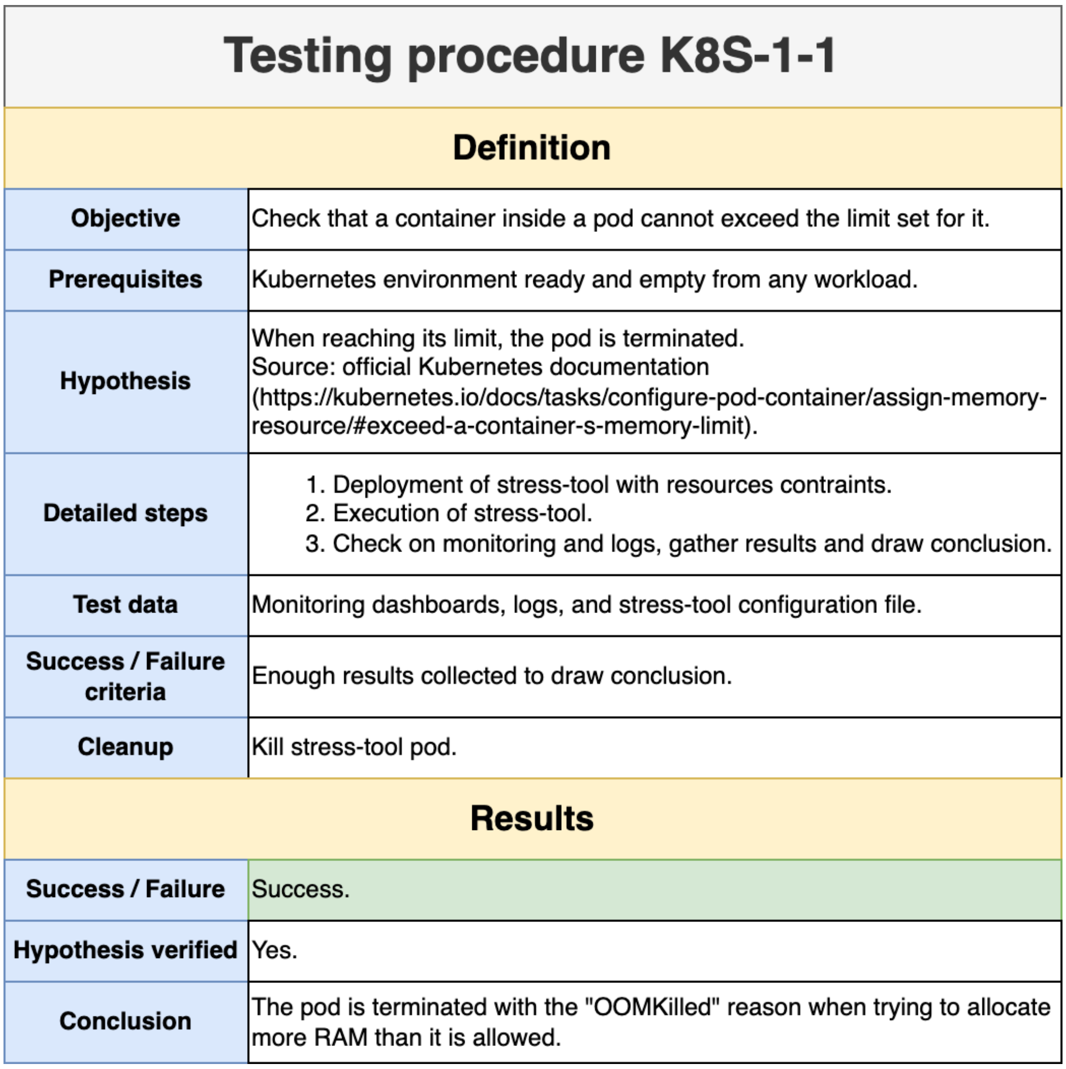
```
"State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": true,
            "ExitCode": 137
        }
```

The "OOMKilled" value is set to "true", validating the fact that the container was killed by the engine.

In all Grafana dashboards, it is possible that the line representing the RAM consumption of the container/pod continues even after the container/pod has been killed. This is due to cAdvisor behaviour.

5.1.2 Testing procedure K8S-1-1

The testing procedure K8S-1-1 is described below with associated results:



Testing procedure K8S-1-1	
Definition	
Objective	Check that a container inside a pod cannot exceed the limit set for it.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	When reaching its limit, the pod is terminated. Source: official Kubernetes documentation (https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/#exceed-a-container-s-memory-limit).
Detailed steps	1. Deployment of stress-tool with resources contraints. 2. Execution of stress-tool. 3. Check on monitoring and logs, gather results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The pod is terminated with the "OOMKilled" reason when trying to allocate more RAM than it is allowed.

Figure 5.4: Testing procedure K8S-1-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1 #100MB
wait 20
allocate 104857600 id2
wait 20
allocate 52428800 id3 #50MB
wait 20
allocate 52428800 id4 # ------ this instruction must kill the pod
wait 20
allocate 52428800 id5
wait 20
```

Note that the pod has been launched with a RAM limit set to 300MB. To confirm the hypothesis, it must be OOMKilled before the 4th allocation instruction. The associated Grafana dashboard is available here:

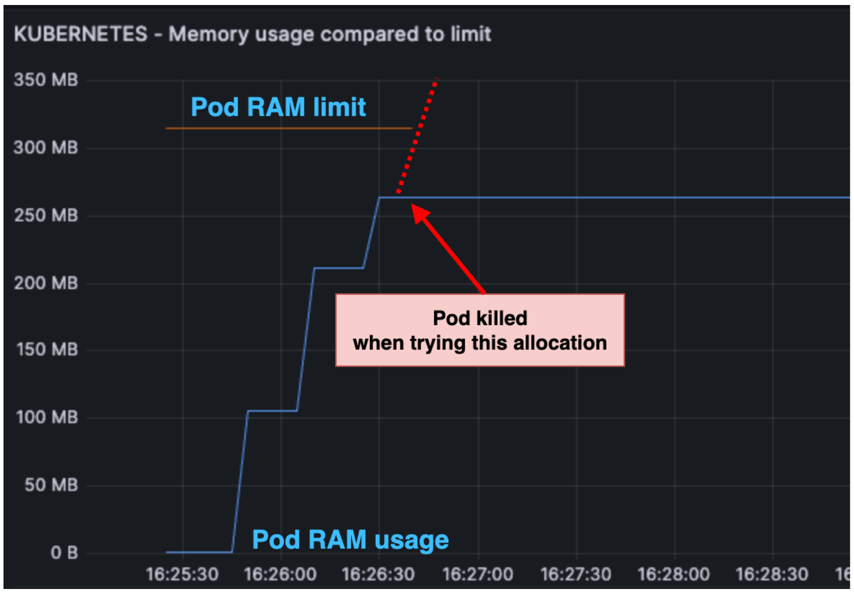


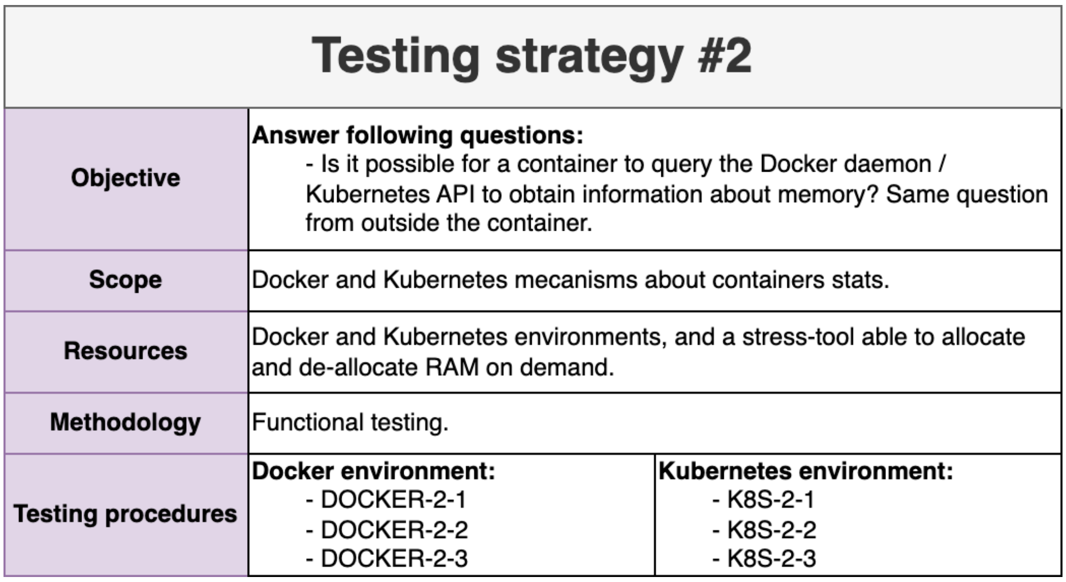
Figure 5.5: K8S-1-1: Grafana dashboard

The horizontal line at the top represents the limit set for the container. This result is the same as for the DOCKER-1-1 testing procedure. The 4th allocation is asking for a too big amount of memory, so Kubernetes kills the pod because it would have exceeded its limit. An other way to see that the pod has been killed is by querying the Kubernetes API:

```
[PS6] ubuntu@master01:~# kubectl get pods
NAME           READY    STATUS      RESTARTS    AGE
stress-tool    0/1     OOMKilled    0          7m13s
```

5.2 Testing strategy 2

The second testing strategy of the project is described below:



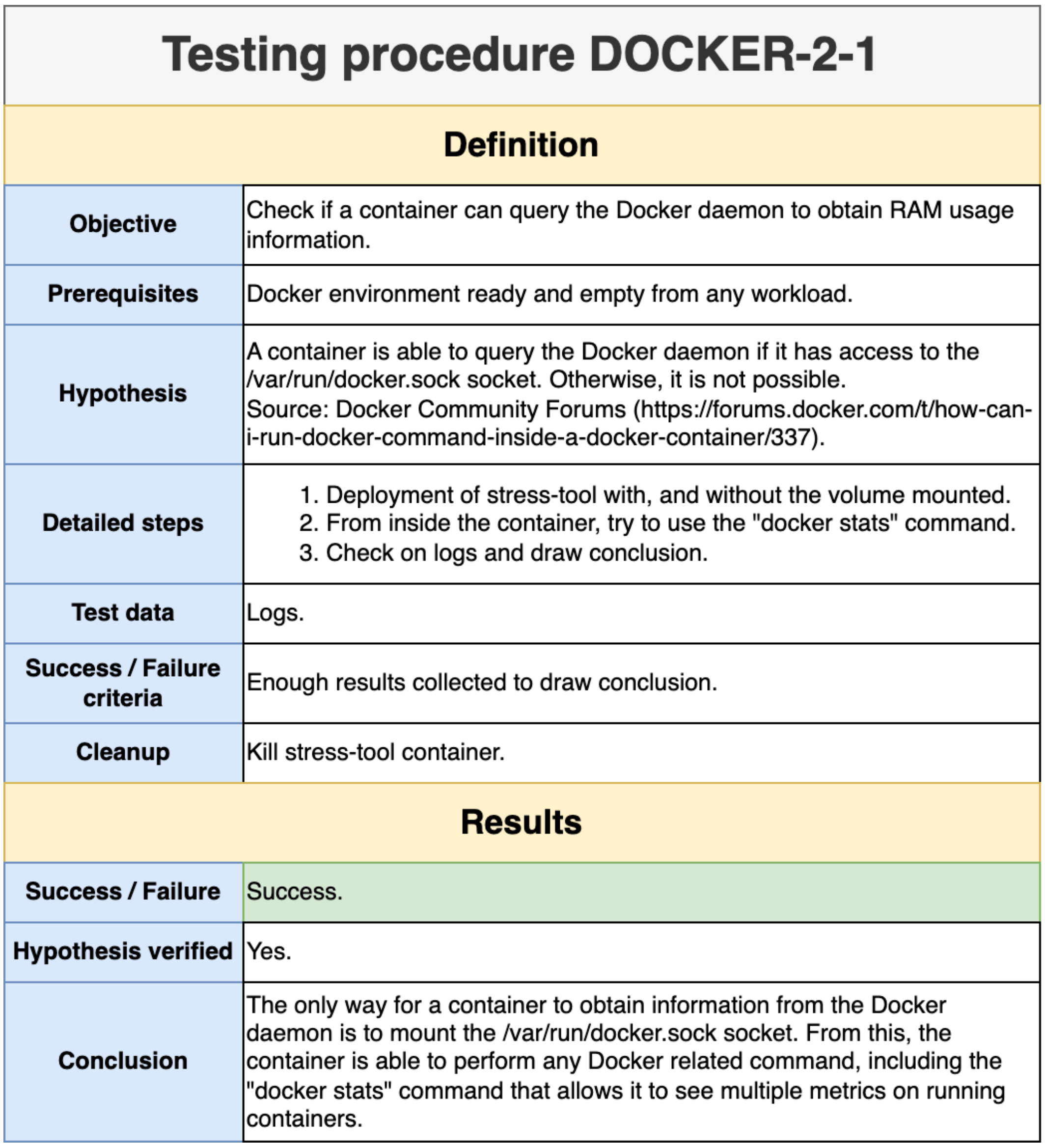
Testing strategy #2		
Objective	**Answer following questions:** - Is it possible for a container to query the Docker daemon / Kubernetes API to obtain information about memory? Same question from outside the container.	
Scope	Docker and Kubernetes mecanisms about containers stats.	
Resources	Docker and Kubernetes environments, and a stress-tool able to allocate and de-allocate RAM on demand.	
Methodology	Functional testing.	
Testing procedures	**Docker environment:** - DOCKER-2-1 - DOCKER-2-2 - DOCKER-2-3	**Kubernetes environment:** - K8S-2-1 - K8S-2-2 - K8S-2-3

Figure 5.6: Testing strategy 2

Its purpose is to answer the second question asked in the objective section of this report.

5.2.1 Testing procedure DOCKER-2-1

The testing procedure DOCKER-2-1 is described below with associated results:



Testing procedure DOCKER-2-1

Definition	
Objective	Check if a container can query the Docker daemon to obtain RAM usage information.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	A container is able to query the Docker daemon if it has access to the /var/run/docker.sock socket. Otherwise, it is not possible. Source: Docker Community Forums (https://forums.docker.com/t/how-can-i-run-docker-command-inside-a-docker-container/337).
Detailed steps	1. Deployment of stress-tool with, and without the volume mounted. 2. From inside the container, try to use the "docker stats" command. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The only way for a container to obtain information from the Docker daemon is to mount the /var/run/docker.sock socket. From this, the container is able to perform any Docker related command, including the "docker stats" command that allows it to see multiple metrics on running containers.

Figure 5.7: Testing procedure DOCKER-2-1

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

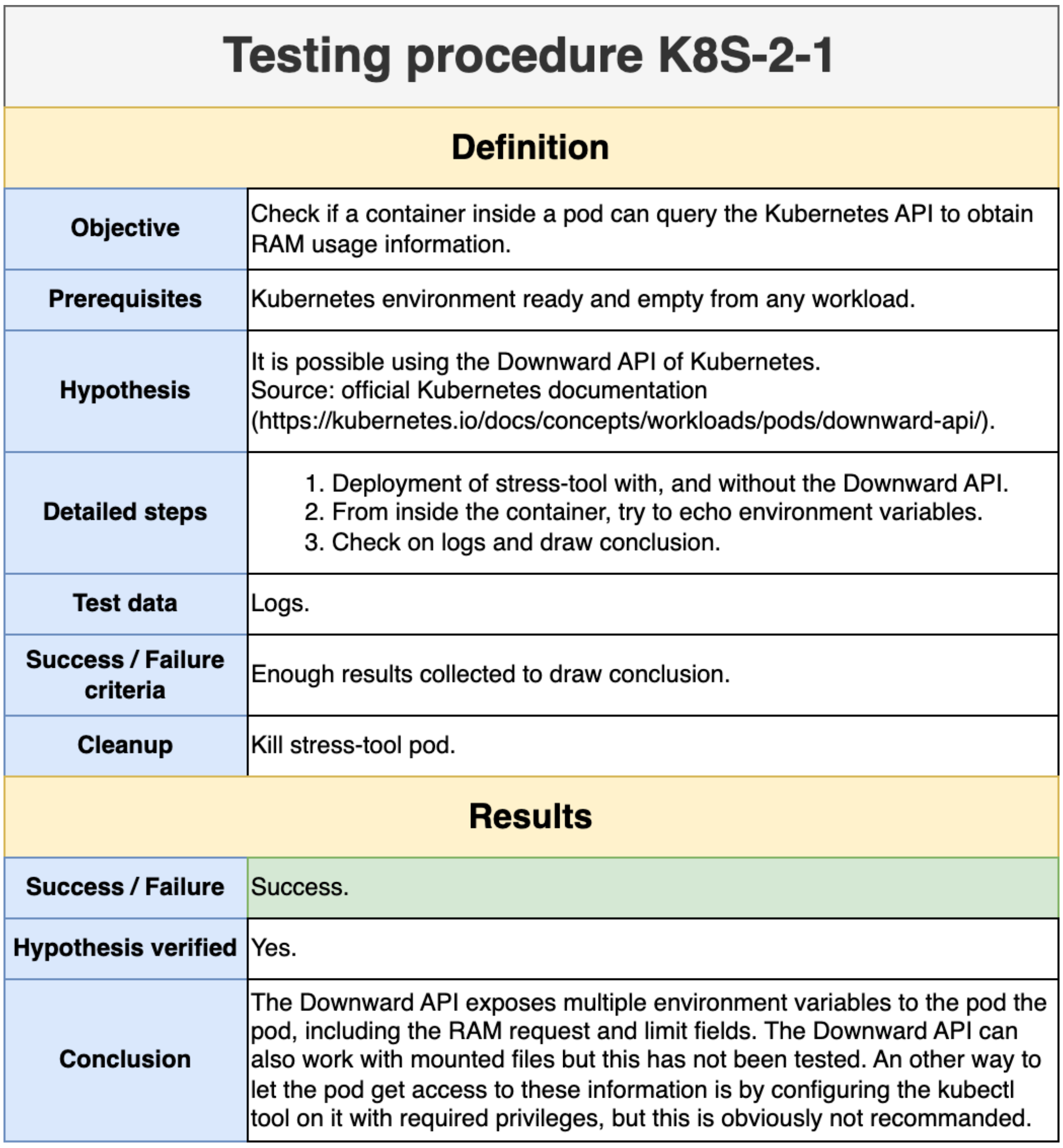
When the volume */var/run/docker.sock* is mounted to the container, it is possible to query the Docker API to gather information on running containers. For example, from inside the stress-tool container, once the Docker client in installed, the output of the "docker stats" command is the following (network I/O have been truncated for a better readibility):

