# Version

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Change** |
| 03-03-2025 | 0.1 | Created first draft. Added chapters ‘theoretical background’ & ‘Research questions’ |
| 10-03-2025 | 0.2 | Added chapter ‘Assignment’ |
| 13-03-2025 | 0.3 | Added chapter ‘Theoretical framework’ |
| 26-03-2025 | 0.4 | Added CPU tests ‘Methods and Technology E1’ |
| 27-03-2025 | 0.5 | Added chapter ‘Hypothesis, CPU’, updated research questions, added OS, scripting software and container engine to chapter ‘Justification of choices’. |
| 04-04-2025 | 0.6 | Added chapter ‘Methods E2’ |
| 10-04-2025 | 0.7 | Added chapter results ‘Results CPU’ |
|  |  |  |

# Executive summary

# Preface

Contents

[Version 2](#_Toc195192636)

[Executive summary 3](#_Toc195192637)

[Preface 4](#_Toc195192638)

[Assignment 7](#_Toc195192639)

[Context 7](#_Toc195192640)

[Goals 7](#_Toc195192641)

[Research questions 9](#_Toc195192642)

[Theoretical background 10](#_Toc195192643)

[Containers 10](#_Toc195192644)

[Real-time embedded systems 11](#_Toc195192645)

[Real-time embedded containers 12](#_Toc195192646)

[Theoretical framework 13](#_Toc195192647)

[Kernel 13](#_Toc195192648)

[Container process 14](#_Toc195192649)

[Process isolation: namespaces 14](#_Toc195192650)

[Resource management: cgroups 15](#_Toc195192651)

[Filesystems and storage: union file system 16](#_Toc195192652)

[Hypothesis 17](#_Toc195192653)

[General hypothesis 17](#_Toc195192654)

[CPU limitations 17](#_Toc195192655)

[Memory limitations 17](#_Toc195192656)

[I/O limitations 17](#_Toc195192657)

[Methods and techniques 18](#_Toc195192658)

[Base Raspberry Pi configuration 18](#_Toc195192659)

[Methods CPU limitations 18](#_Toc195192660)

[CE1 Container overhead single container 18](#_Toc195192661)

[CE2 Increasing running containers 21](#_Toc195192662)

[Results 23](#_Toc195192663)

[CPU experiment results 23](#_Toc195192664)

[CE1 Container overhead single container 23](#_Toc195192665)

[CE2 Increasing running containers 25](#_Toc195192666)

[Discussion 26](#_Toc195192667)

[Conclusion 27](#_Toc195192668)

[Recommendations 28](#_Toc195192669)

[Justification of choices 29](#_Toc195192670)

[Experiment hardware 29](#_Toc195192671)

[Operating system 30](#_Toc195192672)

[Scripting software 30](#_Toc195192673)

[Container engine 31](#_Toc195192674)

[Description of realized professional products 32](#_Toc195192675)

[Evaluation of quality criteria 33](#_Toc195192676)

[Process evaluation 34](#_Toc195192677)

[Sources 35](#_Toc195192678)

[Appendix 36](#_Toc195192679)

# Assignment

## Context

Even though container software like Docker was initially created for server environments (Johnston, 2024), they are significantly more used in real-time embedded systems. The incorporation of containers in these systems presents numerous functional advantages. For instance, firmware is no longer confined to a single device and software becomes easily reusable (Bodet, 2023). However, the effect of containers on real-time performance remains relatively unknown (see chapter theoretical background, containers).

Containers consist of various abstraction layers. These different layers are responsible for the isolation of processes and resources, limitation and monitoring of resources and memory isolation for the container. The abstraction layers could potentially cause a lot of overhead. Therefore, it is important to conduct research to gain a better understanding of the possible unknown disadvantages and limitations of using containers on real-time embedded systems.

## Goals

The aim of this research project is to map the boundaries of containers on real-time embedded systems. To achieve the goal both theoretical and experimental research needs to be done.

This research is mostly conducted from a curiosity perspective on behalf of the product owner. The results will not have direct impact on the projects that Capgemini does for its clients. The research results will mostly be used for reference within the organization for colleagues to gain knowledge about the subject in a relatively short time span.

To successfully complete the project the products described in the table below need to be realized. The described quality criteria are based on the wishes of the product owner.

Table 1 Products and quality

|  |  |  |
| --- | --- | --- |
| **Product** | **Description** | **Quality** |
| Automated experiment | Code must be written to automatically carry out several tests to map the boundaries. | ***Repeatability***: The experiment needs to be repeatable within the timespan of a weekend by a colleague.  ***Reproducibility***: The experiment needs to be written in a way it can be repeated on different hardware. |
| Research document | A document must be written in which the choices and research results are described. | ***Readability***: The document must be written in clear and understandable English. The document needs to provide insight in the experiments, the results and the conclusions in the form of a ‘blog post’ format.  ***Substantiated***: the conclusions in the document must be clearly substantiated using graphs and diagrams. The sources for the conclusions need to be included. |

In addition to the products that needed to be provided for the company, the student worked on this thesis document to prove to demonstrate the necessary skills for graduation.

# Research questions

To obtain meaningful research results, the following research questions are being used.

***Main question:***

*What are the practical limits when using containers on real-time embedded systems, focused on the potential restrictions in the areas CPU-bound, memory-bound and I/O-bound, and how can these restrictions be explained?*

***Sub questions:***

The following sub questions will be used to answer the main research question.

* *Which Linux kernel features are used to run containers, and what theoretical risks, based on the three restricting factors can these features cause on real-time embedded systems?*

This question focusses on the theoretical background on how containers operate and which building blocks they are made of. The theoretical knowledge will be used to form the hypotheses described in the ‘Hypothesis’ chapter.

* *How can an experiment be designed to map the practical limits of the CPU-bound, memory-bound and I/O-bound factors?*

To test the limitations of containers experiments, need to be designed. The methods used to map the limitations are described in the “Methods and Techniques” chapter.

* *How can the practical limits from the experiment be explained?*

The experiment results need to be explained, which will be done in the ‘Discussion’ chapter.

# Theoretical background

Following the research questions, it is important to explain general concepts and techniques before starting the research. Based on the main question several key terms can be derived. These terms are described in this chapter, focusing on their definition and characteristics.

***Main question:***

*What are the practical limits when using* ***containers*** *on* ***real-time embedded systems****, focused on the potential restrictions in the areas* ***CPU-bound, memory-bound and I/O-bound,*** *and how can these restrictions be explained?*

## Containers

In order to explain containers, virtualization needs to be explained first. Virtualization is a concept originating in the server world. Using virtualization multiple applications can run simultaneously on the same hardware without being able to access each other’s content (PowerCert Animated Videos, 2022).

Over the years two mainstream solutions have been developed to implement virtualization: Virtual Machines (VM’s) and ***containers***. This research project is only focused on containers.

A screenshot of a computer screen

AI-generated content may be incorrect.Containers are virtual environments in which applications run. The environments operate on top of the hardware and system OS.

A container contains the application and additional dependencies necessary to run the application.

Between the OS and the container applications the container engine operates. The container engine makes sure the applications are able to access the right system resources and kernel functions to run the container application (PowerCert Animated Videos, 2022).

***Pro’s:***

* ***Resource usage***: Compared to VM’s containers use very few resources to operate, because the system OS is being used. This result is that there’s no need to install an entire OS. Only the components used to run the container are taking up resources. This applies to the systems memory, CPU-power and RAM (AltexSoft, 2022).
* ***Compatibility***: Containers are built to be platform independent. Software runs the same regardless of which system the application is executed on. Containers run on all systems that support container platforms (TechSquidTV, 2018).
* ***Scalability***: Container applications are easily scalable. A container image can easily be created based on an application container. Using the image new containers van be deployed quickly and easily (AltexSoft, 2022).
* ***Isolation***: When problem occurs within a container or if the container crashes, other container applications on the same system are not affected (Barron, 2023).
* ***Single task execution***: Each container is focused on one task and is responsible for performing that task effectively.

***Con’s:***

* ***Kernel vulnerabilities***: A potential drawback of using containers is that when multiple containers run on the same system, they share the host kernel. If the kernel is compromised, all the containers are affected (Balaban, 2024).
* ***Overview of active containers***: When multiple containers run on the same system, it becomes difficult to track the different active containers and the tasks they are preforming (DuploCloud, 2023).
* ***Data storage***: Containers exist in their own environment. Storing data from a container is possible but must be done intentionally. When the container ceases to exist, the data withing the container is lost (Barron, 2023).

## Real-time embedded systems

Even though virtualization was initially created for server environments, containers are nowadays also being used for software purposes. This research project will be focused on the usage of containers in real-time embedded systems.

An embedded system is a combination of hardware and software designed to perform a specific task (GeeksforGeeks, 2022). When the task is time-critical, it is referred to as a ***real-time embedded system*** (RTOS LOGICS, 2020). These systems are designed to meet strict time constraints (GeeksforGeeks, 2022).

When it comes to real-time systems, a distinction can be made between hard real-time and soft real-time. In a ***hard-real time*** application, missing a deadline has serious consequences and leads to a complete system failure, often leading to injury. This is the case with, for example, medical equipment and in the aviation industry. A ***soft real-time*** system is less critical. When a deadline is missed, the entire system does not fail, but the operational quality is reduced. Examples are online games or streaming services (GeeksforGeeks, 2024).

The time constraint can be determined in various ways. For example, by looking at the maximum allowed duration of executing a specific code segment. For more complex systems, solutions like RTOS (real-time operating system) are being used to schedule and execute critical tasks (Fidus, 2024).

## Real-time embedded containers

By using containers on real-time embedded systems, the benefits of application portability, resource isolation, scalability and single task execution can be realized on the systems (Windriver, z.d.). It is currently unclear what the limits are when containers are being used on real-time embedded systems. Embedded systems are known for having limited resources. Resulting it is important to determine how the use of containers influences the resources.

The following three potential restricting factors will be examined: CPU-bound, memory-bound and I/O bound.

* ***CPU-bound***: The CPU speed is the limiting factor when running a process. This limit could be reached by performing many calculations in quick succession (Mortensen, 2009).
* ***Memory-bound***: A process is limited by the speed at which the memory (RAM) can be read from or written to (Mortensen, 2009).
* ***I/O-bound***: A process is limited by the speed of an I/O subsystem. An example could be reading or writing to external data storage. When the external medium is not able to keep up with the speed it is referred to as being I/O bound (Mortensen, 2009).

# Theoretical framework

To formulate a good hypothesis, background knowledge of the topic is necessary. This chapter focusses on existing sources to gather information and to determine what is already know about the subject. This will be done according to the first research question.

* *Which* ***Linux kernel features*** *are used to run containers, and which* ***theoretical risks****, based on the three restricting factors can these features cause on real-time embedded systems?*

The first part of the question is focused on the theoretical background necessary to determine the potential risks. In this subchapter the Linux kernel and the features being used to run containers are discussed. The next chapter will go into the possible risks and hypothesis accordingly.

### Kernel

The kernel is the bridge between the hardware and the software applications on a system (GeeksforGeeks, 2025). It forms an abstraction layer for the applications to use system resources without directly interacting with the hardware (ARMO, 2024). The kernel manages system resources, such as the CPU, memory and peripheral devices. Additionally, the kernel is responsible for executing processes, facilitating file access and coordinating I/O device connections. It forms the backbone of the operating system, providing the necessary resources for the user interaction with hardware components (GeeksforGeeks, 2025).

A diagram of a computer process

AI-generated content may be incorrect.

*(GeeksforGeeks, 2025)*

There are multiple different kernel types, all working slightly differently. This research project is specifically focused on Linux systems. Linux operates on a monolithic kernel, meaning that all operating system services are executed within the kernel space instead of partially being run in the user space. This makes the monolithic kernel highly efficient. Monolithic kernels usually offer simplicity and lower latencies as most operations are handled directly by the kernel. However, they tend to be less stable and secure, as a single glitch or security breach can compromise the entire system (GeeksforGeeks, 2025).

### Container process

One of the responsibilities of the kernel is executing and managing processes. In Linux a process is any active running instance of a program (Carrigan, 2023). The instance consists of all the services and resources that might be utilized by the process that is being execution (GeeksforGeeks, 2022b).

A container is essentially a Linux process that uses additional isolation mechanisms. Unlike a standard process, the container process does not have the ability to interact with the entire system, but only with what it's specifically authorized to access and manage. The container runs on the same kernel, but the container process is contained and unable to see or effect other parts of the system.

Processes can only be created using the clone() or fork() kernel syscalls, creating a new process out of an existing one. Using the exec() function a new program can be executed replacing the program originally executed by the parent.

