# 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’ |
| 24-04-2025 | 0.8 | Rewrote chapter ‘theoretical framework’. Combined chapter ‘theoretical background’ into ‘theoretical framework’ |
| 28-04-2025 | 0.9 | Rewrote ‘Hypothesis chapter and added the Memory and I/O sub-chapters. Wrote the chapter ‘Methods’. Started rewriting the ‘Results’ section. |

# 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 framework

In this chapter the theoretical foundation for the research will be outlined for running containers on real-time embedded systems.

## 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) , meaning the system must meet strict timing constraints for its operations (GeeksforGeeks, 2022).

Real-time embedded systems are typically classified as hard or soft real-time. In a hard real-time system, every deadline must be met – a missed deadline can lead to complete system failure or unsafe situations (for example, in medical devices or aviation control systems)​file-bmja24z35qx1s6pbgnspdu. In contrast, a soft real-time system can tolerate occasional deadline misses, which only degrade the quality of service rather than cause a total failure (as in the case of multimedia streaming or online gaming)

## Containers in real-time embedded systems

Eventhough container technology originated in server environments, they are increasingly being used in embedded systems to leverage the benefits of portability, isolation, and scalability even on resource-constrained devices. Real-time embedded systems tend to have limited CPU power, memory and storage, resulting in the question what the practical limits are when using containers in these environments. Deploying containerized applications on a small embedded device could introduce overhead or resource contention that would not exist when running the same tasks natively. It’s important to research how containerization might affect system resources and timing.

​For this research three categories of potential performance bottlenecks are considered.

* ***CPU-bound***: A process is CPU bound when the execution speed is limited by the CPU’s processing power. The CPU limit can be reached by performing a lot of simple calculations in quick succesion (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).

# TODO: Add what type of program is suitable to reach the limit with source.

* ***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).

These categories are useful for analyzing container performance because container overhead might exacerbate CPU usage (through virtualization overhead), affect memory usage (through additional layers of abstraction), or impact I/O (through layered file systems). The experimental chapters focus on how containers perform under each of these constraints on a real-time embedded platform.

# TODO: Hier naar kijken met sources en idd kijken naar hoe deze dingen container processen zouden kunnen beinvloeden. → in Hoofdstuk Hypothese verwerkt.

## The Linux Kernel and Container Processes

A diagram of a computer process

AI-generated content may be incorrect.The **Linux Kernel** is the core of the operating system that acts as the bridge between the hardware and software applications on a system . It manages critical system resources, including the CPU, memory, and peripheral devices, and handles low-level tasks such as executing processes, managing memory, accessing files and coordinating I/O operations (GeeksforGeeks, 2025).

One of the responsibilities of the kernel is executing and managing processes.

In Linux, a process is an instance of a running program, including the program’s executed code and its allocated resources. A container is essentially a special Linux process. It runs on the same host kernel but is isolated so that it cannot see or affect the rest of the system outside its designated environment (Mullinix et al., z.d.)​.

### Process Isolation: Linux Namespaces

One of the essential features enabling container process isolation is done by the Linux namespace mechanism (Kuninoto, 2024). When the process is created the namespaces are set to provide isolation.

The Linux kernel implements multiple types of namespaces, each isolating a particular resource domain (Mullinix et al., z.d.).

* PID namespace: Isolates process ID’s. Each container gets its own process ID space, as a result processes in the container can only see and reference process within that container’s PID namespace (Tiwari, 2024).
* Mount namespace (mnt): Isolates the filesystem view. Every container possesses its unique view of the filesystem, independent from the host and other containers (Tiwari, 2024).
* IPC namespace (inter process communication): Facilitates the communication between processes (such as shared memory, semaphores, message queues). This namespace prevents interfering with other IPC’s (Boutnaru, 2023).
* 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: Control Groups (cgroups)

While namespaces isolate the view of resources, Linux control groups (cgroups) regulate the usage of resources by processes (Van Kalken, 2021). The main resource controls provided by cgroups include: CPU scheduling, memory usage, block I/O throughput, and network bandwidth. Using cgroups the percentage of CPU time, a fixed amount of RAM or a capped I/O rate can be limited.

Each container’s processes reside in their own cgroup hierarchy, ensuring that if a container process tries to overuse the CPU, memory or I/O the kernel will throttle it according the limits defined in the cgroup. This prevents one container from starving the host or other containers of CPU cycles or memory.

### Filesystems and storage

Containers not only isolate processes and resource usage, but also the file system. To make container provisioning efficient, container frameworks use the union file systems to manage container images and storage. A union file system allows multiple distinct file system layers to be stacked or overlaid, presenting them as a single coherent file system to the container.

Using a union file system with copy-on-write semantics dramatically saves disk space and memory, since multiple containers can share one copy of an OS image or application binaries without duplication​.

For real-time embedded systems, the use of a union file system could mean that I/O performance might be affected by the extra layering. Reading a file might involve checking multiple layers, and using the copy-on-write principle might lead to a slight overhead.

## Summary

In summary, container technology on Linux relies on kernel-provided features: namespaces to isolate process views, cgroups to control resource usage, and union file systems for efficient storage management. A real-time embedded system using containers will leverage these same mechanisms. Understanding how they work is crucial for anticipating how containerization might introduce performance overhead or resource contention in CPU-bound, memory-bound, or I/O-bound scenarios on real-time embedded systems.

# Hypothesis

In this chapter the expected outcomes of the experiments designed to evaluate the practical limits of containerized applications on real-time embedded systems are discussed. Each hypothesis is based on the theoretical framework established in the previous chapter and focuses on one of the three resource categories: CPU, memory and I/O.

## CPU limitations

*CE1:*

The overhead caused by a single container is expected to be very minimal on a Linux based system. Unlike VM’s containers share the host kernel and do no emulate hardware (Ferraz, 2022). Minor slowdowns can be caused by the various isolation layers, but they are minor and most likely negligible in practice (Morabito, 2016).

However, when running the container engine and the container on the same core, this might lead to longer execution times, since they share the same core and need to perform context switching.

*When running a program in a container minor CPU overhead is expected compared to running the program natively. When running the container and the engine on the same core, minor overhead can be expected.*

*CE2:*

When running multiple containers the primary performance limit is most likely that the CPU cores need to be shared between the processes rather than containerization itself (ZPiDer, 2018). As the number of processes increases, each process gets a smaller share of the CPU cycles, therefore resulting in longer completion times.

*When running multiple containerized programs simultaneously, the runtime should be quite similar to running the same number programs natively.*

## Memory limitations

*ME1:*

Containerized applications introduce additional memory usage compared to running a program natively (Ferraz, 2022). Each container has its own isolated user space that duplicates resources that could be shared in a single native environment (Duffy, 2014).

*When running an application natively compared to containerized, a observable runtime difference is expected due to the overhead caused by the isolated environment.*

*ME2:*

Swap memory is a portion of the storage memory (SD card on Raspberry Pi) that is used to extend the available memory by acting as temporary virtual memory. 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 a lot slower..

*With swap-memory disabled, containers will crash sooner, but maintain higher performance until they crash.*

*ME3:*

When a Linux system runs out of memory the default behavior is for the OOM (out-of-memory) killer to terminate one or more processes to free memory. The amount of memory a container can use can be limited using cgroups. When the limit is exceeded the OOM killer kills the process.

*When limiting the amount of memory the container can use the OOM killer will kill the container process instead of resulting in a system crash.*

## I/O limitations

*EI1:*

Docker on Linux used the Union File System (OverlayFS) for container storage (Heinz, 2021). Write operations go though a copy-on-write layer. The first time the container writes a file that originates from the read-only layer, the file system needs to copy the file to the writable layer first (*OverlayFS Storage Driver*, 2025). This might cause notable overhead.