```
CONTAINER ID   NAME            CPU %     MEM USAGE / LIMIT     MEM %     PIDS
a168b36241f4   stress-tool     0.81%     9.59MiB / 15.62GiB    0.06%     12
dfad28df2a48   node-exporter   0.00%     12.38MiB / 15.62GiB   0.08%     8
6b2cf6441ad5   cadvisor        5.98%     25.3MiB / 15.62GiB    0.16%     26
1c6b66913add   filebeat        0.62%     42.63MiB / 15.62GiB   0.27%     4
```

The cAdvisor container is using the same principle to monitor running containers. Note that mounting the */var/run/docker.sock* socket may represent a serious security issue as the container gain access to the whole Docker API.

5.2.2 Testing procedure K8S-2-1

The testing procedure K8S-2-1 is described below with associated results:



Testing procedure K8S-2-1	
Definition	
Objective	Check if a container inside a pod can query the Kubernetes API to obtain RAM usage information.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	It is possible using the Downward API of Kubernetes. Source: official Kubernetes documentation (https://kubernetes.io/docs/concepts/workloads/pods/downward-api/).
Detailed steps	1. Deployment of stress-tool with, and without the Downward API. 2. From inside the container, try to echo environment variables. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The Downward API exposes multiple environment variables to the pod the pod, including the RAM request and limit fields. The Downward API can also work with mounted files but this has not been tested. An other way to let the pod get access to these information is by configuring the kubectl tool on it with required privileges, but this is obviously not recommanded.

Figure 5.8: Testing procedure K8S-2-1

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

To use the Downward API, the YAML description of the pod has been modified to add environment variables that are given to the pod. The new pod description is the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: stress-tool
spec:
  restartPolicy: Never
  containers:
  - name: stress-tool-container
    image: leichap/heia-stress-tool
    imagePullPolicy: Always
    volumeMounts:
    - name: config-volume
      mountPath: /usr/src/stress-tool/config.txt
    resources:
      requests:
        memory: "300Mi"
      limits:
        memory: "300Mi"
    env:
    - name: MEMORY_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: stress-tool-container
          resource: requests.memory
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: stress-tool-container
          resource: limits.memory
  volumes:
  - name: config-volume
    hostPath:
      path: /home/ubuntu/config.txt
      type: File
```

From inside the pod, the administrator can echo these 2 environment variables and see the results:

```
[PS6] ubuntu@master01:~# kubectl exec -it stress-tool -- /bin/sh
# echo $MEMORY_REQUEST
314572800
# echo $MEMORY_LIMIT
314572800
```

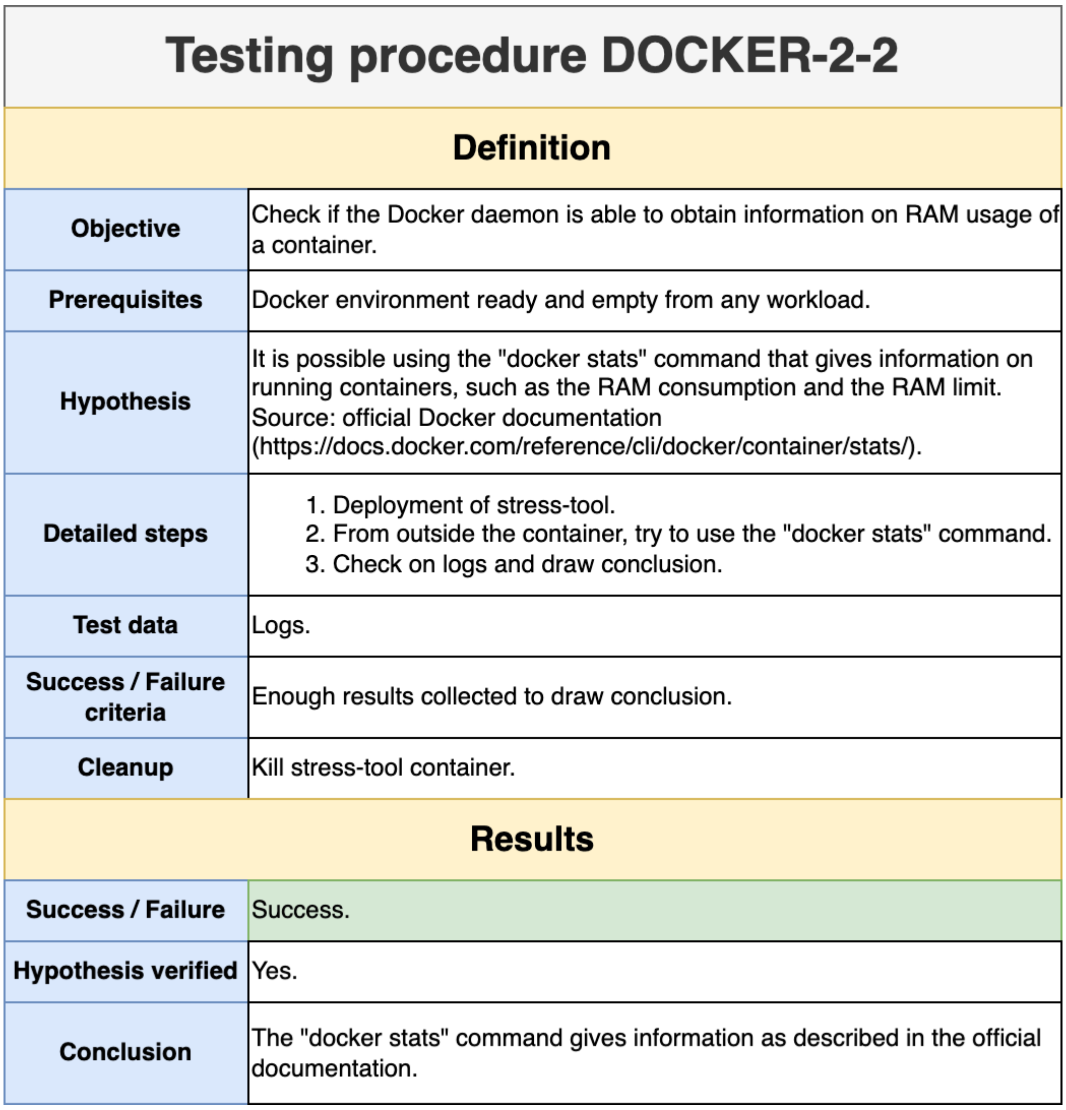
By calculating manually the exact number of bytes:

$$300\,\text{MiB} = 300 \times 1\,048\,576\,\text{bytes} = 300 \times 2^{20}\,\text{bytes} = 314\,572\,800\,\text{bytes}$$

The results are the same. The pod has access to its requests and limits fields using the Downward API in environment variables mode.

5.2.3 Testing procedure DOCKER-2-2

The testing procedure DOCKER-2-2 is described below with associated results:



Testing procedure DOCKER-2-2

Definition	
Objective	Check if the Docker daemon is able to obtain information on RAM usage of a container.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	It is possible using the "docker stats" command that gives information on running containers, such as the RAM consumption and the RAM limit. Source: official Docker documentation (https://docs.docker.com/reference/cli/docker/container/stats/).
Detailed steps	1. Deployment of stress-tool. 2. From outside the container, try to use the "docker stats" command. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The "docker stats" command gives information as described in the official documentation.

Figure 5.9: Testing procedure DOCKER-2-2

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

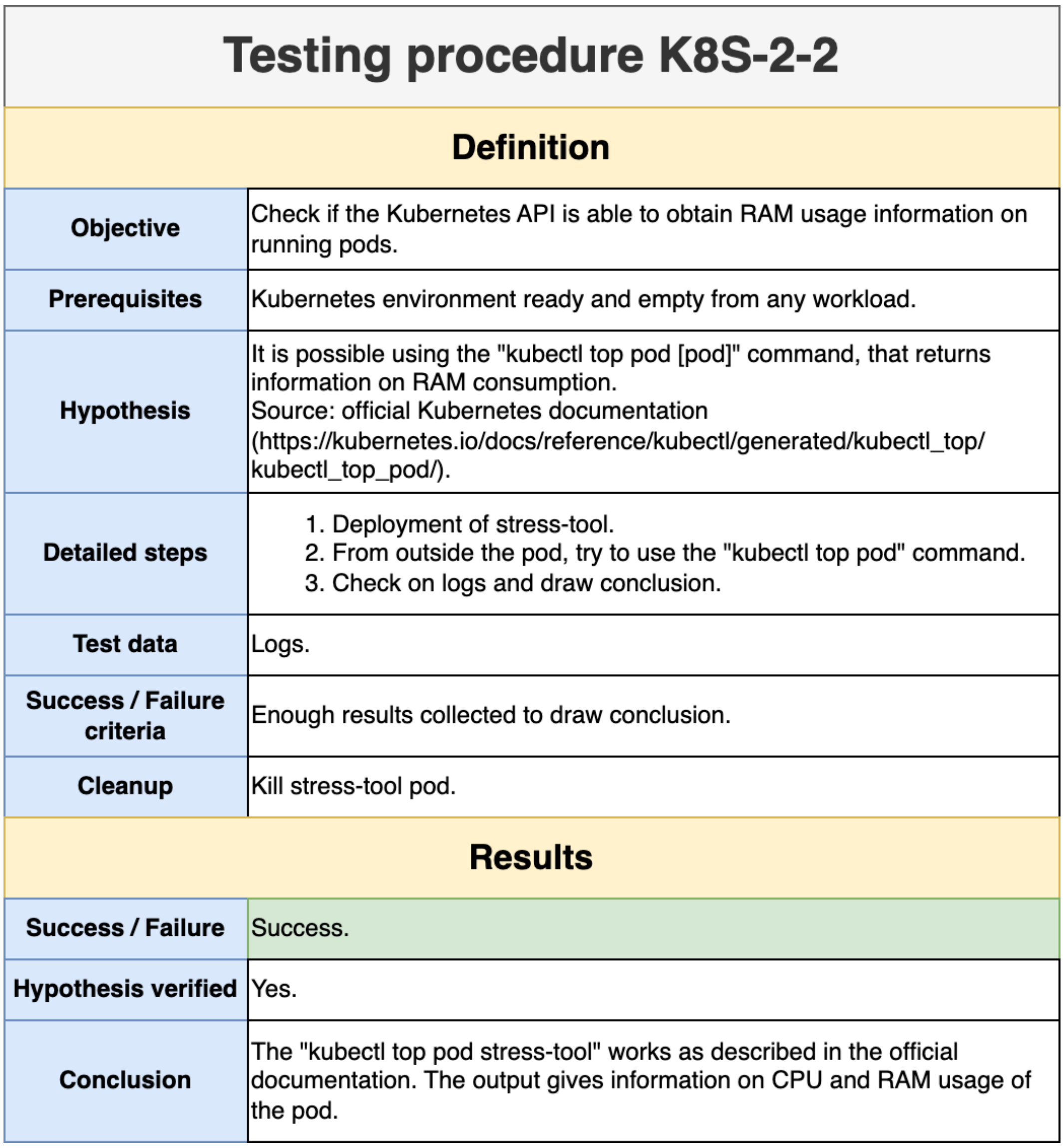
To test the "docker stats" command, the container must be started and the administrator must be logged in SSH on the Docker host. Once connected, they can try the "docker stats" command. The output is the following (network I/O have been truncated for a better readibility):

```
CONTAINER ID   NAME            CPU %     MEM USAGE / LIMIT     MEM %     PIDS
a168b36241f4   stress-tool     0.62%     10.14MiB / 15.62GiB   0.06%     13
dfad28df2a48   node-exporter   0.00%     12.09MiB / 15.62GiB   0.08%     8
6b2cf6441ad5   cadvisor        16.12%    25.12MiB / 15.62GiB   0.16%     26
1c6b66913add   filebeat        0.59%     44.04MiB / 15.62GiB   0.28%     15
```

Note that the values seen from inside and outside of the container are the same, as the same tool ("docker stats" command) is used. This is not true in other experiment cases (DOCKER-2-3 and K8S-2-3).

5.2.4 Testing procedure K8S-2-2

The testing procedure K8S-2-2 is described below with associated results:



Testing procedure K8S-2-2	
Definition	
Objective	Check if the Kubernetes API is able to obtain RAM usage information on running pods.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	It is possible using the "kubectl top pod [pod]" command, that returns information on RAM consumption. Source: official Kubernetes documentation (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_top/kubectl_top_pod/).
Detailed steps	1. Deployment of stress-tool. 2. From outside the pod, try to use the "kubectl top pod" command. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The "kubectl top pod stress-tool" works as described in the official documentation. The output gives information on CPU and RAM usage of the pod.

Figure 5.10: Testing procedure K8S-2-2

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

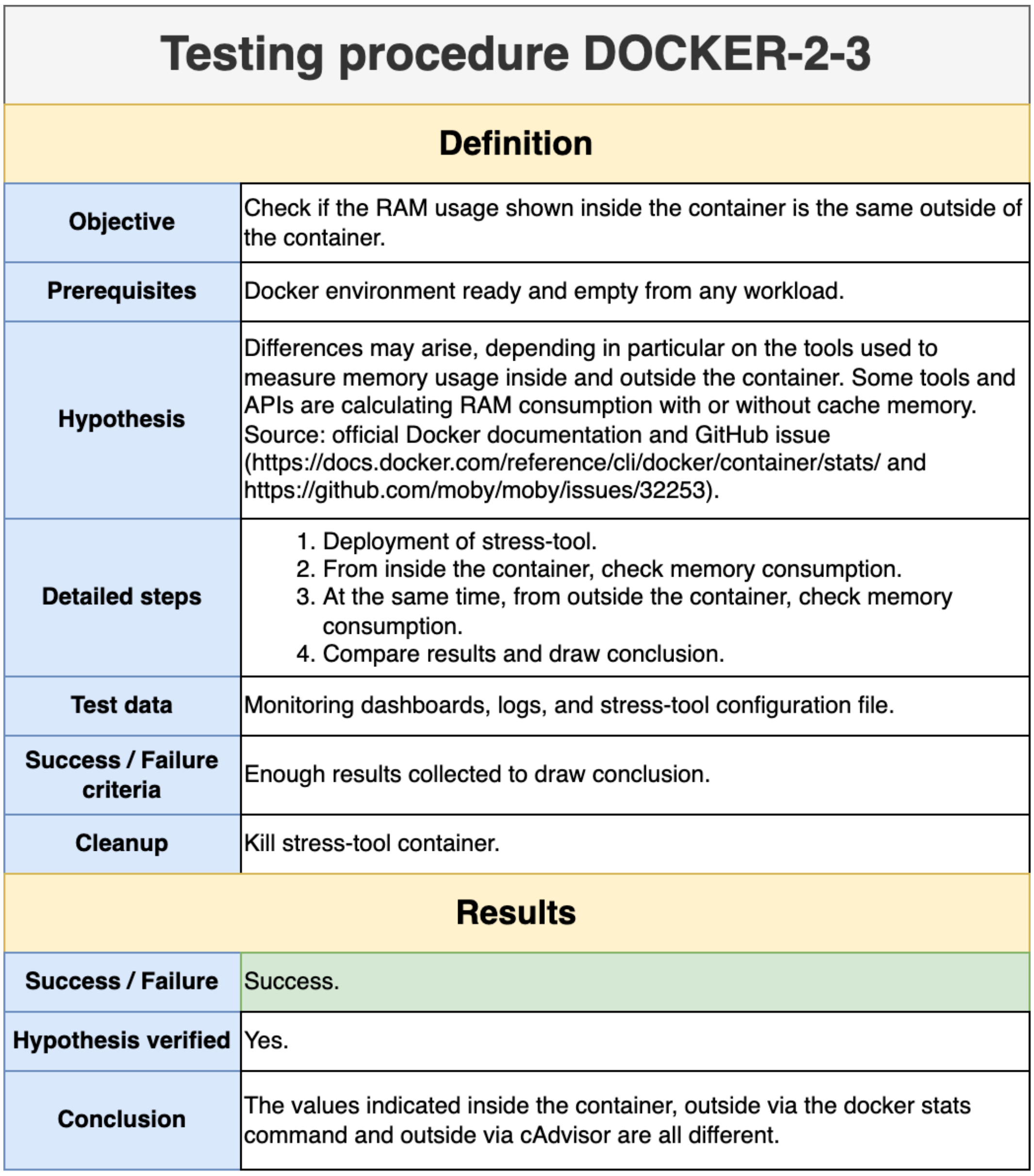
Note that to test the "kubectl top pod [pod]" command, the Metrics API must be installed and operational on the Kubernetes cluster. The result of the command "kubectl top pod stress-tool" is the following:

```
NAME                     CPU(Cores)   MEMORY(Bytes)
stress-tool                    3m           11Mi
```

The output shows the RAM consumption of the workload, from outside the pod.

5.2.5 Testing procedure DOCKER-2-3

The testing procedure DOCKER-2-3 is described below with associated results:



Testing procedure DOCKER-2-3	
Definition	
Objective	Check if the RAM usage shown inside the container is the same outside of the container.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	Differences may arise, depending in particular on the tools used to measure memory usage inside and outside the container. Some tools and APIs are calculating RAM consumption with or without cache memory. Source: official Docker documentation and GitHub issue (https://docs.docker.com/reference/cli/docker/container/stats/ and https://github.com/moby/moby/issues/32253).
Detailed steps	1. Deployment of stress-tool. 2. From inside the container, check memory consumption. 3. At the same time, from outside the container, check memory consumption. 4. Compare results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The values indicated inside the container, outside via the docker stats command and outside via cAdvisor are all different.

Figure 5.11: Testing procedure DOCKER-2-3

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1
wait 20
allocate 104857600 id2
wait 20
free id1
wait 20
allocate 52428800 id3
wait 20
free id2
wait 30
allocate 104857600 id1
wait 20
allocate 52428800 id4
wait 20
allocate 52428800 id5
wait 20
allocate 104857600 id2
wait 20
```

This configuration simulates a more realistic application where allocation and deallocation are made. The results obtained on Grafana are shown below:

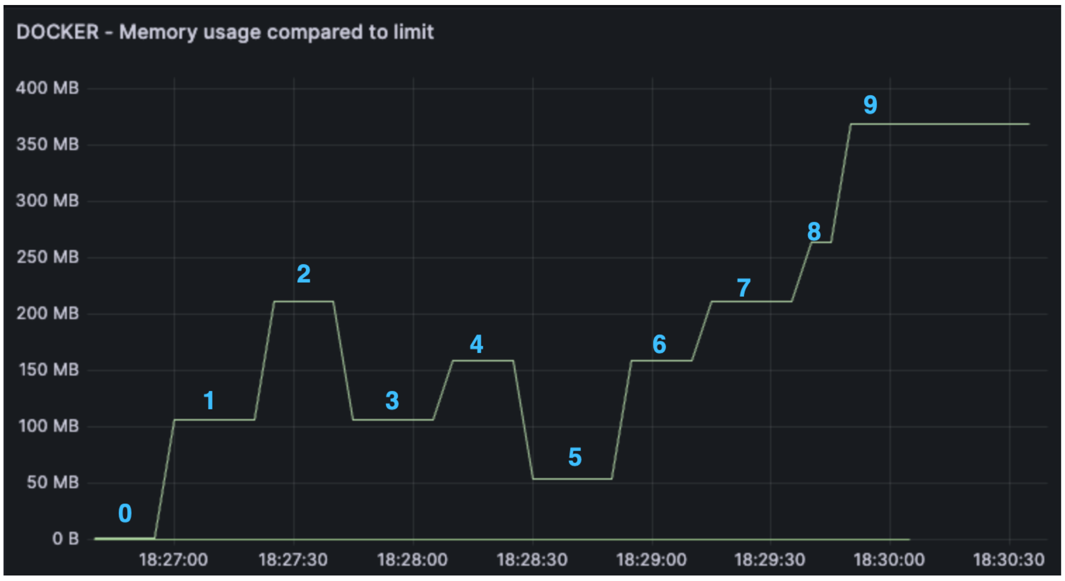


Figure 5.12: DOCKER-2-3: Grafana dashboard

The table below compare the values between the logs of the container (sent by a C++ library that shows the RAM consumption of the application), the result of the "docker stats" command and the Grafana dashboard.

GRAFANA STEP	EXPECTED VALUE (MiB)	GRAFANA VALUE (MiB)	DOCKER STATS VALUE (MiB)	LOGS VALUE (MiB)
0	0	1.02	0.996	1.5
1	100	106	101.3	103.06
2	200	211	201.5	205.46
3	100	106	101.3	103.19
4	150	159	151.4	155.5
5	50	53	51.23	55.53
6	150	159	151.4	155.51
7	200	211	201.5	205.46
8	250	264	251.6	255.54
9	350	369	351.8	355.47

Table 5.1: DOCKER-2-3: RAM usage on each step of the graph

Considering each value on step 0 as the base offset of the application when no allocation has been made, it is possible to substract this offset to other values to obtain the RAM used only by allocation instructions:

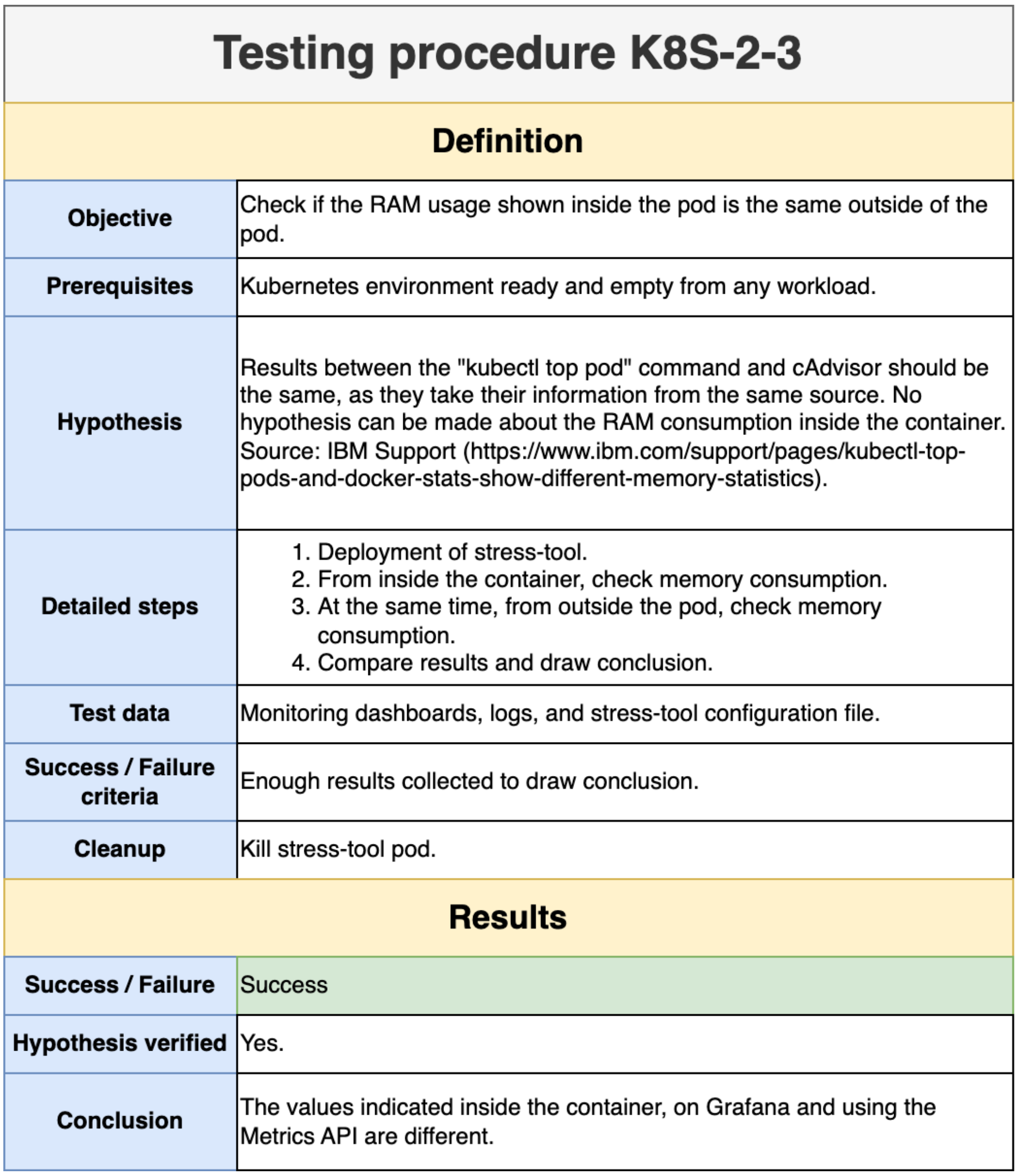
GRAFANA STEP	EXPECTED VALUE (MiB)	GRAFANA VALUE (MiB)	DOCKER STATS VALUE (MiB)	LOGS VALUE (MiB)
0	0	0	0	0
1	100	104.98	100.304	101.56
2	200	209.98	200.504	203.96
3	100	104.98	100.304	101.69
4	150	157.98	150.404	154
5	50	51.98	50.234	54.03
6	150	157.98	150.404	154.01
7	200	209.98	200.504	203.96
8	250	262.98	250.604	254.04
9	350	367.98	350.804	353.97

Table 5.2: DOCKER-2-3: Offset subtracted to each value

It is possible to see that there is no step where the 3 values are equal. Each tool may access and interpret memory usage data in slightly different ways, leading to variations. The C++ library might account for specific overheads or internal metrics, such as memory used for caching or buffering, which cAdvisor or Docker stats might not take into account. In the table, Grafana's values are consistently slightly higher than Docker stats, while the log values are also higher than Docker stats but generally lower than Grafana's values. An explanation to the difference between Grafana (which takes its data from cAdvisor through Prometheus) and the "docker stats" command is that the Docker CLI subtracts cache memory from the total RAM used. The Docker API used by cAdvisor does not do this[7].

5.2.6 Testing procedure K8S-2-3

The testing procedure K8S-2-3 is described below with associated results:



Testing procedure K8S-2-3

Definition	
Objective	Check if the RAM usage shown inside the pod is the same outside of the pod.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	Results between the "kubectl top pod" command and cAdvisor should be the same, as they take their information from the same source. No hypothesis can be made about the RAM consumption inside the container. Source: IBM Support (https://www.ibm.com/support/pages/kubectl-top-pods-and-docker-stats-show-different-memory-statistics).
Detailed steps	1. Deployment of stress-tool. 2. From inside the container, check memory consumption. 3. At the same time, from outside the pod, check memory consumption. 4. Compare results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success
Hypothesis verified	Yes.
Conclusion	The values indicated inside the container, on Grafana and using the Metrics API are different.

Figure 5.13: Testing procedure K8S-2-3

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1
wait 20
allocate 104857600 id2
wait 20
free id1
wait 20
allocate 52428800 id3
wait 20
free id2
wait 30
allocate 104857600 id1
wait 20
allocate 52428800 id4
wait 20
allocate 52428800 id5
wait 20
allocate 104857600 id2
wait 20
```

This configuration simulates a more realistic application where allocation and deallocation are made. The results obtained on Grafana are shown below:

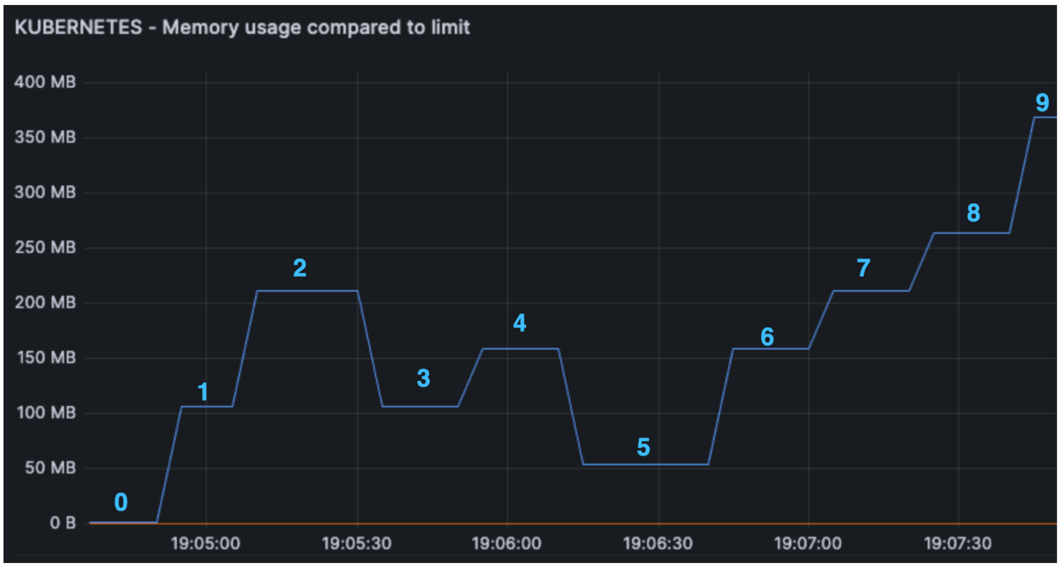


Figure 5.14: K8S-2-3: Grafana dashboard

The tables below compare the values between the logs of the pod (sent by a C++ library that shows the RAM consumption of the application), the result of the "kubectl top pod" command that uses the Metrics API and the Grafana dashboard. The expected value column represents the total of RAM theoretically used by the current program, depending on the configuration of the experiment.

GRAFANA STEP	EXPECTED VALUE (MiB)	GRAFANA VALUE (MiB)	METRICS API VALUE (MiB)	LOGS VALUE (MiB)
0	0	4.44	2	1.5
1	100	106	101	105.5
2	200	211	201	205.45
3	100	106	101	105.5
4	150	159	151	155.5
5	50	53	51	55.52
6	150	159	151	155.5
7	200	211	201	205.46
8	250	264	251	255.53
9	250	369	351	355.47

Table 5.3: K8S-2-3: RAM usage on each step of the graph

Considering each value on step 0 as the base offset of the application when no allocation has been made, it is possible to substract this offset to other values to obtain the RAM used only by allocation instructions:

GRAFANA STEP	EXPECTED VALUE (MiB)	GRAFANA VALUE (MiB)	METRICS API VALUE (MiB)	LOGS VALUE (MiB)
0	0	0	0	0
1	100	101.56	99	104
2	200	206.56	199	203.95
3	100	101.56	99	104
4	150	154.56	149	154
5	50	48.56	49	54.02
6	150	154.56	149	154
7	200	206.56	199	203.96
8	250	259.56	249	254.03
9	250	364.56	349	353.97

Table 5.4: K8S-2-3: Offset subtracted to each value

The main takeaway is that Grafana tends to show slightly higher RAM usage compared to the values recorded in the container logs and the Metrics API, likely due to the factors mentioned previously in the DOCKER-2-3 experiment. Note that even when the base offset is subtracted, the values of RAM usage are not equal to allocation size of each step. The fact that the Metrics API gives values that seem very similar (all ending in 1) may be due to the fact that this API is only used for cluster scaling. The official Kubernetes documentation does not recommend using this API for monitoring, but rather using specialised tools[31]:

> *Integration of a full metrics pipeline into your Kubernetes implementation is outside the scope of Kubernetes documentation because of the very wide scope of possible solutions. The choice of monitoring platform depends heavily on your needs, budget, and technical resources. Kubernetes does not recommend any specific metrics pipeline; many options are available. Your monitoring system should be capable of handling the OpenMetrics metrics transmission standard and needs to be chosen to best fit into your overall design and deployment of your infrastructure platform.*[32]

5.2.7 Comparison of DOCKER-2-3 and K8S-2-3

It may be interesting to compare the values of the tables obtained on Docker and Kubernetes to try and highlight similarities and differences.

Comparison of the values given by cAdvisor
The comparison of the values given by the cAdvisor tool and displayed on Grafana is available in the following table:

GRAFANA STEP	DOCKER VALUE (MiB)	KUBERNETES VALUE (MiB)	DIFFERENCE (MiB)
0	1.02	4.44	335.29%
1	106	106	0.00%
2	211	211	0.00%
3	106	106	0.00%
4	159	159	0.00%
5	53	53	0.00%
6	159	159	0.00%
7	211	211	0.00%
8	264	264	0.00%
9	369	369	0.00%
Average Variance (%)			**33.53%**

Table 5.5: Comparison of DOCKER-2-3 and K8S-2-3: cAdvisor values

The average variance does not reflect the dataset as the first step is the only to have different values. This may be due to an error of measurement, even though the experiment has been done 3 times. If this step is considered as an error and not taken into account, the variance of the cAdvisor datasource is 0.0%, pointing out the fact that cAdvisor gives the same results on Docker and Kubernetes for this workload. The reason why the Grafana and Metrics API values are higher than step 0 on Kubernetes has not been found, especially as the value of the logs for this same step is equal to the value of the Docker environment.

Comparison of the values given by docker stats and kubectl top pod
The comparison of the values given by the "docker stats" and "kubectl top pod" commands is available in the following table:

GRAFANA STEP	**DOCKER VALUE (MiB)**	**KUBERNETES VALUE (MiB)**	**DIFFERENCE (MiB)**
0	0.996	2	100.80%
1	101.3	101	0.30%
2	201.5	201	0.25%
3	101.3	101	0.30%
4	151.4	151	0.26%
5	51.23	51	0.45%
6	151.4	151	0.26%
7	201.5	201	0.25%
8	251.6	251	0.24%
9	351.8	351	0.23%
Average Variance (%)			**10.33%**

Table 5.6: Comparison of DOCKER-2-3 and K8S-2-3: docker stats and kubectl top pod values

As for the latest comparison, the first line seems very out of step with all the others. It's possible that for some reason the offset of a workload is different on Docker and Kubernetes. In any case, omitting this line, the average variance is 0.284%.

Comparison of the values given by the logs of the application
As observed in 5.1 and 5.3 tables, the values given by the logs of the C++ stress-tool application are very similar:

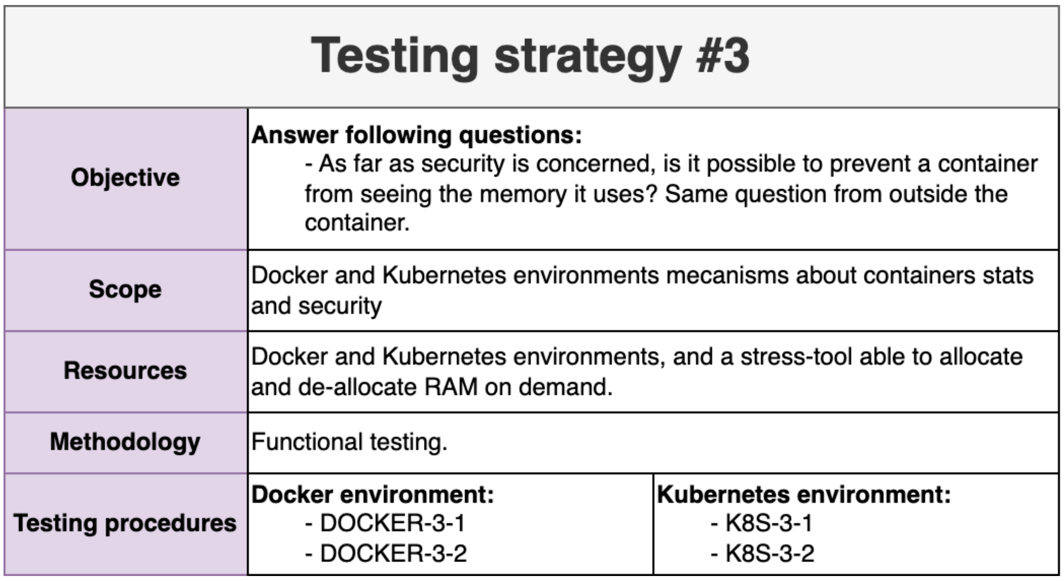
GRAFANA STEP	**DOCKER VALUE (MiB)**	**KUBERNETES VALUE (MiB)**	**DIFFERENCE (MiB)**
0	1.5	1.5	0.00%
1	103.06	105.5	2.36%
2	205.46	205.45	0.00%
3	103.19	105.5	2.24%
4	155.5	155.5	0.00%
5	55.53	55.52	0.02%
6	155.51	155.5	0.01%
7	205.46	205.46	0.00%
8	255.54	255.53	0.00%
9	355.47	355.47	0.00%
Average Variance (%)			**0.463%**

Table 5.7: Comparison of DOCKER-2-3 and K8S-2-3: logs values

The average variance is 0.463%, which is a fairly low value. The C++ application appears to be a reliable source of information regardless of the platform on which it is run.

5.3 Testing strategy 3

The third testing strategy of the project is described below:



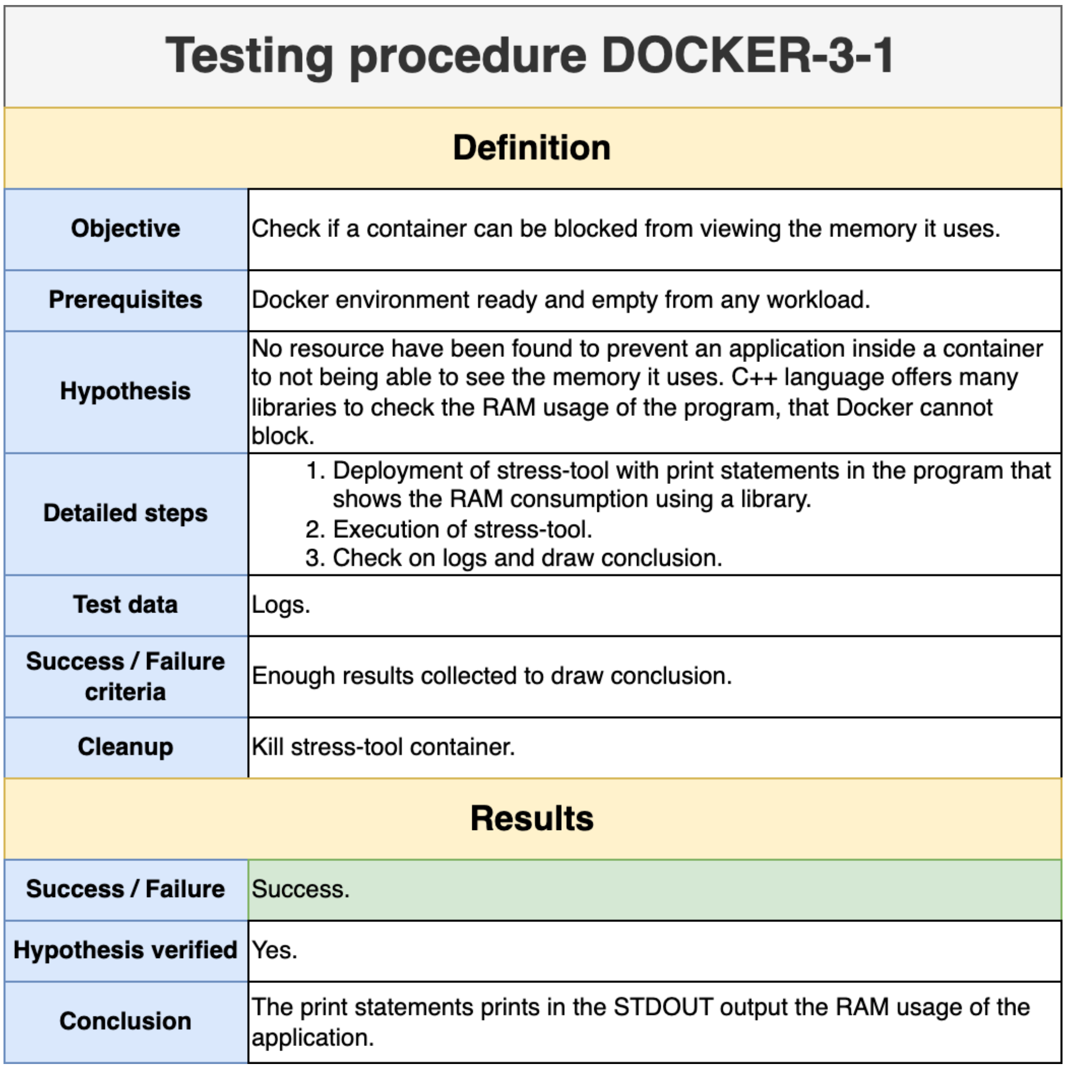
Testing strategy #3		
Objective	**Answer following questions:** - As far as security is concerned, is it possible to prevent a container from seeing the memory it uses? Same question from outside the container.	
Scope	Docker and Kubernetes environments mecanisms about containers stats and security	
Resources	Docker and Kubernetes environments, and a stress-tool able to allocate and de-allocate RAM on demand.	
Methodology	Functional testing.	
Testing procedures	**Docker environment:** - DOCKER-3-1 - DOCKER-3-2	**Kubernetes environment:** - K8S-3-1 - K8S-3-2

Figure 5.15: Testing strategy 3

Its purpose is to answer the third question asked in the objective section of this report.

5.3.1 Testing procedure DOCKER-3-1

The testing procedure DOCKER-3-1 is described below with associated results:



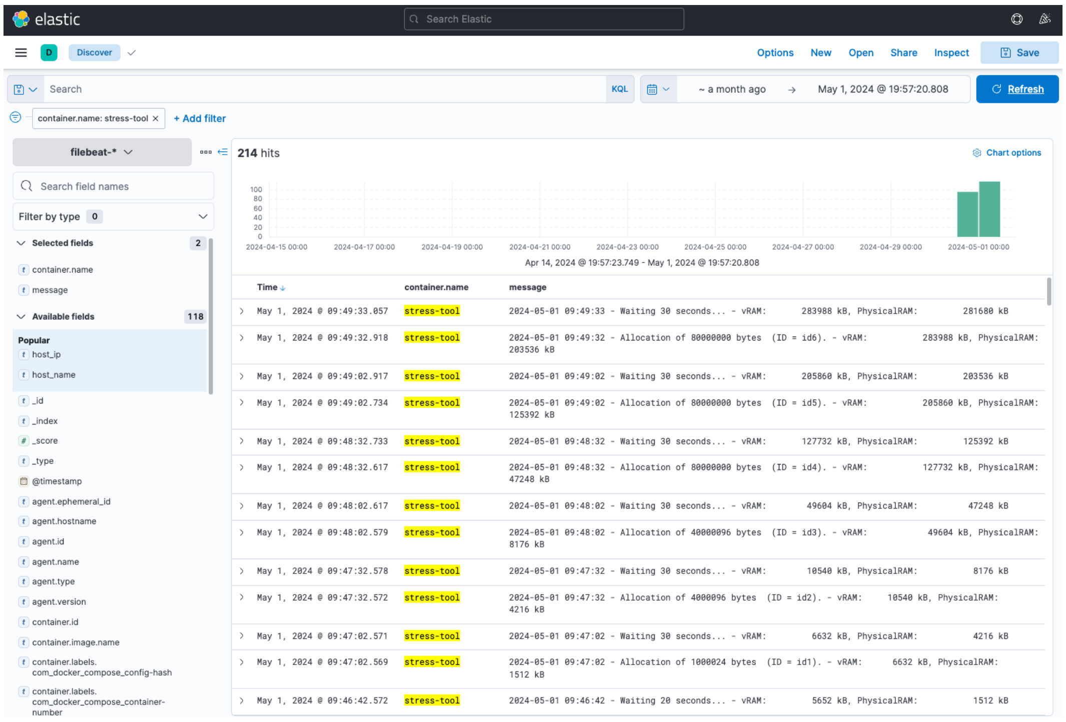
Testing procedure DOCKER-3-1	
Definition	
Objective	Check if a container can be blocked from viewing the memory it uses.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	No resource have been found to prevent an application inside a container to not being able to see the memory it uses. C++ language offers many libraries to check the RAM usage of the program, that Docker cannot block.
Detailed steps	1. Deployment of stress-tool with print statements in the program that shows the RAM consumption using a library. 2. Execution of stress-tool. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The print statements prints in the STDOUT output the RAM usage of the application.

Figure 5.16: Testing procedure DOCKER-3-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1
wait 20
allocate 104857600 id2
wait 20
free id1
wait 20
free id2
wait 20
```

This is a normal allocation procedure without any specificity. The goal is to know if the application can see the RAM it uses. To do so, the logs of the container can be seen on Kibana. For example, this is what Kibana looks like when filtering logs of a specified container:



Time	container.name	message
May 1, 2024 @ 09:49:33.057	stress-tool	2024-05-01 09:49:33 - Waiting 30 seconds... - vRAM: 283988 kB, PhysicalRAM: 281680 kB
May 1, 2024 @ 09:49:32.918	stress-tool	2024-05-01 09:49:32 - Allocation of 80000000 bytes (ID = id6). - vRAM: 283988 kB, PhysicalRAM: 203536 kB
May 1, 2024 @ 09:49:02.917	stress-tool	2024-05-01 09:49:02 - Waiting 30 seconds... - vRAM: 205860 kB, PhysicalRAM: 203536 kB
May 1, 2024 @ 09:49:02.734	stress-tool	2024-05-01 09:49:02 - Allocation of 80000000 bytes (ID = id5). - vRAM: 205860 kB, PhysicalRAM: 125392 kB
May 1, 2024 @ 09:48:32.733	stress-tool	2024-05-01 09:48:32 - Waiting 30 seconds... - vRAM: 127732 kB, PhysicalRAM: 125392 kB
May 1, 2024 @ 09:48:32.617	stress-tool	2024-05-01 09:48:32 - Allocation of 80000000 bytes (ID = id4). - vRAM: 127732 kB, PhysicalRAM: 47248 kB
May 1, 2024 @ 09:48:02.617	stress-tool	2024-05-01 09:48:02 - Waiting 30 seconds... - vRAM: 49604 kB, PhysicalRAM: 47248 kB
May 1, 2024 @ 09:48:02.579	stress-tool	2024-05-01 09:48:02 - Allocation of 40000096 bytes (ID = id3). - vRAM: 49604 kB, PhysicalRAM: 8176 kB
May 1, 2024 @ 09:47:32.578	stress-tool	2024-05-01 09:47:32 - Waiting 30 seconds... - vRAM: 10540 kB, PhysicalRAM: 8176 kB
May 1, 2024 @ 09:47:32.572	stress-tool	2024-05-01 09:47:32 - Allocation of 4000096 bytes (ID = id2). - vRAM: 10540 kB, PhysicalRAM: 4216 kB
May 1, 2024 @ 09:47:02.571	stress-tool	2024-05-01 09:47:02 - Waiting 30 seconds... - vRAM: 6632 kB, PhysicalRAM: 4216 kB
May 1, 2024 @ 09:47:02.569	stress-tool	2024-05-01 09:47:02 - Allocation of 1000024 bytes (ID = id1). - vRAM: 6632 kB, PhysicalRAM: 1512 kB
May 1, 2024 @ 09:46:42.572	stress-tool	2024-05-01 09:46:42 - Waiting 20 seconds... - vRAM: 5652 kB, PhysicalRAM: 1512 kB

Figure 5.17: *Kibana's interface with logs of the stress-tool*

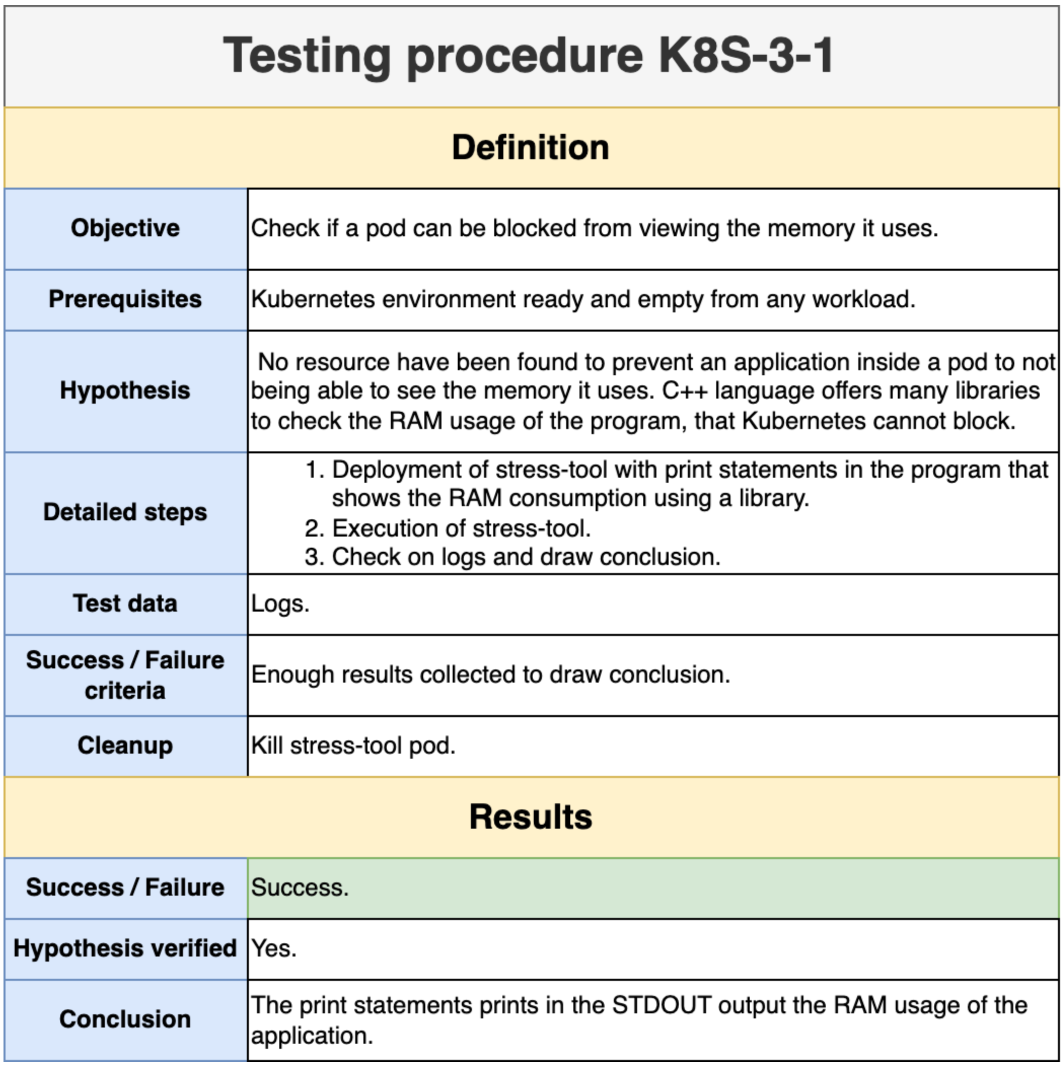
For a better readability, the logs have been summarized in the following text box (the date is omitted to gain space):

```
19:48:17 - Starting stress tool. - vRAM:5652 kB, PhysicalRAM:1508 kB
19:48:17 - Waiting 20 seconds... - vRAM:5652 kB, PhysicalRAM:1508 kB
19:48:37 - Allocation of 104857600 bytes  (ID = id1). - vRAM:108056 kB,
↪  PhysicalRAM:1508 kB
19:48:38 - Waiting 20 seconds... - vRAM:108056 kB, PhysicalRAM:05608 kB
19:48:58 - Allocation of 104857600 bytes  (ID = id2). - vRAM:210460 kB,
↪  PhysicalRAM:105608 kB
19:48:58 - Waiting 20 seconds... - vRAM:210460 kB, PhysicalRAM:208040 kB
19:49:18 - De-allocating memory block with ID = id1. - vRAM:108056 kB,
↪  PhysicalRAM:105792 kB
19:49:18 - Waiting 20 seconds... - vRAM:108056 kB, PhysicalRAM:105792 kB
19:49:38 - De-allocating memory block with ID = id2. - vRAM:5652 kB,
↪  PhysicalRAM:3388 kB
19:49:38 - Waiting 20 seconds... - vRAM:5652 kB, PhysicalRAM:3388 kB
19:49:58 - End of program. - vRAM:5652 kB, PhysicalRAM:3388 kB
```

The containerized application is fully capable of seeing the RAM it uses by reading the values from the /proc/self/status file.

5.3.2 Testing procedure K8S-3-1

The testing procedure K8S-3-1 is described below with associated results:



Testing procedure K8S-3-1	
Definition	
Objective	Check if a pod can be blocked from viewing the memory it uses.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	No resource have been found to prevent an application inside a pod to not being able to see the memory it uses. C++ language offers many libraries to check the RAM usage of the program, that Kubernetes cannot block.
Detailed steps	1. Deployment of stress-tool with print statements in the program that shows the RAM consumption using a library. 2. Execution of stress-tool. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The print statements prints in the STDOUT output the RAM usage of the application.

Figure 5.18: Testing procedure K8S-3-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 104857600 id1
wait 20
allocate 104857600 id2
wait 20
free id1
wait 20
free id2
wait 20
```

This is a normal allocation procedure without any specificity. The goal is to know if the application can see the RAM it uses. To do so, the logs of the container can be seen on Kibana.

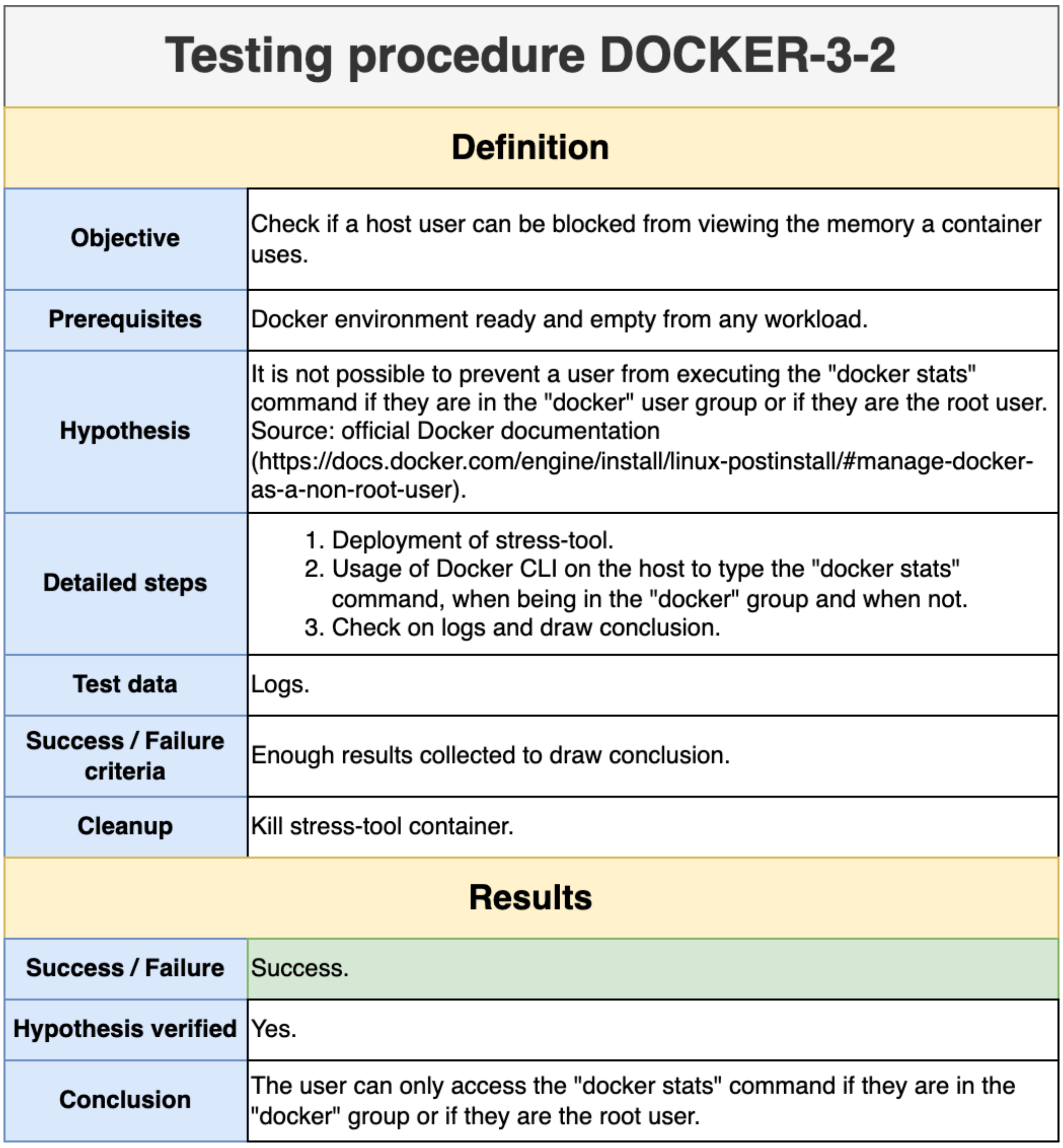
For a better readability, the logs have been summarized in the following text box (the date is omitted to gain space):

```
20:44:22 - Starting stress tool. - vRAM:5652 kB, PhysicalRAM:1512 kB
20:44:22 - Waiting 20 seconds... - vRAM:5652 kB, PhysicalRAM:1512 kB
20:44:42 - Allocation of 104857600 bytes  (ID = id1). - vRAM:108056 kB,
 ↪  PhysicalRAM:1512 kB
20:44:42 - Waiting 20 seconds... - vRAM:108056 kB, PhysicalRAM:105516 kB
20:45:02 - Allocation of 104857600 bytes  (ID = id2). - vRAM:210460 kB,
 ↪  PhysicalRAM:105516 kB
20:45:02 - Waiting 20 seconds... - vRAM:210460 kB, PhysicalRAM:207948 kB
20:45:22 - De-allocating memory block with ID = id1. - vRAM:108056 kB,
 ↪  PhysicalRAM:105696 kB
20:45:22 - Waiting 20 seconds... - vRAM:108056 kB, PhysicalRAM:105696 kB
20:45:42 - De-allocating memory block with ID = id2. - vRAM:5652 kB,
 ↪  PhysicalRAM:3292 kB
20:45:42 - Waiting 20 seconds... - vRAM:5652 kB, PhysicalRAM:3292 kB
2024-05-14 20:46:02 - End of program. - vRAM:5652 kB, PhysicalRAM:3292 kB
```

As for the Docker environment test, the containerized application is fully capable of seeing the RAM it uses by reading the values from the /proc/self/status file.

5.3.3 Testing procedure DOCKER-3-2

The testing procedure DOCKER-3-2 is described below with associated results:



Testing procedure DOCKER-3-2	
Definition	
Objective	Check if a host user can be blocked from viewing the memory a container uses.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	It is not possible to prevent a user from executing the "docker stats" command if they are in the "docker" user group or if they are the root user. Source: official Docker documentation (https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user).
Detailed steps	1. Deployment of stress-tool. 2. Usage of Docker CLI on the host to type the "docker stats" command, when being in the "docker" group and when not. 3. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool container.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The user can only access the "docker stats" command if they are in the "docker" group or if they are the root user.

Figure 5.19: Testing procedure DOCKER-3-2

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

Here is the result when the user "ubuntu" is not in the "docker" group:

```
[PS6] ubuntu@docker:~# docker stats
permission denied while trying to connect to the Docker daemon socket at
 ↪  unix:///var/run/docker.sock: Get
 ↪  "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/version": dial unix
 ↪  /var/run/docker.sock: connect: permission denied
```

The action is forbidden due to a lack of permission. The user cannot access the Docker daemon socket, therefore they cannot query the Docker API to obtain information on RAM usage of containers. This prohibition can be bypassed by retrieving information from other places on the server, for example by using the "htop" command.

To add the user "ubuntu" to the "docker" group, the following command can be run as root:

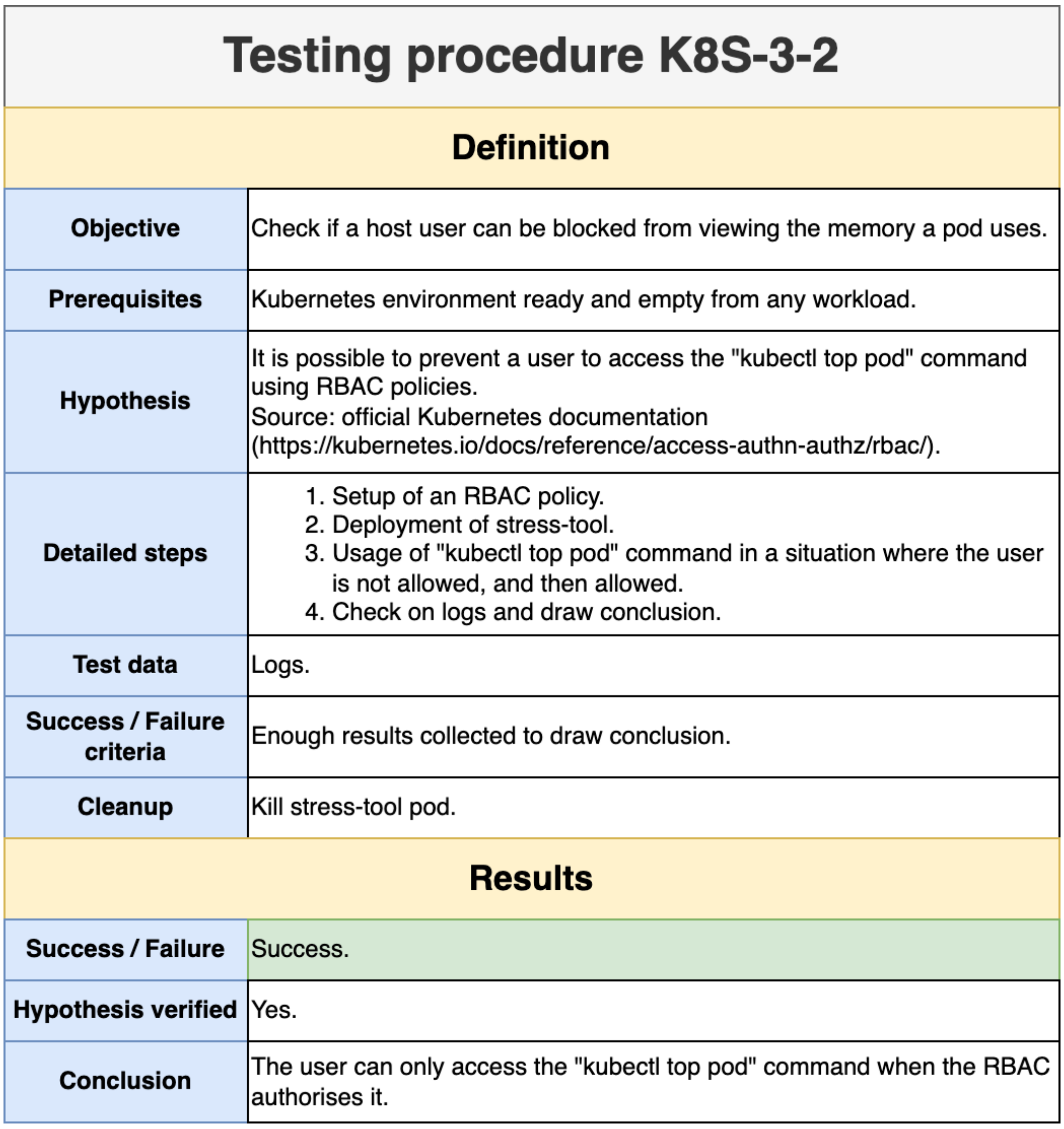
```
[PS6] root@docker:~# sudo usermod -aG docker ubuntu
```

Then, the "ubuntu" user can perform any Docker-related command such as "docker stats", "docker ps"...

```
[PS6] ubuntu@docker:~# docker --version
Docker version 24.0.5, build 24.0.5-0ubuntu1~20.04.1
```

5.3.4 Testing procedure K8S-3-2

The testing procedure K8S-3-2 is described below with associated results:



Testing procedure K8S-3-2	
Definition	
Objective	Check if a host user can be blocked from viewing the memory a pod uses.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	It is possible to prevent a user to access the "kubectl top pod" command using RBAC policies. Source: official Kubernetes documentation (https://kubernetes.io/docs/reference/access-authn-authz/rbac/).
Detailed steps	1. Setup of an RBAC policy. 2. Deployment of stress-tool. 3. Usage of "kubectl top pod" command in a situation where the user is not allowed, and then allowed. 4. Check on logs and draw conclusion.
Test data	Logs.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill stress-tool pod.
Results	
Success / Failure	Success.
Hypothesis verified	Yes.
Conclusion	The user can only access the "kubectl top pod" command when the RBAC authorises it.

Figure 5.20: Testing procedure K8S-3-2

The experiment created for this testing procedure is the following:

```
wait 300 # give time to administrator to enter into the container and type
↪  commands
```

It is possible to restrict the access to the "kubectl top pod" using RBAC (Role-based access control), which is a security framework in Kubernetes that controls access to resources based on the roles of individual users within an organization. It ensures that only authorized users (based on their name or group) can perform specific actions on resources. To restrict the access to the command, it is needed to create 2 Kubernetes objects:

1. First, a ClusterRole with the necessary permissions to access pod metrics. The following ClusterRole has been created:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: top-pod-clusterrole
   rules:
   - apiGroups: [""]
     resources: ["pods"]
     verbs: ["get", "list"]
   - apiGroups: ["metrics.k8s.io"]
     resources: ["pods"]
     verbs: ["get", "list"]
   ```

2. Next, a ClusterRoleBinding to associate the ClusterRole with the specific user that should have access to the kubectl top pod command. The following Cluster-RoleBinding has been created:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRoleBinding
   metadata:
     name: top-pod-clusterrolebinding
   subjects:
   - kind: User
     name: "ubuntu"
     apiGroup: rbac.authorization.k8s.io
   roleRef:
     kind: ClusterRole
     name: top-pod-clusterrole
     apiGroup: rbac.authorization.k8s.io
   ```

Then, the administrator can apply these 2 configurations and the command will be only available to the "ubuntu" user and cluster administrators, that have full permission regardless of the above configuration. When the configuration is applied, the user "ubuntu" can access the command without any problem:

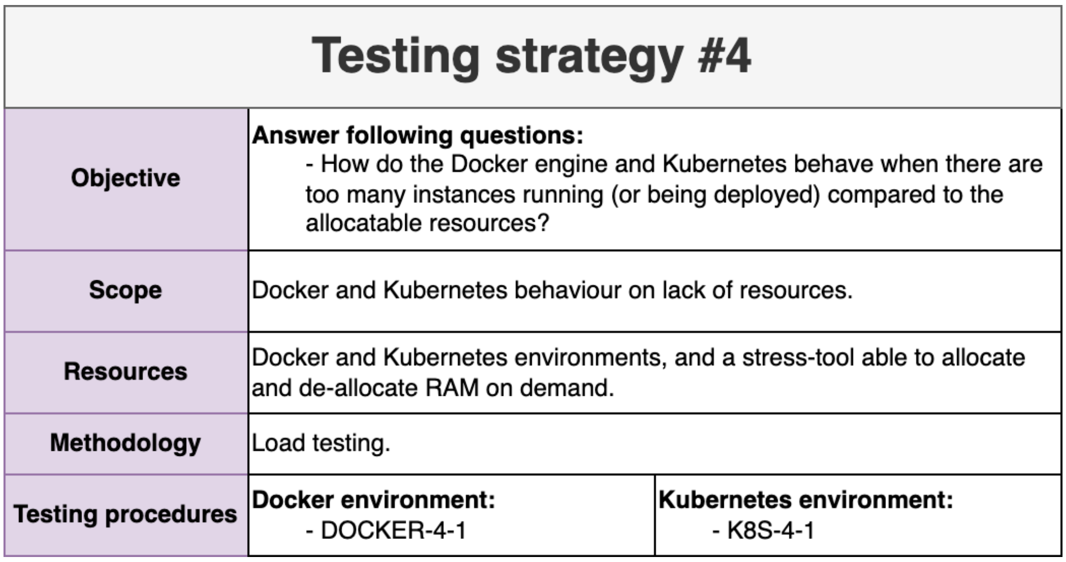
```
[PS6] ubuntu@master01:~# kubectl top pod
NAME           CPU(cores)   MEMORY(bytes)
stress-tool   1m           101Mi
```

When the configuration is deleted, or when the username is changed, the user "ubuntu" gets the following error, stating this action is forbidden as intended:

```
Error from server (Forbidden): pods.metrics.k8s.io is forbidden: User "ubuntu"
↪  cannot list resource "pods" in API group "metrics.k8s.io" in the namespace
↪  "default"
```

5.4 Testing strategy 4

The fourth testing strategy of the project is described below:



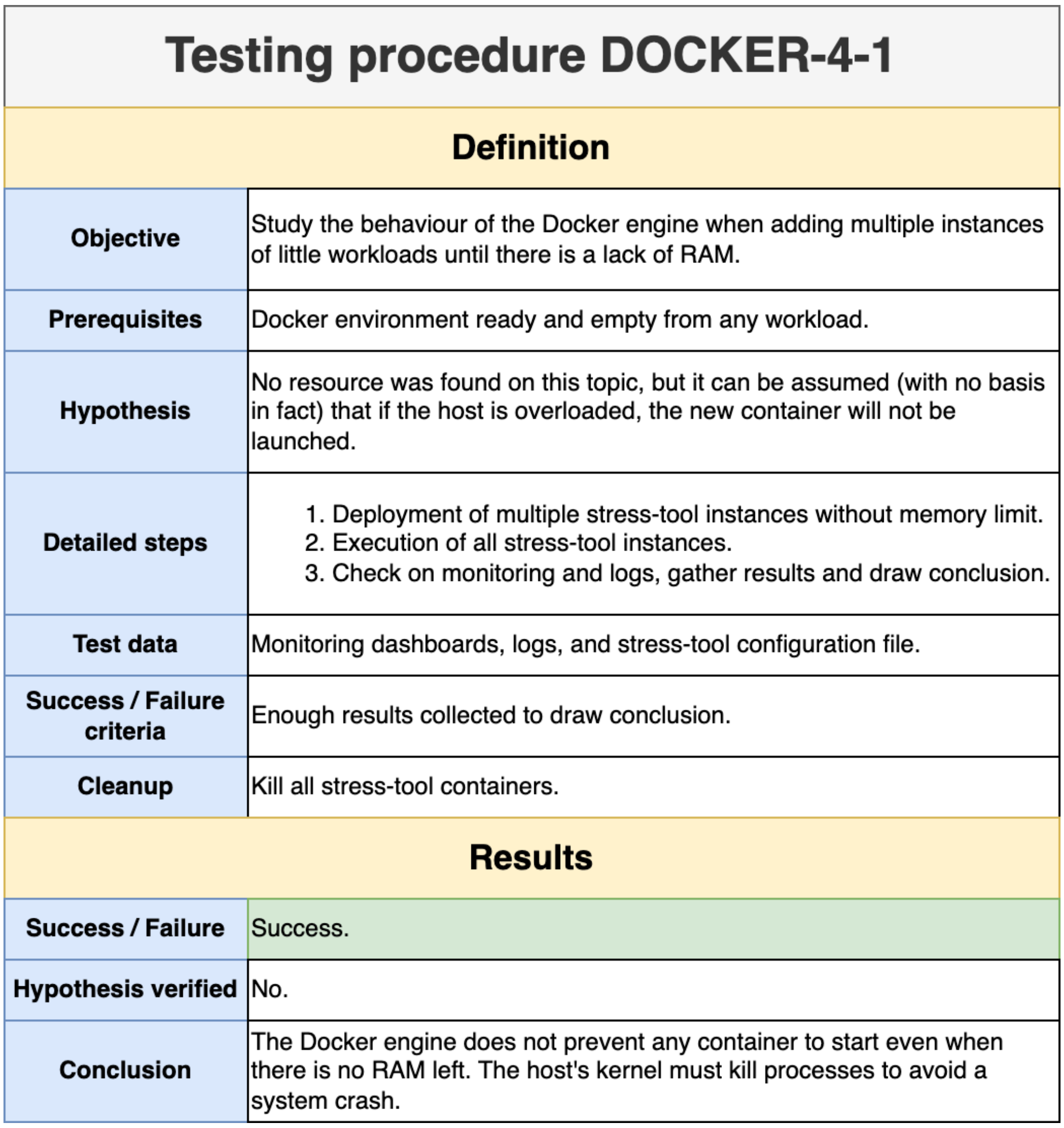
Testing strategy #4		
Objective	**Answer following questions:** - How do the Docker engine and Kubernetes behave when there are too many instances running (or being deployed) compared to the allocatable resources?	
Scope	Docker and Kubernetes behaviour on lack of resources.	
Resources	Docker and Kubernetes environments, and a stress-tool able to allocate and de-allocate RAM on demand.	
Methodology	Load testing.	
Testing procedures	**Docker environment:** - DOCKER-4-1	**Kubernetes environment:** - K8S-4-1

Figure 5.21: Testing strategy 4

Its purpose is to answer the last question asked in the objective section of this report.

5.4.1 Testing procedure DOCKER-4-1

The testing procedure DOCKER-4-1 is described below with associated results:



Testing procedure DOCKER-4-1

Definition	
Objective	Study the behaviour of the Docker engine when adding multiple instances of little workloads until there is a lack of RAM.
Prerequisites	Docker environment ready and empty from any workload.
Hypothesis	No resource was found on this topic, but it can be assumed (with no basis in fact) that if the host is overloaded, the new container will not be launched.
Detailed steps	1. Deployment of multiple stress-tool instances without memory limit. 2. Execution of all stress-tool instances. 3. Check on monitoring and logs, gather results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill all stress-tool containers.
Results	
Success / Failure	Success.
Hypothesis verified	No.
Conclusion	The Docker engine does not prevent any container to start even when there is no RAM left. The host's kernel must kill processes to avoid a system crash.

Figure 5.22: Testing procedure DOCKER-4-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 1073741824 id1 # 1GB
wait 900 # 15 minutes
```

The containers have been launched without any RAM limit. Multiple containers have been started until the system is fully loaded. The RAM usage of the host has been monitored using the Node-Exporter Grafana dashboard:

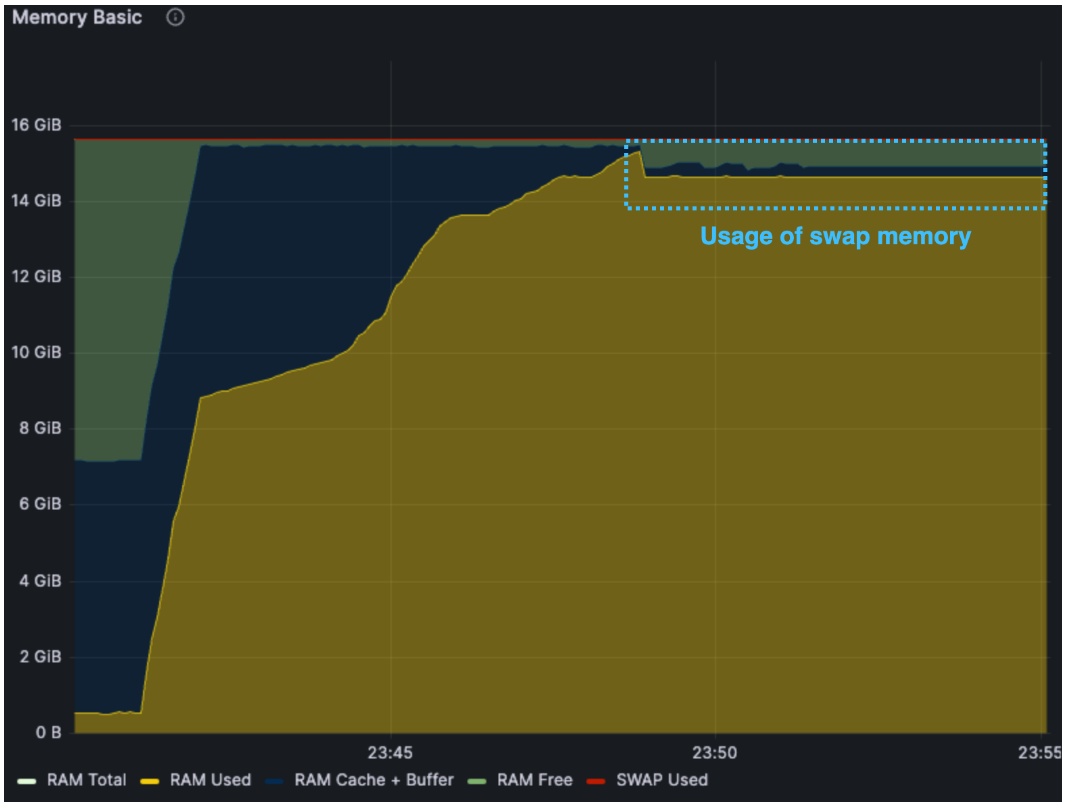
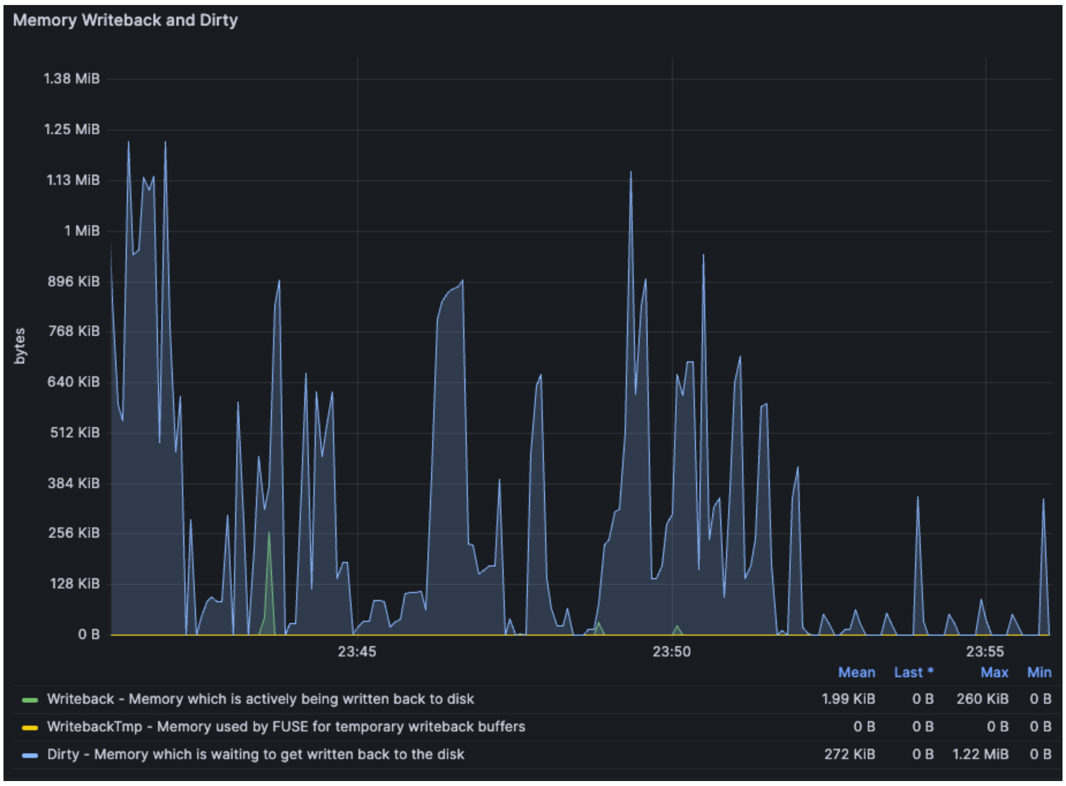


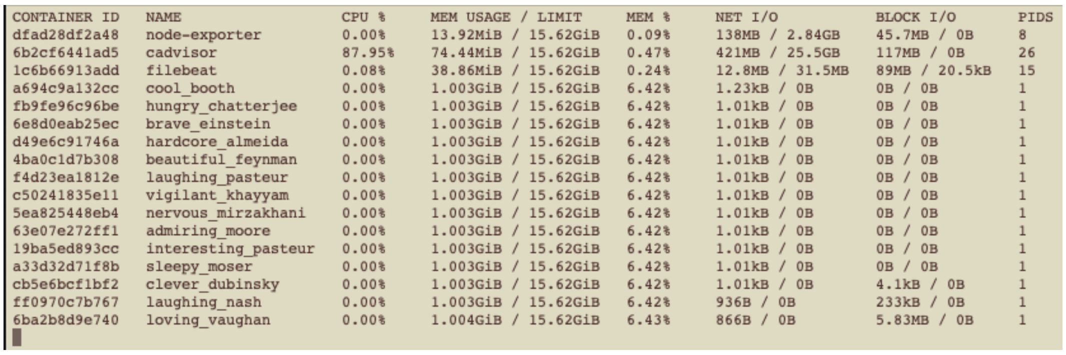
Figure 5.23: DOCKER-4-1: RAM usage of the host

When the RAM reaches its limit on the host, the system frees up space by swapping memory to the disk, as shown on the 2 following images:



	Mean	Last *	Max	Min
🟩 Writeback - Memory which is actively being written back to disk	1.99 KiB	0 B	260 KiB	0 B
🟨 WritebackTmp - Memory used by FUSE for temporary writeback buffers	0 B	0 B	0 B	0 B
🟦 Dirty - Memory which is waiting to get written back to the disk	272 KiB	0 B	1.22 MiB	0 B

Figure 5.24: DOCKER-4-1: The containers are writing and reading data to the disk



```
CONTAINER ID   NAME                  CPU %    MEM USAGE / LIMIT     MEM %    NET I/O             BLOCK I/O         PIDS
dfad28df2a48   node-exporter         0.00%    13.92MiB / 15.62GiB   0.09%    138MB / 2.84GB      45.7MB / 0B       8
6b2cf6441ad5   cadvisor              87.95%   74.44MiB / 15.62GiB   0.47%    421MB / 25.5GB      117MB / 0B        26
1c6b66913add   filebeat              0.08%    38.86MiB / 15.62GiB   0.24%    12.8MB / 31.5MB     89MB / 20.5kB     15
a694c9a132cc   cool_booth            0.00%    1.003GiB / 15.62GiB   6.42%    1.23kB / 0B         0B / 0B           1
fb9fe96c96be   hungry_chatterjee     0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
6e8d0eab25ec   brave_einstein        0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
d49e6c91746a   hardcore_almeida      0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
4ba0c1d7b308   beautiful_feynman     0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
f4d23ea1812e   laughing_pasteur      0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
c50241835e11   vigilant_khayyam      0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
5ea825448eb4   nervous_mirzakhani    0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
63e07e272ff1   admiring_moore        0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
19ba5ed893cc   interesting_pasteur   0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
a33d32d71f8b   sleepy_moser          0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         0B / 0B           1
cb5e6bcf1bf2   clever_dubinsky       0.00%    1.003GiB / 15.62GiB   6.42%    1.01kB / 0B         4.1kB / 0B        1
ff0970c7b767   laughing_nash         0.00%    1.003GiB / 15.62GiB   6.42%    936B / 0B           233kB / 0B        1
6ba2b8d9e740   loving_vaughan        0.00%    1.004GiB / 15.62GiB   6.43%    866B / 0B           5.83MB / 0B       1
```

Figure 5.25: DOCKER-4-1: The "docker stats" command shows the swapping process

Note that when there is enough RAM free on the host, the BLOCK I/O column of the "docker stats" command is empty for all containers. When the RAM of the host is full, some containers starts to exit. By using the "docker inspect [container]", it is possible to check the exit code:

```
"State": {
            "Status": "exited",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 137,
            "Error": ""
        }
```

It is possible to see that the "OOMKilled" boolean is set to false, and the "ExitCode" is 137, which means that the container has been killed using a SIGKILL signal. In fact, the container has been OOMKilled, but not by the Docker daemon because it did not exceed its limit, as it did not have one. This is why the boolean is set to false.

By looking at the */var/log/kern.log* file of the host, it is possible to see that it is the kernel's OOMKiller that killed the container:

```
kernel: [1817509.239456] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cp⌋
 ↪  uset=bcdcf7e91da12b5a25b268e1bb4e4c2732ed073102dce840438d4d85872b6c16,mems_⌋
 ↪  allowed=0,global_oom,task_memcg=/docker/9246b10d05d43a6186f73638bc4c30b1168⌋
 ↪  145dcb3c12b853b4a537dca46f3ee,task=stress-tool,pid=2032871,uid=0
kernel: [1817509.239475] Out of memory: Killed process 2032871 (stress-tool)
 ↪  total-vm:1054232kB, anon-rss:1048784kB, file-rss:0kB, shmem-rss:0kB, UID:0
 ↪  pgtables:2104kB oom_score_adj:0
```

Is is the default behaviour of the Linux's kernel: when there is no more RAM available on the host, the kernel starts to kill processes to take back memory space. As the Docker engine did nothing about the lack of RAM (as it has nothing to do), the kernel did the job.

It is also possible to see that the OOMKiller killed 4 containers on the NodeExporter dashboard:

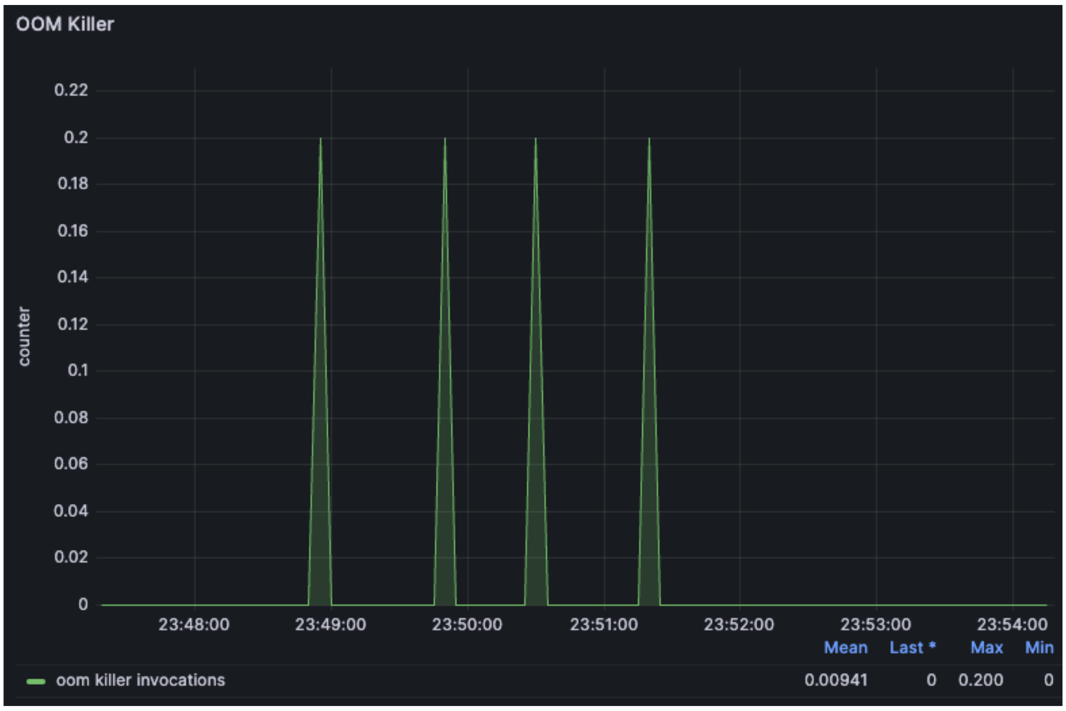
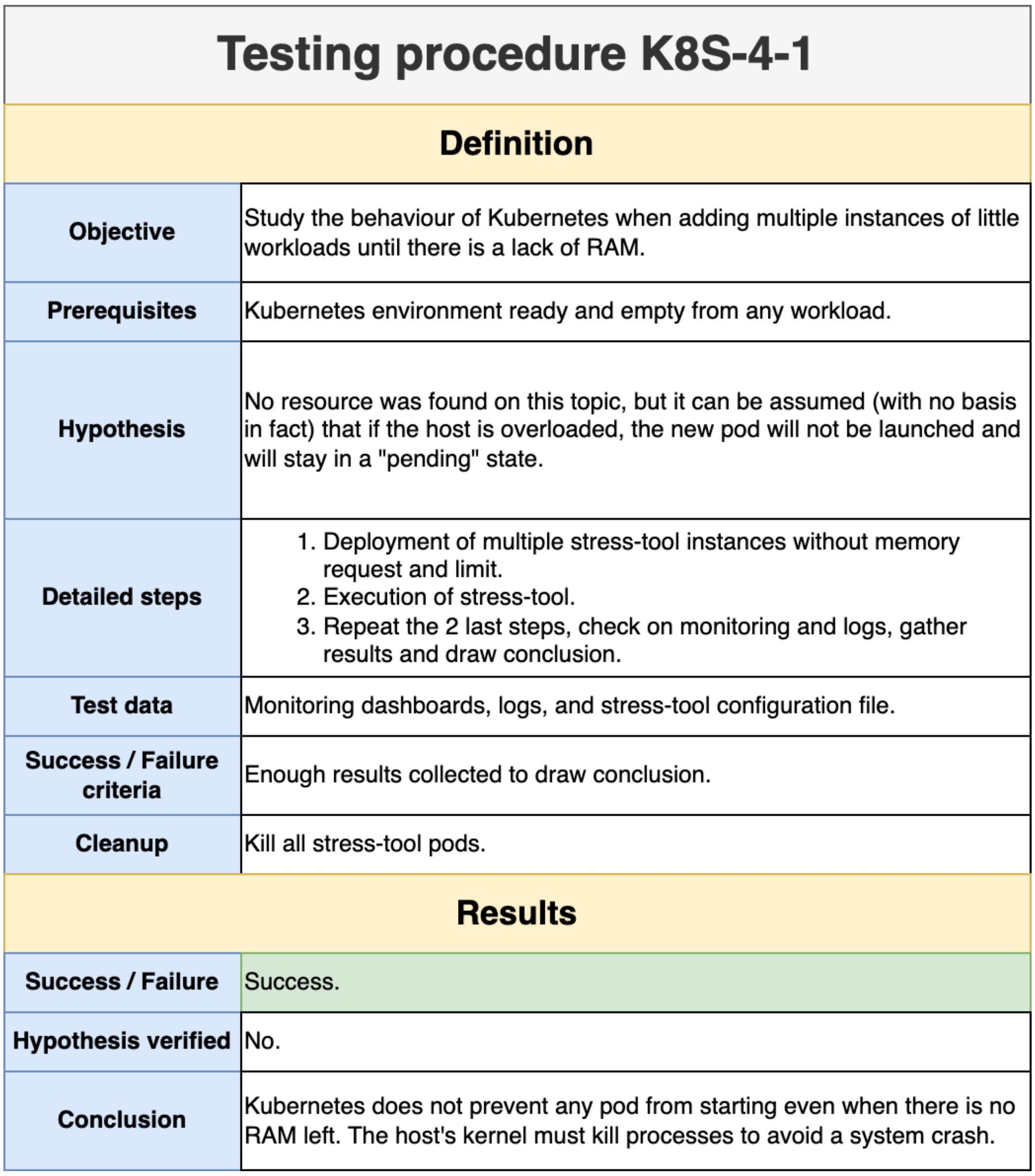


Figure 5.26: DOCKER-4-1: OOMKiller invocations

This experiment points out the need to specify RAM limit to workloads on a Docker environment to avoid the kernel's OOMKiller invocation that may kill randomly processes. The kernel should never kill Docker containers, it is a best practice to leave this job to the Docker engine by specifying RAM limit to workloads.

5.4.2 Testing procedure K8S-4-1

The testing procedure K8S-4-1 is described below with associated results:



Testing procedure K8S-4-1

Definition

Objective	Study the behaviour of Kubernetes when adding multiple instances of little workloads until there is a lack of RAM.
Prerequisites	Kubernetes environment ready and empty from any workload.
Hypothesis	No resource was found on this topic, but it can be assumed (with no basis in fact) that if the host is overloaded, the new pod will not be launched and will stay in a "pending" state.
Detailed steps	1. Deployment of multiple stress-tool instances without memory request and limit. 2. Execution of stress-tool. 3. Repeat the 2 last steps, check on monitoring and logs, gather results and draw conclusion.
Test data	Monitoring dashboards, logs, and stress-tool configuration file.
Success / Failure criteria	Enough results collected to draw conclusion.
Cleanup	Kill all stress-tool pods.

Results

Success / Failure	Success.
Hypothesis verified	No.
Conclusion	Kubernetes does not prevent any pod from starting even when there is no RAM left. The host's kernel must kill processes to avoid a system crash.

Figure 5.27: Testing procedure K8S-4-1

The experiment created for this testing procedure is the following:

```
wait 20
allocate 3221225472 id1 #1GB
wait 300
```

To deploy many pods with the same configuration and without any RAM limit, the following deployment has been applied to the cluster:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: stress-tool-deployment
spec:
  replicas: 17 # > total RAM capacity as each pod allocates 1GB of RAM
  selector:
    matchLabels:
      app: stress-tool
  template:
    metadata:
      labels:
        app: stress-tool
    spec:
      restartPolicy: Always # Unique allowed value
      containers:
      - name: stress-tool-container
        image: leichap/heia-stress-tool
        imagePullPolicy: Always
        volumeMounts:
        - name: config-volume
          mountPath: /usr/src/stress-tool/config.txt
      volumes:
      - name: config-volume
        hostPath:
          path: /home/ubuntu/config.txt
          type: File
```

Once the deployment has been applied, Kubernetes starts to schedule the pods and the pods start to allocate GB of RAM.

The RAM usage of the host has been monitored using the Node-Exporter Grafana dashboard:

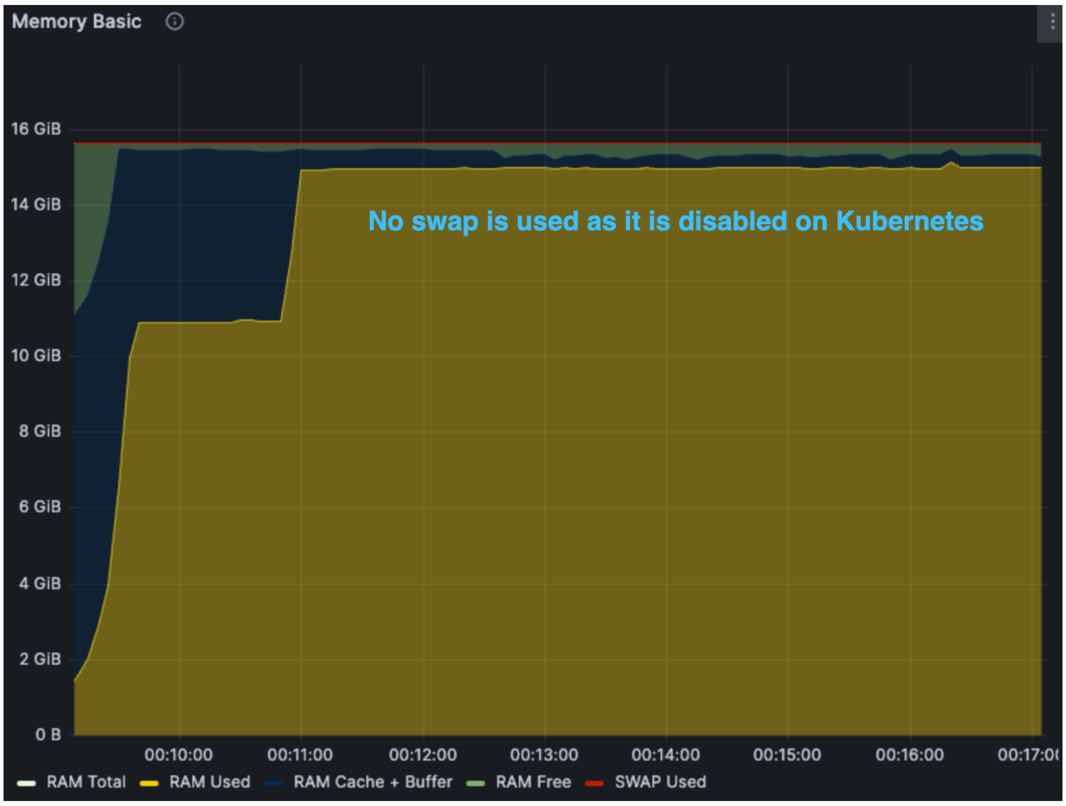


Figure 5.28: K8S-4-1: RAM usage of the host

Unlike Docker, Kubernetes does not use the host's swap memory for its pods. As a result, the amount of RAM used does not fall as it does in the Docker graph.

When the RAM of the host is full, some pods starts to exit. As a result, and because the only *restartPolicy* allowed in a deployment is *Always*, these pods are going to be restarted until Kubernetes detects that they are in a "killed, restarted" loop. These pods ends up being in the "CrashLoopBackOff" state. Each time the pod restarts, Kubernetes increases the wait time, referred to as a “backoff delay.” Throughout this process, Kubernetes shows the "CrashLoopBackOff" error:

```
NAME                           READY   STATUS             RESTARTS        AGE
stress-tool-deployment-2tbdr   1/1     Running            5 (3m39s ago)   8m9s
stress-tool-deployment-5mpjt   0/1     CrashLoopBackOff   5 (14s ago)     8m9s
stress-tool-deployment-bfgct   1/1     Running            0               8m9s
stress-tool-deployment-d69q8   1/1     Running            0               8m9s
stress-tool-deployment-jvx7l   1/1     Running            1 (7m20s ago)   8m9s
stress-tool-deployment-jxjns   1/1     Running            0               8m9s
stress-tool-deployment-kmqw9   0/1     CrashLoopBackOff   4 (57s ago)     8m9s
stress-tool-deployment-l6gzf   1/1     Running            0               8m9s
stress-tool-deployment-mjpfk   1/1     Running            5 (108s ago)    8m9s
stress-tool-deployment-nk9jl   1/1     Running            2 (5m50s ago)   8m9s
stress-tool-deployment-q27b9   1/1     Running            0               8m9s
stress-tool-deployment-q7hcf   1/1     Running            1 (6m54s ago)   8m9s
stress-tool-deployment-rtk8l   1/1     Running            0               8m9s
stress-tool-deployment-slcz6   1/1     Running            0               8m9s
stress-tool-deployment-vqvs7   1/1     Running            0               8m9s
stress-tool-deployment-x2crt   1/1     Running            0               8m9s
stress-tool-deployment-xxnxm   1/1     Running            2 (50s ago)     8m9s
```

When using the "kubectl describe pod" to see the detailed information on a pod which is on the "CrashLoopBackOff" state, the 137 error code appears, meaning that the process has been killed using the SIGKILL signal:

```
Containers:
  stress-tool-container:
    Container ID:   containerd://5e03a4424489226ec470545120fefdbae
    Image:          leichap/heia-stress-tool
    Image ID:       docker.io/leichap/heia-stress-tool@sha256:e85e424e84cb87e7
    Port:           <none>
    Host Port:      <none>
    State:          Waiting
      Reason:       CrashLoopBackOff
    Last State:     Terminated
      Reason:       Error
      Exit Code:    137
```

It is also possible to use the "kubectl events" command to obtain information on last events that occurred on the cluster. As a result, the following lines appears many times (result truncated):

```
REASON       OBJECT          MESSAGE
SystemOOM    Node/worker01   System OOM encountered, victim process:
 ↪  stress-tool, pid: 3067211
SystemOOM    Node/worker01   System OOM encountered, victim process:
 ↪  stress-tool, pid: 3067522
SystemOOM    Node/worker01   (combined from similar events): System OOM
 ↪  encountered, victim process: stress-tool, pid: 3552340
```

The reason says SystemOOM, meaning that Kubernetes did not killed the pod, but the Linux kernel did. Like the experiment on the Docker environment, Kubernetes does not manage machine overload because pods have no memory limit. As a last resort, the Linux kernel kills many processes to free up memory. As a result, Kubernetes relaunches the pods that have been killed, in a loop. It is possible to check the */var/log/kern.log* on the worker node, to see the Linux kernel OOMKiller actions:

```
kernel: [1829337.631193] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),cp⌋
↪  uset=/,mems_allowed=0,global_oom,task_memcg=/system.slice/containerd.servic⌋
↪  e/kubepods-besteffort-pode86da88c_e528_4df4_83bd_58613ea39908.slice:cri-con⌋
↪  tainerd:c94f17d3fe54fec2bb55d348ac24921b8576db56de4a638a54784725f0596c0b,ta⌋
↪  sk=stress-tool,pid=3554030,uid=0
kernel: [1829337.631213] Out of memory: Killed process 3554030 (stress-tool)
↪  total-vm:1054232kB, anon-rss:1048788kB, file-rss:0kB, shmem-rss:0kB, UID:0
↪  pgtables:2100kB oom_score_adj:1000
```

These OOMKiller invocations also appeared on the Node-Exporter Grafana dashboard, as the following image shows:

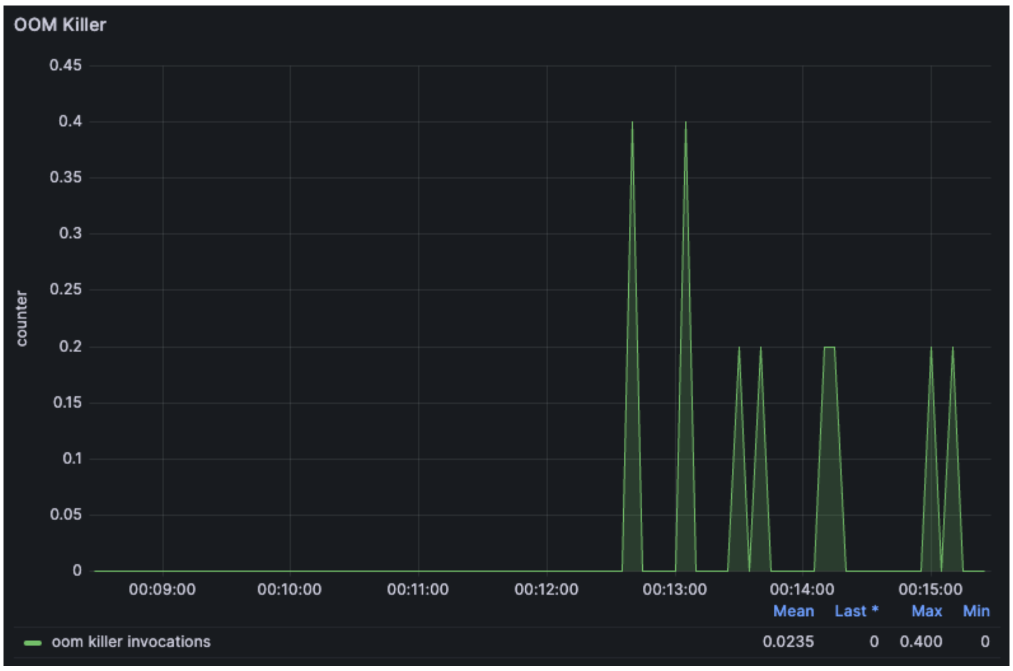
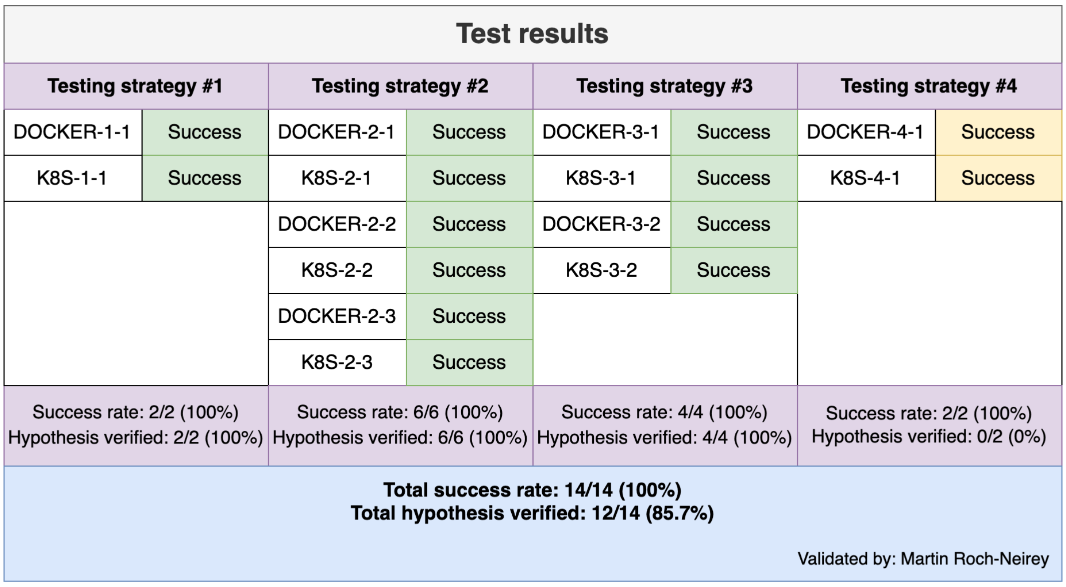


Figure 5.29: K8S-4-1: OOMKiller invocations

To avoid these problems, it is important to define pods requests and limits as seen in the analysis part of this report, so that the Kube-Scheduler can rely on these information to find a node that can execute the pod without any lack of resources. When such requests and limits are defined, Kubernetes can OOMKill pods as seen in the K8S-1-1 experiment, which is much better than calling the Linux kernel OOMKiller. It is considered as a best practice to always define a request field to a pod, and to set its limit to the same value as the request.

5.5 Test results

The table below shows the final test results:



Test results

Testing strategy #1		Testing strategy #2		Testing strategy #3		Testing strategy #4	
DOCKER-1-1	Success	DOCKER-2-1	Success	DOCKER-3-1	Success	DOCKER-4-1	Success
K8S-1-1	Success	K8S-2-1	Success	K8S-3-1	Success	K8S-4-1	Success
		DOCKER-2-2	Success	DOCKER-3-2	Success		
		K8S-2-2	Success	K8S-3-2	Success		
		DOCKER-2-3	Success				
		K8S-2-3	Success				
Success rate: 2/2 (100%) Hypothesis verified: 2/2 (100%)		Success rate: 6/6 (100%) Hypothesis verified: 6/6 (100%)		Success rate: 4/4 (100%) Hypothesis verified: 4/4 (100%)		Success rate: 2/2 (100%) Hypothesis verified: 0/2 (0%)	

Total success rate: 14/14 (100%)
Total hypothesis verified: 12/14 (85.7%)

Validated by: Martin Roch-Neirey

Figure 5.30: Test results

Each test received enough data to draw conclusion. The fourth testing strategy hypothesis were wrong, but they were not taken from any documentation.

5.6 Other experiments driven and ideas

This section reports on various experiments carried out outside the official test cases, and the partial conclusions drawn from them.

5.6.1 Maximal allocatable block size

When trying to allocate a block of 3GB, the instruction is not executed. The container (respectively the pod) is not killed. It is as if the program had skipped the allocation instruction. According to an IBM documentation, the maximal allocatable block size for one *malloc()* call is 2GB[33].

```
23:21:43 - Starting stress tool. - vRAM:5652 kB, PhysicalRAM:1512 kB
23:21:43 - Waiting 20 seconds... - vRAM:5652 kB, PhysicalRAM:1512 kB
[MISSING INSTRUCTION HERE]
23:22:03 - Waiting 300 seconds... - vRAM:5652 kB, PhysicalRAM:1512 kB
23:27:03 - End of program. - vRAM:5652 kB, PhysicalRAM:1512 kB
```

5.6.2 CPU usage during allocation instructions

When deploying multiple containers and pods during the DOCKER-4-1 and K8S-4-1 experiments, the CPU usage of the hosts was very high when allocating multiple GB of RAM. Some containers's RAM usage was growing very slowly to 1GB, as the CPU had to switch between multiple allocation contexts and keep updated it's mapping table that links virtual RAM addresses to physical ones. Here is the example for 15 containers, each requesting 1GB of RAM on a Docker environment:

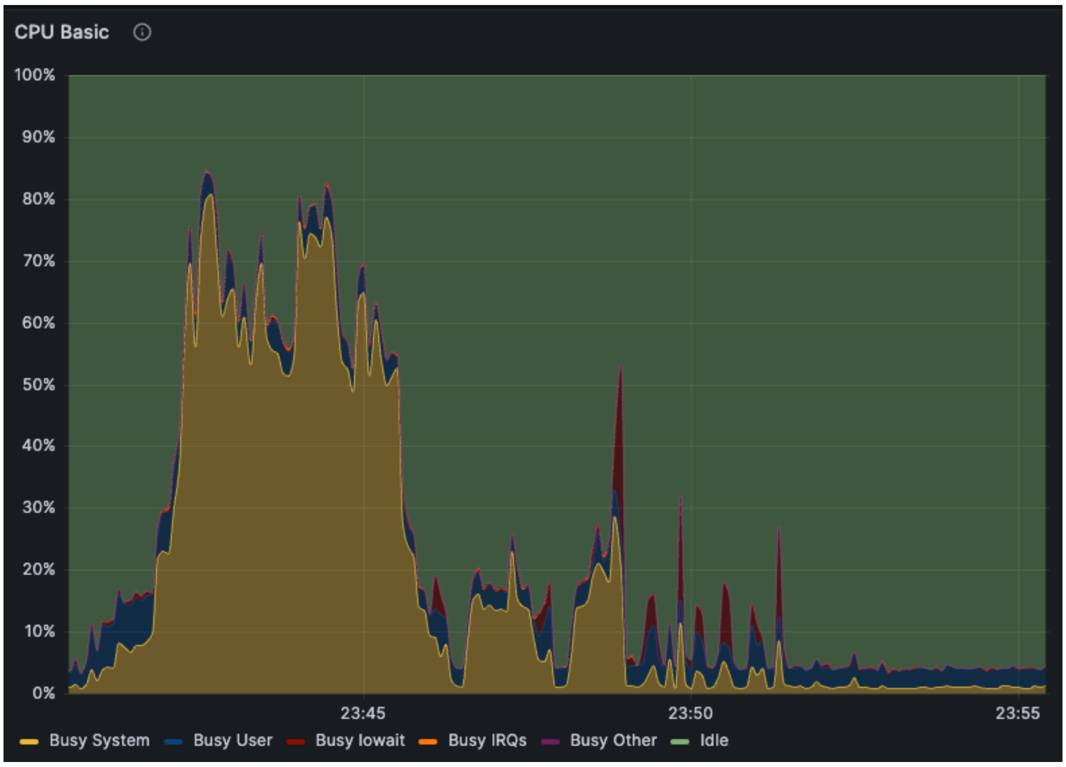


Figure 5.31: CPU usage when allocating multiple GB of RAM

5.6.3 Ideas of experiments

Here are presented some ideas of experiments. They have not been executed by lack of time or because they were out of the scope of the project.

- Replicating this study on managed Kubernetes cluster (Amazon EKS, GKE, AKS...) as they may manage workloads in different ways by implementing Kubernetes operators or other self-developed solutions.

- Perform more DOCKER-4 and K8S-4 experiments, with many configurations like applying requests and limits to pods, trying with multiple deployments, or using Pod Priority and Preemption objects[34].

- A secondary objective of this project was to carry out this study on different containerisation engines. For reasons of time, this was not possible, but certain hypotheses can be put forward.

  Kubernetes

  Since the version 1.20 of Kubernetes, the Docker engine has been deprecated. This has not been a problem, because Kubernetes can still run containers built using Docker's Open Container Initiative (OCI) image format. It is also still able to pull images from the Docker Hub, or other images registries that contains Docker images.

  The default containerization engine used by Kubernetes is containerd, as seen in the analysis part of this report. Docker and containerd are very similar and there is no apparent reason why the two engines should behave differently. In fact, Kubernetes supports other containerisation engines such as CRI-O or Mirantis, and makes no mention of any differences in its official documentation (with some exceptions, as seen in the analysis part). The basic assumption is that the behaviour of a Kubernetes cluster is engine-agnostic, and that the results obtained in this study would be similar with another containerisation engine supported by Kubernetes.

  Docker

  Such an hypothesis cannot be made for a native Docker environment, since in principle the entire platform will be different by using another engine. It is therefore not possible from the results obtained to make any assumptions about the behaviour of other containerisation engines, with the exception of containerd on which Docker is based.

6 | Conclusion

This chapter is the final one of this report, giving the conclusions of the project, the comparison with the initial objectives, the perspectives and a personal conclusion.

The main objectives of the project have been met, as has one of the 2 secondary objectives. For the second, a partial response based on hypotheses was provided. There is no doubt that many behaviours were not observed during this study and that additional test cases can be the subject of a second study to deepen the conclusions and discover new areas of research.

This study showed the importance for production environments to comply with the deployment best-practices, particularly when it comes to limiting the resources accessible to workloads, otherwise the system will randomly penalise processes or, worse still, crash. Administrators must rely on the fact that Docker and Kubernetes are well-designed for deploying workloads that are resources-limited.

It is also important to realise that containerised environments are complex and give rise to a degree of uncertainty when it comes to application metrics. The various overheads associated with complex technical stacks make it more difficult to analyse RAM consumption than if the application were running in a relatively simple native environment. This complexity leads to variations in metrics observed by different tools, as seen in the DOCKER-2-3 and K8S-2-3 experiments.