Like regular processes a container can only be created out of an existing process. When starting a container the new process is created by using the clone() syscall. The clone function is a more advanced version of the fork() syscall that enables the new process to start with the properties of a container.

Different kernel functionalities are used to run containers and their applications. Looking at these features the building blocks namespaces, cgroups and the union file system are most important (Gahlot, 2021). They can be categorized as follows:

* ***Process isolation***: the process is isolated from the rest of the system. This is done by using namespaces.
* ***Resource management***: the resources of the system are controlled using cgroups.
* ***Filesystems and storage***: manages the filesystem (OverlayFS).

### Process isolation: namespaces

One of the main benefits of containers is their isolation between different containers and the host. ***Namespaces*** create a controlled environment for processes, giving them the illusion of isolated system resources (Kuninoto, 2024). Each namespace provides a different type of isolation for the container.

The separation is done by using the namespace kernel feature. When a container is created, the clone() function mounts the new container namespaces, one for each namespace, to the process.

Containers make use of the following namespaces (Gaurav Gahlot, 2021):

* ***PID namespace (process ID)***: The PID namespace ensures that each container has its own unique set of PID’s. The fundamental principle is that the containers visibility is limited to only the PID’s within the namespace. This means the container is not able to perceive or interact with processes outside of its PID namespace (Tiwari, 2024).
* ***Mount namespace (mnt)***: This namespace isolates the view of the filesystem for each container. (Tiwari, 2024). Every container possesses its unique view of the filesystem, independent from the host and other containers.
* ***IPC namespace (inter process communication)***: The IPC namespace facilitates the communication between processes through various methods such as message queues, semaphores and shared memory (Boutnaru, 2023). By default, the namespace is configured to only allow process communication within the container (Datadog, z.d.).
* ***UTS namespace***: Allows the container to have its own host name (Arges, 2020).
* ***User namespace***: Provides user ID (UID) and group ID (GID) separation, allowing containers to have different privilege levels compared to the host system.
* ***Network namespace (net)***: Provides network isolation. This namespace makes sure different containers have different IP addresses while sharing the same hardware (Arges, 2020).

### Resource management: cgroups

When a container is created resources need to be allocated to the process. A ***control group (cgroup)*** is a Linux kernel feature that limits, accounts for, and isolates the resource usage of a collection of processes (Van Kalken, 2021). The following resources are maintained:

* ***CPU***: Restricts the CPU usage per container (Tiwari, 2024).
* ***Memory***: Limits the memory usage of a container (Tiwari, 2024).
* ***I/O***: Controls access to storage devices (Tiwari, 2024).
* ***Network***: Regulates network bandwidth and prioritization.

Each container has one cgroup. Containers usually run multiple processes simultaneously. The cgroup controls the different processes and makes sure they don’t exceed the given resources (Van Kalken, 2021).

Without cgroups containers can use all the system resources, impacting the performance of the host or other containers on the system (Tiwari, 2024).

By modifying the cgroup the number of resources the container is able to use can be modified (Van Kalken, 2021).

### Filesystems and storage: union file system

Many container base images tend to be quite large. Allocating large amounts of storage for multiple containers is highly inefficient when the containers share the same base (Heinz, 2021). The union file system (UFS) is a concept for data storage that solves this issue.

The union file system technology allows multiple layers of a file system to be combined into a single unified view (Appeti, 2023).

The layers creating the file system consist of a bottom- and a top layer. The bottom layer is read-only and is shared between different containers. On top of the read-only layer a writable layer is added. This layer is exclusive to the container and allows for new data to be stored and modified (LearnCantrill, 2023).

Using the union file system, it is possible for the container to write and change data stored in the read-only layers. This can be done using the copy-on-write principle. When data from the base layers needs to be modified, a copy of the data is made and written to the top layer (Appeti, 2023). The container is able to make changes to the data within this layer without affecting other containers using the same base image.

Container software like Docker and Podman use the union file system to save space while still providing isolation between containers (Heinz, 2021).

## Additions theoretical framework

### RAM and swap memory

RAM is a type of volatile memory that enables the quick reading and writing of data. It is used by the system to store data that is actively used by the system processes. In addition to the 512MB of RAM the Raspberry Pi Zero 2 W has swap memory. Swap memory is a portion of the SD card that is used to extend the available memory by acting as temporary storage space. When the RAM runs out of physical memory, inactive areas in the RAM are moved to the swap space to create space for active processes.

Swap is significantly slower because it relies on the speed of the storage device which is generally speaking a lot slower than the RAM.

# Hypothesis

In this chapter the expected results based on the research questions are discussed. The primary objective of this study is to determine the practical limitations of containerized applications on real-time embedded systems, focused on the CPU, memory and I/O limits.

## General hypothesis

Even though containers provide a lightweight alternative compared to traditional virtualization methods, it is expected that they do introduce performance overhead compared to running a program natively. The impact of the overhead caused by containers is expected to be measurable.

## CPU limitations

Using containers brings more overhead to the process. Running a process in a container is more CPU intensive and will take longer to complete.

**CE1: Running a program in a container will take more time than running the process natively. Running the container on the 3rd core and the engine on the 2nd will take longer and running the engine and the container on the 3rd core will be the slowest.**

Since containerized applications share system resources through a virtualization layer, additional processing is required. The overhead created by system calls, resource allocation and scheduling are expected to result in longer execution times compared a natively executed process.

**CE2: Increasing the amount of running containers will lead to a performance loss.**

When gradually increasing the number of containers running on the system the Raspberry Pi will reach its hardware limits resulting in longer runtimes for the programs being executed by the container.

## Memory limitations

## I/O limitations

# Methods and techniques

This chapter outlines the methods used to conduct the experiments designed to *map the practical limits of the CPU-bound, memory-bound and I/O-bound factors* in containers (as formulated in sub-question 2).

Each limiting factor was examined through resource dedicated experiments:

1. CPU limitations
2. Memory limitations
3. I/O limitations

### Base Raspberry Pi configuration

All the experiments were done on a Raspberry Pi Zero 2 W. The Raspberry Pi was configured with Raspberry PI OS Lite (64-bit), with Raspberry Pi Connect installed to remotely simulate the terminal (Raspberry Pi, z.d.-b). The test scripts were executed using Python version … .

These choices are elaborated on in the chapter ‘Justification of choices’.

## Methods CPU limitations

To evaluate the impact of containers on CPU performance, 2 types of tests were designed.

1. Measuring the CPU overhead introduced by a single container.
2. Exploring the CPU limit when running multiple containers simultaneously.

To simulate varying levels of CPU load, three Python scripts were created to perform computations with increasing intensity. These scripts were used for all the CPU experiments.

|  |  |  |
| --- | --- | --- |
| **Script name** | **Description** | **Intensity level** |
| simple\_calculation.py | Performs 100,000 random number generations and sums the values. | Low |
| medium\_calculation.py | Calculates the first 30 Fibonacci numbers. | Medium |
| long\_calculation.py | Calculates all prime numbers bell  ow 20,000,000. | High |

### CE1 Container overhead single container

The objective of this experiment was to measure the impact of containerization on the runtime performance of a CPU-bound program. The aim was to determine if overhead exists and how much the possible overhead was when running a process in a container compared to running it natively.

Each of the scripts described above was executed 10 times to ensure measurement consistency and to account for variability in execution times.

The experiment was conducted in three different test scenarios.

1. Native execution (control):
2. Separate container engine and container:
3. Same core container engine and container:

#### A blue square with white text AI-generated content may be incorrect.Native execution (control)

In the control scenario, all three Python scripts were executed directly on the Raspberry Pi Zero 2 W, without containerization. To ensure consistent performance and limit the interference of different programs the control experiment was done on a single isolated CPU core.

##### CPU core Isolation setup

Core isolation was configured by modifying the kernel startup parameters in /boot/firmware/cmdline.txt.

The following parameters were added:

* isolcpus=3: Prevents non-critical processes from being scheduled on cores 2 and 3.
* nohz\_full=3: Enables tickles mode, reducing the interrupts on the isolated cores.
* rcu\_nocbs=3: Prevents RCU (read-copy-update) callbacks executed on the specified core.

File snippet:

**console**=serial0,115200 **console**=tty1 **root**=PARTUUID=xxxx-xx **rootfstype**=ext4 **elevator**=deadline fsck.**repair**=yes rootwait **isolcpus**=2,3 **nohz\_full**=2,3 **rcu\_nocbs**=2,3

The run the Python file using the isolated core the script needs to be executed using:

taskset -c 3 python automated\_cpu\_test.py

#### A blue squares with white text AI-generated content may be incorrect.Separate container engine & container

For this experiment, the Python scripts were executed within a container on an isolated core, while the Docker engine was running on a separate core.

To achieve the CPU core isolation the steps described previously were used to isolate the cores.

The Docker engine (dockerd) and the container runtime (containerd) were moved to CPU core 2 using the taskset command:

sudo taskset -**pc** 2 <PID>

After pinning the process, they are briefly stopped and restarted to ensure the process migration.

sudo **kill** -**STOP** <PID>  
sudo **kill** -CONT <PID>

The container was isolated by starting the container on core 3:

sudo docker run --cpuset-cpus=3 -v /home/pirinske/Github/ContainersRTES/CPU\_tests/E1/log\_files:/app/logs e1:1.0

#### A blue square with white text AI-generated content may be incorrect.Container engine + container

To measure the total overhead when both the container engine and the container share the same CPU core, all related processes were pinned to core 3.

Both dockerd and containerd were moved to core 3 using the same method described above.

After the move the container running the script was executed on core 3.

sudo docker run --cpuset-cpus=3 -v /home/pirinske/Github/ContainersRTES/CPU\_tests/E1/log\_files:/app/logs e1:1.0

#### Collected data

The primary metric collected during this experiment was the total runtime of each Python script. Each script was executed 10x for the three experiments. The average was calculated for each script. Additionally, CPU core usage and system load were monitored using htop to verify proper isolation and ensure no external processes interfered with the measurements.

### CE2 Increasing running containers

This experiment was conducted to assess the CPU overhead introduced by containerization when increasing the amount of running containers. The goal was to determine if and when the performance starts to decrease.

For both the control and the actual test an automated script was created to spin containers. Each container executed a fixed Python script performing a known CPU-bound workload. The fixed Python scripts being used for this experiment were the simple\_calculation.py, medium\_calculation.py and long\_calculation.py scripts. Each script was executed 10x after which the averages of the output data was calculated.

#### Consecutive spinning (control)

For the control experiment the containers were executed consecutively. The previous container had to be finished before a new one was started. Automatically spinning new containers was done using the ‘Spinning\_x.sh’ script.

The automated test was done three times using the three different CPU load scripts.

The following command was used in the automated script to execute the containers:

sudo docker run -d -v /home/pirinske/GitHub/ContainersRTES/CPU\_tests/E2\_control/medium\_container/log\_files:/host\_log\_files e2:mid\_script

* -v enables the container to write the output file containing logger information from within the container to exist outside the container.

#### Simultaneous spinning

The goal of the experiment was to determine when the container performance starts to decrease when multiple containers run simultaneously. Like the control experiment three tests were performed using the three varying levels of increasing CPU load scripts.

For the actual experiment the same automated spinning script was used.

The command to run the containers was modified within the script.

sudo docker run -d -v /home/pirinske/GitHub/ContainersRTES/CPU\_tests/E2\_test/medium\_container/log\_files:/host\_log\_files e2:mid\_script

* -d causes the container to detach itself, making the container run separately from the script. This enables the script to continue spinning containers without having to wait for the previous container to finish.

#### Collected data

The main data collected during this experiment were the container- and script runtimes.

* ***Container timings:*** The container runtimes were logged using the creation and stop timings of the subprocess created when starting the container. To automatically log the data the script ‘outside\_logger.py’ was created. This script gathers the process timings, converts them to a readable time format and saves the information in a CSV file (container\_outside\_logs.csv)
* ***Script timings:*** The script runtimes were logged inside the container using the script logger.sh. This script was executed on container start up and logs the start and finish time of the script being run in the container. The information was saved and stored in a file outside of the container. Using the logfiles\_formatter.py script the runtime data was converted to a single csv file (container\_logs.sh)

Additionally, the CPU, memory and swap memory usage was monitored during the experiment using htop.

# Results

This chapter describes the results of the experiments done in the methods section.

## CPU experiment results

To determine the limits of using containers on real-time embedded systems multiple CPU tests were performed. In this section the results for the CPU experiments are described.

### CE1 Container overhead single container

To evaluate the impact of containerization of a single program on CPU performance, a series of tests were conducted using three varying CPU-intensive Python programs. Each program was executed under three distinct conditions to assess the overhead introduced by containerization.

1. ***Control (native execution)***: The script was run directly on the host system without containerization.
2. ***Container + engine***: The container engine and the container were isolated on different CPU cores.
3. ***Container & engine***: The engine and container were both pinned to the same CPU core.

#### Execution time results

Table 2 shows the average execution times (in seconds) for each script under the three configurations described above.

Table 2 Mean execution times (in Seconds)

|  |  |  |  |
| --- | --- | --- | --- |
| Script | Control group | Engine & container | Engine + container |
| simple\_calculation.py | 0.08480 | 0.09063 | 0.09061 |
| medium\_calculation.py | 2.40316 | 2.38953 | 2.38974 |
| long\_calculation.py | 41.79462 | 44.97788 | 45.01682 |

* simple\_calculation.py : The table above shows that the execution time for this simple program was slightly higher in the containerized environments (approximately 0.0906 sec) compared to the control group (approximately 0.0848 sec).
* medium\_calculation.py: As shown in the table the execution times for the medium calculation across the different environments were nearly identical, with only minor fluctuations (within ±0.01s).
* long\_calculation.py: In the case of the longest program a more noticeable increase in execution time can be observed. The control group completed execution in approximately 41.79 seconds, while the ‘engine & container’ and ‘engine + container’ configurations took 44.98 and 45.02 seconds (~7-8% increase).

The results described in table 2 are visually represented in figure 1.

A graph with different colored bars

AI-generated content may be incorrect.

Figure 1 Mean execution times (in seconds)

#### Execution time variability

To assess consistency across repeated runs, the standard deviation was calculated for each script and configuration. The results are shown in Table 3.

Table 3 Standard deviation execution times (in seconds)

|  |  |  |  |
| --- | --- | --- | --- |
| **Script** | **Control group** | **Engine & container** | **Engine + container** |
| simple\_calculation.py | 0.00561 | 0.00188 | 0.00196 |
| medium\_calculation.py | 0.00051 | 0.00072 | 0.00059 |
| long\_calculation.py | 0.28433 | 1.96096 | 1.71923 |

The standard deviation was calculated as a percentage of the mean to be able to determine the consistency of the data. The results are shown in table 4.

Table 4 Standard deviation as percentage of the mean

|  |  |  |  |
| --- | --- | --- | --- |
| **Script** | **Control group** | **Engine & container** | **Engine + container** |
| simple\_calculation.py | 6.61% | 2.07% | 2.16% |
| medium\_calculation.py | 0.02% | 0.03% | 0.02% |
| long\_calculation.py | 0.68% | 4.36% | 3.82% |

The standard deviation as a percentage of the mean was highest for the simple\_calculation.py program in the control group (6.61%), while the containerized configurations showed lower relative variability (~2%). The medium\_calculation.py script has the lowest relative variability across all environments (≤ 0.03%). For the long\_calculation.py the control containerized setups note increased variability (3.82%–4.36%) compared to the control group (0.68%).

### CE2 Increasing running containers

The aim of this experiment was to determine the overhead caused when running multiple containers. The following tests were performed:

* ***Consecutive spinning (control)***: For the control experiment, spinning a new container was delayed until the previous container was completed.
* ***Simultaneous spinning (test)***: 10 containers were started in direct concession to measure the performance impact.

The control and actual test were performed using the long\_container.py program to run within the container. Each container was executed 10 times in fast succession, to determine the performance loss when running multiple containers at the same time.

The experiment was performed twice: once on the Raspberry Pi Zero 2 W and once on the Raspberry Pi 5. The actual test failed on the Raspberry Pi Zero 2 W due to repeated crashing, resulting in no data measurements for this test. Consequently, only the results from the Raspberry Pi 5 are considered here.

#### Execution time results

The results of the control experiment and the actual test are displayed in table 5.

Table 5 Results simultaniously running containers

|  |  |  |
| --- | --- | --- |
| **Metric** | **Consecutive (Control)** | **Simultaneous (Test)** |
| mean | 7.55607 sec | 26.06481 sec |
| std | 0.02829 sec | 1.89137 sec |
| Total runtime | 90.17530 sec | 34.27195 |

The average runtime in the control group was 7.56 seconds, while the average runtime under simultaneous spinning conditions rose to 26.06 seconds per container**,** a 249% increase. Additionally, the variance increased substantially (standard deviation of 1.89 seconds vs. 0.03 seconds in the control), highlighting the performance unpredictability introduced by concurrent execution.