*Writing a file natively to an external I/O medium will take notably less time than writing the same file from within a container.*

*EI2:*

The performance of reading data from within a container is much closer to reading the data natively. If the file is not modified by the container the file is read form the lower read-only layer, that can be shared and doesn’t need to be copied for every container instance (LearnCantrill, 2023). \

*Reading from within a container will not cause much overhead compared to reading a file natively, because read only layers are not copied to the container exclusive space.*

*EI3:*

Handling small files tends to be slower than one larger file of the same size. Each file read incurs opening the file, reading its metadata, updating access times and finally reading the file itself.

# Nog even kijken hoe ik dit wil doen: nu vergelijk ik meerdere files tov 1 file binnen de container context maar niks native.

# Methods

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

## Experimental setup

By default all experiments were performed on a Raspberry Pi Zero 2W running Raspberry Pi OS Lite (64-bit). The device features a 1 GHz quad-core ARM Cortex-A53 CPU and 512 MB of RAM, providing a constrained environment to test the practical limits.

The testing programs were created using Python. For each factor (CPU, memory and I/O) three scripts were created simulating varying resource intensities. In addition to the Python programs bash scripts were created to automate doing the tests multiple times and to log the results. Each test was performed ten times to account for variability.

For this research only Docker was used as the container software (see chapter ‘Justification of Choices’.The same base container was used for all the experiments. The container was built including the Python version … with the necessary libraries installed. The files to run the experiments were mounted when the containers were created.

Details on how the experiments can be executed and reproduced can be found in the ‘experiment\_details.md’ file in the accompanying experimental repository.

## CPU Experiments

The CPU-focused experiments were designed to measure the overhead introduced by containerization. To evaluate the impact of containers the following experiments were done.

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

### Experiment C1: CPU overhead single container

**Goal**  
The aim of this experiment was to determine the overhead caused by a single container compared to running natively.

**Method**The experiment was conducted in three different test scenarios.

1. *Native execution (control):* In the control scenario, all three varying intensity Python programs 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.
2. *Separate container engine and container*: The Python programs were executed within a container on an isolated core, while the Docker engine was running on a separate core.
3. *Same core container engine and container*: Both the Python programs and the container engine were executed on the same core.

**Collected data**  
For each test, the execution time required to complete each type of computational program (simple, medium and long) was recorded and automatically logged in a CSV file. Additionally, CPU core usage and system load were monitored using htop to verify proper isolation and ensure no external processes interfered with the measurements.

### Experiment C2: Container overhead multiple containers

**Goal**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 compared to running the same amount of containers natively.

**Method**

# Aanvullen op basis van aanpassingen die gemaakt moeten worden om te vergelijken met native.

**Collected data**The main data collected during this experiment were the container- and script runtimes. The data was stored in a CSV file.

* *Container timings:* The container runtimes were logged using the creation and stop timings of the subprocess created when starting the container.
* *Script timings:*The runtime of the Python program being executed within the container was logged.

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

## Memory experiments

The memory experiments were designed to examine how memory intensive tasks in containerized environments perform. The Following tests were done:

1. Measuring the overhead caused by a single container.
2. Determining the influence of swap memory on a single container.
3. Determining the influence of the OOM killer on containers.

Unlike the CPU experiments there is no option to isolate memory. In addition to the experiments minimal OS processes were running consistently in the background across all experiments.

### Experiment M1: Memory overhead single container.

**Goal**The goal of the first memory experiment was to determine the overhead caused by a single container on the memory usage.

**Method**

The three different memory intensive Python scripts were used in both the control and the actual experiment.

* *Native execution (control)*: The Python programs were performed natively to determine the baseline performance.
* *Container execution*: The Python scripts were executing using the base container image described above.

**Collected data**

The metrics collected during this experiment were the total execution time of the container and the total execution time of the Python program. Additionally the memory usage was logged every 0.1 second.

### Experiment M2: Influence Swap memory on containers

**Goal**

The aim of this experiment was to determine the influence of swap memory on the container.

**Method**

The determine how swap memory impacts the performance it was important to reach the hardware limits of the Raspberry Pi Zero 2W. For this experiment only the most intensive Python program was used.

* *Swap memory enabled (control)*: The swap memory was enabled at the default value of 512MB.
* *Swap memory disabled*: The swap memory was disabled during the runtime of the experiment (experiment\_instructions.md).

**Collected data**

For this experiment both the program runtime and container runtime were logged. The memory being used was logged every 0.1 seconds. Additionally using htop the swap memory was monitored.

### Experiment M3: Influence OOM killer containers

**Goal**

The goal of this experiment is to determine the influence of the OOM killer on containers.

**Method**

# Dit experiment moet eigenlijk ook worden aangepast. Wil eigenlijk juist weten of de limieten te zetten zijn voor de container en dat OOM het process dan juist killt.

**Collected data**

The data collected during this experiment were the container runtime and the program runtime. In addition the memory being used by the Python program was measured for both the control and the actual experiment.

# I/O Experiments

The I/O experiments were used to map the influence of containers on I/O operations. To determine the reading and writing performance in containerization the following experiments were designed:

1. Determining the writing performance of a single container compared to native execution.
2. Determining the reading performance of a single container compared to native execution.
3. Reading multiple files.

The experiments were performed using a … USB. The device was a USB 2.0 device with a theoretical reading and writing speed of … .

### Experiment I1: Writing I/O performance single container

**Goal**

Experiment I1 focused on the performance of writing data to an external USB storage device from within a container. The performance difference between writing data natively compared to writing within a container was measured.

**Method**

To determine the writing perfromance three different intensity Python programs were used. Each program writes a different amount of generated data to a file (created on the USB).

Using the programs the following tests were performed:

* *Writing native (control):* The data generated by the varying Python programs was written to the USB using the native tools.
* *Writing within container*: The Python programs were executed from within the container writing the generated output the USB from within the container.

**Collected data**

The data collected during this experiment consists of the reading and writing speed to and from the USB device. The read and write speeds were monitored every 0.1 seconds. In addition, the program runtime and container time were logged.

### Experiment I2: Reading I/O performance single container

**Goal**

The goal of this test was to measure the performance difference of reading a file from an external I/O device natively compared to running it from within a container.

**Method**

This experiment focused on reading one file. The file that was used for both the control and the actual experiment was a 1GB binary file. The file was stored on the USB.

* *Reading native (control):* The file on the USB stick was read using natively by the Raspberry Pi to determine the reading speed.
* *Reading within container:* The file on the USB device was read from within the container.

**Collected data**

Similar to the experiment above the reading and writing speeds of the I/O device were measured and the container runtime and program runtime were logged.

### Experiment I3: Reading multiple files

**Goal**

The goal of this experiment was to determine the peformance difference between reading one file and multiple files from within a container and natively.

# Experiment nog niet uitgevoerd

# Results

In this chapter the results of the experiments described in the previous chapter are discussed. Each section includes a summary of key findings and average results.

### Experiment C1: CPU overhead single container

Experiment C1 measured the execution times for CPU-bound programs of varying complexities in both native and containerized environments.

Table … shows the experiment results. The average execution time and the standard deviation of the 10 runs were calculated.

|  |  |  |  |
| --- | --- | --- | --- |
| **Program** | **Control group** | **Engine & container** | **Engine + container** |
| *Simple* | 0.08480 | 0.09063 | 0.09061 |
| *Medium* | 2.40316 | 2.38953 | 2.38974 |
| *Long* | 41.79462 | 44.97788 | 45.01682 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Program** | **Control group** | **Engine & container** | **Engine + container** |
| *Simple* | 0.00561 | 0.00188 | 0.00196 |
| *Medium* | 0.00051 | 0.00072 | 0.00059 |
| *Long* | 0.28433 | 1.96096 | 1.71923 |

The table shows a small runtime increase for the simple and long programs. The medium intensity program shows nearly identical run times across the three tests.

# 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

ME2: Most likely more similar runtimes compared to the prediction in the hypothesis. Because the container images and libraries are the same, they can be shared by the containers through the engine?.

# 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: