Tampere University

Leevi Yliollitervo

CONTAINERIZATION OF USER INTERFACE APPLICATIONS

Performance Impact of Containerization

Master's Thesis
Faculty of Information Techonology and Communication Sciences
Examiners: Lehtori Jyke Savia
Pasi Sorsa
May 2025

ABSTRACT

Leevi Yliollitervo: Containerization of User Interface Applications
Master's Thesis
Tampere University
Joint Studies in the Master's Degree Programme in Computer Science
May 2025

Containerization has become a dominant strategy in modern software deployment, offering benefits such as improved portability, security, and environment consistency. Although its advantages in server-side applications are well-established, its impact on graphical user interface (UI) applications remains underexplored, particularly in embedded systems with constrained resources. This thesis investigates the performance implications of containerizing UI applications by conducting empirical tests on a Chromium-based browser operating in both native and containerized environments on an ARM-based embedded Linux platform.

The study evaluates performance on key metrics, including CPU usage, memory consumption, JavaScript execution, and rendering throughput. Quantitative analysis revealed that containerization introduces negligible CPU overhead (approximately +0.9%) and has no significant impact on JavaScript execution performance. However, it significantly increases memory usage by 31.7% and reduces rendering performance by 7.84% on average, as measured by standardized MotionMark benchmarks. The findings indicate that while scripting-heavy applications may remain viable within containers, graphics-intensive workloads are more susceptible to container-induced performance degradation.

These results suggest that performance trade-offs must be carefully considered when deploying containerized user interface applications in latency sensitive or resource-constrained environments. The study contributes to understanding the practical limitations of containerization and outlines directions for future research, including GPU acceleration and a broader evaluation across UI frameworks and hardware platforms.

Keywords: containerization, user interfaces, Chromium, embedded systems, performance analysis, Wayland, benchmarking

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Leevi Yliollitervo: Käyttöliittymäsovellusten kontainerisointi
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelyopin maisteriohjelma
Toukokuu 2025

Kontainerisointi (containerization) on noussut keskeiseksi menetelmäksi nykyaikaisten ohjelmistojen jakelussa, tarjoten etuja kuten siirrettävyyttä, tietoturvaa ja yhtenäisiä ajoympäristöjä. Vaikka kontainerisoinnin hyödyt palvelinsovelluksissa ovat laajalti tunnettuja, sen vaikutuksia graafisten käyttöliittymäsovellusten suorituskykyyn on tutkittu vain vähän, erityisesti sulautetuissa järjestelmissä, joissa resurssit ovat rajalliset. Tämä tutkielma tarkastelee kontainerisoinnin suorituskykyvaikutuksia empiirisesti vertaamalla Chromium-pohjaisen selaimen toimintaa natiivissa ja kontainerisoidussa ympäristössä ARM-pohjaisessa sulautetussa Linux-laitteessa.

Tutkimuksessa mitattiin suorituskykyä keskeisten mittarien avulla: suorittimen kuormitus, muistinkulutus, JavaScript-suorituskyky sekä renderöintiteho. Tulokset osoittavat, että kontainerisointi aiheuttaa vain vähäistä prosessorikuorman nousua (+0,9 %) eikä vaikuta merkittävästi JavaScriptin suoritukseen. Sen sijaan muistinkulutus kasvoi merkittävästi 31,7 % ja renderöintisuorituskyky heikkeni keskimäärin 7,84 % MotionMark-suorituskykymittauksen perusteella. Tulokset viittaavat siihen, että skriptipohjaiset käyttöliittymät soveltuvat kontainerisointiin kohtuullisesti, mutta graafisesti raskaat sovellukset kärsivät suorituskyvyn heikkenemisestä.

Tutkimuksen havainnot korostavat tarvetta punnita kontainerisoinnin hyötyjä ja kustannuksia erityisesti järjestelmissä, joissa vasteaika ja resurssien hallinta ovat kriittisiä. Tutkielma tarjoaa pohjan jatkotutkimukselle, esimerkiksi grafiikkalaitteistokiihdytyksen hyödyntämisen ja eri käyttöliittymäkehysten arvioinnin osalta.

Avainsanat: kontainerisointi, kontitus, käyttöliittymät, Chromium, sulautetut järjestelmät, suorituskykyanalyysi, Wayland, suorituskykymittaus

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☐ No
☒ Yes

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
ChatGPT	4o
Copilot	...

Purpose of the use of AI

ChatGPT was used in a limited capacity to improve the overall readability of the thesis. Its uses included refining the structure of selected sections and clarifying the presentation. The use of the tool was restricted to text adjustments at the surface level and did not influence the generation or interpretation of the original research findings.

Copilot was used to generate code to automate data collection.

Parts of this work, where AI was used

All chapters include improvements in readability made by AI.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

This thesis was conducted as a commissioned project for KONE Oy. I am sincerely grateful for the opportunity to conduct this research and for financial support throughout the project. I would like to extend my deepest appreciation to Sorsa Pasi for enabling this collaboration and for his continued support. I also extend my appreciation to my colleagues for their valuable assistance and support throughout the research and writing process. Finally, I am especially thankful to Savia Jyke for his thorough review and constructive feedback on the final manuscript.

Tampere, 28th May 2025

Leevi Yliollitervo

CONTENTS

GLOSSARY

ANGLE	Almost Native Graphics Layer Engine
API	Application Programming Interface
ARM	Advanced RISC Machines
CLI	Command-Line Interface
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CUDA	Compute Unified Device Architecture
DDR4	Double Data Rate 4
DOM	Document Object Model
DVFS	Dynamic Voltage and Frequency Scaling
FPS	Frames Per Second
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
I/O	Input/Output
IPC	Inter Process Communication
JIT	Just-in-Time
LPDDR4	Low Power Double Data Rate 4
OCI	Open Container Initiative
OS	Operating System
QEMU	Quick Emulator
RAM	Random Access Memory
RISC-V	Reduced Instruction Set Computer Five
UI	User Interface
URL	Uniform Resource Locator
VA-API	Video Acceleration API

1. INTRODUCTION

In modern software development, containerization has emerged as a transformative technology that improves the deployment, scalability, and management of applications. By encapsulating applications and their dependencies in lightweight and portable containers, consistency can be ensured across different computing environments (Eder 2016). Although containerization offers various benefits, its impact on performance, particularly in the context of user interface (UI) applications, remains a critical area of study.

User interface applications demand responsiveness, low latency, and seamless interaction to deliver a high-quality user experience (Fu et al. 2022). As organizations adopt more containerized architectures, understanding the trade-offs between the flexibility of containers and their potential overhead on UI performance is important. Factors such as resource allocation, container orchestration, and interprocess communication can influence the efficiency and responsiveness of UI applications running in containerized environments.

This study aims to answer the following questions: "How large is the performance impact of containerization on UI applications?" and "How large is the overhead of containerization on system resources?" This study tries to answer these questions by analyzing performance metrics, including CPU and memory utilization, as well as execution and rendering performance. By conducting empirical evaluations on native and containerized platforms, this research aims to provide understanding into the viability of containers for UI applications and identify optimization strategies to mitigate performance bottlenecks.