When comparing the total runtime, the control experiment took much longer to complete (90 sec) compared to running the containers simultaneously (34 sec).

# Discussion

# Conclusion

# Recommendations

# Justification of choices

## Experiment hardware

To successfully complete the experiments, selecting the appropriate hardware is necessary. The choice of hardware depends on the scope and requirements of the project. By comparing the different options available within the defined scope, the right hardware can be selected.

#### Scope

Due to time constraints, choosing the hardware for the experiment will be limited to the Raspberry Pi range (see chapter on demarcation). The Raspberry Pi series consist of four primary models: Flagship series, Zero series, Compute Model series and Pico microcontrollers (Contino Nate, 2024).

#### Requirements

***Must haves:***

* ***Support Linux OS version*:** As clarified in the demarcation chapter, this project will exclusively focus on Linux OS containers. Therefore, the selected container must support Linux OS versions.
* ***Availability*:** The controller needs to be available within 4 weeks after ordering to be able to meet upcoming deadlines.
* ***Budget***: The project has a maximum budget of €500. While it is unlikely that the hardware will exceed this limit within the Raspberry Pi range, the controller cannot exceed the limit. Als long as the controller is priced below €500 the price will not impact the final decision.

***Deciding factor:***

* ***Limited resources***: The experiment aims to test the limits of containers on the selected controller. If the hardware is too powerful, it could prolong the process and complicate reaching the limits of container performance. Selecting a controller with limited resources is important.

#### Comparison

Based on the scope and requirements, the four main Raspberry Pi models are compared. Due to time constraints only the newest model in each series will be considered. The table below provides an overview of the key specifications, including pricing and availability as of March 2025. Pricing is an estimate based on various suppliers.

Table 6 Comparison table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Flagship (5) | Zero (2 W) | Compute module (4) | Pico (2) |
| Linux OS support | Yes | Yes | Yes | No |
| Availability | < 1 week | < 1 week | < 1 week | < 1 week |
| Budget | €100, - | €20, - | €45, - | €9, - |
| RAM | 2GB, 4GB, 8GB | 512 MB | 1GB, 2GB, 4GB, 8GB | 520KB |
| CPU | Quad core 2.4 Ghz | Quad core 1.0 Ghz | Quad core 1.5 Ghz | Dual core 150 Mhz |

#### Choosing hardware

Based on the comparison, three models meet the essential requirements of supporting Linux OS, budget and availability. Only the Raspberry Pi Pico 2 is not able to meet the criteria as it does not support a Linux operating system.

This leaves three suitable options: the Raspberry Pi 5, the Raspberry Pi Zero W and the Raspberry Pi Compute 4.

After evaluating these models, the **Raspberry Pi Zero 2 W** emerges as the most fitting choice. While it has the least powerful hardware, the key factor is the project’s focus on limited resources.

## Operating system

To be able to use the Raspberry Pi Zero 2 W it is required to install an operating system. Raspberry Pi provides different alternatives. Since the Raspberry Pi Zero 2 W is equipped with a 64-bit processor downloading a 64-bit OS is recommended. Raspberry Pi offers 3 different 64-bit versions(Raspberry Pi, z.d.-a):

* Raspberry Pi OS with desktop
* Raspberry Pi OS with desktop and recommended software
* Raspberry Pi OS Lite

When comparing these OS versions, the biggest difference is that the Lite OS doesn’t include the desktop environment. It is the most basic OS version only including a text only console (klricks, 2022).

## Scripting software

To create the scripts running in the container a scripting language had to be chosen. One of the main requirements of this project was that it would be easily understandable and repeatable by colleagues. Even though a language like C++ generally offers better performance, Python remains the more accessible language. Therefore, the use of Python was preferred over other languages.

## Container engine

To create, manage and run containers a container engine had to be decided on. Even though there are many different services providing container engines, Docker is the most widely used. As a result, there is a lot of documentation available.

One of the main requirements of this project is accessibility and reproducibility. For this reason, Docker was chosen for this project due to Dockers popularity and the availability of documentation.

# Description of realized professional products

# Evaluation of quality criteria

# Process evaluation

# Sources

# Appendix

E1: