FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Adopting Containers in Microcontrollers for the IoT

Xavier Ruivo Pisco

Mestrado em Engenharia Informática e Computação

Supervisor: João Pedro Dias

Second Supervisor: André Restivo

July 27, 2023

Adopting Containers in Microcontrollers for the IoT

Xavier Ruivo Pisco

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

July 27, 2023

Abstract

Over the last few years, there has been an increase in the number and computing capabilities of Internet of Things (IoT) devices. At the same time, DevOps has been revolutionizing how software is created, mostly through the appearance and evolutions of the cloud, virtualization, and containerization.

More capable microcontrollers have allowed containerization to be used in these pieces of equipment with the surge of container platforms like Toit [40] and Wasmico [33]. Full-fledged computers use containerization for various reasons, including easier portability between different hardware and isolation of tasks running in the same device; the same reasons can also be proven helpful in the IoT context. Containerization tools for microcontrollers are still fairly new and lack important features so that they can be used more frequently.

The tools that are normally used in computers have good-looking user interfaces that allow users to easily know which containers are running and to quickly manage them with options such as starting and stopping the containers. This is not a reality in containerization tools for microcontrollers. These tools generally lack an easy-to-use user interface that gives information about the running containers and helps manage them.

In devices capable of running Linux and Docker, some solutions allow the user to save the state of containers and continue them, either on the same device or on a different one [22]. With the emergence of containerization for lower-end microcontrollers, we want developers to be able to use this feature when working with these devices. Microcontrollers are used in many conditions, which will utilize containerization differently, creating the need for this feature, *e.g.*, orchestrating multiple microcontrollers while changing the devices responsible for each task or checkpointing tasks and resuming them in case of failure.

This thesis aims to adapt one of the current containerization tools used in low-end microcontrollers and improve it. We will change it by adding both features that we mentioned, a user interface that gives information about the containers running in each microcontroller while helping users manage them and allow them to pause and resume those containers on different devices with low latency while minimizing the side effects on the running system. The resulting system should give all the necessary information to users, be easy to use by developers, and enable the user to save the state of a running task to memory and continue the application in the same state using the saved memory.

To validate it, we will develop the described system and evaluate both parts separately. We will evaluate the time consumed when saving the task's state, moving it, and restarting a task from a saved state. With the results, we can determine if the system is quick and small enough to be used in real scenarios. Then, we will conduct a user experiment with possible users of the tool and evaluate it based on their appreciation of the tool.

Keywords: Microcontroller, IoT, Internet of Things, Container, Migration, State, User Interface

Resumo

Durante os últimos anos, tem existido um grande aumento no número e nas capacidades computacionais dos equipamentos da Internet of Things (IoT). Ao mesmo tempo, DevOps tem revolucionado a maneira como o software é desenvolvido, maioritariamente através do aparecimento e evolução da cloud, da virtualização e da containerização.

A evolução das capacidades dos microcontroladores tem permitido que a containerização comece a ser utilizada nestes equipamentos com o surgimento de aplicações como Toit [40] e Wasmico [33]. Containerização é utilizada em computadores por várias razões, incluindo a facilidade de migração de tarefas entre diferentes computadores e o isolamento de tarefas a correr no mesmo dispositivo, e estas mesmas razões podem ser aplicadas no contexto de IoT. As ferramentas de containerização para microcontroladores são relativemente novas e ainda não têm algumas funcionalidades importantes para que sejam usadas mais frequentemente.

As ferramentas de containerização normalmente são usadas em computadores e têm boas interfaces que facilitam ao utilizador o controlo sobre que containers estão a correr e rapidamente geri-los através de opções como começar e parar os containers. Isto ainda não existe nos microcontroladores pois estas ferramentas normalmente não têm uma interface fácil de usar que mostre informação sobre os containers que estão a correr e que ajude na gestão dos mesmos.

Em dispositivos capazes de correr Linux e Docker, já existem soluções que perimetem ao utilizador guardar os estado de containers e continuá-los no mesmo dispositivo ou num differente [22]. Com a chegada da containerização aos microcontroladores, nós queremos que os programadores consigam usar esta funcionalidade quando estão a trabalhar com estes dispositivos. Os microcontroladores são utilizados em muitas condições, que vão utilizar a containerização de modo difrente, criando a necessidade desta funcionalidade, *e.g.*, facilitar a orquestração de múltiplos microcontroladores ao alterar os dispositivos responsáveis por cada tarefa ou guardar o estado das tarefas e continuá-las nesse estado no caso de um falha.

Esta tese vai adaptar uma das ferramentas de containerização existentes para microcontroladores e melhorá-la. Vamos alterar o programa ao adicionar as duas funcionalidades que mencionámos, uma nova interface que mostre informação sobre os containers que estão a ser executados em cada microcontrolador enquanto ajuda os utilizadores a geri-los e permitir a pausa e resumo dos containers em diferentes dispositivos com uma baixa latência minimizando os efeitos secundários no sistema. O programa final deverá mostrar a informação necessária aos utilizadores, ser fácil de utilizar por programadores e permitir que o utilizador salve em memória o estado de uma tarefa que esteja a correr e continuar essa aplicação no mesmo estado usando a memória guardada.

Para validar a nossa tese, vamos implementar o sistema que foi descrito e avaliar cada uma das partes em separado. Vamos avaliar o tempo necessário para guardar o estado de uma tarefa, para movê-lo e para recomeçar a tarefa a partir do estado guardado. Com os resultados vamos poder determinar se o sistema é rápido e leve o suficiente para ser usado em cenários reais. Depois, vamos fazer experiências com possíveis utilizadores da ferramenta e avaliá-la com base na sua avaliação da ferramenta.

Palavras-chave: Microcontrolador, IoT, Internet of Things, Container, Migração, Estado, Interface

Acknowledgements

First of all, I would like to express my gratitude to my supervisors, João Pedro Dias and André Restivo, who not only taught me as well as guided me through the process of writing this dissertation, but also assisted me and motivated me to complete it.

During the course of these five years, it was a blessing to be able to spend the majority of the good as well as the difficult times with my FEUP friends, who assisted me through all of the various stages of getting this degree. My friends "da terrinha" were always there for me, and I will never forget how helpful they were.

In closing, I would like to express my gratitude to my family for their unwavering support throughout my life and throughout the pursuit of this degree, as well as for the fact that they were always supportive of the choices I have made throughout my life.

Xavier Pisco

"No matter how hard or how impossible it is,
never lose sight of your goal"

Monkey D. Luffy

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

API	Application Programming Interface
CLI	Command Line Interface
CRIU	Checkpoint/Restore In Userspace
FaaS	Function-as-a-Service
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
OS	Operating System
OTA	Over the Air
RAM	Random-access memory
ROM	Read-only memory
SLR	Systematic Literature Review
SSH	Secure Shell
SUS	System Usability Scale
TCP	Transmission Control Protocol
TUI	Terminal User Interface
UI	User Interface
VM	Virtual Machine
WUI	Web User Interface

Chapter 1

Introduction

In this chapter, the topics that are discussed in the document are introduced. The problem we tackled and the main reasons it is essential to achieve a solution are presented. Lastly, this chapter identifies the main goals of this document and its structure.

The Section 1.1, p. 1 gives the context of this dissertation. Section 1.2, p. 3 explains the problem explored in the rest of the document. Section 1.3, p. 3 contains the motivation for the developed work. In Section 1.4, p. 4, we will present the goals we want to achieve. Lastly, Section 1.5, p. 4 describes the structure of this document.

1.1 Context

The Internet of Things (IoT) has emerged as a paradigm-shifting technological innovation, revolutionizing our digital and physical interactions. With its ability to seamlessly connect devices, objects, and systems, IoT has the potential to revolutionize industries, increase productivity, and improve our daily lives. As a result, it has attracted the attention of researchers, technologists, and businesses, paving the way for a period of unprecedented connectivity and innovation.

Since the first time it was presented by Kevin Ashton in 1999 [21], the term IoT has had many definitions, and most of them refer to the Internet of Things as a global network composed of different devices that monitor and interact with the environment around and with the ability to self-organize and operate autonomously without the need for human intervention [20, 3, 12, 14].

In recent years, IoT has been used in a variety of fields, like home automation and Industry 4.0, and that's provided a steady growth that is predicted to continue in the future. In 2022, more than 13 billion IoT devices were connected; by the end of the decade, that number is projected to surpass 30 billion [37] as you can see in Figure 1.1, p. 2.

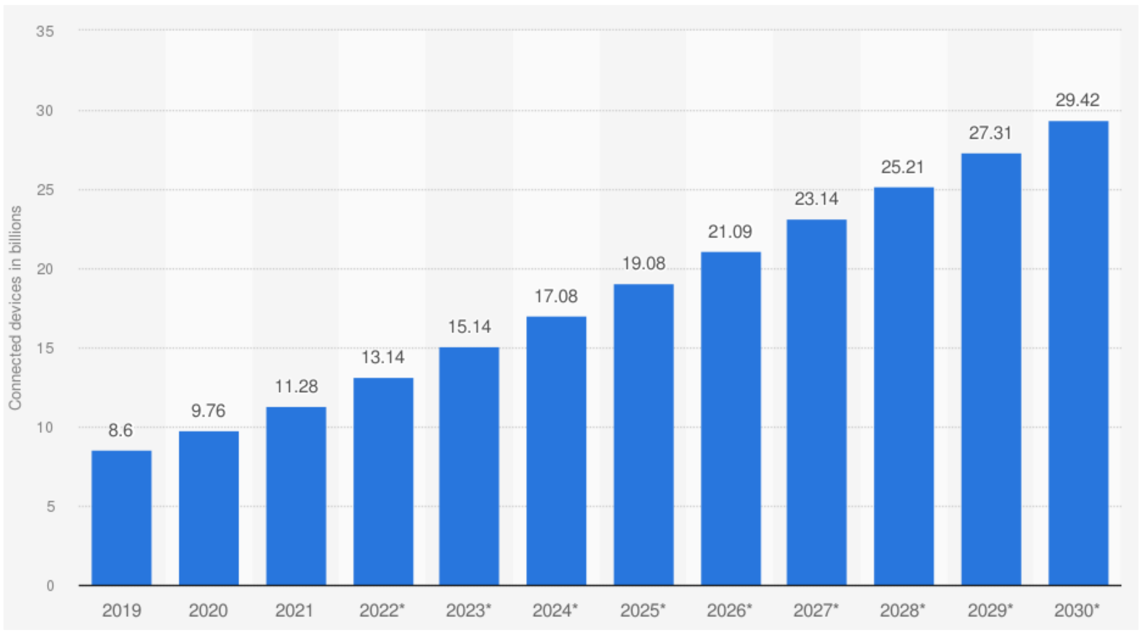


Figure 1.1: Evolution of the number of IoT devices and expected growth according to Statista [37].

This expansion has propelled IoT devices to unprecedented levels of power and capability. IoT devices have undergone substantial improvements in processing power, connectivity, and data-handling capabilities over time.

Alongside this evolution in microcontrollers, virtualization and containerization have undergone a remarkable evolution, revolutionizing the landscape of software deployment and resource management. Initially, virtualization enabled users to run multiple operating systems on a single physical server by creating virtual machines. This innovation improved hardware utilization and scalability, but it was resource-intensive. In response, containerization emerged as a lightweight alternative. Containers offer a more effective method, allowing applications and their dependencies to be packaged in a portable and isolated manner. Currently, containerization is utilized extensively in cloud computing, microservices architectures, and DevOps practices. It allows for rapidly deploying applications, scalability, and simplified management of complex distributed systems. Containerization improves portability, accelerates development cycles, and enables seamless deployment across different environments by encapsulating applications and their dependencies into self-contained units.

For many years, these shifts in the day-to-day practices of developers were not readily available for microcontrollers. With their limited resources and constrained environments, microcontrollers couldn't do virtualization and containerization. However, this scenario has changed with the recent introduction of tools and frameworks such as MicroPython [29] and Toit [40].

1.2 Problem

The usage of high-level programming languages and frameworks has been made possible thanks to virtualization-like abstractions on microcontrollers. By abstracting the hardware being used, the virtualization and containerization tools make programs more portable and reusable than classic solutions. This makes the work done while programming in microcontrollers much more straightforward and more appealing to developers, which in turn improves the cycle of development and operations and reduces the amount of time that developers need to spend programming, debugging, and porting their programs to multiple architectures.

Even though these tools have shown that they can be effective in making the process of developing software for microcontrollers easier, we believe they are still missing important capabilities that would make them better and facilitate the development process for the Internet of Things. Microcontrollers are used in high-mobility environments, which makes them battery dependent and prone to disconnections from other devices due to increases in their distance; thus, the ability to pause a job that is now being executed on one device and resume its execution later, either on the same device or on a different device, is one of the potential improvements that might be made [15].

There are solutions for this problem designed specifically for devices with higher processing power capable of running operating systems like Linux and state-of-the-art container platforms like Docker [16, 22]. However, microcontrollers are not powerful enough to execute these programs, and the available containerization options lack this and other capabilities.

Some solutions have more functionalities than others, but currently, no solution in the market includes all the capabilities that we consider that are fundamental for a containerization tool in microcontrollers. For instance, Toit [40] has a decent user interface but forces users to use Toitlang [41], whereas Wasmico [33] allows users to use more common programming languages but only offers a hard-to-use and mostly incomplete command line interface.

We will explore both of these problems with the containerization tools that are currently available in Chapter 4, p. 19, where we define what we think a good solution would be.

1.3 Motivation

Based on the problems that we mentioned before, allowing the migration of a container's state between multiple microcontrollers and the lack of a solution that combines the best functionalities of the existing containerization tools, we think that a solution for those problems would bring advantages to microcontrollers' developers, such as:

Improve the orchestration of multiple devices. If a developer can migrate tasks from one device to another, it will bring new possibilities to the orchestration of IoT networks. This solution would allow the developer to change the task that is being run to another device to free the initial microcontroller to execute different, and possibly more important, tasks;

Checkpoint the state of a task. By checkpointing the state of a task the user can backup a running task and resume it in the same state after a failure, *e.g.*, if a device runs out of battery, it will be possible to restore the task that was being executed and continue it. By saving the state in an external computer, the user can use the checkpoint as a way to pause the task, fully clean the microcontroller, and use it for other tasks that are more important.

Encompasses most of the benefits available in one tool. Users may be uncertain of which tools to choose from because of the different advantages and disadvantages of each tool, so we think it will be better to change one of the tools to guarantee the best possible one, especially in features that we consider important, such as the already mentioned usability of multiple programming languages, which would allow users to choose the language they prefer to solve for their specific problem, and a good user interface for managing the tasks, which would also facilitate the orchestration of microcontrollers.

To summarize, our main motivation is to improve app development and container management in microcontrollers through good containerization tools.

1.4 Goals

The main objective of this dissertation is to enhance one of the existing containerization options for microcontrollers and expand it through the development of new features.

After this dissertation, we want to have a tool that has the ability to quickly migrate containers from one device to another and save a state for future use, maximize the number of programming languages with which a developer can work with the tool, and have an easy-to-use user interface that allows users to manage and orchestrate the containers across multiple microcontrollers.

When the implementation is concluded, we will evaluate the potential benefits that these enhancements may bring to its users, both on the improvements in the time needed for some tasks execution and the usability of the tool interface.

To determine which containerization tool we will be working with, we will evaluate all of the available options and select the one that best meets our requirements. Then, we will proceed to work on the selected tool in order to make it a more effective tool that will be of assistance to developers in the future.

1.5 Document Structure

This document is structured into seven chapters that can be described as follows:

- Chapter 1 (p. 1), **Introduction**, introduces the problem that we will try to solve, why we'll be working on that problem, and what achievements we want to accomplish.

- Chapter 2 (p. 6), **Background**, presents some concepts that will be important to understand the whole document.

- Chapter 3 (p. 10), **State of the Art**, describes the solutions discovered through a Systematic Literature Review on the topics relevant to this dissertation, virtualization and containerization in microcontrollers and other IoT devices and the migration of containers in computers.
- Chapter 4 (p. 19), **Problem Statement**, presents and describes the problem that this dissertation intends to address, what our ideal solution is, how we're going to solve it, and how we'll validate that it works and is beneficial for developers.
- Chapter 5 (p. 24), **Solution**, describes how we implemented the new features on the existing tool and the limitations it has that this dissertation did not address.
- Chapter 6 (p. 42), **Evaluation and Validation**, presents the evaluations that we did in order to validate some of the research questions that were proposed.
- Chapter 7 (p. 53), **Conclusions**, contains a summary of the work that was done, a recap of the hypothesis that was presented, and some ideas of future work that we think is important to be done in this area.

Chapter 2

Background

In this chapter, we will introduce some concepts that will be utilized throughout the remainder of the document. In Section 2.1, we will define containerization, discuss its relationship to virtualization, and outline its benefits. Section 2.2, p. 7 provides a simple introduction to what a user interface is and the four types that will be discussed in this dissertation.

2.1 Containerization

Containerization is a widely-used modern software development and deployment approach that facilitates the packaging and execution of applications and their dependencies in a lightweight, isolated environment known as a container. Each container encapsulates an application, its libraries, and any required dependencies, providing a portable and consistent runtime environment. Containers offer a greater level of abstraction than traditional virtualization, allowing for efficient system resource utilization and simplified application management [1].

This method originates from virtualization. In virtualization, virtual machines (VM) simulate physical machines without needing distinct hardware; thus, a single computer can simultaneously operate multiple virtual machines. The virtual machines operate on top of a hypervisor, which administers them, the computer's resources, and their allocation to the respective VMs, and that can operate natively on the hardware or on top of an operating system. By using virtualization, people are able to use their computers and servers more efficiently by utilizing the same hardware for multiple purposes. Additionally, it is simpler to deploy and interrupt machines on demand, as well as to update them for security or functionality purposes [2].

Containers are defined by container images, which comprise the application's code, dependencies, and configurations. The container will subsequently be executed on top of an engine such as Docker [16] or Kubernetes [25]. These engines construct containers based on the specifications from the container's files and act as an intermediary between the application and the host operating system to ensure that the application is able to run on any operating system supported by the engine.

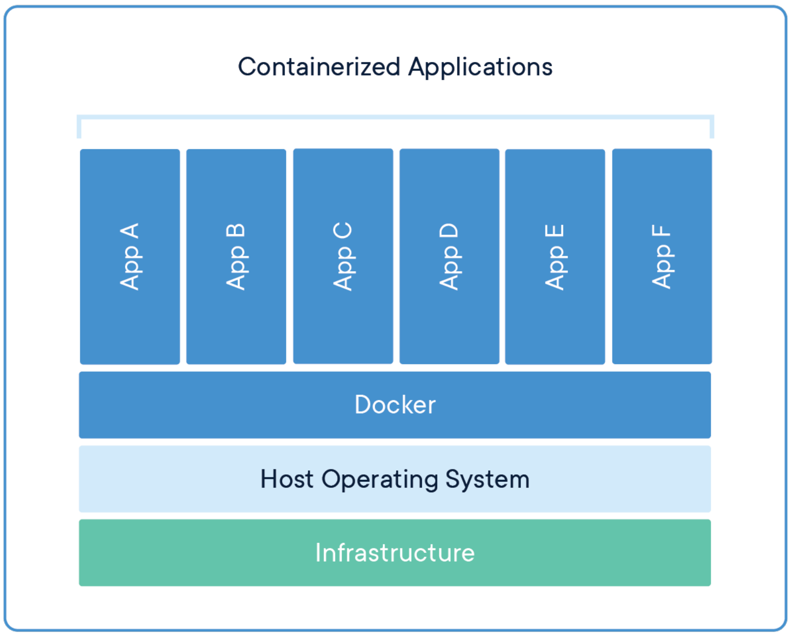


Figure 2.1: Example of containerization architecture using Docker. The containers (Apps) are run on top of Docker, which ensures isolation from each other and the hosting OS [16].

Containerization has resulted in a number of advancements in software development and deployment processes. A software developer is able to create their program on their own personal computer and then use it on any other computer, regardless of the machine's architecture, operating system, or hardware. Since a container is more compact and utilizes its resources more effectively than a virtual machine, a greater number of containers may be operated concurrently on the same physical computer. While running concurrently, containers are separated from one another and from the host system to ensure isolation between all the tasks. This means that they are unable to access contents that are not related to their own application, and in the event that a problem arises with one of the containers, it will not have an impact on the operation of the other containers.

2.2 User Interfaces

According to Debbie Stone *et al.* [38], user interfaces serve as the interface between humans and computers, allowing for efficient human-computer interaction. They offer users a visual representation of the software or application, shielding them from the underlying complexities and presenting only the essential information and actions.

A good user interface is designed to encourage users to execute their desired duties without difficulty, providing a seamless and intuitive experience. It emphasizes plain and concise communication to guide users through the interface and simplify complex actions. and prioritizes usability and user-centric design principles to improve efficiency and user satisfaction. A poor user interface, on the other hand, introduces frustration and complicates users' goals. It may have cluttered layouts, confusing navigation, or non-intuitive controls, impeding the ability of users to complete tasks efficiently and creating unneeded obstacles, hindering productivity, and can result in higher user dissatisfaction.

For this dissertation, we will need to explain and discuss the advantages and disadvantages of four different types of user interfaces: (1) command line interface, (2) terminal user interface, (3) graphical user interface, and (4) web user interface.

All these user interfaces are widely used in the context of IoT systems programming and orchestration, being some of the most widely used ones PlatformIO (CLI), and Node-RED (Web UI) [42, 36, 14].

2.2.1 Command Line Inteface

The Command line interface (CLI) is a text-based interface that enables users to interact with a computer system or program by typing commands. Users input specific commands into a CLI, and the system responds accordingly [18]. These interfaces have their own set of benefits and drawbacks.

One of the major advantages of CLIs is their resource efficiency. Since this type of interface typically only works with text and does not maintain state across multiple usages, it executes input commands quickly and consumes significantly fewer resources than the alternatives we're discussing. In automation scripts, CLIs are frequently combined with commands from other tools to automate complex duties with a single command. Due to their compatibility with low-bandwidth connections, CLIs are widely utilized for remote server and computer access.

Conversely, CLIs can be intimidating and difficult to use for users unfamiliar with terminals. They can have a steep learning curve, requiring users to memorize commands and comprehend their syntax, and without external resources or documentation, commands and options may not be easily identifiable to newcomers. Furthermore, CLIs may lack visual feedback and interactivity, making complex tasks involving multimedia or graphical content potentially more cumbersome.

2.2.2 Terminal User Inteface

Terminal user interfaces (TUIs) are text-based interfaces that run on a terminal by providing a visual representation of what is happening and allow the user to interact with it both through mouse and keyboard inputs [44].

TUIs are well-known for their adaptability, as they can be used in a variety of contexts, including terminals and remote sessions, which makes them useful for system administration, remote access, and software development, as they provide a consistent and familiar interface across platforms and environments. Terminal user interfaces typically demand fewer resources than graphical user interfaces, making them faster.

However, TUIs offer a limited visual representation compared to other options, particularly because the interface must be created using text. TUI-based applications are typically less appealing to users, particularly non-technical users who may not be acquainted with the terminal and will therefore have a steeper learning curve.

2.2.3 Graphical User Inteface

Graphical user interfaces (GUIs) are visual interfaces that enable users to interact with computer systems or software via graphical elements such as windows, icons, menus, and buttons, where users can move the mouse to select an object and click or drag it to interact with it. Over the years, GUIs have become more common than TUIs and are currently one of the most prevalent interface paradigms [27].

An important advantage of graphical user interfaces is their intuitive and user-friendly nature. With graphical representations of elements and visual signals, GUIs facilitate user comprehension and interface navigation. In addition, GUIs are user-friendly through menus, tooltips, and context-sensitive help and can manage complex visualizations and multimedia content.

Nevertheless, GUIs are typically resource-intensive applications that operate more slowly than their terminal-based counterparts, and they cannot be accessed via terminals, which reduces their versatility, particularly in servers and remote sessions.

2.2.4 Web User Inteface

Web user interfaces (WUIs) are comparable to graphical user interfaces (GUIs), but instead of running directly on the operating system, they operate atop web browsers. Their functionalities and use cases are nearly identical.

The primary advantage of WUIs over GUIs is their ability to operate on any operating system or browser with minimal modification.

2.3 Summary

In this chapter, we introduced different topics that are necessary for a full understanding of this dissertation and all the topics that we will explore in the rest of the document.

In Section 2.1, p. 6, we explain the concept of containerization. We dive into the concept of containerization and its capabilities, what virtualization is and how it affects and compares to containerization, and how containers are used in modern computing and their benefits.

Then, in Section 2.2, p. 7, we explain and compare four different types of user interfaces, command line interface, terminal user interface, graphical user interface, and web user interface. These user interfaces are commonly used today, so we discussed the advantages and disadvantages of each one.

Chapter 3

State of the Art

This chapter discusses the current state of container migration, the Internet of Things, and microcontrollers as they relate to this dissertation. Firstly, we will clarify the methodology we used to comprehend the current state of these topics in Section 3.1. Then, we will present our findings regarding virtualization and containerization in microcontrollers and container migration in the Internet of Things in Section 3.2, p. 13 and in Section 3.3, p. 16, respectively. Lastly, in Section 3.4, p. 18, we will present our conclusions regarding the state of the art based on our findings.

3.1 Methodology

We conducted a *Systematic Literature Review* (SLR) to comprehend better the current publications and initiatives related to the already discussed topic. An SLR is a study of the existing literature that strives to be as thorough as possible while minimizing our bias, it is described as a set of procedures that should be taken to obtain the most comprehensive understanding of the state of the art [23, 24]. Firstly, we must identify the topics we want to explore by defining the main research questions. Next, we must decide which databases should be used to search. Lastly, we must develop inclusion and exclusion criteria that will be applied to the retrieved results. In addition to the SLR, we made the decision to broaden the search to find various results that were not present in the selected databases.

3.1.1 Research Questions

To research the state of the art related to this thesis, we decided to separate the problems described in Section 1.2, p. 3 into two different parts, the first related to virtualization and containerization, and the second associated with the migration of tasks.

3.1.1.1 Virtualization and Containerization in Microcontrollers

What are the existing solutions for virtualization and containerization in microcontrollers?

This question was used to know what solutions are currently available that allow developers to program using virtual machines or containers in microcontrollers. This allows us to get an overview of the recent developments in this area and explore the solutions to find the most suitable one to be enhanced in this thesis.

In the queried databases, we executed this search using the following query:

(virtuali OR "virtual machine" OR container*) AND (iot OR "Internet of Things" OR microcontroller OR micro-controller)*

3.1.1.2 Container Migration in IoT

What are the existing solutions for migrating containers between IoT devices?

The second question is used to understand how container migration is currently done in IoT devices. With this question, we can understand the solutions found to migrate containers not only in microcontrollers but also in other devices used in IoT, *e.g.*, Raspberry PIs. Even if these solutions use more advanced tools than the ones found in the previous question, they will give us ideas and a direction toward the development we'll do.

The query that was used to find results was:

(migrat OR checkpoint*) AND (iot OR "Internet of Things" OR microcontroller OR "micro-controller")*

3.1.2 Databases

We selected the following databases for the Systematic Literature Review: ACM Digital Library [26] and IEEEXplore [45]. Both databases are renowned for their reliability and extensive collections of diverse papers in this dissertation's field. In addition to their databases, we used their search systems that provided us with the results from the mentioned queries and the information needed to apply the inclusion and exclusion criteria. We employed their search systems in addition to their databases, which gave us the results from the aforementioned searches as well as the information we needed to apply the inclusion and exclusion criteria. Apart from these two databases, we also used the Google platform to execute a broader search and identify other solutions that weren't found in the previous databases.

3.1.3 Inclusion and Exclusion Criteria

After collecting all the results, we had to sort through them to determine which were essential for our Systematic Literature Review and which weren't, so we used the inclusion and exclusion criteria detailed in Table 3.1 to filter the results.

I/E	ID	Description
Inclusion	IC1	Solution is related to virtualization, containerization, or execution environment
	IC2	Solution is applied to microcontrollers
	IC3	Solution allows updates over the air
	IC4	Solution allows migration of a container's state between devices
Exclusion	EC1	The work is duplicated
	EC2	Work is not in English
	EC3	Solution uses Linux or another non-RTOS
	EC4	Solution is not a migration tool

Table 3.1: Inclusion and exclusion criteria that were used to filter the results obtained after doing the research questions on the databases.

IC1 and IC2 are the two inclusion criteria defining this work's domain, we want to work in microcontrollers with architecture-agnostic tools, such as virtual machines, containers, or execution environments. IC3 and IC4 are the desired features that we want to explore, we want our work to have updates OTA and to allow migration of the state of the tasks between different devices, preferably also over the air.

EC1 and EC2 are two exclusion criteria that serve as a deduplication of work and as a focus on the solutions we are able to read. EC3 refers to all the found solutions that would work on devices with higher capabilities that can run normal operating systems, such as Raspberry PIs, which makes them unusable in microcontrollers. Lastly, EC4 is meant to exclude all the found papers about migration that focus on the orchestration of the migration and not on the specifics of how to migrate a task.

For each of the proposed research questions in Section 3.1.1, p. 11, we used multiple of the shown criteria in a series of steps to filter the results we achieved from the mentioned databases.

We executed the following filtering steps after searching for virtualization and containerization in microcontrollers:

1. **Filtering #1**: Remove all the duplicates and non-English results (EC1 and EC2) based on the papers' title.

2. **Filtering #2**: Select the results that were specific to the domain we want (IC1 and IC2) and remove the ones that used non-RTOS operating systems (EC3). For this stage, we read through the abstracts of the remaining results and filtered them.

3. **Sorting by Relevance**: By applying the IC3 and IC4 criteria, we defined which results were more important based on their title, keywords, and abstract.

4. **Final Filtering**: We read carefully through the remaining solutions and applied the used criteria again in case some of the results seemed relevant but were not.

For the results from the second research question, we applied the following criteria in a similar process:

1. **Filtering #1**: Remove all the duplicates and non-English results (EC1 and EC2) based on the papers' title.
2. **Filtering #2**: We tried to apply the same criteria as in the previous question, but after applying the IC2 and EC3, we were left with no results. This was explained by the fact that the found papers used Raspberry PIs or similar IoT devices that aren't considered microcontrollers. So, in this phase, we only applied IC1 to the papers based on their abstract.
3. **Filtering #3**: In this question, both IC4 and EC4 were major criteria, and, as such, they were enforced based on the papers' titles, keywords, and abstracts.
4. **Final Filtering**: After reading the whole solutions, we did a final filtering based on the used criteria.

Through the application of these two processes, we ended our search with some of the papers that will be detailed in the rest of this chapter.

3.1.4 Broad Search

After filtering all the results found in ACM Digital Library and IEEEXplore we broadened our search by doing manual searches on Google. We used keywords based on the presented search queries and added the results we got to the ones we already had.

3.2 Virtualization and Containerization for Microcontrollers

Over the last few years, there has been a development of new technologies which allow for the use of virtualization and containerization in microcontrollers. These technologies have been able to help developers with advantages, such as the possibility to use high-level languages in these devices, better isolation of tasks that increase fault tolerance and security of the whole system, and better code portability between devices with different architectures.

The solutions we found about this topic are described in this list.

Micropython [29] is described as a Python compiler and runtime that enables the use of Python in microcontrollers. Microptyhon compiles Python code into bytecode that is then executed on a Virtual Machine.

This program was developed to allow developers to create code meant to run on microcontrollers without the need to program in a low-level language. Micropython has some differences from the more commonly used CPython in order to be usable in microcontrollers, but it allows Python to be used on a variety of architectures and different devices.

This solution brings the advantages of Python to IoT microcontrollers, such as dynamic typing and high-level language readability. Still, it also comes with the disadvantages, such as a slower speed than low-level languages like C.

Wasm3 [43] is a similar application to Micropython that enables programmers to write code in high-level languages and execute it on microcontrollers. Wasm3 is a WebAssembly interpreter and runtime used in microcontrollers to execute code written in different programming languages.

To use it, developers must compile their code into WebAssembly, and the Wasm runtime will then execute the application on the device. This tool supports multiple architectures, allowing developers to separate their code from vendor-specific architectures.

When compared to the previously mentioned Micropython, Wasm3 has lower hardware requirements, executes more quickly, and consumes less memory.

Toit [40] is a product for managing and deploying containers in ESP32 microcontrollers. This program allows users to deploy and manage containers over the air by creating a web server on the microcontroller that exposes an API that can be accessed via HTTP requests. It allows users to deploy their containers without needing to connect via USB to the microcontroller or restart the device and monitor those containers for quicker and easier maintenance.

To avoid the need for low-level languages for programming for microcontrollers, Toit developers created their language, a language designed specifically for programming microcontrollers in IoT [41].

One of the main goals of this application is to be as secure as possible, so the developers' focus is on ensuring that even if the uploaded code has bugs or is exploitable, the device running it should still be able to function and continue to receive other commands via the mentioned API.

This solution has some overhead expected to exist compared to running code in bare metal. However, compared to similar solutions, such as Wasmico, it is still slower and uses more memory to execute similar tasks.

Golioth [19] is a tool similar to Toit, defined as an IoT development platform built for scale. This tool consists of a program running on the microcontroller and creates a Web server to deploy and manage containers into the device.

Golioth is compatible with over 100 boards and includes ZephyrOS and FreeRTOS, meaning users can choose their operating system. Communication with the microcontroller can be done via their web console, REST APIs, or a CLI. All the telemetry and logs can be stored in multiple databases, allowing for easier debugging in case of errors.

While their program allows the user to run their code in a container, it focuses on updating the code OTA and doesn't allow running multiple containers simultaneously. This tool is mostly designed to be used in DevOps as a simple way to deploy different applications to multiple microcontrollers.

Femto-Containers [47] is a system that produces multiple concurrent VMs using the multi-threading capabilities of the RIOT OS [34] which are created in an event-based model that responds to events like internet connections, sensor readings, or scheduled events. This architecture allowed the authors to create a hardware-agnostic tool that can be run on various microcontrollers with differing capabilities and architectures and eases the developers' work by easing code portability between those devices.

Besides allowing developers to run multiple applications concurrently, the authors wanted to minimize the overhead created by their program when compared to running the applications on bare metal. They compared the ROM and RAM usage, and the cold start and run times between WASM3 [43], rBPF [46], Micropython [29], and RIOTjs, and decided to proceed with the rBPF based virtual machines, which demonstrated negligible memory overhead. To prevent malicious code from accessing memory areas that are owned by other applications, those virtual machines are isolated from one another.

Wasmico [33] is a containerization tool that allows the development of applications for microcontrollers in the most popular programming languages. In their paper, Ribeiro *et al.* explain how they developed a program that enables users to deploy and manage containers in microcontrollers remotely by allowing users to upload, edit, start, stop, delete, pause, and unpause containers over the air. All these functionalities are available via a command line interface that sends information via HTTP to the microcontroller to request the execution of one of the operations.

On the microcontroller, the tool consists of a running web server that handles the mentioned HTTP connections and the FreeRTOS operating system [17], which has a scheduler for the parallel execution of containers and controls the memory allocations for those tasks. The FreeRTOS is part of the Espressif IoT Development Framework, which was used as the development framework for the ESP32 devices. Each container is a separate task that the FreeRTOS control, which runs on top of the WebAssembly interpreter, Wasm3. All the information that needs to be saved in non-volatile memory is stored in the SPIFFS filesystem, which allows for information to be consistent even after a device reboot. This architecture can be seen in Figure 3.1, p. 16, which is included in the Wasmico dissertation.

With this program, we can circumvent some of the disadvantages usually associated with microcontrollers, such as allowing the deployment and management of tasks OTA, which avoids the typical USB connection and reboot needed to update the code on the device.

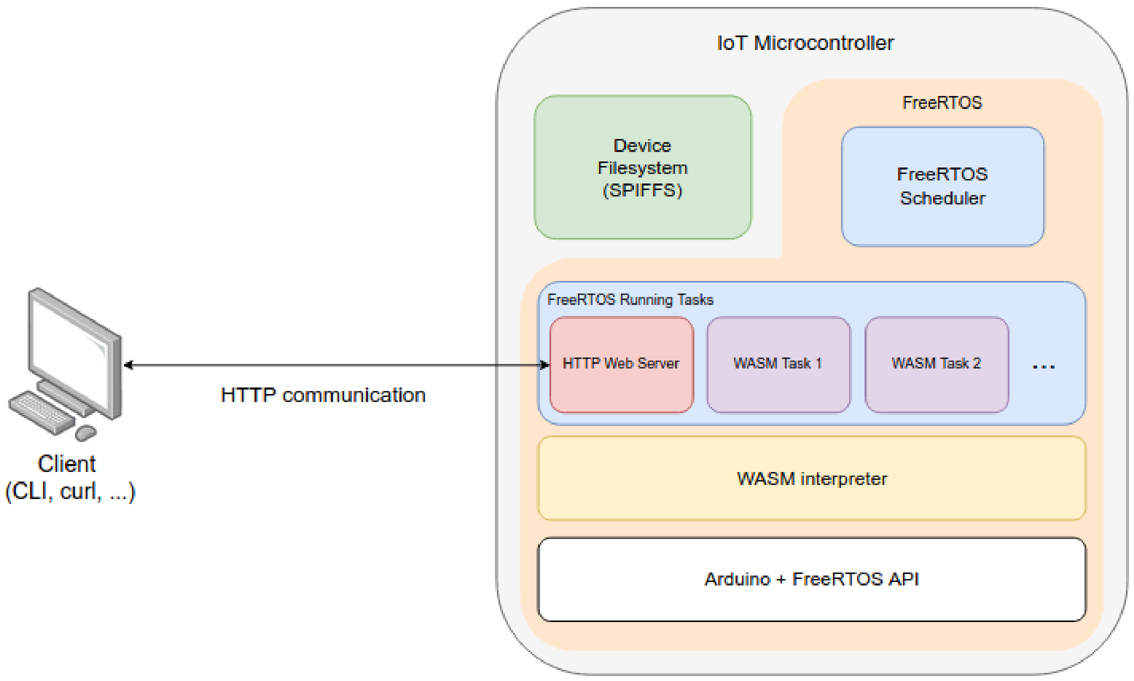


Figure 3.1: Wasmico Architecture Overview. [33]

Another advantage is the possibility of developing code in high-level languages familiar to developers instead of low-level languages, such as C or Assembly.

On the other hand, by using Wasmico, we are introducing some expected overhead compared to native code execution. Still, it's significantly less than the overhead introduced by other tools such as Micropython.

3.3 Container Migration in IoT

In more powerful computers used in IoT, such as the Raspberry Pi, there has been a development that led to the possibility of creating checkpoints of tasks by saving their state and migrating it to be used in other microcontrollers. It is now possible to save the running state of a program and continue when it's needed, where it's needed.

These are the publications with different solutions and different goals that solved this problem:

Checkpointing and Migration of IoT Edge Functions [22] describes how to create checkpoints in Docker containers that are running on a Raspberry PI device. This paper's primary goal is to enable the function-as-a-service (FasS) model that is widely used today in regular computers to be used in edge devices, in this case, in Raspberry PIs.

Karhula *et al.* created a solution using Linux's CRIU functionality on top of Docker to enable developers to checkpoint their programs and save the state of the containers. With their tool, they managed to pause tasks that were executing functions for a long time and functions with established TCP connections and continue both in their expected state.

In this paper, they also managed to migrate the containers after checkpointing to another device and continue the tasks in their previous state. The paper's results show that this feature is achieved with a low memory footprint that allows many containers to be on the same device.

Efficient service handoff across edge servers via docker container migration [28] done by Ma *et al.* show in detail how to migrate docker containers from one edge server to another. This paper's authors explain in depth how docker works and define the nine steps needed to complete the migration of a container from one device to another. These are the steps:

1. **Synchronize Base Image Layers.** Ensure that the Docker image layers are equal in the destination and the source servers.
2. **Pre-dump Container.** Dump a snapshot of the container runtime memory.
3. **Migration Request Received on Source Server.** A request is sent to the source server to start the migration of the task
4. **Checkpoint and Stop Container.** After receiving the request, the server will checkpoint the container and stop it.
5. **Container Layer Synchronization.** Send the runtime and configuration files from the container to the destination server.
6. **Docker Daemon Reload.** Reload Docker in the destination source to recognize the new container.
7. **Get, Send, and Apply Memory Difference.** Compare the new memory dump with the memory dump from step 2 and send the memory differences to the destination server.
8. **Restore Container.** Restore the container in the destination server and ensure it's running as expected.
9. **Clean Up Source Node.** Remove the container and all the associated information from the source server.

This solution focuses on using the minimal needed information to migrate the containers by creating small files that can be transferred quicker between the devices.

Checkpoint/Restore In Userspace (CRIU) [9] is a Linux software that can do many tasks on containers, including migration. It is designed to work with many different types of containers, including Docker, LXC, Podman, and many others. CRIU allows users to migrate containers between two devices, speed-up long boots by saving the state after boot, take snapshots of applications, etc. [10]

In Linux, everything is a file, and based on that, CRIU uses the files created by running containers and saves them; this includes the running task, file descriptors, network connections, and other task information. This idea is not usable on microcontrollers since most of the information is stored on RAM instead of files.

3.4 Conclusions

After doing the *Systematic Literature Review* on the topics that we will work on, we found that there are multiple solutions for microcontrollers that give developers the freedom of choice when choosing the programming languages to work on, even languages that are less common on microcontrollers development, like Python with Micropython [29] or Rust and Golang with Wasm3 [43]. Recently there were developments with containers in microcontrollers with the appearance of tools like Toit [40] and Golioth [19]. Wasmico [33] is one of those recent tools that allows both the choice of language and the containerization of tasks at the same time.

On the topic of container migration, we found different solutions that were capable of doing it, but none of the solutions was viable for use in microcontrollers. Both CRIU [9] and the Karhula *et al.* solution [22] use the Linux filesystem to save the state of the running containers, and the Ma *et al.* tool [28] is done specifically for Docker containers, which are not available for microcontrollers. Since all three solutions use software that cannot be used in microcontrollers, we can only use them as proven ideas for what we want to do in our solution.

Finally, none of the found solutions can use both features that we discussed, so if we are able to change one of the available containerization tools and add the option to migrate its containers between devices, we will create a solution in a new area that has no tools yet.

Chapter 4

Problem Statement

In this chapter, we will describe in detail the problem we intend to address in this dissertation, as well as the strategy we will employ to do so. Firstly, we will outline the prevalent problems with containers in microcontrollers in Section 4.1. In Section 4.2, p. 20, we construct desiderata based on the problems we identified and our ideal solution for those problems. Then we will present our hypothesis in Section 4.3, p. 21 and outline the research questions we will be working on in Section 4.4, p. 21. In Section 4.5, p. 22, we will present the scope of the work that will be done in this dissertation. In Section 4.6, p. 22, we will present how we will validate that the solution we discovered is capable of resolving the problems described. Lastly, we will summarize this chapter in Section 4.7, p. 23.

4.1 Current Issues

Even though microcontrollers are known as low-end devices that have constrained computing abilities, recent developments have increased their capabilities, and those devices are more developed than ever. This increase in computing abilities has provided significant developments in IoT and, in recent years, has allowed bringing DevOps to the code development in microcontrollers. One of the DevOps tools now available in microcontrollers is the containerization of tasks. Some new containerization tools, which were detailed in Section 3.3, p. 16, such as Wasmico [33], work on microcontrollers, which brings many benefits to developers who use these devices.

Those tools allow developers to abstract their code from each vendor's existing architectures, code in different languages besides the usually used low-level programming languages, and execute multiple isolated programs concurrently. But they still lack some features that would be helpful for developers to use the devices' capabilities as much as possible.

We have pointed out two main issues with the existing solutions, (1) lack of container's state migration and (2) bad user interfaces. The first issue is already solved for regular computers that can run operating systems with containerization tools, such as the already mentioned Karhula *et al.* solution [22], so we can build our solution with similar ideas.

4.2 Desiderata

The following desiderata can describe our vision of the ideal containerization tool for microcontrollers:

D1: Architecture agnostic tool. The ability to operate in any of the several microcontroller architectures would be necessary for our tool to be useful. Otherwise, users would be trapped in the architectures we choose to offer, and the solution's applicability would be significantly reduced.

D2: Run multiple containers in parallel. A good containerization tool has to be capable of running multiple containers at the same time.

D3: Support multiple programming languages. Our perfect solution would be able to run any programming language. By doing that, we are giving users the freedom to work on the languages they are already acquainted with.

D4: Over the air updates. To avoid having to physically connect to each device each time a change has to be made, the user should be able to carry out updates over the air.

D5: Migrate containers' state between devices For the tool to be fully functional, we want it to allow users to migrate the state of running tasks between microcontrollers.

D6: User-friendly interface. This tool is intended to be used by developers; thus, it must have an intuitive user interface that doesn't obstruct their work.

D7: Incorporation into other applications. For a containerization tool to reach its true potential, it should have an API that developers can use to integrate it into their applications and use it in automatic tasks, such as CI/CD.

From the tools described in Section 3.2, p. 13, four of them can be described as containerization tools for microcontrollers, (1) Toit [40], (2) Golioth [19], (3) Femto-containers [47], and (4) Wasmico [33]. We decided to compare these solutions based on their fit with our description of the ideal tool in Table 4.1, p. 21.

None of the existing tools fit our description of the perfect tool for containerization in microcontrollers, which means there's a gap between what exists today and what we want to use.

Tool	D1	D2	D3	D4	D5	D6	D7
Toit	✓	✓	-	✓	-	-	✓
Golioth	✓	-	-	✓	-	✓	✓
Femto	✓	✓	-	✓	-	-	-
Wasmico	✓	✓	●	✓	-	-	-

Table 4.1: Comparison of containerization tools for microcontrollers according to their completion of our desiderata. (●) means WebAssembly compilable languages.

4.3 Hypothesis

This dissertation's work will be focused on the following hypothesis:

> *"Through a user-friendly interface, as well as the facilitation of containers' state migration between constrained devices, quality of service in microcontrollers' containerization will be increased by simplifying containers management and reducing the time to recover from failures."*

We aim to improve the users' experience with the existing tools by incrementing one of them with an updated and user-friendly interface and validating the usability of the new interface with user tests based with a validation based on the System Usability Scale [7].

We also want to allow developers to migrate containers from one device to another to help developers reduce the downtime of tasks and validate it by comparing the time to recover from failures with the previous versions of the tool. This is based on the fact there already exist migration solutions for some IoT devices and that applying the same logic to containers in microcontrollers will bring similar benefits.

4.4 Research Questions

We have separated our hypothesis into three research questions, which, if validated separately, will validate our hypothesis.

RQ1 *Can we provide a way to migrate the state of a container from one microcontroller to another?*

To allow the continuation of a running container in a different microcontroller, we should be able to copy the state it contains to the new microcontroller and start that container with the copied state.

RQ2 *If we find a solution for **RQ1**, does it help developers by reducing the time to recover from failure?*

If we manage to develop a solution for the **RQ1**, we need to validate if the state migration helps developers and, for that, we will need to compare the time to recover from failure with and without the state migration.

RQ3 *Will a new user interface be easy to use and help users manage the containers running in each microcontroller?*

After implementing the new user interface for the containerization tool, we will need to test it with potential users to know if it improves and simplifies container management in microcontrollers.

We will dwell on these three questions and create solutions that together will represent an answer to our hypothesis.

4.5 Scope

After the comparison shown in Table 4.1, p. 21, we decided to develop our solution using Wasmico. It was one of the tools that fulfilled more of our desiderata, and besides that, we had two other reasons:

- Firstly, it is an open-source tool that allows us to understand and expand the existing functionalities quickly. By having access to the source code, it will probably be easier to understand how the tool works and how to increment its functionalities.
- The other reason is the already implemented functionalities. Wasmico already allows its users to pause and resume containers by sending commands via HTTP, so our work will use those functionalities and expand on top of them by allowing users to copy the state of the paused container to another microcontroller and resume it in the new device.

This dissertation's scope is to expand the existing Wasmico tool. As a first step, we will develop the container migration functionality, which involves capturing the state of a running container on a designated device and subsequently transferring that state to another device for future utilization when the container is initiated. After that, we will focus on developing a user-friendly interface to ease the use of all the functionalities that are implemented.

4.6 Validation

After completing the implementation that's defined in Section 4.5, we must validate that our implementation confirms our hypothesis. We will need to validate our implementation in two parts, (1) migration of a task's state between two microcontrollers and (2) the usability of the new user interface.

To validate the quality of service improvement caused by the migration of a task's state, we will compare the recovery time from failures when beginning the task from its initial state versus from a previously saved state. We will need to determine how long it will take to store the state of a task and upload it to a new microcontroller, how long it will take to start a task with and without a state, and ultimately, how advantageous it will be to be able to migrate the state.

To validate the usability and simplicity of our user interface, we will conduct a user survey based on the System Usability Scale [7] to determine whether the interface improves the users' satisfaction with the tool. In addition to the SUS queries, we will also include questions to identify user groups and determine if any of their characteristics will influence their evaluation of the interface.

4.7 Summary

In this chapter, we discussed the problems that exist in the currently available containerization programs for microcontrollers. After defining the existing problems, we specified what our ideal containerization tool would be able to do in our desiderata and contrasted the features of the solutions presented in the state of the art with the features that we specified. From this, we formulated our hypothesis regarding what we wish to be able to do and how this will enhance the work of developers. This hypothesis was divided into three research questions that are simpler to comprehend and validate separately. After delineating what needed to be done, Section 4.5, p. 22 describes how it will be accomplished. Finally, we described how we will validate the hypothesis by testing our implementation.

Chapter 5

Solution

In this chapter, we will describe the work we performed on top of Wasmico in order to solve the previously mentioned problems. In Section 5.1, we begin with a comprehensive overview of Wasmico's capabilities before our work. In Section 5.2, p. 27, we will describe some minor changes that were made to Wasmico that did not correspond directly to the previously mentioned issues but were nonetheless deemed necessary. Then we explain how we implemented the container's state migration in Section 5.3, p. 29. In Section 5.4, p. 31, we discuss the ideas we had for the user interface and explain the implementation we chose to proceed with. In Section 5.5, p. 38, we discuss the tool's limitations following our implementations. We conclude by summarizing our work in Section 5.6, p. 40.

5.1 Wasmico

Our solution will expand Wasmico, so we will start by doing an in-depth analysis of the tool. We will start with an overview of the architecture of the tool, then we will explain each of the available operations, and lastly, how the code should be structured before being compiled to Web Assembly and used in Wasmico.

5.1.1 Architecture

Wasmico was built to work on different microcontroller architectures, so it is built on top of the FreeRTOS operating system [17]. This operating system can run on ESP32s, Raspberry Pi Picos, Arduinos, and many more devices, thus bringing Wasmico to those devices. FreeRTOS and Arduino are the base of the Wasmico architecture and represent the API that connects the rest of the code to the hardware.

On top of the FreeRTOS, Wasmico has two different services, (1) a web server and (2) a wasm interpreter. The web server is responsible for allowing the users to do updates over the air, it receives HTTP connections, and based on the request, it will execute the code corresponding to the requested functionality. The wasm interpreter is Wasm3 [43] which has one environment setup for an easy start of new runtimes that will be launched and execute each new task.

All the running tasks started by the users are a thread on FreeRTOS. For each different task that the users start, Wasmico launches a new thread which starts a Wasm3 runtime that executes the WebAssembly code that was previously uploaded. Wasmico can manage all these threads to allow pause, resume, and stoppage of the running containers.

Lastly, Wasmico uses the SPIFFS filesystem[39], which allows Wasmico to store the uploaded files and read them when needed.

5.1.2 Operations

Before our additions to Wasmico, this tool offered the following operations:

Task Upload. This operation allows users to upload a WebAssembly file to the microcontroller. This file should contain the task to be executed and, additionally, four parameters can be sent: (1) reservedStackSize, (2) memoryLimit, (3) reservedInitialMemory, and (4) liveUpdate. The reserved stack size is the size of the stack memory that the program will need to execute. It needs to include the minimum memory for a FreeRTOS thread, 768 bytes, and the memory needed for the program. The memory limit is the stack size for the Wasm interpreter, and it should be equal to the *stack-size* parameter set when compiling the original code to WebAssembly. The reserved initial memory refers to the *initial-memory* parameter also set when compiling. Lastly, live update is a boolean that defines if the task should be updated if it's running. If it is set to true, the task will be stopped and started with the new code.

Edit Task Details. This operation allows users to edit the four parameters set on task upload. It is supposed to be used when users want to change any of those values but don't need to change the code meant to be executed.

Task Start. This operation allows users to start tasks that were previously uploaded. When this operation is called, Wasmico will check if the code for the task has been uploaded, the details were correctly set, the task is not already running, and enough resources are free to execute the task. If all the conditions are set, Wasmico starts a new FreeRTOS thread that will load the file, prepare the Wasm runtime and start executing the WebAssembly code. The needed information about the running thread is stored in memory to be used by some of the other available operations.

To execute this operation, the uploaded code must implement the *_start* function. This function is the main function of the program; it is normally divided into two parts, the setup, where all the initial variables and configurations are set, and the main loop, an infinite loop that will execute the task periodically.

Task Stop. This operation allows users to stop a running task. Before stopping a task, Wasmico verifies if the task is running, and if it is running, it is stopped. When a task is stopped, Wasmico removes the task from the FreeRTOS task list and erases all the info about the running thread. This sets the microcontroller in the same state as if the task was not started, which means, the WebAssembly file remains in the device, all the execution parameters are still available, but all the information created after starting the task is lost.

Task Deletion. When a task is no longer needed in a device, it should be deleted by the user. Even if a task is not running, its code and some of its information still occupy space that other tasks may need, so users should make sure to delete all the tasks that are not needed in a device. When this operation is called, Wasmico will first check if the task exists and if it is running. If the task is being executed, Wasmico will cancel the operation as running tasks should be first stopped, and only then can they be deleted. If the operation proceeds, the WebAssembly file and the details for that task will be removed from the device, and after deletion, if the user needs to rerun the task, it must be uploaded, and only then can it be started again.

Task Pause. If the user wants to pause a task so that it can be continued again in the same state later on, he should call the pause operation. This operation saves the current state of the running task on the microcontroller's memory and pauses its execution.

This operation relies on the *_pause* function. The *_pause* function should guarantee that the program is in a savable state and proceed to save it. To save the state, the user can import the *pauseTask* function from the Wasm interpreter, which allows users to save an array of bytes in the device's memory for later use.

Task Unpause. This operation allows users to unpause a task that has been previously paused. To use this operation, the task has to have been uploaded, started, and paused. The *_start* is called again, and if the user wants to retrieve the previously saved state, he should call the function *resumeTask*, which loads the same byte array that was previously stored on the *pauseTask* call.

List Task Details. Users can access the information that was specified for each of the uploaded tasks using this operation. It is used to identify which tasks were uploaded to the device and for troubleshooting reasons.

List Active Tasks. This operation lets users know which tasks are being executed, their current status (running or paused), and the task's thread's maximum memory utilization. It may be used to keep track of the tasks that are running on each device and to determine the *reservedStackSize* that should be set on a task upload.

Get Heap Info. This operation shows the current memory available in the device. It provides details on the amount of free memory on the device as well as the size of the largest available contiguous memory block.

Our work will ensure that these operations will continue to be available while adding new ones as we need them.

5.1.3 Code Structure

When developing code to be run in Wasmico, the code should be structured into three different parts:

1. **Import link functions.** At the beginning of the program, it is important to import the needed functions provided by the Wasm interpreter. Wasmico defines these functions, which includes *delay*, *print*, *pauseTask*, *resumeTask*, and others. The functions are linked by Wasm3 to the executed code and connect the high-level programming to the low-level programming specific to microcontrollers.
2. ***_start* function.** This function is the entry point of the program. It is equivalent to the *main* function of a C program. Normally it contains an initial setup that, if needed, should get the previously saved state and a main loop that executes infinitely doing the main task.
3. ***_pause* function.** This function is only called when a program is paused. The function should make sure that the program is in a normal state, and if needed, it should save the state at that moment.

From these sections, only the *_start* function is necessary for the program to be run. All the others are extensions that may be needed for some functionalities but are not mandatory for the task to be executed.

5.2 Improvements

In this section, we will cover some improvements we made to the tool. These changes were made to clarify some doubts that users had and to allow for new functionalities that were not important for the hypothesis being tested in this dissertation.

5.2.1 Pause and Unpause Tasks

When pausing a task, firstly, Wasmico executes the *_pause* function that should be defined in the code compiled to WebAssembly. That function can use the *pauseTask* function, that's defined in Wasmico and can be imported from the Wasm interpreter, which will store an array of bytes of any length that will be considered the task's state. After that call, Wasmico will completely stop the running task and free all the memory it was using.

This way of pausing a task means that when resuming it, the task will be completely loaded again, and the unpause task operation will work the same way as the start task operation. In fact, both of these operations start a new FreeRTOS thread which calls the *_start* function to run the task, with the only difference being checking if the task was previously paused.

The implementation of task state migration also means that the program would behave strangely under certain circumstances. With the previous Wasmico pause and unpause logic, if we upload a task's state that is supposed to be a backup state to a device while the task is paused, that state will be used on the unpause function instead of the state that existed at the time of the pause, meaning that the unpause operation would resume the container in a different state than when it was paused.

To avoid this strange behavior, we changed the implementation of the pause and unpause operations to utilize the FreeRTOS thread management features instead of saving the state and starting based on that state. Instead of calling the *_pause* function that's inside the user's program and saving the state, we call the *vTaskSuspend* function from the FreeRTOS API, this function changes the state of the task to *eSuspended*, which means that the microcontroller will never give any processing time to the thread executing the task. When unpausing the task, we call the *vTaskResume* function, which will resume the task. By doing the pause and unpause operations this way, we enforce that when unpausing a task, it continues in the exact same state it was when it was paused.

While changing the way these operations work, we also decided that "resume" would be a better name for "unpause" so we changed the names and, in the rest of this document, we will be using the name "resume".

5.2.2 Device Status

When working with Wasmico, especially while doing the user interface, we found that we needed a command which would quickly check if a device was running Wasmico or not. With the old Wasmico version, this could be done by making a request to any of the available endpoints that existed and ignoring the content of the response, but this would bring more workload to the microcontrollers. To lower the amount of processing needed to check if a microcontroller was running Wasmico, we created a new endpoint, which we called "wasmico" that responds to the request with a confirmation that the device is on.

5.2.3 Restarting the Microcontroller

On some occasions, while developing code to be executed in Wasmico, we created bad code that would use too much memory or processing time, and we would need to restart the microcontroller manually. Since Wasmico was made to work over the air, we thought that it would be helpful if we could restart it with a command, so we created the "restart" endpoint, which restarts the microcontroller, thus stopping all the tasks and freeing all the memory used by the tasks.

5.3 Task State Migration

Task state migration is separated into three parts, saving the state of the task that is running in a device, uploading that state to a new device that will run that task, and finally, starting the task on the last device.

5.3.1 Save the Task's State

The first step in migrating the state of a task between two microcontrollers is saving the state of the running task on the original device. Wasmico was already doing this on the pause operation before our changes, so we kept the idea and changed the *_pause* function to a similar *_state* function and changed the *pauseTask* function to the *saveTaskState* function.

As part of this operation, we had to allow the users to export that state. When a user calls the save state operation, Wasmico will call the *_state* function to retrieve the current state of the task and save it in memory as an array of bytes together with an integer corresponding to the array's length. After that, it will respond to the request in JSON notation containing the *data* and *len* fields. The *data* field is a hexadecimal string containing the state of the task. For example, if the task needed to save a state which is the bytes 0xDE, 0xAD, the *data* field would be "DEAD". The *len* field is the length of the sent data in the number of bytes. In the same example, the *len* would be the number 2.

At the end of the operation, the state of the task is both in the response from the device and in the device's memory, which already contains each task's parameters and runtime. By saving the state in the device, we allow users to use it as a backup without having to upload the state at a later time, this means that we users can do the same steps involved in the task state migration without uploading it to restart the task from a checkpoint in the same microcontroller.

The process of saving a task's state can be seen in Figure 5.1.

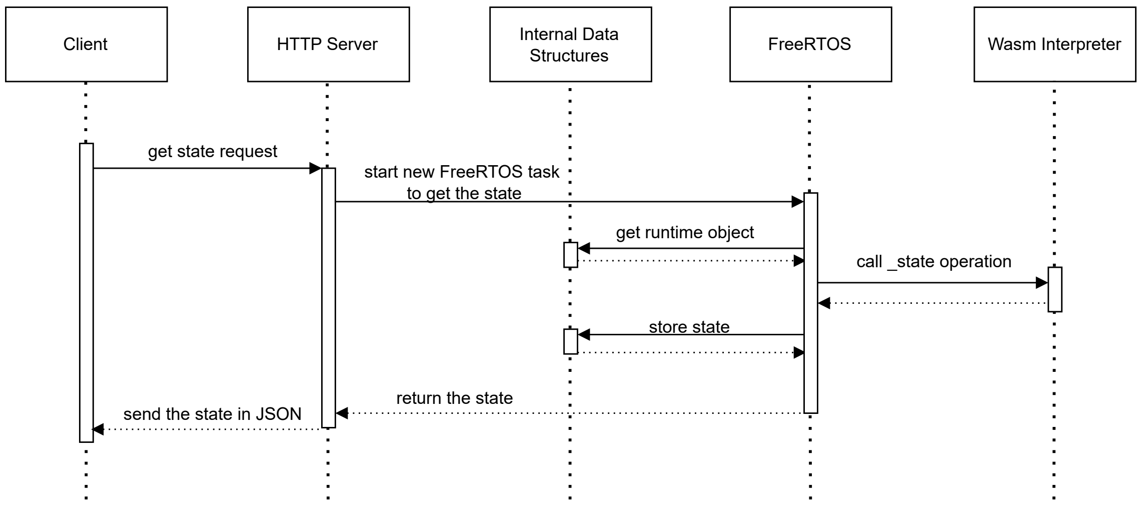


Figure 5.1: Sequence diagram of the get state operation. To avoid cluttering the image, we removed the check operations.

5.3.2 Upload the Task's State

The second step in migrating a task to a different microcontroller is uploading that task to the destination device.

To avoid changes between the state that was saved and the state that is uploaded, we decided to keep the same format across both steps, this means that when uploading the state, the user should send a JSON object containing, at least, the *data* and *len* fields. The uploaded state is stored in the microcontroller's memory in the same way that the save state operation stores it to be used when starting the task.

Users can upload a state for the same container as many times as they want, but to reduce the memory used on each microcontroller, the device will only store the most recent one. This allows for periodic backups of tasks in different microcontrollers without drawbacks besides the time needed to upload and store the state.

In Figure 5.2, you can see the sequence of operations done when uploading a state.

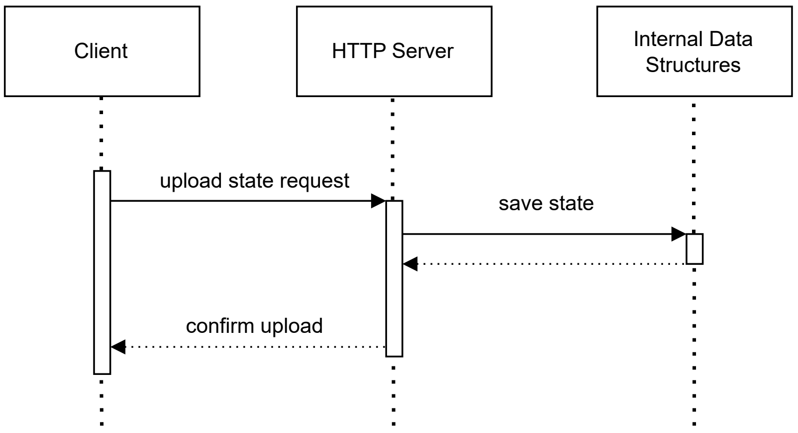


Figure 5.2: Sequence diagram of the upload state operation. To avoid cluttering the image, we removed the check operations.

5.3.3 Starting the Task

If a task's state is saved on the device, when the task is started, in the *_start* function, the user can call the *getTaskState* function, which allows the user to get the saved state from the Wasmico data structures and use it instead of the task's initial state. The state will be available in the same state as it is saved, an array of bytes together with the length of that array.

To avoid the confusion mentioned when pausing and resuming tasks, if the users want to update a task's state, that task should be stopped and started again, as the pause and resume will ignore the new state and continue with the same state the task had when it was paused.

The sequence of operations that are executed in the microcontroller can be seen in Figure 5.3, p. 31

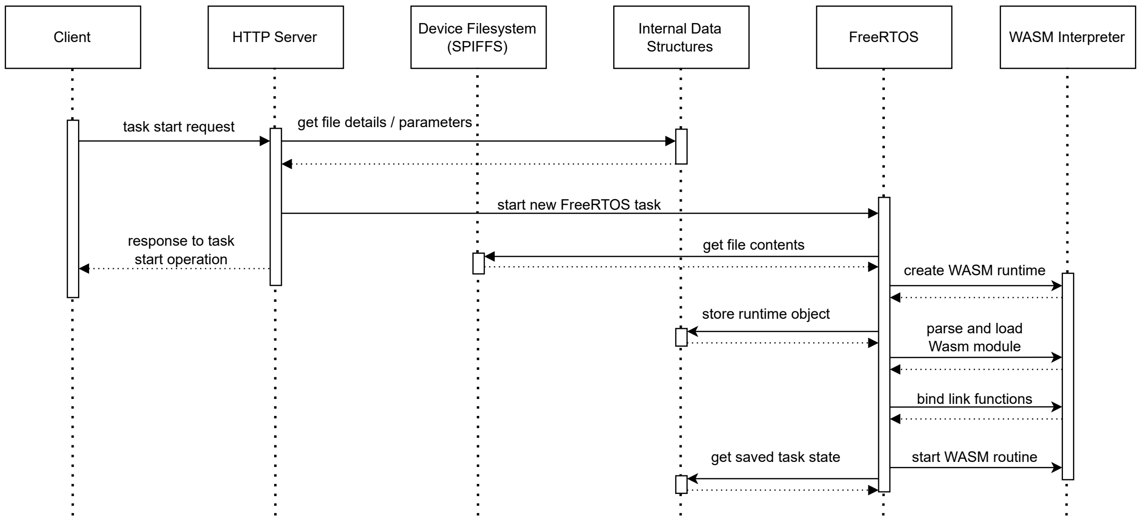


Figure 5.3: Sequence diagram of the start task operation with a saved state. To avoid cluttering the image, we removed the check operations. This was mostly taken from the paper about Wasmico [33].

5.4 Terminal User Interface

To simplify container management and facilitate the use of Wasmico, we created a terminal user interface. While developing it our main objectives were for the TUI to be simple and easy to use, even for developers unfamiliar with Wasmico or microcontroller-related tools.

Wasmico already had a command line interface developed using NodeJS [30]. To reuse as much as possible of the existing code, we also decided to use NodeJS and separated the creation of the TUI into two different parts. Firstly, we created a core library with an API that can be used on other NodeJS applications. From that, we restructured the CLI and built the TUI using that API. In the end, we delivered an API that can be used for other use cases that use Wasmico, such as the automatic deployment of code to microcontrollers and two different terminal tools that can be used to do container management with Wasmico.

5.4.1 Wasmico API

The Wasmico API is a NodeJS module that allows its users to call functions that we created that execute the needed HTTP requests to run the commands on the microcontroller. With this API, developers can create their own programs that use Wasmico while avoiding constantly checking the endpoints for each of the available commands.

Through the use of this API, users can execute the following functions:

uploadTask. This function receives the IP address of the microcontroller, the path to the file that will be uploaded, and the task parameters that need to be set when uploading. It reads the file, extracts the filename from the path, and sends all the information to the microcontroller.

deleteTask, startTask, pauseTask, and resumeTask all receive the IP address of the microcontroller and the filename of the task and send a request to the microcontroller to execute the corresponding command.

getTaskDetails. This function receives the IP address and requests the available information about the tasks that were previously uploaded.

editTaskDetails. This function receives the IP address, the filename, and all the parameters that will be edited for the corresponding task and sends this information to be edited in the microcontroller.

getTaskState. This function receives the IP address and the filename of the running task and requests the state to the microcontroller, which, in case of success, will return a JSON message with the state of the task.

uploadTaskState. This function receives the IP address, the filename, and the state of the task and, after checking the validity of the state, sends it to the microcontroller, where it will be stored.

getActiveTaskDetails. This function receives the IP address of the microcontroller and requests the details about the running tasks.

getFreeHeapSize. This function receives the IP address of the microcontroller and returns the information about its heap.

restartDevice. This function restarts the device at the given IP address.

scanNetwork. This function will scan a given network for microcontrollers running Wasmico. It pings all the possible IPs on the network, and, for the found devices, it sends a request to the “wasmico” endpoint to verify if that device is running Wasmico.

pingDevice. This function receives an IP address and sends a request to the “wasmico” endpoint, which should only respond if the device is running Wasmico.

These functions correspond almost one-to-one with the available endpoints implemented in Wasmico as they connect the command-line interface, the terminal user interface, and possible future applications to the Microcontrollers using Wasmico.

Even though this API was not part of the hypothesis that we wanted to validate, we found that it would facilitate the development of the user interface and be beneficial for future tool users in developing their own applications. Besides different user interfaces for the different needs that each user may have, we also thought that this API could be useful for CI/CD applications, where the developers could test their code automatically on some devices and easily deploy the new and tested applications to multiple devices.

5.4.2 User Interface Requirements

Before choosing the type of user interface that we would implement for Wasmico, we had to decide the specifics of the interface that we wanted, and we determined that our interface would need to have three major functionalities, (1) display information about the microcontrollers using Wasmico, (2) manage microcontrollers individually, and (3) manage microcontrollers as a group.

Firstly, the user interface would need to show the microcontrollers using Wasmico. For each of the connected devices, the interface should display a custom name for it, the IP address of the device, if the device is on or off, and, if it's turned on, the tasks that it's running. The device's name should serve as an identifier so that the user can distinguish between devices quicker than by using the IP address. The IP address of the device is essential for identifying the devices in a network and for troubleshooting. The device's status should be used to determine whether the device is available or not, as well as whether the interface-using computer can connect to the microcontroller. The device's tasks should aid container management by providing information on which tasks are operating on which microcontrollers and preventing simple errors, such as stopping a non-running task.

In addition to displaying the information, the interface should allow users to send all API commands to the various microcontrollers. It is essential that the interface we design is able to utilize Wasmico's features and that users can easily send the appropriate commands and set the correct parameters. The tool should display a list of the available parameters, plainly indicate the required parameters when executing a command, and, if necessary, provide suggestions regarding particular parameters.

Lastly, in cases where the user has many microcontrollers and needs to manage them all, it's important that the user can set them in groups and manipulate them in the same way he can manage individual devices. The user should be able to create new groups, add devices to the groups, and send commands to all the devices in the group simultaneously. This should be beneficial in situations involving many devices, such as a phased task deployment to multiple devices or a separation of devices with different functionalities.

5.4.3 User Interface Type

When deciding what would be a good interface, we debated over the four different types that we might use: (1) command line interface, (2) terminal user interface, (3) graphical user interface, and (4) web user interface and the advantages and disadvantages of each one.

5.4.3.1 Command Line Interface

The first option we considered was a command line interface. Before this dissertation's work, there was already a CLI as part of Wasmico, but it didn't feel easy to use for us, and, most important, it did not fulfill the requirements that we had set.

Command line interfaces normally don't store information between multiple usages, if there's information that is needed to execute the program, it usually comes either from the program arguments or the standard input. This means that to keep track of the available information about the containers in the microcontrollers, we would need to send new commands, which would be additional work and give less information than automatic updates periodically.

This type of interface would be a good option to send a single command to a single microcontroller, as it would be quick and simple, but when dealing with multiple microcontrollers or more than one command, this interface would require multiple commands that would have to be typed separately, meaning this interface type would not be a good solution.

Due to their normal format, CLIs are usually used as part of larger programs that use the tool as part of a bigger task. Users will use a command line interface to execute one-time tasks or in junction with other tools where they can chain the information from previous commands to be used in the tool and use the output as input for another tool.

Concluding, even though a CLI is useful when integrated with other tools in more complex scripts or when sending single commands to the microcontrollers, it was not a good user interface to manage the containers running in those microcontrollers.

5.4.3.2 Terminal User Interface

Terminal user interfaces are normally designed to be used when the tool is only meant to run in a terminal or when the user may not have access to a desktop environment where there's the possibility of using applications apart from a terminal. This means that a TUI can be used in most situations, even in special cases, such as in servers where the user may only have terminal access or through a remote shell to another computer with tools such as SSH.

With a TUI, users can easily keep track of the state of the tool that is being used; in our case, keep track of the containers that are running in each microcontroller. TUIs normally are tools that are continuously running and, thus, have a state which is kept, at least, while the user doesn't terminate it.

By using this type of interface, we can create menus with which the user interacts to send commands, and we can create groups to send commands to multiple devices at once without requiring more effort from the user's perspective.

5.4.3.3 Graphical User Interface

Graphical user interfaces usually are large programs with a wide variety of features or programs that are used not only with a keyboard but also with a mouse. Normally these programs consume more resources from the computer and require the computer where it's used to have a screen and a desktop environment, which may not be available on servers and smaller computers, which may work as routers to the microcontrollers' network.

On the contrary, graphical user interfaces are easier for most people, especially those who may not be used to terminals. This would make the tool more friendly for non-developers needing to work with it.

Just like with TUIs, GUIs would be able to save the information of multiple containers and display menus and tooltips to help users send the needed commands to the microcontrollers, thus, managing them easily and efficiently.

5.4.3.4 Web User Interface

The last option that we considered was a web user interface. WUIs are similar to GUIs, but instead of running the user interface directly on the operating system, the interface is designed for browsers. The advantages and disadvantages are mostly the same, with some exceptions. In the process of developing a web user interface, it is not typically necessary for developers to modify their code or tools in order to ensure compatibility with the majority of web browsers. In the process of creating a graphical user interface, the requisite code and tools will likely vary depending on the specific operating system for which the program is intended. Also, web user interfaces are more similar across multiple programs, which makes the adaptability to our program much faster than the other options.

5.4.3.5 Conclusions

After deliberating about the possible user interfaces that we could create for Wasmico, we found that a terminal user interface would be the one with the most benefits and least drawbacks. We decided to keep the existing command line interface but not develop it further because, in the end, the tool would not be a good management tool. We think that the tool is still important to build scripts and to send single commands quickly, but not for our main purpose.

Both the GUI and the WUI would require more resources and larger computers to be executed in, and, thus, they would not be suitable for use in servers or routers, which may be important for companies with separated networks for microcontrollers and other devices.

Lastly, the biggest disadvantage that the TUI had, not being user-friendly for non-developers, can be mostly ignored since Wasmico is for microcontrollers' developers, and a terminal is already a frequent tool when working with those devices.

5.4.4 Implementation Details

The solution we implemented is a terminal application, which can be divided into two parts, the table display, which shows information to the user, and the command menu, which allows users to send commands to the microcontrollers shown in the table.

The table appears always before the menu and contains the following information for each of the devices:

Name. The device's name is a unique identifier set by the user to distinguish between multiple devices. When sending commands from the command menu included in the TUI, the name will be used to choose the device to send the commands. By default, the name of a device is the IP address, as it should be unique across all devices.

Group. When the user has many devices which execute the same tasks, for example, multiple temperature sensors in different rooms, they should be set in the same group. Groups work the same way as names for the devices, but instead of selecting one device at a time, the user can select the whole group at once. This may be useful to execute the same command on all the devices of a group simultaneously.

IP address. Each device's IP address is an identifier similar to the name, but the IP address should not be used to choose the device to work with. Since the communication with the microcontrollers is done via HTTP requests, the IP address is used to send the commands.

Status. The status of a device corresponds to whether the device is on or off. It is determined by whether or not the computer operating the interface can communicate with the microcontroller. To test this communication, the interface periodically sends a ping command to the Wasmico microcontroller and waits for a positive response.

Number of running tasks. To easily keep track of the tasks running on a microcontroller, the TUI shows the user how many tasks are running on each device for each online device.

Free Space. Lastly, for all the online devices, the TUI shows the free space available in the microcontroller's heap. This is the space that will be occupied by uploading and starting tasks, so the user should check this value before executing memory-consuming tasks to guarantee that there's enough memory for the task.

Under the table containing this information is a menu from which the user can select various operations to carry out. Depending on the selected operation, the TUI will request additional information required to execute it.

The available commands and the needed information are the following:

Add / Rename a device. This option enables the user to add a device not already present in the table or rename an existing device. This operation requires the following information for each microcontroller: the name that the user wants to use for future references to the device, the IP address of the device, which is especially important for devices whose IP address is in a network that you can't or shouldn't scan, and the group that the device belongs to; by default, the group is named ”default,” and should be used if the user does not need to set a specific group.

Remove a device. If the user doesn't need a device anymore, it should be removed from the interface. This command removes the information that the tool has about a specific microcontroller. To remove the device, the user only needs to select it from a list of all the names of the configured devices.

Scan network. When the user wants to add all the microcontrollers running Wasmico that exist in a specific network, such as the company network, this option scans the whole network and finds all the devices whose HTTP server responds with status 200 to a request to the "/wasmico" endpoint. To run this command, the user must only specify the network address according to the CIDR [4] format (IP address and suffix). To facilitate, the active network interface of the computer running the TUI is the default option that corresponds to the network that the user's currently using the most.

Upload file. This option will execute the API calls needed to upload the file to all the selected devices. The user starts by selecting all the devices to which he wants to upload the file by name, group, or both. Then enters the path to the file that's been compiled to wasm relative to the terminal being used. Lastly, the user enters the reserved stack size, reserved initial memory, and memory limit, which are needed when uploading the file and whether to live to update the task. Finally, the tool will output the answer given by each microcontroller that was chosen.

Remove file. Once a file is not needed anymore, the user should use this command to remove it from the device. Microcontrollers are devices that have low memory and low storage space, so the users should not keep files that aren't needed on devices. To remove them, the TUI will ask which device to remove from and the file's filename that should be removed. After sending the request, the console will print the response given by each of the devices.

Start task. After uploading a file to a microcontroller, the user needs to start the task for it to execute. The parameters needed by the TUI are the devices to start the task on and the filename of the task.

Stop task. To stop a task that has been previously started and is running, the user will need to select the devices to stop the task on and then select one of the tasks currently running on any of those.

Pause task. The conditions and parameters needed to pause a task are the same as stopping a task.

Resume task. Resume task is similar to the pause task option, but instead of selecting a task that is running, the user selects a task that has been paused before.

Save task state. This option allows users to save the state of a running task. To save the state, the user needs to specify the device running the task, the task whose state is supposed to be saved, and a filename to output the state to that file. In the end, either the console prints an error or the given file will contain the task's state.

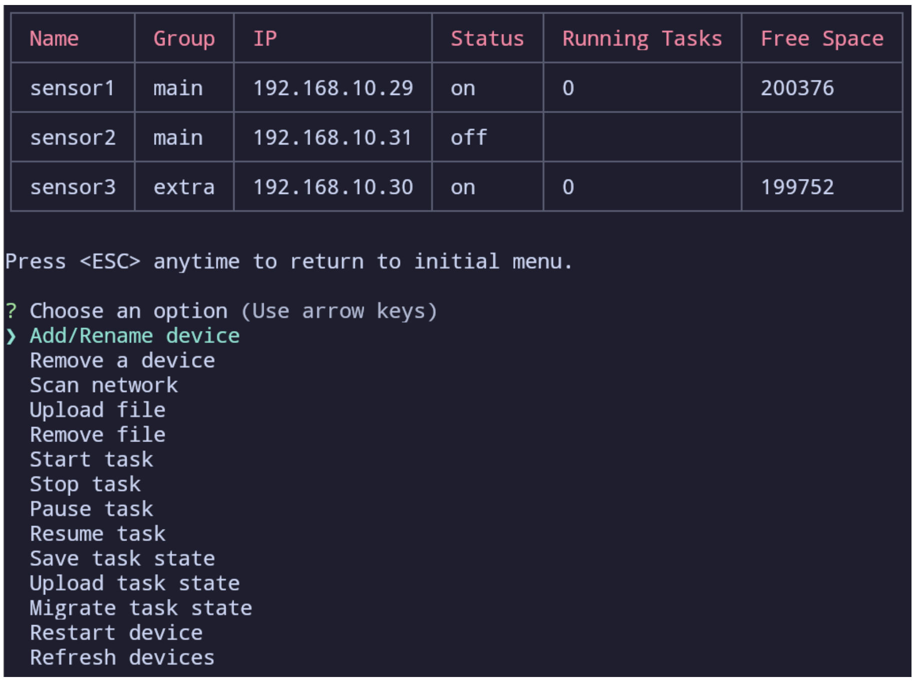
Upload task state. In conjunction with the save task state options, the upload task state option should be used to upload a state that was previously saved. There needs to be a device to upload the state, a file with the state in the same format in which it was saved, and the filename of the task whose state belongs to.

Migrate task state. This option does the same as the save and upload task state options together without storing the state in a file. In this case, the user needs to specify the device that is running the task, the task whose state will be migrated, and the device that will receive the state of the task.

Restart device. After selecting this option, the user needs to select one or more devices, and they will be restarted.

Refresh devices. This option refreshes the table display and the information about the devices. It will update the device's state, the number of tasks running, and the free space available on each microcontroller. These updates are done automatically every five seconds, but the table information is only updated after running a command. If the user wants to update the information without sending it using other commands, this option forces a refresh of information and updates the table.

