The Journal of Systems and Software 214 (2024) 112081

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

WASMICO: Micro-containers in microcontrollers with WebAssembly ☆

Eduardo Ribeiro [a], André Restivo [a,b], Hugo Sereno Ferreira [a,c], João Pedro Dias [a,*]

[a] DEI/FEUP, Rua Dr. Roberto Frias, Porto, Portugal
[b] LIACC, Rua Dr. Roberto Frias, Porto, Portugal
[c] INESC TEC, Rua Dr. Roberto Frias, Porto, Portugal

ARTICLE INFO

Keywords:
Internet-of-Things
Containers
WebAssembly
Microcontroller
DevOps

ABSTRACT

The Internet-of-Things (IoT) has created a complex environment where hardware and software interact in complex ways. Despite being a prime candidate for applying well-established software engineering practices, IoT has not seen the same level of adoption as other areas, such as cloud development. This discrepancy is even more evident in the case of edge devices, where programming and managing applications can be challenging due to their heterogeneous nature and dependence on specific toolchains and languages. However, the emergence of WebAssembly as a viable solution for running high-level languages on some devices presents an opportunity to streamline development practices, such as DevOps. In this paper, we present WASMICO — a firmware and command-line utility that allows for the execution and management of application lifecycles in microcontrollers. Our solution has been benchmarked against other state-of-the-art tools, demonstrating its feasibility, novel features, and empirical evidence that it outperforms some of the most widely used solutions for running high-level code on these devices. Overall, our work aims to promote the use of well-established software engineering practices in the IoT domain, helping to bridge the gap between cloud and edge development.

1. Introduction

Over the past few decades, traditional computational system software development, deployment, and maintenance have been revolutionized. Advancements in virtualization and software containerization have streamlined development processes and enabled continuous integration and deployment practices. The fundamental practices of the DevOps philosophy[1] aim to "reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" (Zhu et al., 2016), but they are not directly applicable to the IoT domain (Lwakatare et al., 2016; Baccelli et al., 2018; Zandberg and Baccelli, 2021), despite being widely used in other software development domains (Erich et al., 2017). Workarounds such as simulating and emulating hardware devices have been used as a partial solution to the lack of automation in IoT software development tasks, enabling quick replacement of running software without manual or multi-step processes (Dias et al., 2019). However, such workarounds may only partially consider the reality of the target device and deployment environment and often rely on leaky abstractions of hardware features.

Meanwhile, we are witnessing an ever-increasing number of IoT devices, and most do not benefit from these advances. This is mainly due to the very nature of the equipment: they are restricted and constrained devices with a small memory size and low computational power, typically present in the most peripheral layer. They are usually tasked with simple roles, such as the collection of data from their physical environment (*e.g.*, the periodic measurement of the temperature in a room) or performing a simple action in the physical environment in which they are inserted (*e.g.*, opening a door remotely through a mobile application). For example, the ESP32 microcontroller, a commonly used IoT edge device, only has around 520 KB of RAM and can have about 16 MB of flash memory (Hübschmann, 2020).

Typically, these devices run a lightweight operating system with only essential features due to their limited computational resources. In some cases, application code is executed directly on the hardware, while in other cases, it runs on top of a lightweight real-time operating system (RTOS). Updating the device firmware with a code update can be slow, often requiring a complete reboot (Baccelli et al., 2018). Nonetheless, there are use cases where it is necessary to run non-predetermined logic before deploying the device.

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail address: jpmdias@fe.up.pt (J.P. Dias).

[1] While DevOps include other aspects beyond technical ones (*e.g.*, culture), this work only focus on the technical aspects.

https://doi.org/10.1016/j.jss.2024.112081
Received 29 July 2022; Received in revised form 19 April 2023; Accepted 22 April 2024
Available online 24 April 2024
0164-1212/© 2024 Elsevier Inc. All rights reserved.

Thus, working with these devices is typically cumbersome, and programming is, in some cases, only possible through closed-source toolchains that are vendor- and device-specific. These characteristics limit the adoption of widely used software engineering tools and approaches in the IoT scope. Currently, no established solution encompasses the entire software development lifecycle in this context, as highlighted in recent literature (Dias et al., 2022).

This stark contrast between traditional software development and firmware development for microcontrollers and other embedded devices can be attributed to several factors: (1) developers are often limited to using only one programming language, such as C or C++, due to hardware constraints, (2) deploying new code can be a tedious and error-prone process that involves connecting to the device's serial port via USB (or using an over-the-air (OTA) mechanism if available), programming the device using the official flash tool, and verifying if the process was successful by monitoring the serial port, and (3) update cycles can be slow due to the limited resources of the device and the need for manual intervention. These factors collectively result in a poor developer experience when working with embedded systems.

One of the main consequences of these limitations is the difficulty in implementing opportunistic computing, which involves dynamically allocating computation across available resources. In other words, it is challenging to update the code or logic of a device on-demand, over-the-air, in a process that should take a few seconds at most.

As of now, virtualization (Virtual Machines or VM) and containerization are not established solutions for constrained devices. This means that the benefits that these concepts could bring to the IoT scope, such as code portability through hardware/OS abstraction, isolation, security, and over-the-air updates, cannot be fully utilized. This makes it more difficult to implement common DevOps processes, such as continuous integration and continuous deployment (CI/CD), as there is no straightforward way to automatically deploy new code to all devices.

This work introduces WASMICO, a comprehensive solution for managing the entire software development lifecycle of micro-containers on low-end, resource-constrained IoT devices. WASMICO leverages the Wasm3 interpreter and the FreeRTOS operating system to enable the deployment and concurrent execution of WebAssembly tasks. With this micro-containerization platform, developers can write programs in various programming languages, compile them into WebAssembly using existing tools, and remotely upload, start, and manage all tasks via an HTTP API and command-line interface. WASMICO also provides abstractions for device capabilities such as temperature sensing or lightbulb control, allowing developers to use them in higher-level specifications. This solution addresses the challenges of deploying and managing software on constrained devices, such as limited programming language support, slow update cycles, and closed-source vendor-specific toolchains. With WASMICO, DevOps processes like continuous integration and deployment can be applied to IoT devices, enabling rapid development and deployment of applications in this space.

To validate and evaluate the effectiveness of our solution, we carried out two main phases. In the first phase, we aimed to verify the correctness of the developed system to ensure that there were no potential bugs or flaws. We conducted various tests and experiments using physical IoT devices such as the ESP32 microcontroller to test the system's functionality thoroughly. The second phase evaluated the solution's performance, including computation and memory requirements. We conducted several experiments to generate relevant metrics and statistics and compared our solution against other state-of-the-art alternatives to benchmark its performance and feasibility.

Our evaluation reveals that WASMICO outperforms other state-of-the-art solutions in terms of efficiency and resource usage, thus showcasing its feasibility. The tool's ability to execute WebAssembly tasks concurrently on resource-constrained IoT devices, using a micro-containerization platform built on top of the Wasm3 interpreter and the FreeRTOS operating system, allows for better performance compared to existing alternatives. Additionally, the platform's HTTP API and command-line interface enable remote upload, start, and management of tasks, making firmware development and deployment more akin to traditional software development.

The remainder of this paper is organized as follows. In Section 2, we review the relevant literature from both scientific and gray sources. Section 3 describes the WASMICO tool, including the approach and implementation details, while Section 4 lists and explains the supported operations. In Section 5, we present the experiments we conducted to evaluate the tool's performance and discuss the results. Section 6 examines some of the validation threats to our work. Finally, in Section 7, we provide concluding remarks and highlight areas for future research.

2. Related work

To gain a better understanding of current solutions for running programs on microcontrollers, we conducted a review of both scientific and gray literature resources. Our analysis covered a range of approaches, including language interpreters and runtime environments that enable high-level languages to run on microcontrollers and solutions that allow developers to run tasks in a container-like fashion. We organized these solutions into three main categories, each of which is described in the following subsections: (1) *Runtime Environments*, which presents runtimes capable of running high-level languages on microcontrollers; (2) *WebAssembly Runtimes*, which enable microcontrollers to run WebAssembly binaries; and (3) *Micro Containers*, solutions that provide container-like functionality for running tasks on microcontrollers. It is worth noting that these categories are not mutually exclusive and that many solutions may fall into multiple categories.

2.1. Runtime environments

MicroPython (MicroPython, 2023a; RIOT OS, 2023; Tollervey, 2017) is a lean and efficient implementation of the Python 3 programming language designed to run on microcontrollers. It comes with a subset of the standard Python libraries and modules that allow developers to interact with low-level hardware, such as the devices' I/O pins. Code can be executed interactively through a prompt called the REPL or by uploading and running files on the device's built-in file system, using tools like `ampy` (Adafruit Industries, 2021). MicroPython supports a wide range of devices, has a garbage collector for heap memory management, and provides multi-threading support through libraries like `uasyncio` (MicroPython, 2023b).