As containerization continues to evolve, this research seeks to bridge the gap between theoretical benefits and practical performance considerations, enabling more informed decision making in the deployment of UI applications within containerized ecosystems.

The structure of this thesis is as follows: Chapter 2 provides background information on application containerization, UI architectures, display servers, and performance metrics. Chapter 3 outlines the research methodology, including experimental setup and data collection techniques. Chapter 4 presents the results, and Chapter 5 presents the analysis of the data. Chapter 6 follows with a discussion of the findings and potential optimizations. Finally, Chapter 7 concludes the thesis with key takeaways, practical recommendations, and future research directions.

2. BACKGROUND

There is a growing need for software that is more scalable, portable, and secure. The increasing diversity of computing environments, from cloud data centers to embedded devices, requires implementations that allow software to run seamlessly across different platforms with minimal modification. Containerization addresses this requirement by providing a lightweight, consistent, and isolated runtime environment that abstracts applications from the underlying operating system (OS) and hardware.

Traditionally, deploying software applications involves installing dependencies, configuring environments, and ensuring compatibility with the host system. This approach often leads to issues related to dependency management, conflicts between different software versions, and difficulties in maintaining consistency across multiple devices. Containerization simplifies deployment by isolating applications along with their dependencies in a container that can be executed consistently across different environments.

Scalability is another important factor for containerization. Modern applications must be able to scale efficiently to meet differing workloads and demand. Containers provide a modular approach to application development and deployment, allowing individual components to be copied, updated, or replaced independently. This enables more efficient resource utilization and faster response to changes in system requirements. (Merkel et al. 2014)

Security is also a primary concern in software development. By isolating applications and their dependencies within containers, potential vulnerabilities can be contained, preventing them from affecting other parts of the system. Containerization enforces process separation by providing an additional layer of isolation beyond traditional process boundaries, including distinct namespaces and dedicated file systems. This minimizes the risk of unauthorized access or malicious exploitation. In addition, container security policies and best practices can further enhance protection against threats. (Boettiger 2015)

Containers provide an abstraction layer similar to traditional virtualization, but with some significant differences. Virtual machines emulate entire hardware systems along with the OS environments, while containers share the host system's kernel while maintaining isolation for applications. This lightweight approach reduces overhead, enabling faster startup times and improved performance. (Eder 2016)

A containerized application consists of an image that includes the software, libraries, runtime environment, and necessary configurations. When deployed, this image is executed as an isolated process with the container runtime. Technologies such as Podman, Docker, and Kubernetes help container management, enabling developers to build, run, and manage containerized applications efficiently. (Pahl 2015)

Containerization allows abstracting applications from the underlying hardware and operating system, containers provide a consistent and efficient runtime environment. For UI applications, containerization provides several advantages, including improved maintainability, enhanced security and potentially simplifying deployment. As container technology continues to develop, it is becoming a necessary tool for modern software development, allowing applications to run reliably across different environments with minimal effort.

2.1 Chromium

Initially, the plan was to use Qt webengine (Qt Company 2025). Since Qt webengine is based on chromium, implementation was simplified to use chromium directly. Using only chromium removes the Qt dependency from the container, which both simplifies the build process and decreases the container image size.

Chromium provides support for window system protocols, which are used to transfer the image from the application inside the container to a display server on the host system. Display data is transferred over a socket, as opposed to shared memory access. This allows for a greater separation between the container and the host (Chromium Development Team 2025).

Chromium-based applications rely on a multi-process architecture and a complex graphics pipeline designed to balance performance, security, and responsiveness. Key components in this pipeline include the Blink rendering engine, which handles HTML, CSS and DOM processing; Skia, Chromium's 2D graphics library responsible for rasterization, and the compositor, which coordinates frame generation and display scheduling (Skia Contributors 2025).

For hardware-accelerated rendering, Chromium utilizes ANGLE (Almost Native Graphics Layer Engine), a compatibility layer that translates OpenGL ES or WebGL calls into Direct3D, Vulkan, or OpenGL depending on the host platform. This translation allows Chromium to maintain cross-platform consistency while leveraging native GPU APIs for performance. When GPU access is restricted or unavailable, as is common in some containerized environments, Chromium falls back to software rasterization, which significantly reduces rendering throughput and affects visual fluidity.

Chromium also features an asynchronous input event and a rendering model. Input events (e.g., mouse, keyboard, touch) are routed through an event queue and processed by the appropriate thread, typically the main thread or compositor thread, before triggering UI updates. Frame composition and display are scheduled by the composer thread, which interacts with the GPU process to finalize rendering. The overall pipeline is designed to target a consistent frame rate (e.g., 60 FPS), but can be impacted by IPC latency (mainly sockets in this case), GPU driver behavior, and thread scheduling.

Performance measurement within Chromium is facilitated by built-in tools such as Chrome Tracing and the DevTools Performance panel, which provide access to event timing, rendering stages, frame intervals, and dropped frame diagnostics. These tools are essential for identifying bottlenecks in rendering and input responsiveness, particularly in containerized environments where system-level constraints can exacerbate latency or reduce throughput.

2.2 X11 and Wayland Display Servers

The choice of display server protocol is an important aspect of UI architecture, as it determines how graphical applications interface with the underlying operating system and hardware. In Unix systems, X11 (the X Window System) has historically been the dominant display server protocol, providing network transparency and a well-established ecosystem. However, X11's design introduces inefficiencies, including the overhead of interprocess communication and the reliance on extensions for modern graphical features. (Peersman et al. 2011)

For Linux systems, Wayland has become a modern alternative to X11, designed to provide a more efficient and secure compositing model (Wayland Development Team 2025). Unlike X11, which separates the roles of the window manager and the display server, Wayland combines these functionalities into a single compositing server, reducing latency and improving graphical performance (Peersman et al. 2011). Chromium supports both X11 and Wayland, allowing developers to select the appropriate backend depending on the target environment. Wayland's advantages, such as reduced input lag and direct rendering capabilities, make it particularly suitable for embedded applications where responsiveness is important.

2.3 Performance Metrics in Software Systems

Evaluating the performance of user interface applications requires selecting appropriate metrics that accurately reflect system behavior. Choosing the right performance metrics is important, as they help isolate the effects of containerization on an application running in a controlled environment. Direct measurement of performance impact down to the clock cycle level is impractical because modern operating systems abstract hardware details through complex scheduling, memory management, and virtualization layers

(Shirazi 2003). Furthermore, features such as dynamic frequency scaling, process prioritization, and context switching obscure direct attribution of performance variations to specific instructions (Pillai and Shin 2001). Instead, we must rely on statistical methods and well-defined performance indicators to estimate the impact with reasonable accuracy.

2.3.1 CPU Usage

CPU utilization is a commonly used metric that measures the percentage of time the CPU spends executing an application's processes. This metric is valuable because it measures how efficiently an application uses processing power. Higher CPU usage can indicate computationally intensive operations, while lower CPU usage may suggest idle time or inefficient resource usage (Tanenbaum and Bos 2015, Chapter 4). However, CPU utilization alone does not always provide a direct measure of application performance, as it can be influenced by factors such as background processes and CPU scheduling policies.

Despite its usefulness, CPU utilization has several limitations. CPU utilization can be affected by background processes running in the operating system, making it difficult to isolate the impact of the application itself. This is especially relevant in embedded systems, where resource constraints require precise measurements (Jayaraj et al. 2006). Many embedded devices, employ dynamic CPU scaling techniques such as Intel's SpeedStep or ARM's DVFS (Dynamic Voltage and Frequency Scaling) to conserve energy. These mechanisms adjust the CPU frequency depending on load, which can distort direct CPU utilization measurements (Pillai and Shin 2001).

Despite these drawbacks, CPU utilization is a widely used performance metric due to its ease of implementation and broad applicability. For this study, CPU usage will serve as a primary metric for evaluating the impact of containerization on UI-applications.

2.3.2 Memory Usage

Memory usage is another important metric that tracks the amount of memory an application consumes during execution. Unlike CPU usage, memory consumption does not directly affect application speed, unless the device runs out of memory. When a system runs out of memory, it can trigger paging (if virtual memory is available) or cause application crashes due to memory allocation failures (Hennessy and Patterson 2011, Chapter 2). Tracking memory usage is particularly important in embedded systems, because they often have constrained memory resources compared to general-purpose computers. Efficient memory management is important to prevent memory leaks and excessive resource consumption (Wang 2017, Chapter 3.1.2). Containerized applications may consume additional memory due to runtime dependencies, container engines, and process isolation mechanisms (Merkel et al. 2014). Analyzing memory usage helps quantify the impact of these factors. Some UI frameworks, rely on caching techniques to improve performance. Monitoring memory usage can show inefficiencies in cache management or excessive memory allocation for rendering operations (Wang 2017, Chapter 3.1.2).

2.3.3 Latency

For user interface applications, responsiveness and rendering performance are important factors in perceived usability. One metric to measure this is latency. Latency refers to the time delay between user input, such as touch interaction or touch events, and the corresponding visual response in the application. Lower latency improves user experience, especially in responsive systems such as digital signage or control panels (Attig et al. 2017).

In containerized environments, latency can be affected by factors such as containerized execution overhead, the additional layer introduced by container platforms that can introduce slight delays in IPC mechanisms (Fu et al. 2022). The Chromium rendering pipeline, which uses the Skia graphics engine and hardware acceleration via ANGLE or Vulkan backends, can incur delays during rasterization, texture uploads, compositor frame scheduling, or GPU command submission.

2.3.4 Frames Per Second

Alongside latency, frames per second (FPS) measures rendering performance, which is a main factor in perceived usability. FPS measures how many frames per second the application renders. Higher FPS values generally indicate smoother UI performance. However, FPS is constrained by hardware limitations and rendering efficiency. The main factors affecting FPS include the efficiency of the graphics pipeline. Some containerized environments introduce GPU passthrough challenges that may impact rendering performance (Felter et al. 2015).

2.3.5 Synthetic Benchmarks

Benchmarks can be used to test specific aspects of application performance, helping to identify potential performance bottlenecks. Synthetic benchmarks are designed in particular to isolate and evaluate the efficiency of specific subsystems, such as the JavaScript engine or the rendering pipeline. Although they may not fully represent real-world workloads, they provide repeatable and comparable metrics that are useful for performance analysis in different environments or configurations. (Hennessy and Patterson 2011, Chapter 1) (Joshi, Eeckhout, and John 2008)

2.4 Related Work

The performance implications of containerization have been investigated in several prior studies, although most have focused on server-side workloads, cloud computing contexts, or generic performance metrics. The research by Felter et al. 2015 provided an early benchmark comparison between containers and virtual machines, showing that containers generally have lower overhead due to their reduced abstraction layers. However, this study emphasized CPU-bound and I/O-bound tasks in server environments, offering limited insights into user interface application scenarios.

Several studies have explored containerized deployments for embedded and edge devices. Noronha et al. 2018 discussed the feasibility of running containerized applications on resource-constrained IoT devices. Their research demonstrated how lightweight container runtimes such as LXC and Docker can improve portability while maintaining low overhead. Similarly, Morabito 2017 examined the performance of containerized applications on ARM-based embedded platforms and highlighted the trade-offs between isolation, resource usage, and performance. More recently, Kadulla 2025 investigated the use of container-based virtualization in edge computing environments, emphasizing the benefits of container orchestration to enhance resource management and scalability.

Kämäräinen et al. 2015 investigated the performance of virtual machines versus containers in cloud gaming systems, which is a domain that shares some characteristics with UI rendering, such as the need for low latency and high frame rates. Their findings revealed that container-based virtualization (via Docker) significantly outperforms QEMU-based virtual machines, delivering gaming performance almost equivalent to native execution (QEMU Contributors 2025). However, they also emphasized the sensitivity of rendering pipelines to system-level abstractions and hardware access limitations, particularly with GPU passthrough.

Recent work by Fu et al. 2022 examined how system resource utilization affects UI responsiveness in Android applications, identifying CPU and memory contention as primary factors impacting latency. Although focused on mobile platforms, this study supports the hypothesis that overhead caused by containerization can degrade user perceived responsiveness in user interface applications.

Despite this growing amount of research, there remains a significant gap in the literature addressing the specific performance impact of containerization on UI applications, especially under embedded Linux systems. Most prior studies have either abstracted away the graphical layer or evaluated containerization in high-resource, cloud-based contexts. This thesis contributes to narrowing this gap by conducting empirical analysis of Chromium-based user interface workloads, measuring both system-level and application-level performance indicators in native and containerized environments.

3. RESEARCH METHODOLOGY

This research used a comparative performance analysis to evaluate the impact of containerization on user interface (UI) applications. The primary objective was to measure differences in performance metrics, such as CPU utilization, memory consumption, and frame rate, between containerized and native execution environments. The experimental setup was designed to reflect a realistic deployment scenario for devices, where resource constraints and responsive rendering requirements are critical.

The study focused on isolating the performance overhead introduced by containerization. The experimental design ensured that both configurations executed comparable application code with equivalent workloads, thereby attributing any performance differences directly to the effects of containerization.

3.1 Experimental Setup

The experimental platform utilized in this study was an ARM-based embedded Linux system. The system included a quad-core ARM Cortex-A processor, 4 GB of LPDDR4 memory, and an integrated GPU that supported hardware-accelerated graphics through OpenGL ES and Vulkan interfaces. Despite the device's potential support for hardware acceleration via GPU, it was not utilized for this study. Enabling hardware acceleration in containerized environments introduced additional challenges with driver compatibility, and thus a simpler approach with software rendering was chosen.

This hardware configuration was selected to emulate the computing environment of an elevator display panel, where limitations on processing power, thermal dissipation, and memory capacity are typical. Elevator display systems represent a class of embedded devices that must maintain high reliability and responsiveness despite operating with limited resources. They are tasked with rendering dynamic graphical content, such as floor indicators, animations, status messages, and advertising material, frequently under continuous operation and with limited software maintenance windows. Therefore, evaluating the performance characteristics of containerized UI applications in this context offered direct relevance for industrial deployment.

3.1.1 UI Application

The implementation used a Chromium browser. Chromium conveniently allowed for benchmarking through its integrated support for JavaScript and rendering performance instrumentation. Two execution environments were constructed: a native configuration, where the browser was compiled and run directly on the host operating system, and a containerized configuration, in which Chromium was executed within an OCI-compliant container using Podman as the container runtime.

3.1.2 System Setups

In the native setup, the host operating system was configured with a Wayland display server, Mesa3D rendering drivers, and a minimal service footprint (see Appendix B) to eliminate background interference. The containerized environment mirrored this configuration as closely as possible. Podman was selected for container orchestration due to its support for daemonless and rootless execution, features that enhanced compatibility with embedded deployment and improved system security. The container image was constructed using Alpine Linux to reduce memory and storage overhead and included only the necessary runtime dependencies for Chromium. In both configurations, Chromium was launched in kiosk mode with identical command-line arguments, including flags to disable hardware acceleration and to enforce the use of the Wayland protocol.

3.2 Data Collection Methods

To evaluate performance, two distinct categories of data were collected. The first included system-level resource utilization metrics, including CPU and memory usage, while the second focused on application-level performance metrics, particularly rendering throughput and JavaScript execution efficiency.

3.2.1 CPU & Memory Usage

System-level performance data were collected using established Linux command-line tools. The `top` utility was executed in batch mode with a one second sampling interval over a continuous five-minute execution period for each test. This configuration captured per-core and aggregate CPU utilization, distinguishing between user and system time, as well as idle states. Memory usage was also recorded using `top`, which provided measurements of total, used and available memory, along with memory consumption per process. Each execution scenario was repeated five times, and all resource usage data were recorded at fixed intervals to allow consistent sampling and temporal analysis. Data collection was done during 4K video content playback to simulate a high-demand multimedia workload and to ensure that resource measurements reflected real-world usage conditions.

Listing 3.1. *System resource usage snapshot from `top` command*

```
top - 15:31:55 up 2 days,  9:58,  1 user,  load average: 1,04, 0,24, 0,12
Tasks: 366 total,   2 running, 363 sleeping,   0 stopped,   1 zombie
%Cpu(s): 62,3 us, 19,2 sy,  0,0 ni, 17,8 id,  0,0 wa,  0,0 hi,  0,7 si,  0,0 st
MiB Mem :  24031,7 total,  11922,3 free,   5344,9 used,   6764,5 buff/cache
MiB Swap:   2048,0 total,   2048,0 free,      0,0 used.  17871,6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4193 leevi     20   0   32,9g 428572 349312 S 775,0   1,7  21:31.29 code
```

3.2.2 Benchmarks

To evaluate application-level performance, two standardized benchmarks from BrowserBench.org were used: JetStream 2 and MotionMark 1.3.1. JetStream 2 was designed to measure the execution speed and responsiveness of the JavaScript and WebAssembly engines. It included subtests covering object property manipulation, cryptographic computations, memory management, and language features common to modern web applications. The benchmark output a composite score that aggregated the performance of all subtests, allowing direct comparison across execution environments.

MotionMark 1.3.1 assessed graphics rendering performance through a series of animation workloads that simulated interactive user interface interactions. The benchmark measured average frames per second under varying graphical features, such as compositing effects, vector graphics transformations, and DOM manipulation. The benchmarks were executed in Chromium using flags (see Appendix A.3.4) to ensure consistent behavior, and each benchmark was run five times in each configuration to allow averaging and variance analysis.

3.3 Data Analysis Techniques

The analysis of the collected data followed a structured statistical and visual approach. Descriptive statistics were computed for all quantitative metrics, including mean, median, standard deviation, and interquartile range. These statistics helped to understand the distribution and variability of performance measurements in both execution configurations.

To determine whether the observed performance differences were statistically significant, paired two-tailed t-tests were performed with a confidence level of 95%. This inferential technique was chosen due to its suitability for comparing matched pairs of measurements under two related conditions. For benchmark results, the relative performance degradation due to containerization was calculated as a percentage of the native configuration score. This enabled estimation of the overhead caused by containerization for both JavaScript execution and rendering throughput.

Graphical representations of the data were created using the Python-based visualization library `matplotlib`. Time-series plots were used to show trends in CPU and memory usage during the execution period. Scatter plots were used to compare benchmark scores across test repetitions and to illustrate the spread of performance data. These visualizations provided an intuitive way to identify performance trends, outliers, and the degree of consistency between test runs.

4. RESULTS

This chapter presents the empirical results obtained from performance tests conducted in both native and containerized environments. Two principal sets of data were collected: system-level resource utilization, focusing on CPU and memory usage, and application-level benchmark scores, which reflect JavaScript execution and graphical rendering performance.

4.1 Resource Usage Results (top)

CPU and memory usage was measured using the `top` command utility in batch mode, with sampling intervals of one second over a continuous runtime of five minutes. CPU usage is reported as the average percentage of time that CPU cores were engaged in user and system processes related to the application, excluding idle time. Memory usage is reported in megabytes (MB) and represents the average resident memory utilized during each execution.

Table 4.1. *Native System Resource Usage Test Results*

Average CPU Usage%	Average Memory Usage (MB)
86.7	2365.3
86.7	2386.4
86.9	2363.8
86.5	2366.2
86.4	2375.4

Table 4.2. *Containerized System Resource Usage Test Results*

Average CPU Usage%	Average Memory Usage (MB)
86.6	3125.9
87.5	3120.7
87.3	3108.4
88.2	3128.0
87.5	3127.0

4.2 Benchmark Results

Application-level performance was evaluated using two benchmark suites: JetStream 2 and MotionMark 1.3.1. JetStream 2 evaluates the execution performance of JavaScript and WebAssembly through a series of subtests and returns a composite score. The resulting score is unitless and is intended only for comparative analysis, where higher values represent better performance. MotionMark 1.3.1 assesses rendering performance by measuring average frames per second under various animation workloads, again reporting a composite score. (BrowserBench.org 2025a; BrowserBench.org 2025b)

Table 4.3. *Native Application Performance Test Results*

JetStream 2	MotionMark 1.3.1
93.760	178.93
93.267	168.73
87.463	192.70
93.333	185.60
93.432	180.55

Table 4.4. *Containerized Application Performance Test Results*

JetStream 2	MotionMark 1.3.1
93.797	130.82
92.680	150.19
91.793	214.72
92.911	150.90
92.666	190.68

5. ANALYSIS

This chapter presents a detailed statistical and comparative analysis of the data collected during performance evaluation of the Chromium-based user interface (UI) application in native and containerized execution environments. The analysis investigates differences in CPU usage, memory consumption, and application-level performance metrics (JetStream 2 and MotionMark 1.3.1). The goal is to determine whether containerization introduces statistically significant performance overhead in an embedded Linux system representative of constrained deployment environments.

5.1 CPU Usage

The average CPU utilization values for five runs in the native setup ranged narrowly between 86.4% and 86.9%, yielding a mean of 86.6%. For the containerized setup, CPU utilization ranged from 86.6% to 88.2%, resulting in a slightly higher mean of 87.4%.

The mean difference in CPU usage between containerized and native environments is 0.8 percentage points, which translates to an increase of approximately 0.9% in the containerized environment. The standard deviation in CPU usage for the native setup was 0.17, whereas for the containerized setup it was 0.59, indicating a marginally higher variability in CPU demands when Chromium is executed inside a container.

A two-tailed paired t-test was performed on the CPU usage data, resulting in a p-value of 0.087 (α = 0.05). Thus, the null hypothesis cannot be rejected, indicating that the increase in CPU usage is not statistically significant. However, while not significant in the statistical sense, the increase in average utilization and its higher variance in the containerized setup may still be operationally relevant in real-time or thermally constrained embedded environments. The increased variability may point to sporadic scheduling latency or container-induced jitter, potentially linked to additional kernel context switches, namespace isolation, and interprocess communication overhead.

5.2 Memory Usage

Memory usage data showed a pronounced disparity between the two environments. The native application consumed an average of 2,371.4 MB, with individual runs ranging from

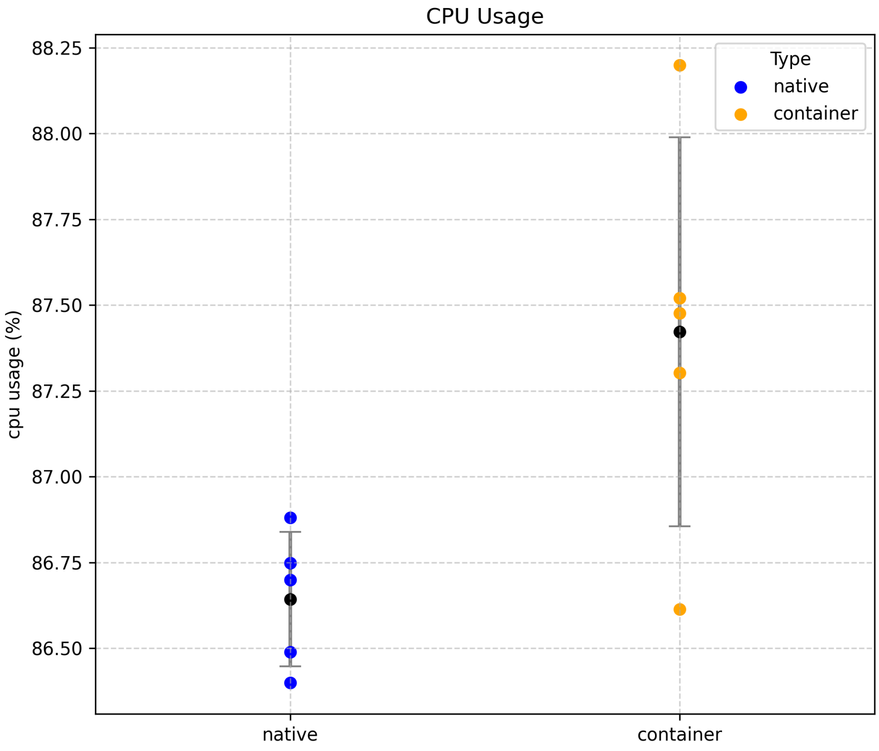

Figure 5.1. *Categorized scatter plot and standard deviation of CPU usage data. (Lower is better)*

2,363.8 MB to 2,386.4 MB. In contrast, the containerized version showed a significantly higher average memory usage of 3,122.8 MB, ranging from 3,108.4 MB to 3,128.0 MB.

This increase of 751.4 MB represents a 31.7% higher memory footprint in the containerized setup. The containerized environment showed slightly more stable memory usage (standard deviation: 8.03) compared to the native setup (standard deviation: 8.10), although both were relatively low, reflecting steady-state execution.

Statistical testing with the paired t-test yielded a p-value of < 0.001, strongly rejecting the null hypothesis and confirming that the memory overhead introduced by containerization is statistically significant. This can be attributed to several architectural factors. First, containerized applications must duplicate runtime dependencies within the container image, eliminating the opportunity for shared memory across processes that exist natively on the host. Second, memory usage by container management processes, such as Podman's OCI runtime and associated namespaces, adds to the system's overall footprint. Finally, the lack of access to certain host-level optimization mechanisms (e.g. shared libraries and graphics driver state) may increase memory duplication in rendering and process management. In memory-constrained devices, this overhead could lead to degraded system responsiveness, increased swap usage, or application instability under high concurrency.

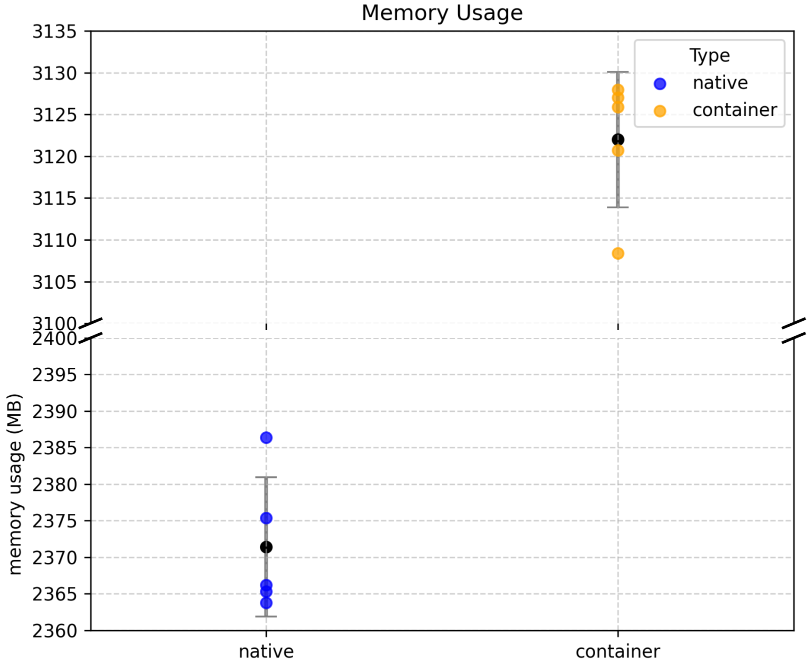

Figure 5.2. *Categorized scatter plot and standard deviation of memory usage data. (Range is cut for readability between 2400 and 3100) (Lower is better)*

5.3 Synthetic Benchmarks

5.3.1 JetStream 2

The native configuration achieved a mean JetStream 2 score of 92.251, with a standard deviation of 2.504. The containerized configuration recorded a slightly higher mean of 92.769, with a standard deviation of 0.813. The difference of 0.518 points corresponds to a relative increase of approximately 0.56% in the containerized setup.

Despite the higher mean, a paired t-test indicates that there is no statistically significant difference (p = 0.439). This result suggests that the JavaScript execution environment within Chromium remains largely unaffected by containerization when executed on the same physical hardware. Because JavaScript execution is CPU-bound and relies on just-in-time compilation within the V8 engine, this outcome confirms that containerization does not impede dynamic code generation or execution in this context.

5.3.2 MotionMark 1.3.1

The native environment achieved a mean score of 181.70, with results ranging from 168.73 to 192.70. The containerized setup yielded a mean of 167.46, with individual runs ranging from 130.82 to 214.72. The average difference observed is -14.24 points,

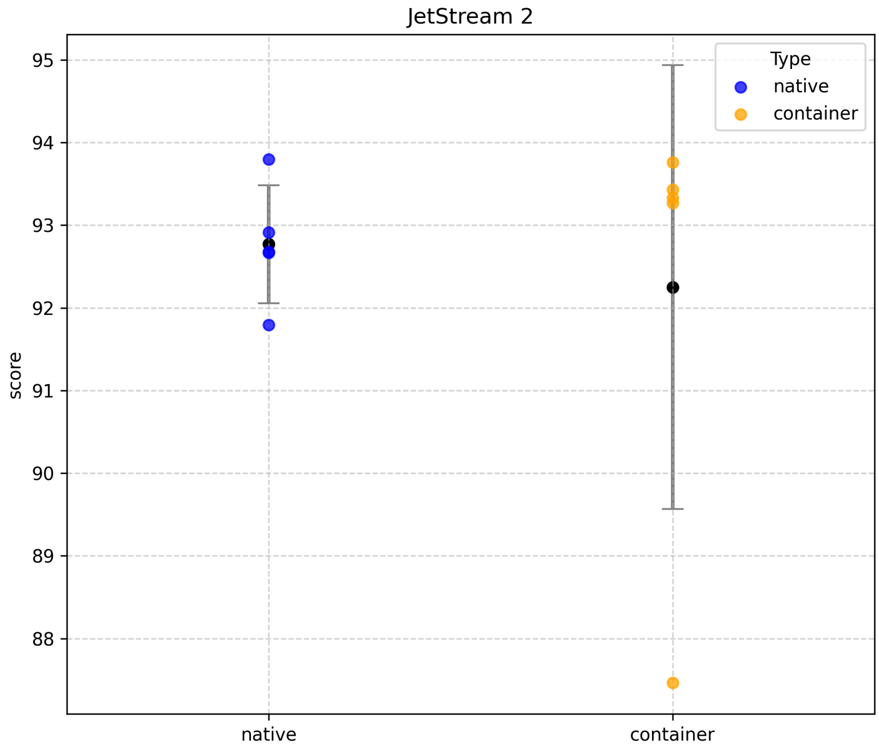

Figure 5.3. *Categorized scatter plot and standard deviation of JetStream 2 data. (Higher is better)*

equivalent to a performance reduction of 7.84% under containerization.

In particular, containerized scores showed higher variance (standard deviation: 29.12) compared to the native case (standard deviation: 8.89), suggesting reduced determinism in rendering performance. A paired t-test confirms that this reduction is statistically significant ($p = 0.038$), supporting the conclusion that containerization introduces measurable performance degradation for rendering tasks. Contributing factors may include additional memory pressure, container-induced scheduling jitter, and elevated interprocess communication latency. These elements collectively diminish the predictability of frame scheduling and increase the variability of animation workloads.

5.4 Variability and Outlier Analysis

Both CPU usage and memory usage did not have significant outliers with low standard deviations.

The MotionMark results in the containerized environment contain a notable outlier (214.72), significantly exceeding the native high (192.70). This may result from transient system states, such as momentary release of memory pressure or reduced thermal throttling. Despite this anomaly, the consistency and central tendency still demonstrate inferior performance under containerization.

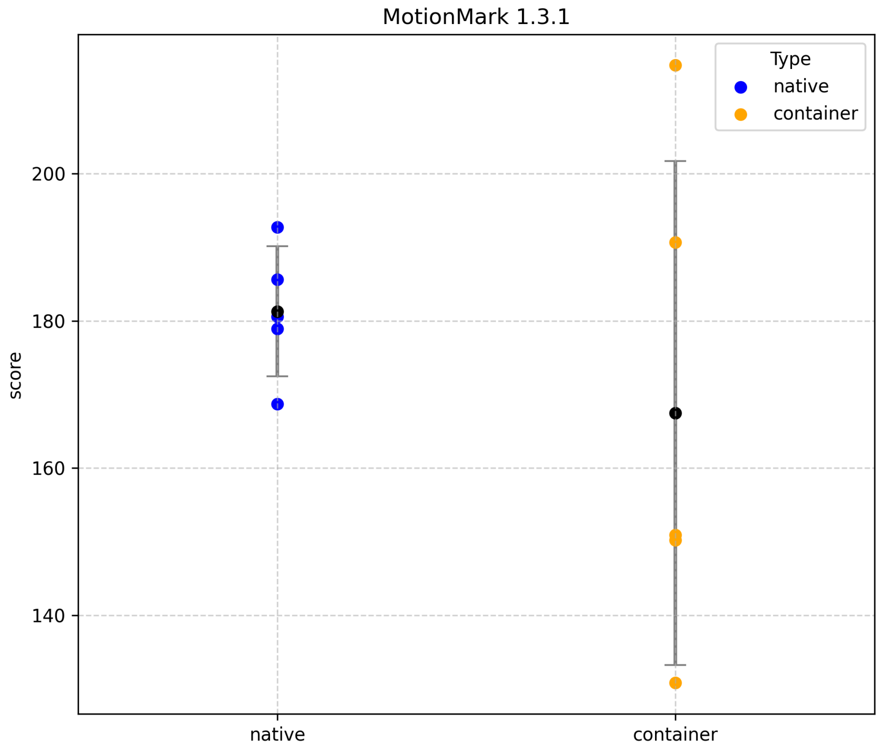

Figure 5.4. *Categorized scatter plot and standard deviation of MotionMark 1.3.1 data. (Higher is better)*

JetStream scores, in contrast, showed less inter-run variability and no discernible outliers, reinforcing its suitability as a stable metric for scripting engine evaluation.

5.5 Summary of Findings

Table 5.1 consolidates the observed performance differences and their implications:

Table 5.1. *Native System Resource Usage Test Results*

Metric	Native Avg	Cont. Avg	Rel. Change
CPU Usage (%)	86.6	87.4	+0.9%
Memory Usage (MB)	2,371.4	3,122.8	+31.7%
JetStream 2	92.57	92.25	-0.56%
MotionMark 1.3.1	181.70	167.46	–7.84%

These results show that while CPU performance is minimally affected, containerization introduces statistically significant memory overhead and a measurable reduction in rendering performance. The consistency of JavaScript engine performance across environments suggests that CPU-bound operations remain largely unaffected, whereas graphics-intensive tasks have more substantial losses.

6. DISCUSSION

6.1 Interpretation of Results

The empirical findings indicate that containerization has a measurable, although context dependent, impact on the performance of user interface applications. CPU utilization in containerized environments showed a slight increase (approximately 0.9%) compared to native execution, although this difference was not statistically significant. However, the observed increase in variability could have practical implications for interactive systems where consistent CPU behavior is critical.

Memory usage shows a more notable difference. The containerized environment consumed approximately 31.7% more memory than the native configuration. This increase was statistically significant and was primarily attributable to duplication of runtime dependencies, container engine overhead, and restricted access to host-level optimizations such as shared memory and library usage. These findings are consistent with prior literature that shows resource duplication and isolation costs as inherent limitations of container-based deployments (Felter et al. 2015).

In contrast, JavaScript execution performance measured by JetStream 2 showed almost no difference between the two environments. This suggests that CPU-bound operations executed within a just-in-time (JIT) compiled virtual machine environment are largely unaffected by containerization, provided that the host system resources are comparable.

However, the rendering performance measured by MotionMark 1.3.1 showed a statistically significant decrease in the containerized setup, with a reduction of 7.84% in the average score and greater variability between runs. This aligns with the hypothesis that containerization introduces latency and jitter in graphics intensive operations, potentially due to less efficient driver integration and increased context switching within the container runtime.

6.2 Practical Implications

The results of this study have practical implications for developers and system architects working with containerized UI applications, particularly in embedded or resource-constrained environments. The minimal impact on JavaScript execution performance implies that scripting-heavy UI applications may be viable candidates for containerization without a significant loss of computational efficiency.

However, a larger memory footprint and reduced rendering consistency might be a concern for systems with limited RAM. Applications that rely on high frame rates, such as digital signage, control panels, or infotainment systems, may suffer from perceptible reductions in responsiveness and visual fluidity when executed in containerized environments.

These considerations are particularly relevant for the deployment of Chromium-based applications, such as many modern applications like Visual Studio Code, Slack and Spotify. Also many web browsers are based on chromium such as Google Chrome, Microsoft Edge and Opera. Developers must weigh the trade-offs between the security, portability, and maintainability benefits of containers and the performance costs associated with graphics processing and memory consumption.

6.3 Limitations

Several limitations constrain the generalization of the results. First, the study was conducted on a single hardware configuration (ARM-based embedded platform), which limits the extrapolation to x86 or high performance desktop systems. Second, only one browser-based UI framework (Chromium) and a limited set of benchmarks were evaluated. Although representative of common deployment scenarios, these may not capture the full range of UI application behaviors.

Moreover, the benchmarks used, particularly MotionMark, simulate rendering workloads under controlled conditions, but may not fully reflect all relevant real-world UI interactions. The containerization approach was also limited to Podman, and the results may differ under alternative runtimes (e.g., Docker, containerd) or with varying container image compositions and runtime configurations (Felter et al. 2015).

6.4 Future Research

Future research should explore the generalization of containerization impacts across a wider range of GUI frameworks, hardware platforms, and container runtimes. While this study examined Chromium on an ARM-based system using Podman, repeating the analysis with different runtimes (e.g., Docker, containerd) and across additional platforms (e.g., x86-64, RISC-V) could reveal different performance results and container-host interactions. Also, evaluating other UI toolkits, such as Qt Quick, GTK, or Electron, may highlight framework-specific issues to constraints caused by containerization in rendering pipelines or memory management.

Perhaps the most interesting direction lies in investigating the role of GPU hardware acceleration in mitigating performance penalties observed in containerized environments. In this study, hardware acceleration was disabled to isolate container related overheads. However, enabling GPU acceleration using Vulkan, OpenGL ES, or vendor-specific APIs (e.g., VA-API, CUDA) could offload rasterization and compositing workloads from the CPU, improving rendering throughput and reducing input-to-display latency (Kämäräinen et al. 2015).

To achieve this, future work should examine the mechanisms by which containers access GPUs, such as through device passthrough, runtime hooks, or container-specific GPU plugins. Solutions such as NVIDIA Container Toolkit and Intel GVT-g enable direct hardware acceleration within isolated environments and have shown promise in accelerating both graphical and compute-bound workloads (Satzke et al. 2020).

However, GPU integration has challenges. Security implications, including kernel-level driver exposure and namespace isolation, must be evaluated to ensure that container boundaries are not compromised. Moreover, resource contention between containers that share a single GPU may require advanced scheduling and priority management. (Satzke et al. 2020)

7. CONCLUSION

This study evaluated the performance impact of containerization on a user interface application within an embedded Linux environment. By comparing native and containerized execution using CPU utilization, memory consumption, and standardized benchmark scores, it was possible to quantify both the overhead and variability introduced by containerization.

The findings indicate that containerization introduces negligible CPU overhead and does not affect execution performance. However, it significantly increases memory usage and decreases rendering throughput and consistency, particularly in graphics-intensive scenarios. These effects can be a practical concern in resource-constrained or interactive contexts where predictability and responsiveness are critical.

Finally, while containerization remains a viable deployment strategy for user interface applications, especially where maintainability and security are priorities, careful performance evaluation is essential before adopting containers in latency sensitive or memory limited systems. Future optimizations and refinements in container technology may further reduce these drawbacks, enabling greater adoption in embedded user interface domains.

REFERENCES

Attig, Christiane et al. (2017). “System latency guidelines then and now–is zero latency really considered necessary?” In: *Engineering Psychology and Cognitive Ergonomics: Cognition and Design: 14th International Conference, EPCE 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part II 14*. Springer, pp. 3–14.

Boettiger, Carl (2015). “An introduction to Docker for reproducible research”. In: *ACM SIGOPS Operating Systems Review* 49.1, pp. 71–79.

BrowserBench.org (2025a). *About MotionMark 1.3.1*. URL: https://browserbench.org/MotionMark1.3.1/about.html.

– (2025b). *JetStream 2.2 In-Depth Analysis*. URL: https://browserbench.org/JetStream/in-depth.html.

Chromium Development Team (2025). *Chromium*. URL: https://www.chromium.org/Home/.

Eder, Michael (2016). “Hypervisor-vs. container-based virtualization”. In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 1.

Felter, Wes et al. (2015). “An updated performance comparison of virtual machines and linux containers”. In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, pp. 171–172.

Fu, Jiaojiao et al. (2022). “How resource utilization influences UI responsiveness of Android software”. In: *Information and Software Technology* 141, p. 106728.

Hennessy, John L and David A Patterson (2011). *Computer architecture: a quantitative approach*. Elsevier.

Jayaraj, PB et al. (2006). “Effective Performance Analysis And Visualization On Embedded Linux”. In: *Software Engineering Applications*, pp. 27–29.

Joshi, Ajay, Lieven Eeckhout, and Lizy John (2008). “The return of synthetic benchmarks”. In: *2008 SPEC Benchmark Workshop*, pp. 1–11.

Kadulla, Sumanth (2025). “Edge Computing and Cloud Synergy: Optimizing Distributed Deployments”. In: *Journal of Computer Science and Technology Studies* 7.4, pp. 125–132.