The user interface that corresponds to this description can be seen in Figure 5.4.



```
┌─────────┬───────┬───────────────┬────────┬───────────────┬────────────┐
│ Name    │ Group │ IP            │ Status │ Running Tasks │ Free Space │
├─────────┼───────┼───────────────┼────────┼───────────────┼────────────┤
│ sensor1 │ main  │ 192.168.10.29 │ on     │ 0             │ 200376     │
├─────────┼───────┼───────────────┼────────┼───────────────┼────────────┤
│ sensor2 │ main  │ 192.168.10.31 │ off    │               │            │
├─────────┼───────┼───────────────┼────────┼───────────────┼────────────┤
│ sensor3 │ extra │ 192.168.10.30 │ on     │ 0             │ 199752     │
└─────────┴───────┴───────────────┴────────┴───────────────┴────────────┘

Press <ESC> anytime to return to initial menu.

? Choose an option (Use arrow keys)
❯ Add/Rename device
  Remove a device
  Scan network
  Upload file
  Remove file
  Start task
  Stop task
  Pause task
  Resume task
  Save task state
  Upload task state
  Migrate task state
  Restart device
  Refresh devices
```

Figure 5.4: Terminal User Interface developed for Wasmico. On top there's a table displaying the information about the microcontrollers, and below that a menu containing all the available commands.

5.5 Known Limitations

No tool is perfect, and Wasmico is a tool.

While working with the tool for this dissertation, we discovered additional constraints beyond those already known to Wasmico. While some of these issues may be crucial for Wasmico to gain developer support, others may be less crucial but nonetheless worth mentioning.

5.5.1 Previous Limitations

In their paper, Eduardo Ribeiro *et al.* mention three different limitations in Wasmico. Since our work was not focused on any of those topics, the tool still has those problems. We will explain them, but if you need more information, you should read their paper.

The first limitation happens when stopping a task. After the user stops a task, not all the allocated memory is deallocated. In their paper, they show an experiment in which they start and stop a task, and the available memory after stopping the task is smaller than the memory available before starting the task. Even though the cause of this is not precisely defined, they mention that the Wasm interpreter may cause it because the same happens when running a Wasm task directly on top of FreeRTOS.

Then, they mention the lack of adaption of the stack size to the needed memory. Wasmico doesn't allow a task to grow or shrink the memory they need at runtime; the memory needed needs to be set before starting a task and is constant during the execution of the program. This means that if the task needs more memory than what was set, it will fail to allocate it and crash, and if the task doesn't need as much memory as initially predicted, it locks that memory from other tasks.

The last limitation described in the paper is related to the overhead created when starting a task. When the user calls that operation, Wasmico launches a new FreeRTOS thread and, inside that thread, reads the file content, prepares the Wasm runtime, and only then starts the task. All of these tasks require memory, which may push the required stack size to a higher value than the need for the task execution.

These three limitations still exist in Wasmico, but in our vision, they weren't as important as the work we've done on other parts of the tool.

5.5.2 Including Library Functions

In the current version of Wasmico, developers cannot easily include functions not implemented in the language's main library. When using functions from an external library, we had to include them in Wasmico and expose them via a Wasm link imported into the code.

We didn't find a way to do static linking when compiling using Wasic++, which was already being used in the previous version of Wasmico, and we believed that was the cause of not being able to utilize libraries when using Wasmico.

The other compilers we found didn't allow setting the parameters needed for *reservedStack-Size*, *reservedInitialMemory*, or *memoryLimit*, and the *wasmer* compiler documentation didn't specify how to do static linking with the libraries we needed.

In the end, we didn't solve this problem because we didn't find a solution, but it may be easier to solve in the future since WebAssembly is also a recent technology being developed.

5.5.3 Migration During Long Operations

Container's state migration is currently divided into three steps, getting the state of a task, uploading the state to another device and starting the task with that state. The first part of the migration is done through a call to the *_state* fuction that should be implemented in the WebAssembly code, where the user is responsible for storing the stateusing the *saveTaskState* function.

This execution will run in parallel with the main task which was previously started and that may be executing an operation which requires a long time to finish, *e.g.*, during HTTP connections with a server. If this is the case, the state may be in an inconsistent state when *_state* is called which would make the developer's job of ensuring that the saved state is consistent and usable harder or even impossible.

Karhula Pekka *et al.* [22] solve this problem in their paper by dumping all the information about a docker container and using that as its state, but we didn't found a similar function to get all the information about a FreeRTOS thread, which would be a solution that would eliminate this problem and remove the need of the *_state* and *saveTaskState* functions that are currently needed to save the task's state.

5.6 Summary

This chapter described Wasmico and the modifications we made to it throughout this dissertation which are available in the github repository [1].

We began with an overview of Wasmico, in which we presented the instrument as designed and implemented by Eduardo Ribeiro *et al.* . We discussed the Wasmico architecture, how the tool was divided into two services, and how it handled multiple containers operating in parallel. We review in depth every operation available on microcontrollers operating Wasmico. And finally, the structure that the code should have in order to be utilized in Wasmico.

Then, we discussed the minor modifications we made to the code, ranging from small quality-of-life features, such as restarting a microcontroller remotely and verifying if a device is connected or disconnected, to renaming and modifying the functionality of pause and resume operations.

In Section 5.3, p. 29, we demonstrated how we reused the previous code from Wasmico for the pause and unpause operations to implement the migration of a task's state by separating it into three distinct parts: saving the state outside of the device, uploading the state to another device, and starting the task from the new device.

Following that, we presented our user interface development. First, we created an API based on the existing code to facilitate the creation of Wasmico-based applications. Then, we presented

[1]https://github.com/SIGNEXT/wasmico-ng

the requirements that we believed were essential for the user interface and a comparison of different types of user interfaces. Finally, we elaborated on the implementation's specificities and described the information the interface contained and the commands it permitted users to execute.

Lastly, we discussed the most significant limitations that we believe are still present in the tool, beginning with those that existed in the previous version of Wasmico and concluding with those we discovered while using it.

Chapter 6

Evaluation and Validation

In this chapter, we will present the evaluations and the validations that we did in order to test our hypothesis. In Section 6.1, we will define what the validations that we want to achieve are. In Section 6.2, p. 43, we describe the experiments that we are going to do and how they will proceed. We will present the results of our experiments and how they prove our goals in Section 6.3, p. 44. Then, we will show some threats to the validity of the experiments in Section 6.4, p. 51. Finally, we will summarize all the information in this chapter in Section 6.5, p. 52.

6.1 Validation Goals

By revisiting our hypothesis and our research questions, defined Section 4.3, p. 21 and Section 4.4, p. 21 respectively, we can define the tests that we need to do to validate the hypothesis and answer the **RQ2** and **RQ3** questions.

We can separate our dissertation's validation into two smaller validations, validating that the migration of a container's state between microcontrollers can be used to reduce the time to recover from failures and validating that the user interface that was built is easy to use by people in software development or related areas.

To ensure that migrating the state of a task between two microcontrollers will benefit microcontrollers developers, we have to calculate the total time it takes to save the state and upload it. Then, we need to compare the time the user needs to complete the migration and the time needed for the task to reach a similar state when starting without a state.

For the usability of the terminal user interface that was created, we will run a controlled experiment where the users execute some predefined tasks using it and evaluate the tool in a post-experience questionnaire.

6.2 Experiments

We conducted different experiments for each of the validations that we mentioned before. Still, through all the experiences, all the microcontrollers used were AZ-Delivery ESP32 Dev Kit C V2. This type of device has 4MB of external flash memory, 512KB of RAM, and an ESP32-Wroom 32 chip which guarantees a frequency of up to 240MHz [1].

6.2.1 Task's State Migration

For the validation of the task's state migration, we conducted speed tests to ensure that the migration would occur quickly, thus minimizing the time required to recover from failures.

The first decision we made was to calculate the duration of a state migration. This time could be defined as the time it takes the user to send the necessary commands, the time it takes Wasmico to save the task's state, and the time it takes the task to start in its new state. For this, we measured each part separately.

All of the experiments utilized the "temperature" task, which measures the temperature every 5 seconds and outputs the moving average of the most recent readings. The number of readings in the average corresponds to the state's size. For testing purposes, all of the temperatures in this task are determined at random.

First, we determined how long it would take Wasmico to save the state of a task and then the time required for each saved state to be uploaded to another device. We used the temperature task with three varying state sizes: 10 bytes, 100 bytes, and 1000 bytes, and we expect that with these three different sizes, we would start seeing some patterns in the time needed for the operations.

After that, we realized it was crucial to understand if such timings would alter if the microcontroller was already working on other projects; thus, we conducted the same experiments while running a version of the temperature job that would run every 0.5 seconds.

Last but not least, we had to compare how starting the task was affected by the state, so we timed how long it took to launch each of the tasks used before, both with and without a previously stored state.

To prevent significant discrepancies in the obtained data, all measurements were performed 20 times, a local internet connection was used to minimize latency, and the time required to query the utilized devices was determined as a baseline for the minimum time to execute an operation.

[1]The information mentioned here was obtained from the AZ-Delivery website (`https://www.az-delivery.de/en/products/esp32-developmentboard`) on the 12th of June 2023

6.2.2 User Interface

To validate the user interface that we built, we decided to conduct a user experiment with master's students in Informatics or Electrical related fields. This experiment is divided into two different parts, a task that is supposed to be done by the users and a post-experience questionnaire to evaluate their experience using the tool.

The task they had to do (*cf.* Appendix A, p. 60) is divided into four parts, an initial explanation of the experiment and the tool to familiarize the users, a setup part where the user prepares their computer and all the files that will be needed, a tutorial to familiarize the user with the user interface, and the task that they need to execute. Throughout the first three sections, users were assisted with setting up everything correctly, and any questions that may have arisen were clarified. On the experimental task part, little to no assistance was provided, as users were expected to complete everything independently; we only intervened in cases of minor confusion. During the final phase of the experiment, we measured how much time each user required for each segment.

The post-experience questionnaire (*cf.* Appendix B, p. 63) comprises two parts, an initial part to identify users by their experience and familiarity with some concepts related to this dissertation and a second part that corresponds to the SUS questions [7]. We decided to use the SUS because it's a system that has been widely used and corroborated by multiple researchers, and the original author, John Brooke, did a retrospective on why SUS was still good based on multiple reviews done by other people [8]. This provided us with a systematic way to validate the usability of the created interface that removed a possible bias in the evaluation that was done.

We decided to conduct this experiment with finalist students of informatics or electrical engineering masters. Informatics engineering students are expected to be familiar with programming and the terminal, while electrical engineering students will probably know more about microcontrollers. We decided to proceed with the experiments with eleven different students who would execute the proposed tasks and answer the post-experience questionnaire.

6.3 Results

After doing the experiments, we got the results that will be shown and discussed in this section.

6.3.1 Task's State Migration

Before starting the validation of the state migration, the first experiment we did was to test the ping command on Wasmico. This will be useful as a baseline for the minimum time that a command may need to be executed, which already encompasses the latency time of the internet, the time for establishing the connection with the microcontroller, and waiting for a response, which should be the fastest in Wasmico. The statistics of the results we gathered can be seen Table 6.1, p. 45 and show us that the commands sent to Wasmico will always take around 120 milliseconds just for the basic connection. Only the remaining time of the operation will be on processing the given information.

Task	min	max	x	x	σ
ping	60	141	122	123	16

Table 6.1: Statistics from pinging a microcontroller running Wasmico in milliseconds. These times correspond to the time needed to connect to the microcontroller and receive a simple response in the network used for the tests.

To validate the state migration across microcontrollers we started by measuring the time Wasmico needed to get the state from one device and the time needed to upload the device to another. As mentioned Section 6.2.1, p. 43, this was performed with similar tasks whose only difference was the size of the state. This experiment produced the results shown in Table 6.2. The first thing we noticed was that most operations had a high standard deviation, which was caused by some operations taking three to four hundred more milliseconds than the average time for the operation. Even though we have no clear explanation for this, since the device only has one thread, it may be caused by other non-Wasmico computations that may be running in the background that delay the operation. The operations that involved the tasks with 10 and 100 bytes had average durations between 0.1 and 0.15 seconds, which means that the duration of the operations had almost no difference from the ping command. The operations with 1000 bytes had much higher durations, especially the upload operation; still, in the worst case, the operation needed a little over 2 seconds but, on average, only 0.5 seconds.

Task	min	max	x	x	σ
Get 10 Byte State	103	515	153	113	120
Get 100 Byte State	99	129	108	106	7
Get 1000 Byte State	64	552	217	146	165
Upload 10 Byte State	100	506	129	109	87
Upload 100 Byte State	98	514	134	115	87
Upload 1000 Byte State	273	2282	516	288	541

Table 6.2: Comparison between the time needed to get the state of a task with a state of 10, 100, and 1000 bytes and the time needed to upload the same states in milliseconds.

In order to verify the prior results, we repeated the experiment, but this time included a temperature task that measured the temperature every 0.5 seconds while the operations were being called and compared to the results from the previous experiment. This time around, there were, as was to be expected, even bigger variances. This is because there were conflicts in CPU time with the new container that was running. However, as can be seen by the values in Table 6.3, p. 46 and the plots in Figure 6.1, p. 47, the average and median duration of each of the tasks did not increase by a significant amount. This can be explained by the fact that the operation was just as quick when both tasks weren't competing for processing time. The only operation whose duration increased significantly was the upload of the 1000-byte state because it was, on average, longer than the loop time of the running task.

Task	min	max	x	x	σ
Get 10 Byte State	100	617	266	122	197
Get 100 Byte State	90	518	185	109	157
Get 1000 Byte State	71	158	127	135	30
Upload 10 Byte State	96	511	129	109	88
Upload 100 Byte State	102	520	156	116	121
Upload 1000 Byte State	287	2300	844	676	744

Table 6.3: Comparison between the time needed to get the state of a task with a state of 10, 100, and 1000 bytes and the time needed to upload the same states when the microcontroller is overloaded with another task in milliseconds.

Lastly, we compared how much these tasks would require to start when loading the state from Wasmico and when there's no state to be loaded. The statistics of the results we got can be found in Table 6.4 and show us that there's no real difference between starting the task with and without the state. Even though there are some differences in the average time to start each of the tasks, with and without a state, those differences are not significant, especially due to the high standard deviation that exists in the data we gathered.

Task	min	max	x	x	σ
Start 10 Byte Without State	141	2020	682	476	563
Start 100 Byte Without State	136	2479	968	416	915
Start 1000 Byte Without State	139	2457	694	373	690
Start 10 Byte With State	135	2850	821	461	906
Start 100 Byte With State	139	2477	763	548	705
Start 1000 Byte With State	139	2477	426	166	540

Table 6.4: Comparison between starting a task with and without a previously uploaded state in milliseconds. The tasks used had 10, 100, or 1000 bytes-long states.

Looking back at all the parts that constitute the migration of a task's state between two microcontrollers, we can separate into saving the state, uploading the state, and starting the task with the state. After this, the new device should be running the task in the same state as it was in the old device.

When using tasks with small states, *i.e.*, less than 100 bytes, the first two steps are negligible. The time between when the user begins saving the state from one device and when the state is stored on another device is likely between 0.3 and 0.4 seconds, plus the time required to send the commands, which could be reduced to almost zero with automation. Starting the task after migrating the state would likely take between 0.5 and 1 seconds, but this time would always be used, with or without state migration, so the migration does not affect the task's beginning.

In the case of tasks with a larger state, there will be a high increase in the time needed to save and upload the state. Still, larger states usually require more time to be achieved, which means that, for tasks with larger states, the migration will save more time, probably compensating for the

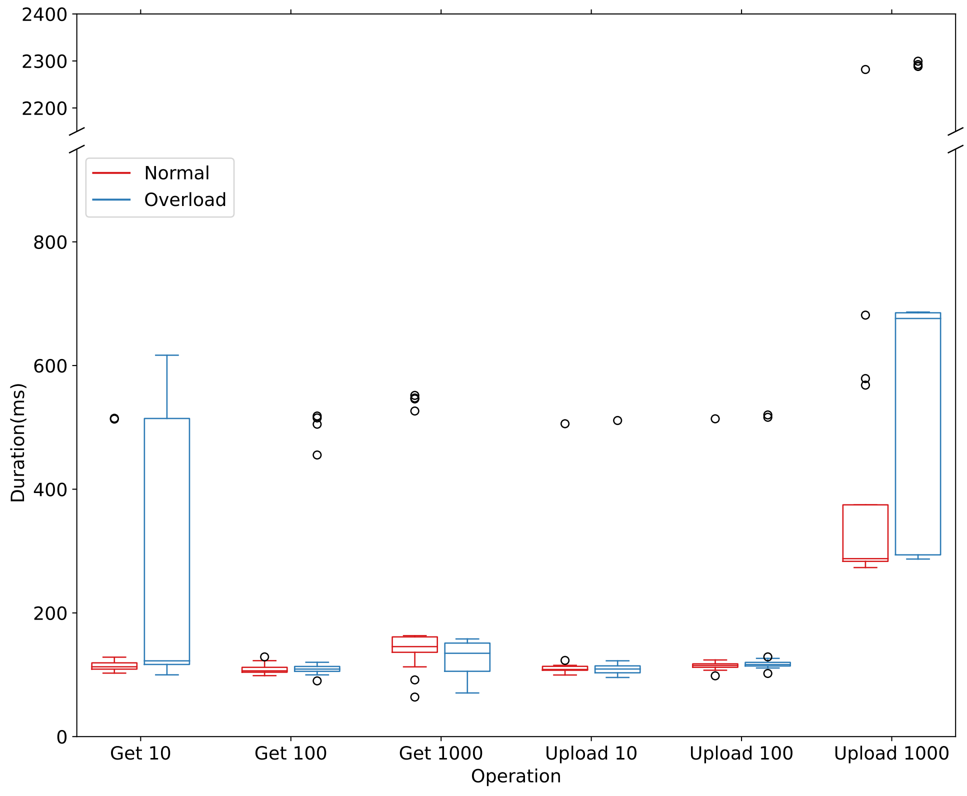


Figure 6.1: Comparison between get and upload state with and without a task running in the background. The “Get N” operation corresponds to getting the state with N bytes, while “Upload N” corresponds to uploading the state with N bytes.

extra time needed to migrate. For example, with the temperature task that was explained before, for the task to get a state of 1000 bytes, which corresponds to 1000 temperature readings, the user would have to wait at least 5000 seconds, or 1 hour 23 minutes and 20 seconds, which would be much higher than the average of 1 second we measured to save and upload the state.

Lastly, some tasks might have an important state that is not replicable. For example, if we have a microcontroller that monitors the heart rate of an ill person and needs to record all the timestamps when the heart rate falls below a certain threshold, that state is not replicable. If the state is not saved in an external computer, that data may be lost in the case of a failure and may never be recovered. This makes it important not only to be able to migrate a task’s state from one microcontroller to another but also to save the current state in a different machine.

6.3.2 User Interface

The experiment we conducted produced two distinct results: the time required by each participant to complete each task and their responses to the questionnaire (*cf.* Appendix B, p. 63).

We will begin by discussing the results of the time measurements taken during the experiment. While the users were doing the tasks, we measured the time needed to complete each part of the experiment, then those values were rounded down to the nearest five, *e.g.*, 103 seconds was rounded to 100 seconds, to accommodate for small differences between the user finishing the task and the timer being stopped. Table 6.5 displays the statistics from our measurements. The duration of the experimental task was, on average, 8 minutes and 30 seconds. Figure 6.2 demonstrates that, with the exception of the first section, where some users were still becoming acquainted with the instrument, there were no significant differences. Without the outliers, the results of the second part varied between 150 and 190 seconds; in the third part, the results were between 30 and 80 seconds; and those of the last part were between 50 and 120 seconds, with one user requiring 180 seconds.

	Part 1	**Part 2**	**Part 3**	**Part 4**	**Total**
Min	120	120	30	50	380
Max	240	270	110	180	700
Average	174	185	60	91	510
Median	180	180	60	90	480
Standard Deviation	37	44	21	39	104

Table 6.5: Statistics for the time spent by the users on each part of the experimental task in seconds.

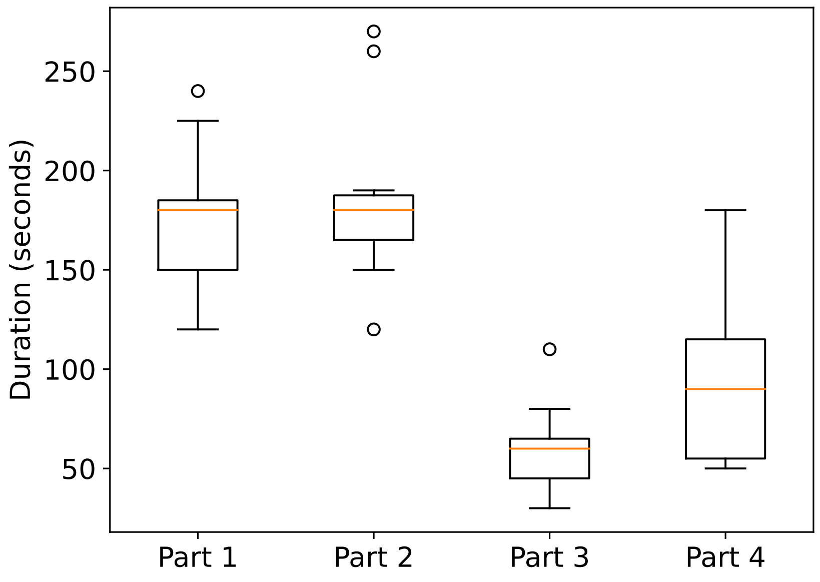


Figure 6.2: Time spent on each part of the experimental task.

The majority of users completed the task in between 380 seconds (6 minutes and 20 seconds) and 540 seconds (9 minutes), while two users required 700 seconds (11 minutes and 40 seconds) each. These two users represent most of the outliers found in the time needed to execute each part of the task. The majority of variances in time, specifically outliers, can be attributed to the type of person using the tool, some users were attempting to complete the tasks swiftly and accurately, while others were exploring the interface and attempting to comprehend the Wasmico tool without a focus on speed.

After the experiment, we conducted a questionnaire that started by determining the familiarity and experience with certain Wasmico-related concepts. We found that 36.4% of the users we surveyed had never worked with microcontrollers, while the rest had varying degrees of experience, with some having used it infrequently and others quite frequently and all of them having worked with Arduino or ESP devices using the Arduino IDE. When asked about their expertise with containerization tools such as Docker, the majority of users reported having at least some container experience (*cf.* Figure 6.3).

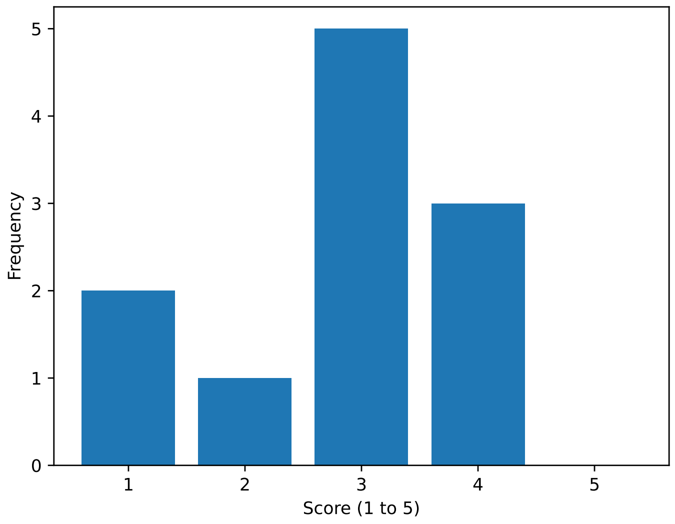


Figure 6.3: Answers given by users when asked about their experience with containerization tools. Score 1 corresponds to very inexperienced, and 5 corresponds to very experienced.

Also included in the questionnaire were the questions written by John Brooke when defining the System Usability Scale [7]. On the basis of these queries, we calculated each user's SUS score, which can be seen in Table 6.6, p. 50. According to Aaron Bangor *et al.* [5], an app with a score between 70 and 80 is considered to have decent usability, while an app with a score between 80 and 90 is considered to be good, and a score of more than 90 defines an app as the best imaginable. Other authors, such as Jonh Brooke, cite 68 points as the average result, separating bad and good interfaces. Only two users found this user interface to have poor usability, and the average score was 80.7, indicating that this tool was rated as a strong tool, 12.7 points above the average rating of 68. The two negative scores did not come from any of the above-mentioned outliers or the users who required 700 seconds to complete the task, further proving our hypothesis that the variation in time was not due to user difficulties with the tool.

User Number	1	2	3	4	5	6	7	8	9	10	11
SUS Score	70	55	82.5	97.5	80	95	92.5	57.5	85	80	92.5

Table 6.6: SUS Score for each user. A score of 68 is considered an average score. The users were numbered based on the order of the experiments for anonymization.

When examining specific responses to SUS statements, we discovered some noteworthy patterns. First, we discovered that users believed Wasmico was easy to use in general, two out of eleven users who worked with the user interface disagreed with the statement "I thought Wasmico was easy to use", and only one user responded that Wasmico was unnecessarily complex.

Then, we realized that users think of Wasmico's TUI as easy to learn and use. No one said they would need the support of a technical person to be able to use Wasmico, and no one said it would take a long time to learn how to use Wasmico. On average, users said they didn't need to learn much before beginning to use Wasmico and probably only needed the basic information given at the start of the validation task.

We calculated multiple Pearson correlation coefficients on multiple pairs of questions that we deemed important in determining whether they are related or not in order to determine if there are any correlations between the users' experience with the various concepts mentioned in the questionnaire and the results obtained from the SUS questions. We did the calculations as defined by Jacob Benesty *et al.* [6], and we used a probability of 95% that the results obtained for each of the variables are correlated ($P < 0.05$), even though this value has been contested in recent years as not significant enough to prove correlations [11].

The most important relation that we found in the answers was that the users who used more microcontrollers were the same that agreed with the statement, "I think that I would like to use Wasmico frequently when working with microcontrollers.". When comparing the answers to these two statements, by using Pearson's correlation coefficient, we got a p-value of 0.0274 which means there's a high probability these values are correlated, and a correlation value of 0.6589, which proves there's a direct relation, *i.e.*, the users who used microcontrollers more are the same that would use Wasmico more. This relationship was significant to us because these users are already familiar with some of the tools used when interacting with this type of device and are, therefore, potential future users of this tool, which could give Wasmico the upper hand when selecting containerization tools for microcontrollers.

The other relations that we had in our answers were not relevant to our purpose, as they mostly related two similar questions or opposite questions that we already expected users to answer in similar ways, *e.g.*, the users that mentioned they had more experience with containerization tools were the same that had more experience with orchestration tools, and the users that answered that Wasmico was unnecessarily complex also answered that they wouldn't imagine that most people would learn to use Wasmico very quickly.

6.3.2.1 Improvements

At the end of the questionnaire, we asked users to leave some observations about the tool. Some of the observations done were recurrent across multiple users, so we think they're important to address.

Add / Rename a device is confusing. During the experiment, users had to change the names of the devices that were previously scanned on the network, and most of the users were confused at first when doing that. This confusion was due to the ordering of the introduction of the device's information. When renaming a device, the menu asks first for the name that the device will have and then the IP address of the device that's being changed. Most people confused this and thought that the name that was asked was the current name of the device instead of the name that should appear after renaming. After asking those users they said that a simple order change between the name and the IP address, or a better tip, would clear this issue.

Choosing multiple tasks at once. In multiple menus, it was impossible to choose more than one task at once, *e.g.*, when pausing a task, the user had to choose a device and pause one task and repeat it for all the other tasks he wanted to pause. Most users said it would be helpful to be able to choose multiple tasks when pausing, migrating, stopping, and other task-related functions.

There were some other observations, such as saying the tool was really simple to use and understand, and some design choices with which we disagree, *e.g.*, allowing CTRL+C to return to the main menu instead of closing the interface, that we didn't include in this improvements because we thought were not relevant for this dissertation.

6.4 Threats to Validity

In both parts of our validation of the development we did, there were some limitations that can affect the validity of our results.

6.4.1 Task's State Migration

The majority of the validation procedure for the migration of a task's state involved measurements of the time required to complete specific operations. Several of the presented statistics indicate a high degree of variation in the results we obtained, which suggests that if we were to repeat the experiences, the outcomes might differ from those we obtained. We've already mentioned that, based on our comprehension of this issue, we believe that the high variances are caused by factors outside of Wasmico that we can't fix or that would require a modification to some of Wasmico's foundations, which are not addressed in this dissertation.

Even with this limitation, the results were satisfactory for our validation, and even if they lose their validity, the last reason for having task migration that we mentioned, saving an unreplicable task state, is a strong reason why this section of our work may be important to future Wasmico users.

6.4.2 User Interface

Our validity of the user interface stands on the tests that were made and the answers given by the users that did those tests. The selected people that responded to the questionnaire and the answers they gave could have some bias, and we identified three types of different biases that may influence the results and impair the replicability of our results.

Sampling bias. As mentioned, eleven students with master's degrees in informatics engineering and electrical engineering were interviewed, but they all attended the same two universities, which may have created a bias in terms of the type of individuals who filled out the survey. In our work samples, we excluded students from other universities, localities, and courses that may work with microcontrollers, such as computer science.

Courtesy bias. Some users may have provided answers with higher values than intended to improve our results. We attempted to mitigate this by incentivizing interviewees to provide what they believed to be the most appropriate response for how they felt and not to be biased towards assisting our validation, as this would reduce the validity of the results, whether they were positive or negative.

Sample size. In our experiments, we used eleven people, which is a small number of people. We recognize that with a larger sample, the results may have been different and more significant than the results we had.

We attempted to mitigate these threats to the greatest extent possible, but they still exist and may therefore have influenced our results.

6.5 Summary

In this chapter, we validated the research questions **RQ2** and **RQ3** through some experiments.

Firstly, we validated that by allowing users to migrate the state of a container between micro-controllers, we reduce the time to recover from failures. We measured the time needed to get the state from one device, upload it to another device, and start the task and concluded that, in many cases, it would be faster than starting a task again and waiting until a similar state is reached. Besides that, this migration allows users to save states which are not replicable.

Then we validated the user interface we created by doing an experiment where the users worked with the TUI and answered a form to evaluate the tool. Through this, we found that users think the tool is easy to use, and the more experience with microcontrollers a user had, the more they wanted to use the tool.

Chapter 7

Conclusions

In this chapter, we will reflect on the work done in this dissertation and the work that's still to be done. Firstly, we will summarize this document in Section 7.1. Then we will revisit our hypothesis and research questions in Section 7.2, p. 54. Lastly, in Section 7.3, p. 54, we will show some topics in which there's still some work to do

7.1 Summary

With the expansion that has occurred and is expected to continue in the number of IoT devices, it is more necessary than ever to assist developers working with low-end devices such as microcontrollers. At the same time, we witnessed significant revolutions in computer software development with the introduction and advancement of the DevOps paradigm. Thus, incorporating DevOps best practices into microcontroller programming is one strategy to improve development for these devices [32, 13].

Containerization has been introduced to microcontrollers as an important DevOps paradigm in recent years, but the available solutions still lacked some of the capabilities we believe are vital in today's software development. As a result, this dissertation enhanced one of the current tools in order to improve its functionality and ease of use.

These needs led us to expand the Wasmico tool and implement the migration of a container's state between multiple microcontrollers. We re-used the code from the existing pause and unpause operations and changed it to allow users to migrate the state to another microcontroller and to back up their tasks on those devices or a computer. After that, we created a new user interface that allowed easier management of the containers running in each microcontroller from a separate computer. We decided to go with a terminal user interface that shows the main information about the microcontrollers and the containers running on them and a menu with all the available commands that facilitate executing the operations in the devices.

Lastly, we validated both of these new features to ensure that they would be useful for developers. We started by measuring the time needed to migrate a container's state between two microcontrollers and the time needed to start the task after the migration and found that the time needed to execute those operations was small enough that, in most cases, it would be faster to migrate and start a task than to start a task and wait until it got a similar state. We also validated the usability of the user interface through a user experiment which gave us positive results validating that the TUI we created was easy to use and facilitated container management.

7.2 Hypothesis Revisited

Considering our hypothesis as it was defined in Section 4.3, p. 21, it encompasses the issues that we identified on the tools that existed and the validations that would need to be done. It was defined as:

> *"Through a user-friendly interface, as well as the facilitation of containers' state migration between constrained devices, quality of service in microcontrollers' containerization will be increased by simplifying containers management and reducing the time to recover from failures."*

We separated this hypothesis into three different research questions, which were easier to validate and, when proved, would also prove this hypothesis.

The first research question debates whether it is possible to migrate the state of a container between two microcontrollers, similar to how this is already done in computers. This was proven possible by our implementation, which is described in Section 5.3, p. 29.

The other questions were confirmed in Chapter 6, p. 42. **RQ2** was confirmed by our validation that the state migration would be useful for microcontroller developers. **RQ3** was confirmed with the results we got from our user experiment, which proved that the tool was easy to use and helped users manage the containers in each microcontroller.

By answering the three research questions separately, we have confirmed our hypothesis. Following the implementation of a user-friendly interface and container state migration, we improved the quality of service in microcontroller containerization by simplifying container management and reducing the time required to recover the state of a task in the event of failure.