JerryScript is a lightweight JavaScript engine designed for resource-constrained devices, such as microcontrollers, with limited RAM (less than 64 KB) and ROM (less than 200 KB) for engine code. JerryScript allows users to access the device's peripherals on the developed applications through JavaScript abstractions.

Toit (2023a,b) is a micro-containerization solution specifically designed to run applications written in the Toit language (Toit, 2023b) on IoT end devices, supporting the ESP32 microcontroller. It offers a platform to create and deploy containers with IoT applications, which can be managed and orchestrated through a client API, a web dashboard, and a command line interface. To configure a device to use Toit, one must install the Toit firmware on the microcontroller. From then on, all communication can be conducted remotely and wirelessly over-the-air (*e.g.*, using the Toit cloud). The remote interaction between the client side and the Toit firmware installed on the devices is conducted through the built-in WiFi or any other added communication module, such as NB-IoT. The solution also handles any faulty application, ensuring that only the application with bugs/crashes is affected while everything else running on the device, like the Toit firmware and other containers, remains unaffected. This makes updating and deploying new code risk-free, as it guarantees that any consequences are contained within the application environment. This approach enables developers to set up a continuous delivery pipeline and reduce the deployment of new code from several minutes to just a few seconds.

Golioth (2023) is a commercial IoT development platform that offers scalability. It is similar to Toit in that it requires a firmware component installed on the IoT device and has a cloud component, enabling developers to build, deploy and manage applications remotely easily. Golioth is built on top of ZephyrRTOS (ZephyrRTOS, 2023), an RTOS designed for IoT devices with broad support among hardware vendors. Developers can communicate with deployed devices remotely via a web console, REST API, or CLI, retrieving readings and data from the hardware. Like Toit, it supports over-the-air live code updates to IoT devices, enabling developers to iterate and execute various code versions on deployed hardware at the desired location. This provides the flexibility to modify the firmware throughout the product's lifetime. Golioth is compatible with several devices (*i.e.*, any device compatible with ZephyrRTOS), and communication can be done through NB-IoT/Cat-M1, Wi-Fi, or Ethernet, using the CoAP protocol.

These runtime environments enable the deployment and execution of code on resource-constrained IoT devices without requiring device resets. They also prioritize the development lifecycle by offering command-line tools that integrate with DevOps frameworks and tools. However, it is important to note that these solutions do not provide the same level of isolation and portability as containers, which may make them unsuitable for certain use cases. Specifically, they may not be suitable for use cases requiring high-security levels or the ability to move code between different devices and environments.

2.2. WebAssembly runtimes

Wasm3 (Wasm3, 2023; Wasm3-arduino, 2023) is a fast and lightweight WebAssembly (WebAssembly, 2023; Haas et al., 2017) interpreter compatible with various types of microcontrollers such as the ESP32, Arduino, and others. One of its advantages over a solution like Toit is that it allows developers to compile widely used, general-purpose programming languages such as C++, Go, Rust, TypeScript, and others to WebAssembly, instead of requiring the use of a specially built language, which may present a learning curve for the user. Wasm3's requirements for memory and computational power are minimal, requiring only around 64 KB of flash memory and 10 KB of RAM for minimal functionality. However, Wasm3 provides fewer security and isolation guarantees than a more comprehensive solution like Toit, nor does it include any features for managing the software lifecycle on these devices.

Gurdeep Singh and Scholliers (2019) conducted a study on the feasibility of using WebAssembly to program IoT devices compatible with the Arduino framework. Their work focused on extending the standard WebAssembly VM with features such as safe live code updates, remote debugging, and programmer-configurable Arduino modules. The outcome of their study is WARDuino, which outperforms most language interpreters that can run on IoT devices, such as MicroPython (MicroPython, 2023a; RIOT OS, 2023) and JerryScript (Gavrin et al., 2015). WARDuino supports a wide range of programming languages since it uses WASM, which can be compiled from many languages. It also reduces the development and update cycle to just a few seconds.

The key advantages of using WebAssembly in IoT devices are: (1) compared to interpreted languages, its superior performance offers faster execution times and lower memory usage, (2) it is (by default) sandboxed execution environment that provides some degree of isolation and security, (3) many programming languages can be compiled into WebAssembly, and (4) it can run on various devices and architecture.

2.3. (Micro) containers

Femto-containers (Zandberg and Baccelli, 2021) is a solution developed for IoT devices that enables the containerization and secure deployment of software modules over low-power networks. It is built on top of RIOT (RiotOS, 2023), a low-power IoT operating system supported by several microcontrollers, and supports multi-threading, which is fundamental to the architecture of the approach. Each femto-container consists of a hardware-agnostic lightweight VM that runs in a separate thread managed by the OS's scheduler. The VM is sandboxed and uses a generic interface to connect to hardware I/O. The execution of the code inside each VM uses eBPF virtualization, with an rBPF interpreter and the eBPF instruction set (Fleming, 2017; Zandberg and Baccelli, 2020). The approach is event-based, meaning that the launch and execution of femto-containers are done on-demand when a specific event calls for it, such as network packet reception or a sensor reading input. The authors benchmarked the solution against other tools and approaches in terms of performance, speed, flash, and RAM overhead, and the results are overall very positive. The memory overhead is considered negligible, being less than 10% on both flash and RAM, and the installation and execution of femto-containers are reasonably optimized operations.

Baccelli et al. (2018) focuses on remote over-the-air (OTA) code updates to IoT edge devices. Their solution is based on the RIOT (RiotOS, 2023) operating system and the JavaScript programming language. It consists of (1) middleware that binds a lightweight script engine, like MicroPython (MicroPython, 2023a; RIOT OS, 2023) or JerryScript (Gavrin et al., 2015), to the key APIs of the OS, enabling scripts to interact with the device's hardware; (2) a Web resource that exposes a REST API, which allows the cloud to interact with the device, so that new code can be sent over the air, and data and logs can be extracted from running apps; (3) a security component that ensures the integrity of the OTA-received scripts and maintains transport layer security and standard DTLS; and (4) device registration and discovery mechanisms that can be implemented in the cloud via standard CoAP (Bormann et al., 2012) resource discovery. A benchmark of a proof-of-concept implementation of the proposed approach shows that it runs on low-end IoT equipment with memory as low as 32 KB. However, the total overhead can be considered significant, particularly regarding flash memory, but it is still affordable for most IoT devices.

The Velox VM (Tsiftes and Voigt, 2018) provides a secure execution environment for IoT applications written in Scheme, a high-level LISP programming language, and Cyclus, which has a C-like syntax, on resource-constrained edge devices running the Contiki OS (ContikiOS, 2023). It allows developers to configure bandwidth, energy, and memory thresholds for each application and define actions (*i.e.*, policies) when the thresholds are exceeded. The VM's core components include a policy enforcer and a preemptive scheduler that selects the application to be executed at a given time. If an application violates a policy or exceeds a threshold, the scheduler will only pick it up when the policy is fulfilled again. Initial benchmarks show that Velox's bytecode is significantly more compact compared to other solutions (PyMite (Hall, 2011), TakaTuka (Aslam et al., 2010), and Darjeeling (Brouwers et al., 2009)), occupying less memory, and it can enforce thresholds, including energy-related ones.

VMs and containers provide numerous advantages for deploying applications on IoT devices, such as efficient resource usage, portability, easy deployment, security, and policy enforcement. However, some existing solutions prioritize containerization benefits over integration with DevOps workflows, potentially posing challenges for teams seeking to implement DevOps processes in their IoT development. Moreover, many of these solutions rely on interpreted languages, which may lead to a performance penalty on devices with limited resources.

2.4. Summary

After reviewing the literature on virtualization, containerization, and execution runtimes for IoT devices, it is clear that there are numerous solutions available, each addressing different aspects and challenges. For instance, the Velox VM (Tsiftes and Voigt, 2018) emphasizes security and ensuring that applications run efficiently and within expected parameters, while solutions like Baccelli et al.'s work (Baccelli

Table 1
Comparison of the *desirable features* for the lifecycle management of software (*i.e.*, DevOps) for IoT devices. The ● stands for fully support and automated, ⦿ stands for partial support and not automated by default, and ○ stands for no support. In the update method column, "cloud" means that it supports OTA updates but they are cloud-dependent.

Platform	Develop / Test	Deliver / Change	Monitor / Operate	Update Method
WASMICO	●	●	●	OTA
MicroPython	●	⦿	○	USB
JerryScript	●	⦿	○	USB
Toit	●	●	●	CLOUD
Golioth	●	●	●	CLOUD
Wasm3	●	⦿	○	USB
WARduino	●	⦿	○	USB
Femto-conts.	●	⦿	○	OTA
Baccelli et al.	●	⦿	○	OTA
Velox VM	●	⦿	○	OTA

et al., 2018) focus on remote updates and deployment of new containers, as well as language interpreters such as MicroPython (MicroPython, 2023a; RIOT OS, 2023) and JerryScript (Gavrin et al., 2015), which enable devices to execute code snippets on demand in specific languages.

Our approach to container management and code updates over-the-air is similar to projects such as Toit (Toit, 2023a), Golioth (Golioth, 2023), and WARDuino (Gurdeep Singh and Scholliers, 2019). However, these solutions have limitations and drawbacks, such as: (1) Toit requires developers to use its proprietary programming language, (2) both Toit and Golioth depend on a cloud-based platform for deployments, management, and operation, thus limiting its usage on systems without direct access to the Internet, (3) Toit only runs on ESP32 devices, and Golioth only officially supports 4 microcontrollers, and (4) only Golioth supports direct integration – via its cloud platform API – in continuous deployment pipelines. Consequently, there is a gap in the literature for a solution that encompasses the full container management lifecycle and facilitates code updates and deployments over-the-air (OTA) while supporting popular general-purpose programming languages and a plethora of devices. The Wasm3 (Wasm3, 2023; Wasm3-arduino, 2023) interpreter is a promising candidate, as it enables the compilation of multiple languages into WebAssembly and runs on more than 15 different types of devices.[2]

From the perspective of DevOps practices, most of the reviewed approaches have significant limitations when it comes to supporting Delivery and Change, such as Continuous Integration and Continuous Deployment. In most cases, the solutions are only capable of building artifacts, such as binaries, which must then be flashed into the device either via Over-the-Air or Serial over USB, often relying on proprietary toolkits and requiring device full resets (resulting in mandatory downtime). Furthermore, they do not offer any mechanisms for monitoring, such as telemetry about RAM and storage, or operating the code that is running on the device, such as starting, pausing, or removing it. Currently, only Toit and Golioth provide solutions that address all of these practices, but they also suffer from the limitations and drawbacks mentioned above. Table 1 presents a side-by-side comparison of these solutions.

3. WASMICO approach

Our proposal includes all the essential tools, including firmware, required to develop a comprehensive micro-containerization solution for microcontrollers. WASMICO supports container management and its complete lifecycle, including remote deployment and over-the-air updates. Moreover, it provides a user-friendly client-side API and a command line interface that allows developers to monitor and perform various operations on the deployed containers easily. Our solution is open-source, and the code is available at https://github.com/SIGNEXT/WASMICO.

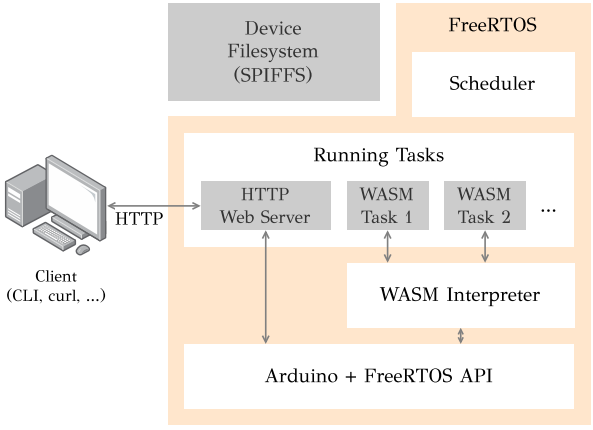

Fig. 1. Architecture overview.

WASMICO utilizes the Wasm3 VM (Wasm3, 2023; Wasm3-arduino, 2023), a lightweight WebAssembly VM, to execute the code. This VM was selected due to its compatibility with a broad range of devices, and because many commonly used programming languages can be compiled to WebAssembly. The developer can access each device's unique features and capabilities, such as temperature readings or controlling switches and lightbulbs, through high-level abstractions that obscure low-level circuit details, device ports, and sensors.

Fig. 1 provides a diagram that displays the various components of the system. The tool was developed utilizing the Arduino (2023) framework on top of the official Espressif IoT Development Framework (ESP-IDF) (Espressif Systems, 2023a) for ESP32 microcontrollers. ESP-IDF is designed around the widely used FreeRTOS (FreeRTOS, 2023), a real-time microcontroller operating system. The impact of each component of used on WASMICO was evaluated in terms of resource usage and impact of code execution time being this data presented and discussed in Section 5.2.3, but, in sum, WASMICO is slightly slower than WASM running on a default configuration and has a slightly lower RAM usage.[3]

The subsequent subsections describe each system component, their respective roles, and their interaction with the rest of the system.

3.1. Architectural modules

HTTP web server. The solution includes an HTTP server that offers endpoints representing different operations supported by the system, such as uploading and creating new tasks. This is how the client communicates with the device and the platform, and how the tool enables processing operations and over-the-air updates without requiring a physical connection.

Device filesystem. When uploading new WASM files, their content is permanently stored on the device's filesystem using SPIFFS (Espressif Systems, 2023b), a simple file system based on flash memory that is compatible with the ESP32 microcontroller. As flash memory is non-volatile, the WASM files remain stored on the device even after a full reboot. While we chose SPIFFS for its simplicity, it would be ideal to have the ability to store the WASM files in any other ESP32 compatible

[2] Wasm3 supported devices, https://github.com/wasm3/wasm3/blob/main/docs/Hardware.md

[3] The reason of the slightly lower RAM usage is further discussed in the same subsection, Section 5.2.3.

non-volatile store, using any other type of filesystem (such as LittleFS[4]) and make it possible to use any available storage (such as an external SD Card). By abstracting this storage layer, the solution would be compatible with more devices without requiring modifications.

Freertos scheduler. FreeRTOS enables concurrent execution of multiple threads or tasks, and WASMICO utilizes this feature to run various WebAssembly tasks alongside the server. The FreeRTOS scheduler implements a fixed-priority preemptive scheduling policy and round-robin time-slicing of tasks with equal priorities. Since all tasks created by WASMICO have the same priority, the scheduler ensures that each thread takes turns executing and switches between tasks on each tick interrupt.

WASM interpreter. To execute WebAssembly routines, WASMICO uses the Wasm3 interpreter (Wasm3, 2023; Wasm3-arduino, 2023). This interpreter provides an API to create a WebAssembly runtime object, parse and load the module from the saved WASM file, and call the desired function from the module. When a start task operation is received, the HTTP server creates a new FreeRTOS task and calls the necessary Wasm3 API to start the execution of the respective WASM task. Moreover, Wasm3 allows the creation of new functions that can be imported and used by the client code compiled into WebAssembly. Although WASM routines typically lack access to low-level functions, the WASM interpreter serves as a middle layer between high-level WebAssembly routines and low-level functions of the Arduino and FreeRTOS APIs. It also abstracts these APIs to control their use in higher-level specifications while exposing device capabilities, such as peripherals.

Arduino and FreeRTOS API. This layer provides access to the device's hardware and low-level operating system's capabilities. Examples of functions from these APIs include `digitalWrite`, which writes a value to a device pin, `delay`, which pauses the execution of the program for a specified period, and `Serial.print`, which prints a message to the serial port. The use of this layer is controlled in the high-level WebAssembly routines that the client specifies and uploads to the device through the link functions defined at the WASM interpreter level.

CLI. To simplify the platform usage and streamline the HTTP server operations, a minimalist command-line interface (CLI) was implemented. Clients can conveniently use the CLI to create and manage their tasks.

3.2. Code file structure

Each C++ code file must follow a concise structure to ensure proper execution on the device. The file must contain the functions outlined in the following paragraphs.

Importing link functions. Each code file in C++ must begin by importing all the required link functions declared and made available by the platform through the WASM interpreter. As previously mentioned, Wasm3 enables the creation of functions that can be linked to the WASM modules being executed, making the WASM environment act as an intermediary layer between the high-level client-specified programs and the device's low-level capabilities. To make these functions available in the programs, they must be imported and declared.

`_start` function. The main routine of the file should be contained within the `_start` function. Although this function can technically have any structure, it is recommended to follow an Arduino-like form with a `_setup` function executed in the beginning, followed by an infinite loop of `_loop` function calls. These two functions can be declared in the file like regular functions. Typically, the `_setup` function should be structured to receive a potentially restored state through a resume operation.

[4] LittleFS, https://github.com/littlefs-project/littlefs.

`_pause` function. Additionally, the file should contain a `_pause` function, which is executed when a pause operation is initiated. To ensure that the state stored by the pause operation can be accessed within this function, it should be declared as a global variable in the file. Furthermore, the main routine should be structured in such a way that the state can be modified and updated as needed.

3.3. Generation of WASM applications

We chose C++ as our programming language for WASM compatibility, and this section will focus on using C++ and the tools required to deploy the code. However, the various steps of the WASM compilation process are likely to be similar across different compilation tools and programming languages.

To compile the C++ programs to WASM, we utilized `wasic++`, which acts as a wrapper on top of `clang++` (Clang, 2023). `wasic++` is capable of compiling C++ projects to WebAssembly + WASI. WASI (Anon, 2023), also known as the WebAssembly System Interface, is a modular collection of standardized APIs for WASM projects. The `wasic++` tool takes a C++ file as input and produces the resulting WebAssembly file, ready to be executed in any WASI WebAssembly runtime or the browser.

To run the resulting application correctly, the `wasic++` command requires specific parameters to be specified. These parameters include the `stack-size`, which specifies the allocated memory in bytes for the task, the `initial-memory` parameter, which denotes the minimum memory space allocated for the task, and the optional `max-memory` parameter, which sets the maximum memory space. Both `initial-memory` and `max-memory` parameters have a minimum value of 65536 bytes and must be a multiple of that. The `reservedInitialMemory` parameter, which should have the same value as `initial-memory`, may also be provided. Finally, the `allow-undefined-file` flag needs to be set to the name of a file containing all the link functions declared by the WASM interpreter and imported and used in the client program.

After compiling the program and generating the WASM file, we utilized two tools to optimize the resulting file. The first tool is `wasm-opt` (wasm-opt, 2023), which applies various optimizations to the file, and the second tool is `wasm-strip` (wasm-strip, 2021), which removes any unnecessary sections from a WebAssembly binary file.

The resulting output is a streamlined and efficient WebAssembly routine suitable for upload and execution on WASMICO. Listing 1 provides an example script for compiling a C++ program named **app.cpp** and generating a corresponding WASM file named `app.wasm`. The string **app.syms** should be replaced with the file name specifying the names of all imported link functions.

```
> wasic++ -Os \
  -z stack-size=4096 \
  -Wl,--initial-memory=65536 \
  -Wl,--allow-undefined-file=app.syms \
  -Wl,--strip-all -nostdlib \
  -o app.wasm \
  app.cpp

> wasm-opt -O3 app.wasm -o app.wasm
> wasm-strip app.wasm
```

Listing 1: Script to compile a C++ program file to WebAssembly.

3.4. Known limitations

Some limitations exist in the current implementation of WASMICO but do not significantly impact the tool's usage. These limitations mainly affect its performance and memory consumption and are described in the following paragraphs.

Deallocation of heap memory. The first limitation concerns the deallocation of resources when stopping a task. As previously mentioned, a FreeRTOS thread is created when a task starts, which calls the necessary functions from the WASM interpreter to run the WASM routine. The memory used as the thread's stack is dynamically allocated from the FreeRTOS heap and automatically released when the task stops. However, after a task stops and its memory is freed, the free size of the FreeRTOS heap is slightly smaller than its previous value, indicating that not all task resources are being released. This phenomenon may be caused by objects and pointers created through the WASM interpreter that are used to create new WASM runtimes, parse, and load WASM modules and are not properly freed when the task stops. The same issue also occurs when running a task on the base Wasm3 interpreter on top of FreeRTOS without any modifications. This limitation mainly affects the tool's memory consumption and performance and may eventually lead to the depletion of memory available for task allocation.

Listing 2 presents a small experiment that we conducted to illustrate this limitation. In this experiment, we started and stopped a previously uploaded WASM task while monitoring the FreeRTOS free heap size at three stages: before starting the task, during its execution, and after stopping it. As expected, the heap size reaches its lowest value during the task's execution. However, after stopping the task, the free size of the heap is slightly lower than it was before starting the task.

```
> micro heap
{ free_heap_size: 199364,
  largest_free_block_size: 113792,
  status_code: 0 }
> micro start test1.wasm
{ status_code: 0,
  message: 'task started successfully' }
> micro heap
{ free_heap_size: 141772,
  largest_free_block_size: 113792,
  status_code: 0 }
> micro stop test1.wasm
{ status_code: 0,
  message: 'task stopped successfully' }
> micro heap
{ free_heap_size: 191924,
  largest_free_block_size: 113792,
  status_code: 0 }
```

Listing 2: Experiment showcasing that the start and stoppage of tasks slowly depletes the FreeRTOS heap, as there are some resources that are not freed.

Automatic adaptation of the stack size of a running task. When starting a new task, the allocated stack size must be specified as one of the parameters. Although some mechanisms and operations assist in selecting an appropriate stack size for a specific task, there is currently no feature to dynamically adjust the stack size of a task without stopping its execution and providing a new value. To modify the stack size of a task in WASMICO, the task must first be stopped, its details updated (such as its reserved stack size), and a restart requested, making it a manual process. For instance, in the Toit solution, each task begins with an initial stack size of 4 KB, which can grow on demand.[5] WASMICO should provide a similar mechanism where the stack size can be automatically increased or decreased based on the resource consumption of the task and the available resources, without requiring manual intervention.

[5] As described in https://github.com/toitlang/toit/discussions/23.

Overhead created from task creation setup. When a task creation operation is requested, a setup process must be performed before executing the WebAssembly task. This process involves creating a Wasm3 runtime object, opening the corresponding WASM file from the device's filesystem, parsing and loading the WASM module, and binding the declared link functions. Upon receiving a request to start a task, a new FreeRTOS thread is created, and the thread executes the described setup before calling the main WASM routine, *i.e.*, the `_start` function. However, performing the setup inside the new thread can negatively impact its memory consumption. If we move the setup process outside the thread creation, such that the thread only calls the WASM routine, the occupied heap size could potentially decrease, allowing the task to execute successfully with a smaller reserved stack size.

4. Supported operations

This section outlines the operations supported by WASMICO and presents the CLI usage. We have categorized the operations into two groups: (1) operational, which is related to the lifecycle of tasks such as creating, stopping, pausing, and resuming tasks; and (2) telemetry, which provides information about task performance and resource usage, as well as the device's overall functioning.

To deploy programs to devices in the network using the CLI, the CLI must be aware of the available devices and their IP addresses. There are two primary ways to create a registry of devices: (1) using the CLI itself with the command `wasmico config <deviceName> <deviceIP>`, where `<deviceName>` is an arbitrary string that identifies the device or (2) by manually editing the `devices.txt` file with one `<deviceName> <deviceIP>` pair per line.[6] The command `wasmico devices` can be used to list all registered devices, and `wasmico select <deviceName>` to choose a specific target for deployment.

4.1. Task upload

The task upload operation enables the client to upload a new WebAssembly file containing a task to be executed on the device. In addition to specifying the WASM file to be uploaded, this operation requires four additional parameters to be specified:

`reservedStackSize`: This parameter specifies the desired stack size, in bytes, for the FreeRTOS task that will execute the WASM routine. The developer estimates the stack size and depends on various factors such as the program size, variables used, functions called, and their call depth. One way to estimate an appropriate stack size for a task is to initially assign it a large stack size, run the task, and observe its performance and memory consumption using other platform operations. Based on the results, the stack size can be reduced until a desirable value is reached, which allows the task to run without errors or crashes while not consuming unnecessary memory that can be used for other tasks. The minimum recommended value for this parameter is 768 bytes, which is the minimum stack size for a FreeRTOS task.

`memoryLimit`: the `stack-size` parameter used in the compilation of the routine to WebAssembly, specified in bytes. It is used when starting the execution of the task, to indicate to the Wasm3 interpreter the memory limit of the WASM program, which is equal to the stack size used in the WASM compilation of the task. Generally, the reserved stack size needs to be bigger than the memory limit.

[6] As a potential future improvement, the CLI could perform auto-discovery of WASMICO-ready devices in the Local Area Network.

reservedInitialMemory: the `initial-memory` parameter used in the compilation of the routine to WebAssembly, specified in bytes. It must have a minimum value of 65536 bytes and always be a multiple of this number. Although the platform does not currently use this parameter when creating WASM tasks, it is still stored on the device.

liveUpdate: This flag is only necessary if an older version of the task is currently being executed on the device. It instructs the WASM interpreter to stop and delete the old task after the new version has been uploaded and replace it with the latest version. This eliminates the need for a start operation after uploading a new version of the WASM file, resulting in a faster development cycle.

The parameters `reservedStackSize`, `memoryLimit`, and `reservedInitialMemory` are stored on the device and applied when a task start operation is requested.

Once the CLI sends a request to the device, it should receive a response indicating whether the file was successfully uploaded or if an error occurred. If the operation is successful, the WASM file will be stored permanently in the device's flash memory and persist on it.

Using the developed CLI, the upload of a new task can be done by executing the command present in Listing 3. The command executes the upload operation of a file named `app.wasm`, indicating to the platform that the WASM task that results from the execution of this file should have 10000 bytes of stack size, 65536 bytes of initial memory and 4096 bytes for the memory limit.

```
> wasmico upload app.wasm 10000 65536 4096
```

Listing 3: WASMICO CLI file upload command.

4.2. Edit task details

This operation enables the client to modify the details of a previously uploaded task without re-uploading it, as long as its code has not changed. In addition to the task name, it accepts the same parameters as the upload operation, namely, `reservedStackSize`, `reservedInitialMemory`, `memoryLimit`, and `liveUpdate`. After the operation, the new values are saved on the device, and the task is restarted with the updated parameters, depending on whether it was already running and whether the `liveUpdate` flag was set to true. The client will then receive a response confirming whether the operation was successful.

The command at Listing 4 executes the edit task details operation for a file named `app.wasm` that is expected to be on the device. The command specifies that the resulting WASM task from executing the file should have a stack size of 10000 bytes, an initial memory of 65536 bytes, and a memory limit of 4096 bytes.

```
> wasmico edit app.wasm 10000 65536 4096
```

Listing 4: WASMICO CLI edit task details command.

4.3. Task creation

This operation enables the user to initiate the execution of a previously uploaded WASM task on the device, as specified in Listing 5. The operation takes only one parameter, the task's name, for example, `app.wasm`. The stack size and memory limit used for the task are determined by the values specified during the task upload or updated through the edit task details operation. The sequence diagram illustrated in Fig. 2 depicts all the steps involved in a task creation request and the interaction among the various components.

```
> wasmico start app.wasm
```

Listing 5: WASMICO CLI task creation command.

When starting a new WASM task, the system first checks if enough resources are available on the device to execute the task, using the `reservedStackSize` value specified when the task was uploaded. The task is only executed if its reserved stack size is less than or equal to the size of the largest free continuous block available in the heap memory. If there is sufficient memory, a new FreeRTOS thread is created. This thread opens and reads the contents of the specified file and calls the necessary Wasm3 API functions to create a new WebAssembly runtime, parse and load the WASM module with the contents of the WASM program, bind the link functions with the runtime, and run the `_start` function of the program, thus executing its main routine. The program runs concurrently with any other tasks that were previously running, as well as the FreeRTOS thread used for the HTTP server.

Under the hood of this operation, the FreeRTOS task is created using the function `xTaskCreate`.[7] It dynamically allocates only the memory that is strictly necessary to run the required task, thus ignoring the `initial-memory` parameter used when compiling an application to WebAssembly. However, if the stack size demand suddenly increases, there is a risk that insufficient memory was allocated for the task, which can result in a crash.

4.4. Task stoppage

This operation enables users to halt a WebAssembly task by providing the task name as the sole parameter (refer to Listing 6). Upon receiving the command, the FreeRTOS task that runs the WASM routine terminates, and its resources are freed, liberating space on the FreeRTOS heap to create new tasks in the future. After the operation, the server should respond to the client, indicating whether the operation was successful. No information about the task's state is saved, which means that it is impossible to resume the task execution after stopping it. In other words, the task will start from the beginning if it is reinitiated.

```
> wasmico stop app.wasm
```

Listing 6: WASMICO CLI task stoppage command.

4.5. Task deletion

This operation allows for the permanent deletion of WebAssembly files from the device's filesystem. It takes the filename (*e.g.*, `app.wasm`) as its only parameter (see Listing 7) and returns a response to the client indicating whether the operation was successful.

[7] `xTaskCreateStatic` is an alternative method of performing this operation, which instead of dynamically allocating RAM for the task stack, takes the stack as a parameter. Another alternative is the `xTaskCreateRestricted` (or even `xTaskCreateRestrictedStatic`), which makes use of the Memory Protection Unit, or MPU, to create memory-restricted tasks.

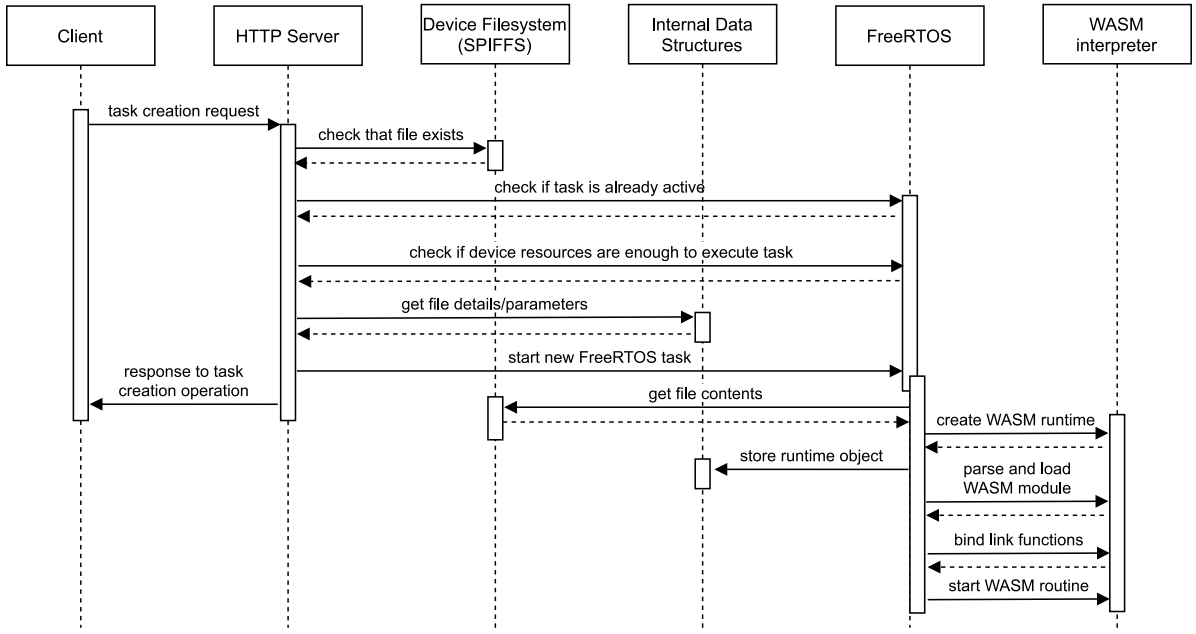

Fig. 2. Sequence diagram of the task creation operation.

```
> wasmico delete app.wasm
```

Listing 7: WASMICO CLI file deletion command.

4.6. Task pause

This operation lets users temporarily pause an ongoing WASM task (see Listing 8). It saves the task's current state to the device memory, allowing the user to resume the task later by restoring the stored state. The operation takes the name of the task (*e.g.*, `app.wasm`) as its only parameter. Once the request is sent, the CLI will print a response indicating whether the operation was successful.

```
> wasmico pause app.wasm
```

Listing 8: WASMICO CLI task pause command.

The sequence diagram depicted in Fig. 3 outlines the stages involved in a task pause request and how the various components interact. When a new WebAssembly task is created using the task creation operation, the Wasm3 API generates a new WASM runtime that is stored in memory for later use. The runtime object contains information about the current execution and state of the WASM task, such as the declared variables and their current values. The pause operation invokes the *_pause* function defined alongside the *_start* function in the uploaded WASM file, reusing the same runtime object. This enables the *_pause* function to access the current state of the task and store it on the device, allowing the task to be resumed later from where it was paused. Once the state is stored, the FreeRTOS thread running the WASM task is terminated, releasing all consumed resources except for the stored state. The state is held in memory until a resume operation is initiated or until the WASM file is deleted from the system.

4.7. Task resume

This operation enables the resumption of a previously paused WebAssembly task, with the only required parameter being the name (*e.g.*, `app.wasm`) of the task (see Listing 9). Upon sending the request, the CLI should respond, indicating whether the operation was successful.

```
> wasmico resume app.wasm
```

Listing 9: WASMICO CLI task resume command.

After storing the current state, the pause operation completely stops the WASM task. The resume operation, similar to the task creation operation, starts a new FreeRTOS thread. This thread calls the Wasm3 API to create a new WASM runtime, parse and load the WASM module with the file contents, and execute the *_start* function. The only notable difference is that the resume operation restores the previously saved state of the application, as shown in Fig. 4. This state can then be processed in any way the user intends.

4.8. List task details

This operation enables the user to list all the WebAssembly files currently stored in the device's file system and their corresponding details, such as their reserved stack size, initial memory, and memory limit (see Listing 10). These values can be specified during the file upload process.

```
> wasmico details
{
  "files": [
    {
      "filename": "example1.wasm",
      "reserved_stack_size": 4000,
      "reserved_initial_memory": 66536,
      "memory_limit": 2048
    },
    {
      "filename": "example2.wasm",
      "reserved_stack_size": 10000,
      "reserved_initial_memory": 66536,
      "memory_limit": 4096
    }
```

```
  ],
  "status_code": 0
}
```

Listing 10: WASMICO CLI list task details command and response.

4.9. List active tasks

This operation lists all active WASM tasks currently running on the device (see Listing 11). It indicates whether each task is paused, and for running tasks, it also returns their stack "high watermark". The stack "high-watermark" is obtained using the FreeRTOS function `uxTaskGetStackHighWaterMark` and represents the amount of unused stack space when the task's stack was at its highest value. A value of 0 typically indicates that the task has overflowed, and a value close to 0 suggests that the task is close to overflowing. This information can be helpful when initially determining an appropriate stack size for a given task. In addition to this information, the operation returns the total number of FreeRTOS threads currently running on the system.

```
> wasmico showactive
{
  "tasks": [
    {
      "filename": "example1.wasm",
      "paused": false,
      "stack_high_water_mark": 675
    },
    {
      "filename": "example2.wasm",
      "paused": true
    }
  ],
  "espressif_num_threads": 11,
  "status_code": 0
}
```

Listing 11: WASMICO CLI *list active tasks* command and example response.

4.10. Get heap info

This operation provides the user with information about the current usage of the FreeRTOS heap and overall memory of the device. Upon execution, it returns two values: (1) the current free heap size, in bytes, and (2) the size of the largest continuous free block on the heap, also in bytes (refer to Listing 12).

```
> wasmico heap
{
  "free_heap_size": 203723,
  "largest_free_block_size": 132921,
  "status_code": 0
}
```

Listing 12: WASMICO CLI get heap info command and example response.

5. Experiments and results

5.1. Experimental setup

This section describes the physical system and environment used to conduct the experiments and the specifications of the used microcontroller. Additionally, it presents the IoT execution environments and containerization tools used as a basis for comparison against WASMICO. A replication package can be accessed on Zenodo (Ribeiro et al., 2022).

5.1.1. System

All experiments were conducted on an *AZ-Delivery ESP-32 Dev Kit C V4* board.[8] The board has 4 MB of flash memory and 520 KB of RAM. It is equipped with a built-in *ESP-WROOM-32* chip, which has two low-power *Tensilica Xtensa LX6* microprocessors running at a frequency of 240 MHz. The board also has built-in wireless connectivity capabilities (2.4 GHz dual-mode Wi-Fi), a TSMC Bluetooth chip, and 40 nm low-power technology.

The host computer used to program and manage the microcontroller for testing was an Apple MacBook from 2020 running the Big Sur operating system. It had an Apple M1 CPU and 16 GB of RAM. In most benchmarked solutions, operations were performed over a local network. The computer and the microcontroller were connected to the same wireless network, and messages were exchanged only locally. The only exception was Toit, which depended on an internet connection to run operations because it was cloud-based.

5.1.2. Benchmarked solutions

This subsection presents the state-of-the-art tools used in the benchmark experiments, covering a range of platforms from low-level solutions closer to native performance to high-level ones with rich features. The validation and benchmarking process involves evaluating these platforms on various parameters.

Arduino. We used the Arduino framework for the benchmark experiments as a point of comparison (Arduino, 2023). The ESP32 microcontroller supports this framework and runs on top of the ESP-IDF, the official development framework for the ESP32. However, it is worth noting that the performance achieved with Arduino on top of ESP-IDF may be worse than that of a device running Arduino in a bare-metal fashion. To minimize this impact, we only used functions from the Arduino API, avoiding those exposed by the ESP-IDF API (Espressif Systems, 2023a). This means that the code has access to the device's features (such as printing information to the serial port, pausing the program for a specified amount of time with functions like *delay*, and reading and writing values to the device's pins) but does not have access to any OS-level API calls, such as the creation of new threads. As Arduino is not a complete operating system, it does not have the concept of threads or concurrency, and thus, tasks can only be executed sequentially.

ESP-IDF. ESP-IDF (Espressif Systems, 2023a) is a modified version of FreeRTOS (FreeRTOS, 2023), which includes a kernel and a set of IoT libraries suitable for various purposes. This tech stack is similar to the previous one, but the ESP-IDF API can be used to create and manage threads and tasks and the concurrent execution of certain operations.

MicroPython. MicroPython (MicroPython, 2023a; RIOT OS, 2023) is an interpreter based on Python3 optimized to run on IoT edge devices, as previously introduced in Section 2. For the experiments, we used the tool `ampy` (Adafruit Industries, 2021) to send Python files to the ESP32 microcontroller through its serial connection. The functions defined in the files can then be called through the MicroPython REPL prompt. Finally, we used the MicroPython library `uasyncio` (MicroPython, 2023b) to enable the concurrent execution of several tasks, as it implements an asynchronous I/O scheduler.

Toit. Toit (2023a), as previously introduced (in Section 2), is a programming language and ecosystem designed for the ESP32 microcontroller, allowing over-the-air code updates to devices without the need

[8] Additional information about the device is available at https://www.az-delivery.de/products/esp-32-dev-kit-c-v4.

Table 2
Results of the experiment showing the correlation between file size and occupied heap size, in bytes.

File	File size	Reserved stack size	Free heap size before start	Free heap size after start	Occupied heap size	Extra occupied heap size	Relative diff. (%)
small_opt.wasm	395	8192	201 788	183 972	17 816	9624	117.4%
		12 288	201 740	179 824	21 916	9628	78.4%
		16 384	201 796	175 764	26 032	9648	58.8%
small.wasm	512	8192	201 756	183 800	17 956	9764	119.1%
		12 288	201 768	179 700	22 068	9780	79.6%
		16 384	201 736	175 588	26 148	9764	59.6%
big_opt.wasm	684	8192	201 752	182 816	18 936	10 744	131.5%
		12 288	201 760	178 728	23 032	10 744	87.4%
		16 384	201 756	174 640	27 116	10 732	65.5%
big.wasm	1834	8192	201 804	182 348	19 456	11 264	137.5%
		12 288	201 788	178 240	23 548	11 260	91.6%
		16 384	201 944	174 308	27 636	11 252	68.7%

Table 3
Results showing the correlation between the `stack-size` parameter used in the compilation process, and occupied heap size, in bytes.

Stack-size used in WASM compilation	Reserved stack size	Free heap size before start	Free heap size after start	Occupied heap size	Extra occupied heap size	Relative difference (%)
2048	8192	201 764	185 984	15 780	7588	92.6%
	12 288	201 736	181 856	19 880	7592	61.8%
	16 384	201 904	177 932	23 972	7588	46.3%
4096	8192	201 788	183 972	17 816	9624	117.4%
	12 288	201 740	179 824	21 916	9628	78.4%
	16 384	201 796	175 764	26 032	9648	58.8%
8192	8192	201 740	179 840	21 900	13 708	167.3%
	12 288	201 756	175 740	26 016	13 728	111.7%
	16 384	201 764	171 480	30 284	13 900	84.8%

for a full reboot. The fleet of devices can be controlled and orchestrated through their cloud API, and the application code must be developed in the Toit programming language. We installed the Toit firmware on an ESP32 microcontroller and then pushed tasks to the device through the Toit CLI. However, since the Toit ecosystem relies on the cloud to control the fleet of devices (unlike the other solutions, which can be controlled over the local network), network conditions may introduce delays in uploading tasks and executing certain commands, which could potentially influence the results.

5.2. Experiments

We conducted experiments in four different categories: (1) sanity checks, which provided empirical evidence on some of the design choices made and ensured that the platform could support basic scenarios; (2) performance tests, which involved executing a single task in all platforms and comparing the execution time and RAM usage; and (3) dynamic tests, which involved issuing a set of device operations, such as starting and stopping tasks, to create a request timeline.

The dynamic tests (3) were only performed on approaches that allow over-the-air updates without resetting the device. The behavior of each tool was observed by measuring the time each request was received, how long it took to process the operation, and the time taken to return a response, among other aspects.

We also performed load tests to evaluate the performance and maximum task capacity of various solutions under concurrent task execution to ensure the consistency of our performance test observations. The results showed that the behavior of WASMICO and its foundation, Wasm3, were similar. As these tests did not provide new information, we have decided not to fully describe them in the paper. However, the data is available in our replication package (Ribeiro et al., 2022).

5.2.1. Sanity checks

Impact of file size and compilation stack size on the occupied heap size. To upload a WebAssembly file to the device using WASMICO, one parameter that needs to be specified is the `reservedStackSize` parameter, which represents the stack size allocated for the FreeRTOS thread responsible for executing the WASM task. Although the stack memory allocated for a FreeRTOS thread comes from the FreeRTOS heap, examining the free heap size before and during the execution of a task reveals that the occupied heap size is slightly larger than the desired stack size. The following sanity checks will demonstrate that the additional space occupied in the heap is directly correlated to the size of the WebAssembly file and the `stack-size` parameter used during compilation.