On a personal note, working on this semester project has been a rewarding experience. It was a pleasure to carry out this study on a technological subject that every IT engineer is familiar with, but which remains fundamental. RAM is one of the most important elements in a computer, and this study shows once again how important it is to manage it correctly. As I said for my semester 5 project report, I sincerely hope that this (long) report conveys the enthusiasm and interest I had to work on this study.

HES-SO legal information

Declaration of honour I, the undersigned, Martin Roch-Neirey, declare on my honour that the work submitted is my own work. I certify that I have not resorted to plagiarism or any other form of fraud. All sources of information used and author citations have been clearly stated.

Déclaration d'honneur Je, soussigné, Martin Roch-Neirey, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toute autre forme de fraude. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Martin Roch-Neirey

A | Use case / Components diagram

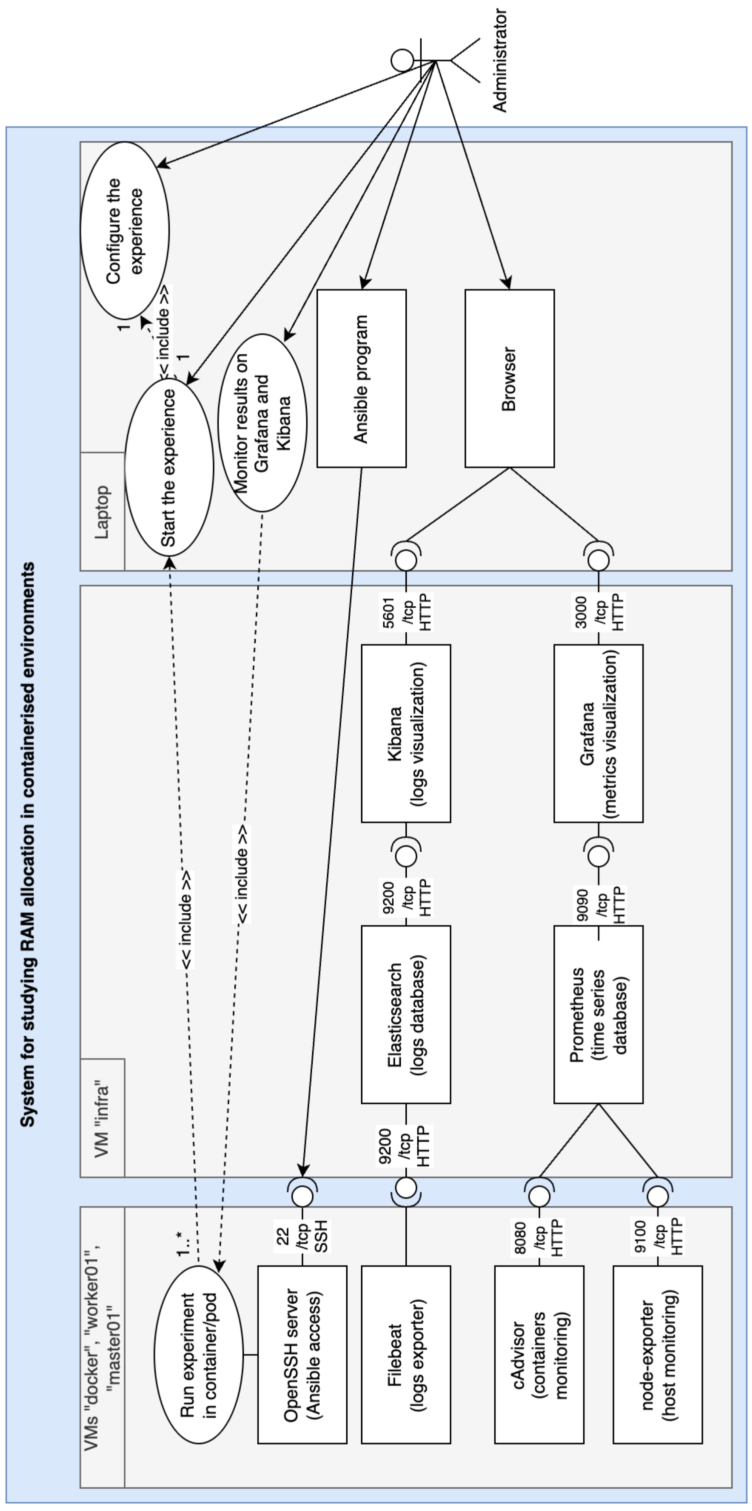

Administrator
System for studying RAM allocation in containerised environments
Laptop
Configure the experience
Start the experience
<< include >>
1
1
Monitor results on Grafana and Kibana
Ansible program
Browser
VM "infra"
Kibana (logs visualization)
Grafana (metrics visualization)
5601 /tcp HTTP
3000 /tcp HTTP
9200 /tcp HTTP
9090 /tcp HTTP
Elasticsearch (logs database)
Prometheus (time series database)
9200 /tcp HTTP
<< include >>
<< include >>
VMs "docker", "worker01", "master01"
1..*
Run experiment in container/pod
OpenSSH server (Ansible access)
22 /tcp SSH
Filebeat (logs exporter)
cAdvisor (containers monitoring)
8080 /tcp HTTP
node-exporter (host monitoring)
9100 /tcp HTTP

Figure A.1: Use case / Components diagram

B | CI/CD pipeline

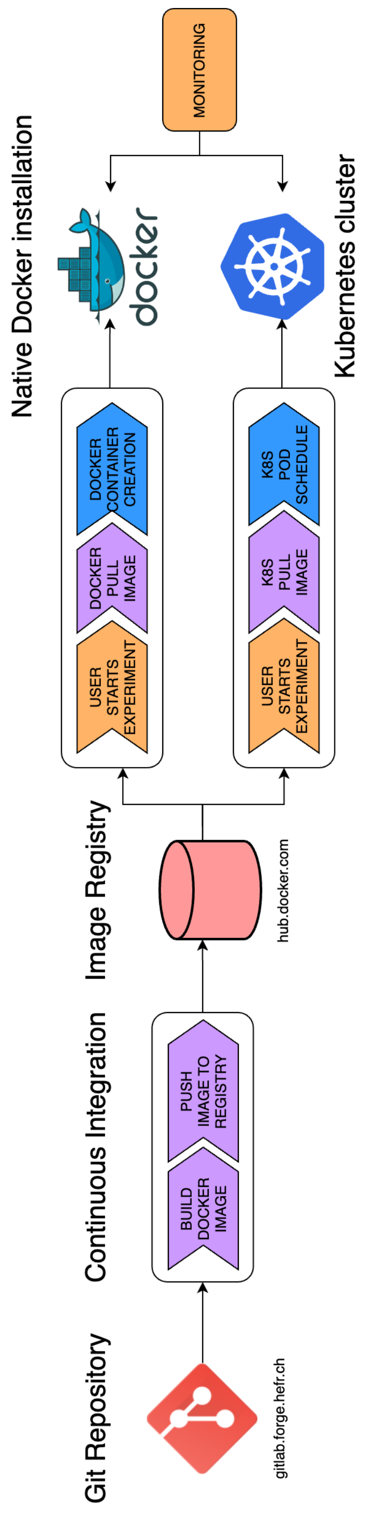


Figure B.1: CI/CD pipeline

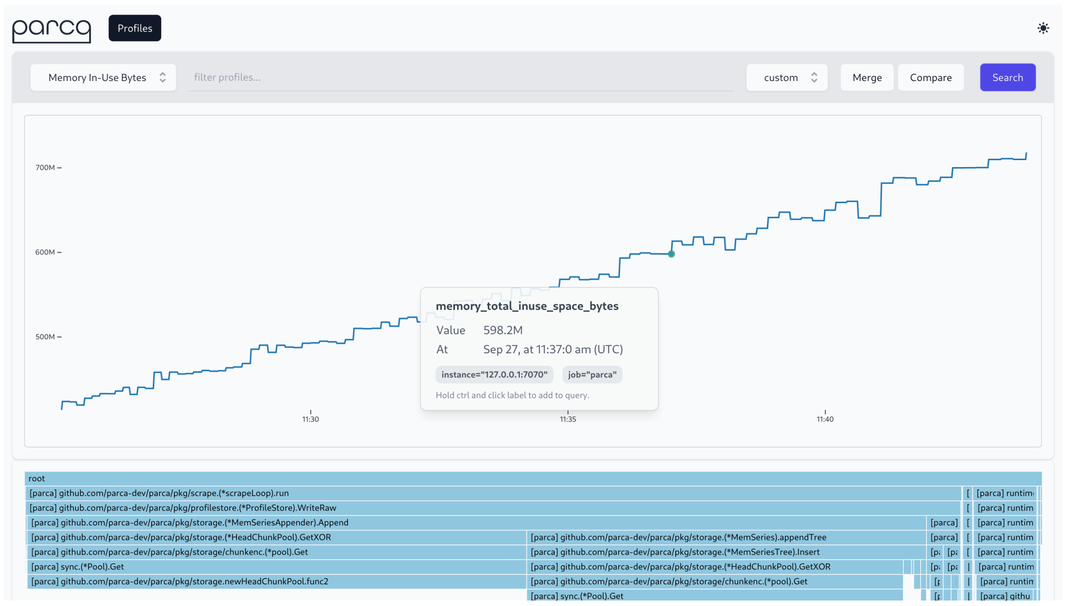
C | Analysis of the parca.dev solution

The parca.dev[35] solution is an open-source project focused on continuous profiling across IT infrastructure. It provides insights about resource usage (CPU, RAM and others), performance bottlenecks, and debugging issues. As their documentation says, parca's continuous data collection is performed with minimal overhead, making it suitable for use in production environments without affecting system performance.

Parca's infrastructure is made of agents and a server. The agents are executed on the monitored hosts while the server is used to store, filter and export data.

The Parca Agent uses eBPF[36] (extended Berkeley Packet Filter), a technology for low-level data collection on Linux hosts. This allows Parca to profile system performance with minimal impact on the system. The use of eBPF ensures that the profiling process is efficient and capable of running continuously without significant performance overhead.

The following image shows the RAM usage of an application on a dashboard with Parca's data.



memory_total_inuse_space_bytes
Value 598.2M
At Sep 27, at 11:37:0 am (UTC)
instance="127.0.0.1:7070" job="parca"
Hold ctrl and click label to add to query.

```
root
[parca] github.com/parca-dev/parca/pkg/scrape.(*scrapeLoop).run
[parca] github.com/parca-dev/parca/pkg/profilestore.(*ProfileStore).WriteRaw
[parca] github.com/parca-dev/parca/pkg/storage.(*MemSeriesAppender).Append
[parca] github.com/parca-dev/parca/pkg/storage.(*HeadChunkPool).GetXOR      [parca] github.com/parca-dev/parca/pkg/storage.(*MemSeries).appendTree
[parca] github.com/parca-dev/parca/pkg/storage/chunkenc.(*pool).Get         [parca] github.com/parca-dev/parca/pkg/storage.(*MemSeriesTree).Insert
[parca] sync.(*Pool).Get                                                    [parca] github.com/parca-dev/parca/pkg/storage.(*HeadChunkPool).GetXOR
[parca] github.com/parca-dev/parca/pkg/storage.newHeadChunkPool.func2       [parca] github.com/parca-dev/parca/pkg/storage/chunkenc.(*pool).Get
                                                                            [parca] sync.(*Pool).Get
```

Figure C.1: Example of a Parca dashboard (Parca)

An interesting blog post[37] from the creators of Parca shows a demonstration of the tool with many information.

References

[1] Arith-Matic authors. *ROM and RAM: An Introduction to Computer Memory*. 2024. URL: https://arith-matic.com/notebook/rom-ram-computer-memory (visited on 05/07/2024).

[2] Wikipedia. *Wikipedia: Thrashing (Computer Science)*. 2024. URL: https://en.wikipedia.org/wiki/Thrashing_(computer_science) (visited on 05/07/2024).

[3] Wikipedia. *Wikipedia: Virtual memory*. 2024. URL: https://en.wikipedia.org/wiki/Virtual_memory (visited on 05/07/2024).

[4] Michael Kerrisk. *cgroups(7) — Linux manual page*. 2024. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 05/16/2024).

[5] Docker Inc. *Docker - Understand the risks of running out of memory*. 2024. URL: https://docs.docker.com/config/containers/resource_constraints/#understand-the-risks-of-running-out-of-memory (visited on 05/07/2024).

[6] Inc. Linux Kernel Organization. *Kernel Chapter 13 - Out Of Memory Management*. 2024. URL: https://www.kernel.org/doc/gorman/html/understand/understand016.html (visited on 05/07/2024).

[7] Docker Inc. *Docker - Runtime options with Memory, CPUs, and GPUs*. 2024. URL: https://docs.docker.com/config/containers/resource_constraints/#limit-a-containers-access-to-memory (visited on 05/07/2024).

[8] containerd Authors. *containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability*. 2024. URL: https://containerd.io/ (visited on 05/07/2024).

[9] The Kubernetes Authors. *Downward API*. 2024. URL: https://kubernetes.io/docs/concepts/workloads/pods/downward-api/ (visited on 05/07/2024).

[10] The Kubernetes Authors. *Kubernetes documentation - Assign Memory Resources to Containers and Pods*. 2024. URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/#if-you-do-not-specify-a-memory-limit (visited on 04/07/2024).

[11] Prometheus Authors. *Prometheus - From metrics to insight*. 2024. URL: https://prometheus.io/docs/introduction/overview/ (visited on 05/10/2024).

[12] Google. *cAdvisor - GitHub repository*. 2024. URL: https://github.com/google/cadvisor (visited on 05/10/2024).

[13] Prometheus community. *Node-Exporter - GitHub repository*. 2024. URL: https://github.com/prometheus/node_exporter (visited on 05/10/2024).

[14] Grafana. *Grafana - The open observability platform*. 2024. URL: https://grafana.com/docs/grafana/latest/ (visited on 05/10/2024).

[15] Elastic. *Elasticsearch - The heart of the free and open Elastic Stack*. 2024. URL: https://www.elastic.co/elasticsearch (visited on 05/11/2024).

[16] Elastic. *Filebeat - Lightweight shipper for logs*. 2024. URL: https://www.elastic.co/beats/filebeat (visited on 05/11/2024).

[17] Elastic. *Kibana - Discover, iterate, and resolve with ES|QL on Kibana*. 2024. URL: https://www.elastic.co/kibana (visited on 05/11/2024).

[18] Elastic. *Elastic - ELK Stack*. 2024. URL: https://www.elastic.co/elastic-stack/ (visited on 05/11/2024).

[19] OpenStack community. *OpenStack - The Most Widely Deployed Open Source Cloud Software in the World*. 2024. URL: https://docs.openstack.org/2024.1/ (visited on 05/11/2024).

[20] Hashicorp. *Terraform - Automate infrastructure on any cloud with Terraform*. 2024. URL: https://developer.hashicorp.com/terraform (visited on 04/23/2024).

[21] RedHat. *Ansible - Ansible offers open-source automation that is simple, flexible, and powerful.* 2024. URL: https://docs.ansible.com/ (visited on 04/23/2024).

[22] Jeison Sánchez (Growth Acceleration Partners). *A Brief History of Microservices - Part I*. 2024. URL: https://medium.com/@wearegap/a-brief-history-of-microservices-part-i-958c41a1555e (visited on 04/19/2024).

[23] LLC. DigitalOcean. *Comparing AWS, Azure, GCP*. 2024. URL: https://www.digitalocean.com/resources/article/comparing-aws-azure-gcp (visited on 04/24/2024).

[24] Janani Ravi. *Cloud Container Services Compared – AWS vs Azure vs GCP*. 2023. URL: https://www.pluralsight.com/resources/blog/cloud/cloud-container-services-compared-aws-vs-azure-vs-gcp (visited on 04/24/2024).

[25] Cody Slingerland. *AWS Vs. Azure Vs. Google Cloud: Which One Should You Use?* 2023. URL: https://www.cloudzero.com/blog/aws-vs-azure-vs-google-cloud/ (visited on 04/24/2024).

[26] Ubuntu MOTU Developers. *Ubuntu Package: stress (1.0.4-6) [universe]*. 2024. URL: https://packages.ubuntu.com/focal/stress (visited on 05/07/2024).

[27] progrium. *Docker-Stress GitHub repository*. 2024. URL: https://github.com/progrium/docker-stress (visited on 05/07/2024).

[28] Martin Roch-Neirey. *PS6-MRN-Tech GitLab repository*. 2024. URL: https://gitlab.forge.hefr.ch/ps6-martin-roch-neirey/ps6-mrn-tech (visited on 01/31/2024).

[29] "Tim" and "Stephen Kitt" StackExchange users. *Which process is '/proc/self/' for?* 2024. URL: https://unix.stackexchange.com/questions/333225/which-process-is-proc-self-for (visited on 05/07/2024).

[30] Michael Kerrisk. *proc(5) — Linux manual page*. 2024. URL: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 05/07/2024).

[31] The Kubernetes Authors. *Metrics Server*. 2024. URL: `https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/#metrics-server` (visited on 05/07/2024).

[32] The Kubernetes Authors. *Tools for Monitoring Resources*. 2024. URL: `https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/` (visited on 05/07/2024).

[33] IBM Corporation. *malloc() — Reserve Storage Block*. 2024. URL: `https://www.ibm.com/docs/en/i/7.3?topic=functions-malloc-reserve-storage-block` (visited on 05/07/2024).

[34] The Kubernetes Authors. *Pod Priority and Preemption*. 2024. URL: `https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/` (visited on 05/07/2024).

[35] Parca contributors. *Parca - Open Source infrastructure-wide continuous profiling*. 2024. URL: `https://www.parca.dev/` (visited on 05/07/2024).

[36] eBPF.io authors. *eBPF - Dynamically program the kernel for efficient networking, observability, tracing, and security*. 2024. URL: `https://ebpf.io/` (visited on 05/02/2024).

[37] Sumera Priyadarsini. *Introduction to Parca - Part 1*. 2024. URL: `https://www.polarsignals.com/blog/posts/2022/07/12/introducing-parca-sequel` (visited on 05/07/2024).

Glossary

Docker Platform that deploys and manages applications within lightweight, portable containers. 2

Infrastructure as Code Practice where the management and provisioning of computing infrastructure are automated and managed using code and software development techniques, rather than through manual processes. 13

Kubernetes Orchestrator used to deploy scalable workloads as containers. 2

OpenStack Open-source cloud computing platform that provides a set of software tools and components to build and manage public and private cloud infrastructure. 13

Random Access Memory Computer memory that provides fast, temporary storage for data and instructions that a computer's processor needs to access quickly. 2

SIGKILL Signal used in Unix-like operating systems to forcefully terminate a process immediately without allowing it to perform any cleanup operations. 8

SIGTERM signal used in Unix-like operating systems to gracefully terminate a process, allowing it to perform cleanup operations before exiting. 8

TSDB Time Series Database. 12