Kämäräinen, Teemu et al. (2015). “Virtual machines vs. containers in cloud gaming systems”. In: *2015 International Workshop on network and systems support for games (NetGames)*. IEEE, pp. 1–6.

Merkel, Dirk et al. (2014). “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux j* 239.2, p. 2.

Morabito, Roberto (2017). “Virtualization on internet of things edge devices with container technologies: A performance evaluation”. In: *IEEE Access* 5, pp. 8835–8850.

Noronha, Vivian et al. (2018). “Performance evaluation of container based virtualization on embedded microprocessors”. In: *2018 30th International Teletraffic Congress (ITC 30)*. Vol. 1. IEEE, pp. 79–84.

Pahl, Claus (2015). “Containerization and the paas cloud”. In: *IEEE Cloud Computing* 2.3, pp. 24–31.

Peersman, Hans et al. (2011). *X Window system*.

Pillai, Padmanabhan and Kang G Shin (2001). “Real-time dynamic voltage scaling for low-power embedded operating systems”. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pp. 89–102.

QEMU Contributors (2025). *QEMU*. URL: https://www.qemu.org/.

Qt Company (2025). *Qt Company*. URL: https://www.qt.io/.

Satzke, Klaus et al. (2020). “Efficient gpu sharing for serverless workflows”. In: *Proceedings of the 1st workshop on high performance serverless computing*, pp. 17–24.

Shirazi, Jack (2003). *Java performance tuning*. " O’Reilly Media, Inc."

Skia Contributors (2025). *Skia*. URL: https://skia.org/.

Tanenbaum, Andrew S and Herbert Bos (2015). *Modern operating systems*. Pearson Education, Inc.

Wang, Jiacun (2017). *Real-time embedded systems*. John Wiley & Sons.

Wayland Development Team (2025). *Wayland*. URL: https://wayland.freedesktop.org/.

APPENDIX A: REPRODUCIBILITY PROTOCOL

This appendix documents the steps used to reproduce the performance evaluation in this thesis, including hardware, software environment, configuration, and command-line tools.

A.1 Hardware Environment

- **Processor:** ARM Cortex-A (quad-core)
- **Memory:** 4 GB LPDDR4
- **Graphics:** Integrated GPU with OpenGL ES/Vulkan

A.2 Software and System Configuration

A.2.1 Host Operating System

Ubuntu 22.04

A.3 Application Deployment

A.3.1 Chromium Setup

Chromium version 136.0.7103.97 was used throughout the evaluation. Chromium launch commands are given in the `Containerfile`.

A.3.2 Containerization

- **Container Runtime:** Podman
- **Base Image:** Alpine Linux

A.3.3 Container Build Command

```
podman build -t chromium-wayland -f Containerfile .
```

A.3.4 Containerfile Example

```
FROM alpine:latest

RUN apk add --no-cache \
    chromium \
    wayland \
    ttf-freefont \
    mesa-dri-gallium \
    mesa-gl \

RUN rm -rf /var/cache/apk/*

ENV WAYLAND_DISPLAY=wayland-0
ENV XDG_RUNTIME_DIR=/run/user/1000

CMD chromium \
    --disable-accelerated-video-decode \
    --disable-background-networking \
    --disable-battery-status \
    --disable-client-side-phishing-detection \
    --disable-component-update \
    --disable-crash-reporter \
    --disable-dev-shm-usage \
    --disable-device-discovery-notifications \
    --disable-dbus \
    --disable-features=ChromeWhatsNewUI \
    --disable-features=RendererCodeIntegrity \
    --disable-gpu \
    --disable-logging \
    --disable-pinch \
    --disable-power-saver-notifications \
    --disable-print-preview \
    --disable-restore-session-state \
    --disable-session-crashed-bubble \
    --disable-sync \
    --disable-translate \
    --enable-features=UseOzonePlatform \
    --fast \
    --fast-start \
```

```
    --mute-audio \
    --no-default-browser-check \
    --no-first-run \
    --no-referrers \
    --no-sandbox \
    --noerrdialogs \
    --overscroll-history-navigation=0 \
    --ozone-platform=wayland \
    --start-maximized \
    --user-data-dir=/tmp/chromium \
    https://browserbench.org/
```

A.3.5 Container Execution Command

```
podman run --rm -it \
    -v /run/user/$(id -u)/wayland-0:/run/user/1000/wayland-0 \
    -e WAYLAND_DISPLAY=wayland-0 \
    -e XDG_RUNTIME_DIR=/run/user/$(id -u) \
    chromium-wayland
```

A.4 Performance Data Collection

A.4.1 CPU and Memory Usage

Collected with `top`:

```
top -b -d 1 -n 300 > usage_log.txt
```

A.5 Workload Scenario

During all tests, a 4K video was played back using Chromium from a local URL to ensure consistent graphical workload.

A.5.1 Benchmark Execution

Benchmarks were run manually using Chromium:

- `https://browserbench.org/JetStream/`
- `https://browserbench.org/MotionMark1.3.1/`

Executed 5 times per benchmark, per configuration.

A.6 Statistical Analysis

- Visualizations: `matplotlib`
- Tests: Paired t-test using Python `scipy.stats`

APPENDIX B: SERVICES

List of background services listed by systemctl.

B.1 Command

```
systemctl list-units --type=service --state=running
```

B.2 Background Services

accounts-daemon.service	Accounts Service
avahi-daemon.service	Avahi mDNS/DNS-SD Stack
bluetooth.service	Bluetooth service
colord.service	Manage, Install and Generate Color Profiles
cron.service	Regular background program processing daemon
cups-browsed.service	Make remote CUPS printers available locally
cups.service	CUPS Scheduler
dbus.service	D-Bus System Message Bus
getty@tty1.service	Getty on tty1
glances.service	Glances
gnome-remote-desktop.service	GNOME Remote Desktop
kerneloops.service	Tool to automatically collect and submit kernel crash signatures
lightdm.service	Light Display Manager
ModemManager.service	Modem Manager
NetworkManager.service	Network Manager
polkit.service	Authorization Manager
power-profiles-daemon.service	Power Profiles daemon
qrtr-ns.service	QIPCRTR Name Service
rsyslog.service	System Logging Service
rtkit-daemon.service	RealtimeKit Scheduling Policy Service

serial-getty@ttyFIQ0.service	Serial Getty on ttyFIQ0
snapd.service	Snap Daemon
ssh.service	OpenBSD Secure Shell server
switcheroo-control.service	Switcheroo Control Proxy service
systemd-journald.service	Journal Service
systemd-logind.service	User Login Management
systemd-oomd.service	Userspace Out-Of-Memory (OOM) Killer
systemd-resolved.service	Network Name Resolution
systemd-timesyncd.service	Network Time Synchronization
systemd-udevd.service	Rule-based Manager for Device Events and Files
udisks2.service	Disk Manager
unattended-upgrades.service	Unattended Upgrades Shutdown
upower.service	Daemon for power management
user@1000.service	User Manager for UID 1000
wpa_supplicant.service	WPA supplicant
xrdp-sesman.service	xrdp session manager
xrdp.service	xrdp daemon