The **first sanity check** focuses on the correlation between file size and occupied heap size. Four files with increasing sizes were used: `small_opt.wasm`, `small.wasm`, `big_opt.wasm`, and `big.wasm`. The `small_opt.wasm` and `small.wasm` are small WebAssembly tasks generated from the same C++ program, with the former being optimized using the `wasm-opt` and `wasm-strip` commands. Similarly, the `big_opt.wasm` and `big.wasm` were generated from a larger C++ program. All files were compiled with a `stack-size` of 4096 bytes. Each file was executed using three reserved stack sizes for the FreeRTOS thread: 8192, 12288, and 16384 bytes. For each test, we recorded the free heap size before and after the task started executing, then subtracted these values to obtain the amount of heap allocated to the task. Using these values, we calculated the difference between the indicated reserved stack size and the actual heap size used, *i.e.*, the additional occupied heap size.

Table 2 presents the results obtained from the experiments. As shown, the difference between the reserved stack size and the occupied heap size varies for each of the four files tested. We can observe that the occupied margin in the FreeRTOS heap increases as the size of the file increases. Additionally, we observed that the difference does not significantly change with the rise of the reserved stack size. For instance, when using the `big_opt.wasm` file, regardless of the reserved stack size, the difference between it and the occupied heap size was always approximately the same, ranging from 10732 to 10744 bytes, as observed in the other files.

While it could be argued that the content of the file could influence the occupied heap size, our results indicate that even for files

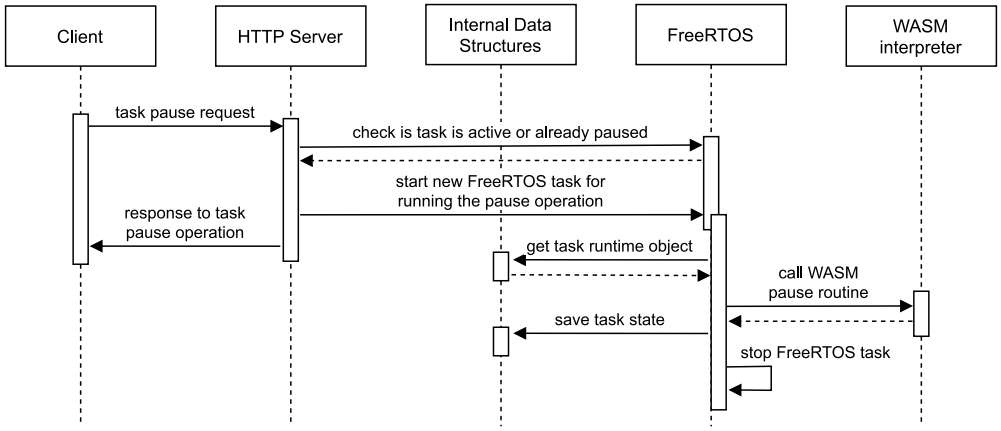

Fig. 3. Sequence diagram of the task pause operation.

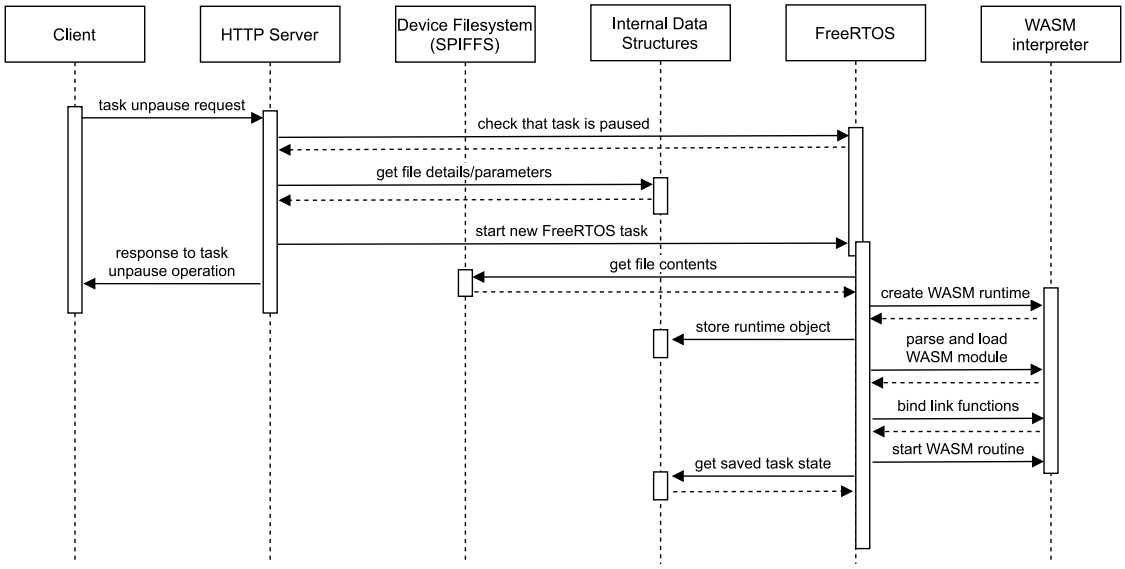

Fig. 4. Sequence diagram of the task resume (*task unpause*) operation.

generated from the same source code (such as `small_opt.wasm` and `small.wasm`, or `big_opt.wasm` and `big.wasm`), there is a noticeable difference in the occupied heap size (although the content of the WASM files differs due to optimization).

The **second sanity check** focuses on the correlation between the `stack-size` parameter used in the compilation process and the occupied heap size. Only one file, `small_opt.wasm`, was used for this check. However, it was compiled using three different `stack-size` values: 2048, 4096, and 8192 bytes. We then executed each file using the three reserved stack sizes for the FreeRTOS thread and recorded all the values.

Table 3 presents the results of the second sanity check. The results show that as the value of the `stack-size` parameter used in the compilation process increases, the difference between the reserved stack size and the actual occupied heap size also increases, indicating a correlation between the two. This increase in occupied heap size is unrelated to the rise in file size, as the `small_opt.wasm` file was compiled with all three `stack-size` values and had a consistent file size of 395 bytes. Therefore, no observable relationship between file size and the `stack-size` value used was found.

Use of stack "high watermark" to determine task stack. This sanity check was designed to illustrate how one can leverage a deployed task's stack "high watermark" to fine-tune and determine an appropriate value for its stack size. This approach helps ensure that the task can run without errors or crashes, while also avoiding the allocation of unnecessary space that could be used for other tasks.

For this experiment, a simple infinite loop was implemented that increments a counter every 2 s. The code was written in C++ and compiled to WebAssembly using the script depicted in Listing 1 with a `stack-size` value of 2048 bytes.

Initially, the task was executed with a reserved stack size of 3000 bytes, a reserved initial memory of 65536 bytes, and a memory limit of 2048 bytes.[9]

The output of the operation is presented in Listing 13. Immediately after starting the task, we can observe that it crashed with the error message "Stack canary watchpoint triggered", indicating a stack overflow.

```
Starting new task /tasks/test_state.wasm...
Counter: 0
Guru Meditation Error: Core  0 panic'ed (
    Unhandled debug exception)
```

[9] As previously mentioned in Section 3, the memory limit value should be the same as the one used in the `stack-size` parameter when compiling the task.

```
Debug exception reason: Stack canary watchpoint
    triggered (/tasks/test_sta)
```

Listing 13: Example output showcasing a stack overflow error due to the reserved stack size being too small.

When we execute the task with a higher reserved stack size value of 10000 bytes, we can observe that it runs successfully, as shown in Listing 14.

```
Starting new task /tasks/test_state.wasm...
Counter: 0
Counter: 1
Counter: 2
...
```

Listing 14: Example output showcasing that the task is running successfully.

When we execute the task with a higher reserved stack size value of 10000 bytes, we can observe that it runs successfully, as shown in Listing 14.

```
> wasmico showactive
{
  "task": [
    {
      "filename": "app.wasm",
      "paused": false,
      "stack_high_water_mark": 6528
    }
  ],
  "espressif_num_threads": 12,
  "status_code": 0
}
```

Listing 15: Output of the *list active tasks* operation, which shows that the active tasks has a large stack "high watermark" value.

In Listing 16, we can see that the stack's "high watermark" has decreased to 532 bytes, indicating less unused space in the task's stack. However, the value is sufficient to ensure the task does not crash. Therefore, the current stack size is appropriate for the task's memory requirements.

```
> wasmico showactive
{
  "task": [
    {
      "filename": "app.wasm",
      "paused": false,
      "stack_high_water_mark": 532
    }
  ],
  "espressif_num_threads": 12,
  "status_code": 0
}
```

Listing 16: Output of the *list active tasks* operation, which shows that the active tasks has a more adequate stack "high watermark" value.