7.3 Future Work

Since this area is still an undiscovered path, there's much to do in order to improve what's already been done. Here, we will present some work that we think should be done, both in relation to Wasmico and this dissertation and in relation to containerization for microcontrollers.

Fix known Wasmico limitations. Wasmico has some limitations that were already presented in Section 5.5, p. 38. These problems may stop Wasmico from being widely used in micro-controller development, so we think it's important to address and try to solve them.

Improve the user-interface. In Section 6.3.2.1, p. 51, we mention some improvements that users mentioned about the tool, which could be implemented to improve its usability.

Improve the user study. The user study that we conducted has some threats to its validity, such as the number of users that were used. With a better user experiment, the results we had would either be confirmed or be denied, but in both cases, it would have an increased validity.

Real-world orchestration. Wasmico was not tested in real-world experiments that could prove its usability in orchestration. A future developer who works with Wasmico should use it in complex orchestrations of microcontrollers to demonstrate the tool's full capabilities [31, 35].

Containerization in microcontrollers is still far from what could be achieved, and several points were not addressed in this dissertation that can be used as future research topics.

References

[1] Inc. Amazon Web Services. What is Containerization? - Containerization Explained - AWS. https://aws.amazon.com/what-is/containerization/. [Online; accessed 2023].

[2] Inc. Amazon Web Services. What is Virtualization? - Cloud Computing Virtualization Explained - AWS. https://aws.amazon.com/what-is/virtualization/. [Online; accessed 2023].

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, pages 2787–2805, October 2010.

[4] AWS. What is CIDR? - CIDR Blocks and Notation Explained - AWS. https://aws.amazon.com/what-is/cidr/. [Online; accessed 2023].

[5] Aaron Bangor. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *J. Usability Studies*, 4(3), 2009.

[6] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson Correlation Coefficient. In *Noise Reduction in Speech Processing*, volume 2, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Springer Topics in Signal Processing.

[7] John Brooke. Sus: A quick and dirty usability scale. *Usability Eval. Ind.*, 189, 11 1995.

[8] John Brooke. SUS: a retrospective. *Journal of Usability Studies*, 8:29–40, January 2013.

[9] CRIU. CRIU. https://criu.org/Main_Page. [Online; accessed 2023].

[10] CRIU. Usage scenarios - CRIU. [Online; accessed 2023].

[11] Giovanni Di Leo and Francesco Sardanelli. Statistical significance: p value, 0.05 threshold, and applications to radiomics—reasons for a conservative approach. *European Radiology Experimental*, 4(1):18, March 2020.

[12] Joao Pedro Dias, Joao Pascoal Faria, and Hugo Sereno Ferreira. A reactive and model-based approach for developing internet-of-things systems. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 276–281. IEEE, 2018.

[13] Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. Testing and deployment patterns for the internet-of-things. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–8, 2019.

[14] Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Designing and constructing internet-of-things systems: An overview of the ecosystem. *Internet of Things*, 19:100529, 2022.

[15] Joao Pedro Dias, Tiago Boldt Sousa, André Restivo, and Hugo Sereno Ferreira. A pattern-language for self-healing internet-of-things systems. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–17, 2020.

[16] Docker. Docker. https://www.docker.com/. [Online; accessed 2023].

[17] FreeRTOS. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. [Online; accessed 2023].

[18] G2. What Is Command Line Interface? Learn the Basics in One Go. https://www.g2.com/articles/command-line-interface. [Online; accessed 2023].

[19] Golioth. Golioth: the straightforward commercial IoT development platform built for scale. https://golioth.io/. [Online; accessed 2023].

[20] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, pages 1645–1660, September 2013.

[21] RFID JOURNAL. That 'Internet of Things' Thing. https://www.rfidjournal.com/that-internet-of-things-thing. [Online; accessed 2023].

[22] Pekka Karhula, Jan Janak, and Henning Schulzrinne. Checkpointing and migration of iot edge functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking - EdgeSys '19*, pages 60–65. ACM Press, 2019.

[23] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele Univ.*, 33, August 2004.

[24] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering – a systematic literature review. *Information and Software Technology*, 51:7–15, January 2009.

[25] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/. [Online; accessed 2023].

[26] ACM Digital Library. ACM Digital Library. https://dl.acm.org/. [Online; accessed 2023].

[27] LINFO. GUI Definition. http://www.linfo.org/gui.html. [Online; accessed 2023].

[28] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 1–13, New York, NY, USA, October 2017. Association for Computing Machinery.

[29] MicroPython. Micropython - python for microcontrollers. http://micropython.org/. [Online; accessed 2023].

[30] Node.js. Node.js. https://nodejs.org/en. [Online; accessed 2023].

[31] Duarte Pinto, Joao Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC)*, pages 1–8. IEEE, 2018.

[32] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Patterns for things that fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs*, pages 1–10, 2017.

[33] Eduardo Carreira Ribeiro. *Micro-Containerization in Microcontrollers for the IoT*. Master's thesis, University of Porto, Portugal, July 2022.

[34] RIOT. RIOT. https://www.riot-os.org/. [Online; accessed 2023].

[35] Margarida Silva, Joao Dias, André Restivo, and Hugo Ferreira. Visually-defined real-time orchestration of iot systems. In *MobiQuitous 2020-17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 225–235, 2020.

[36] Margarida Silva, Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. A review on visual programming for distributed computation in iot. In *International Conference on Computational Science*, pages 443–457. Springer International Publishing Cham, 2021.

[37] Statista. Iot connected devices worldwide 2019-2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. [Online; accessed 2023].

[38] Debbie Stone, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Elsevier, April 2005.

[39] Espressif Systems. SPIFFS Filesystem - ESP32 - — ESP-IDF Programming Guide latest documentation. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html. [Online; accessed 2023].

[40] Toit. Toit. https://toit.io/. [Online; accessed 2022].

[41] Toit. Toit programming language. https://toitlang.org/. [Online; accessed 2023].

[42] Diogo Torres, Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Real-time feedback in node-red for iot development: An empirical study. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8. IEEE, 2020.

[43] Wasm3. Wasm3: A fast webassembly interpreter, and the most universal wasm runtime. https://github.com/wasm3/wasm3. [Online; accessed 2023].

[44] Wikipedia. Text-based user interface. https://en.wikipedia.org/wiki/Text-based_user_interface. [Online; accessed 2023].

[45] IEEE Xplore. IEEE Xplore. https://ieeexplore.ieee.org/Xplore/home.jsp. [Online; accessed 2023].

[46] Koen Zandberg and Emmanuel Baccelli. Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF. In *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*, pages 1–6, December 2020.

[47] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, Middleware '22, pages 161–173, New York, NY, USA, November 2022. Association for Computing Machinery.

Appendix A

Validation Task

In this annex, we present the validation task that was done with the users. It can be divided into four parts, an initial overview, the setup, a tutorial, and the experimental task.

Wasmico Interview

Wasmico is a containerization tool developed for microcontrollers. It allows multiple WebAssembly files (.wasm) to be run on a single microcontroller. The basic flow to get a task running is to upload the code file to the device and start the task, and, if you need, you can pause, resume and stop the running tasks. The tasks may have a state. As an example, if you have a task that counts upwards to infinite, the counter is the state of the task. This state can be saved in your machine and uploaded to a microcontroller or directly migrated from one microcontroller to another.

For this experiment, we'll simulate that each of the 3 microcontrollers is a temperature and CO sensor and that you'll work with 2 of them each time. More details will follow as you progress in the task.

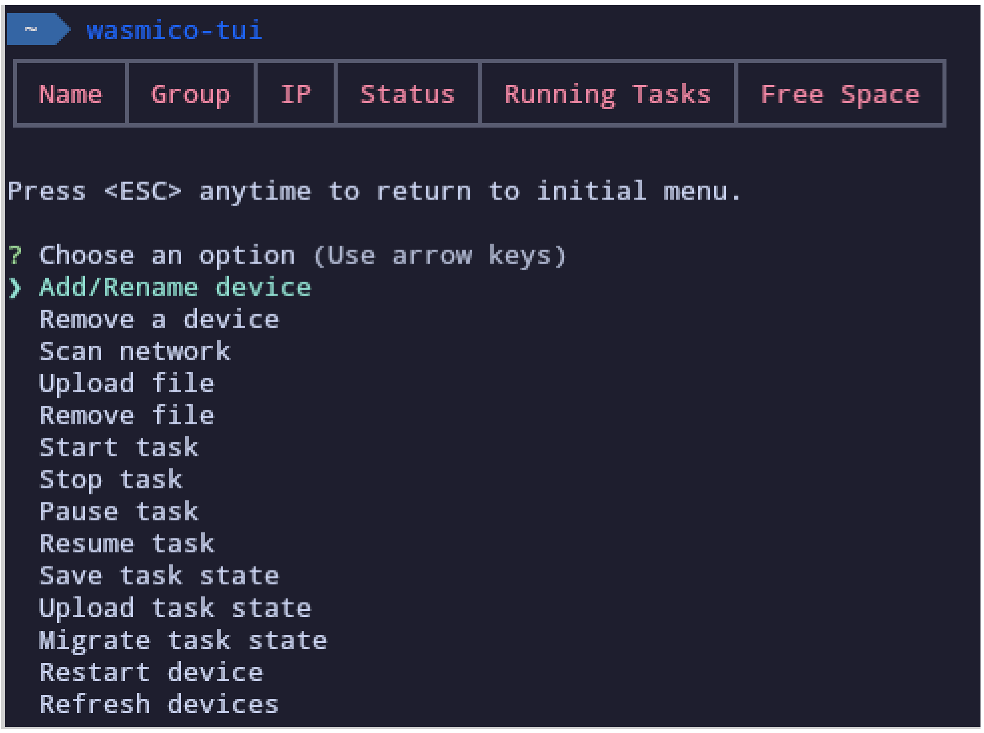
Setup:

- Install Wasmico (npm package) on your computer
- In my computer, there are 3 terminals, one for each microcontroller, that should be used for debugging purposes.
- Ensure your computer is connected to the same network as the microcontrollers.
- Download the `temperature.wasm` and `co.wasm` files.
- You should read the documentation before starting.

Tutorial:

The tutorial will guide you through the TUI until you've set up all the devices on it and uploaded one file that will be needed later.

1. Open the TUI using `wasmico-tui`

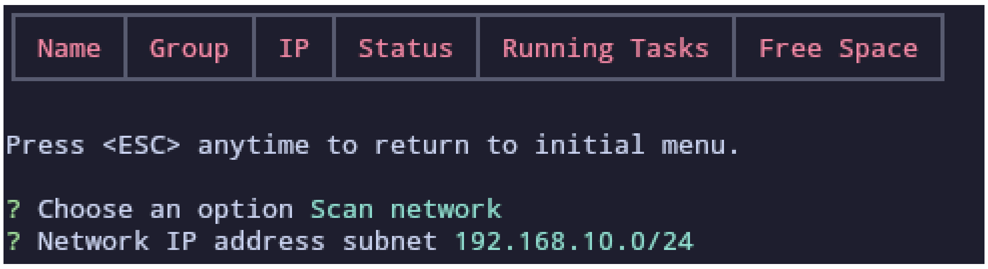


```
~ wasmico-tui
| Name | Group | IP | Status | Running Tasks | Free Space |

Press <ESC> anytime to return to initial menu.

? Choose an option (Use arrow keys)
❯ Add/Rename device
  Remove a device
  Scan network
  Upload file
  Remove file
  Start task
  Stop task
  Pause task
  Resume task
  Save task state
  Upload task state
  Migrate task state
  Restart device
  Refresh devices
```

2. Scan the network that contains the devices

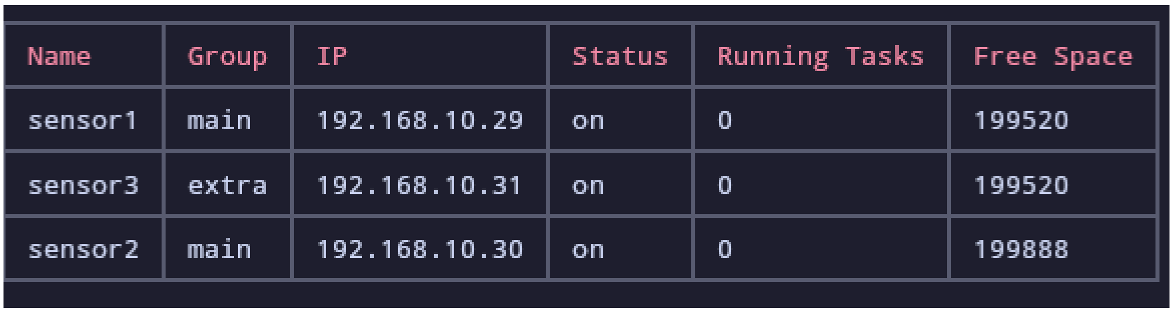


```
| Name | Group | IP | Status | Running Tasks | Free Space |

Press <ESC> anytime to return to initial menu.

? Choose an option Scan network
? Network IP address subnet 192.168.10.0/24
```

3. Rename the devices and attribute groups (2 on "main" group and 1 on "extra" group)



Name	Group	IP	Status	Running Tasks	Free Space
sensor1	main	192.168.10.29	on	0	199520
sensor3	extra	192.168.10.31	on	0	199520
sensor2	main	192.168.10.30	on	0	199888

4. Upload the temperature file to all the devices

Experimental task

Part 1

You have **two sensors** in different rooms. Both sensors can read the temperature and the quantity of carbon monoxide in the air. You start a task that reads the temperature every 5 seconds and prints the moving average (temperature.wasm), and a task that reads the CO quantity every 3 seconds and prints a warning if it's above a threshold (co.wasm) on each sensor.

Pause for at least 30 seconds.

Part 2

While they are running, one of the devices gets low on battery, and you need to replace it with another device. Start **both** tasks in the "extra" device while maintining the state they had in the device that's low on battery.

Pause freely.

Part 3

You need to move one of the devices to a different room and don't want to keep the tasks running while you move it, so you decide to pause them while you move the device.

Pause freely.

Part 4

You don't need those sensors anymore, so you terminate all the tasks and remove all the files.

Appendix B

Validation Questionnaire

This annex includes the questionnaire done to the users after the user experience. This questionnaire was divided into two parts, the part with questions about the knowledge of the user, from question one to question eleven, and the SUS questions part, from question twelve until the end.

Wasmico - Post Experience

This form is a way to get feedback on the experience that was made with the Wasmico tool.

* Indicates required question

1. Which course did/do you attend? *

Tick all that apply.

- ☐ Informatics Engineering
- ☐ Electrical Engineering
- ☐ Computer Science
- ☐ Other: ____________

2. How do you estimate your programming experience? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very experienced

3. How do you estimate your experience with containerization (e.g. Docker or Podman)? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very experienced

4. How do you estimate your experience with orchestration tools (e.g. Kubernetes or Docker Compose)? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very experienced

5. How do you estimate your experience with Web Assembly? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very experienced

6. How familiar are you with the concept of Internet of Things? *

Mark only one oval.

	1	2	3	4	5	
Unk	◯	◯	◯	◯	◯	Very familiar

7. How often do you use microcontrollers? *

Mark only one oval.

	1	2	3	4	5	
Nev	◯	◯	◯	◯	◯	Frequently

8. Select all the devices in this list that you've worked with.

Tick all that apply.

- ☐ Arduino
- ☐ ESP8266 / ESP32
- ☐ Raspberry Pi Pico
- ☐ Other: ____________________

9. Select all the tools in this list that you've worked with.

Tick all that apply.

- ☐ Arduino IDE
- ☐ PlatformIO
- ☐ Other: ____________________

10. Select all the containerization tools for microcontrollers that you've worked with.

Tick all that apply.

- ☐ Toit
- ☐ Golioth
- ☐ Femto-Containers
- ☐ Other: ____________________

11. If you've used any of the tools in the last list, do you think it was easier to use Wasmico compared to the tools you've used?

Mark only one oval.

	1	2	3	4	5	
Muc	◯	◯	◯	◯	◯	Much easier

Wasmico experience

12. I think that I would like to use Wasmico frequently when working with microcontrollers. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

13. I found Wasmico unnecessarily complex. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

14. I thought Wasmico was easy to use. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

15. I think that I would need the support of a technical person to be able to use Wasmico. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

16. I found the various functions in Wasmico were well integrated. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

17. I thought there was too much inconsistency in Wasmico. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

18. I would imagine that most people would learn to use Wasmico very quickly. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

19. I found Wasmico very cumbersome to use. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

20. I felt very confident using Wasmico. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

21. I needed to learn a lot of things before I could get going with Wasmico. *

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

22. Observations