5.2.2. Usage of the WASM runtime memory limit

This sanity check aims to provide evidence that the `memoryLimit` parameter specified when starting a WebAssembly task is crucial for efficient memory allocation by the WASM interpreter. The experiment compares the memory usage of the same WASM task when the `memoryLimit` parameter is specified with a case with no provided value.

The WASM task used in this experiment is identical to the one used in Section 3.4, demonstrating heap memory's deallocation. Building on this experiment, we used the same free heap size values to compare the task's memory usage with and without the memory limit parameter.

The experiment results are shown in Listing 17, indicating that when the memory limit is not provided, the free heap size significantly decreases from 200924 to 79956 bytes upon task execution. Furthermore, upon stopping the task, the free heap size only recovers to 129384 bytes. This demonstrates the critical role of the memory limit parameter in ensuring efficient memory allocation for WebAssembly tasks.

```
> wasmico heap
{ free_heap_size: 200924,
  largest_free_block_size: 113792,
  status_code: 0 }
> wasmico start test1.wasm
{ status_code: 0,
  message: 'task started successfully' }
> wasmico heap
{ free_heap_size: 79956,
  largest_free_block_size: 48228,
  status_code: 0 }
> wasmico stop test1.wasm
{ status_code: 0,
  message: 'task stopped successfully' }
> wasmico heap
{ free_heap_size: 129384,
  largest_free_block_size: 48228,
  status_code: 0 }
```

Listing 17: Experiment showcasing the evolution of the free heap size, if no memory limit is provided.

Specifying the memory limit for each executed WASM task on the system is crucial for efficiently using the FreeRTOS heap. This is likely due to the implementation of the task creation operation. To execute a task, the `xTaskCreate` function creates a FreeRTOS thread that uses the Wasm3 interpreter library to load and perform the task. The task stack is dynamically allocated from the FreeRTOS heap by `xTaskCreate`. When the memory limit is not specified to the Wasm3 interpreter, it cannot determine the exact memory requirement for the task to load correctly, leading to a possible allocation of more memory than needed.

5.2.3. Performance test

The Performance Tests (PT) served two purposes: (1) to understand the impact of each dependency of WASMICO in terms of resource usage and execution time, and (2) to benchmark our solution against other state-of-the-art solutions to establish a baseline for execution times and resource usage, and thereby better understand the trade-offs of our solution compared to other approaches. The performance of the different solutions was evaluated by calculating the first 1500 numbers of the Fibonacci sequence, and to ensure reliable results, we repeated the task 100 times for each platform. This allowed us to obtain a reasonable level of confidence in the measurements. In addition to benchmarking the considered solutions, we also executed the code on Wasm3 to measure the overhead of WASMICO compared to a plain WASM runtime.

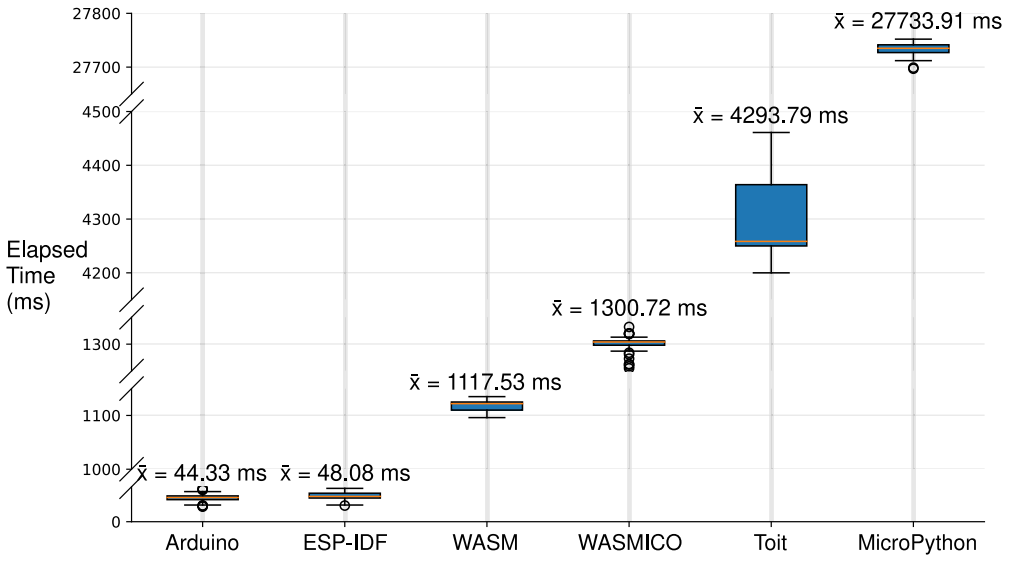

Fig. 5. Execution time results on all six platforms, when executing a single task computing the first 1500 Fibonacci numbers.

Regarding memory usage across platforms, the reserved stack size and the `stack-size` parameter used in the WebAssembly compilation process significantly impact the task's memory requirements. To minimize this impact, we ensured that the task was executed on all platforms with the minimum memory needed for safe execution without any errors or crashes. Specifically, the thread used a stack size of 550 bytes for ESP-IDF. For the base WASM (Wasm3) execution environment, the task was compiled with a `stack-size` value of 1168 bytes and a reserved stack size of 4250 bytes. For WASMICO, the task was compiled with a `stack-size` value of 1168 bytes and a reserved stack size of 4900 bytes. As for Toit, each task was allocated 4 KB of memory, which could grow based on the application's memory demands. The stack size specification was not required for the remaining platforms.

We determined the execution time of the task by logging a timestamp to the serial port at the beginning and end of the 1500 iterations. By comparing the timestamps, we were able to estimate the execution time.

We employed different methods to calculate memory usage depending on the platform. For ESP-IDF, Wasm3, and WASMICO, which run the task on a FreeRTOS thread, we used the FreeRTOS function `xPortGetFreeHeapSize` to determine the free heap size before the task's start and at the end of the 1501st iteration. The extra iteration ensured that the function call did not affect the execution time calculation. The total RAM size occupied by the task was computed by subtracting the two values obtained. For MicroPython, we used the `gc.mem_free` function provided by the garbage collector package. On Arduino, since there is no concept of threads, the task runs directly on the `_setup` function, and we could not use `xPortGetFreeHeapSize`. Instead, we subtracted the stack size allocated to the main routine, 8192 bytes, from the minimum amount of remaining stack space since its start obtained using the function `uxTaskGetStackHighWaterMark`. For Toit, we called the `serial_print_heap_report` function after all iterations, which outputted a table reporting the number of bytes occupied in RAM for each module or action. We used the value under the Toit entry, which indicates the RAM usage for all executing tasks on the device. The Toit garbage collector can be delayed for performance reasons if ample memory is available, which could cause an overestimate in the table entry value. Nevertheless, we considered it precise enough for the experiments.

Results. Regarding the execution time results presented in Fig. 5 and Table 4, it is evident that the Arduino and ESP-IDF environments are the fastest, with mean execution times of 44.33 and 48.08 ms, respectively, and negligible differences between the two. The base WebAssembly interpreter takes slightly over 1 s to execute the task, with a mean value of 1117.53 ms. WASMICO introduces minimal overhead to this performance, with execution times ranging from 1255 to 1332 ms. So far, all the execution times calculated on the same platform have been relatively similar, with low standard deviation and variance values. However, the execution times on the Toit platform exhibit greater diversity and broader variation, with the highest standard deviation and variance among all the platforms. Furthermore, the execution time is worse than the developed solution, taking on average 3.3× more time to complete the task, with an average of 4239.79 ms. Finally, the MicroPython environment has the poorest execution time among all platforms, with the task taking an average of almost 28 s to complete.

Regarding the results of the RAM usage, as shown in Fig. 6 and Table 5, we can observe that the Arduino and ESP-IDF platforms are the most efficient, using 700 bytes and 940 bytes, respectively, for all 100 executions. As expected, WASMICO has a higher RAM usage, with an average of 8.748 KB and similar minimum, median, and mode values, indicating consistent performance in most executions. However, some executions had a higher RAM usage, with a maximum value of 10.388 KB. The Wasm3 interpreter performed worse than WASMICO regarding RAM usage, with readings consistently between 10.212 KB and 10.220 KB. This is likely due to the dynamic allocation of heap size and the optimization techniques used by the Wasm3 interpreter. Toit, a complex tool with many features, had the worst memory usage out of all platforms, ranging from 69.632 KB to 118.784 KB, with an average of 95.887 KB. Finally, the MicroPython environment performed better than Toit but worse than WASMICO, with an average RAM usage of 36.913 KB for the task execution.

In conclusion, the results of the experiments indicate that WASMICO provides reasonable performance in terms of execution time and memory usage for a single task. It can be viewed as a middle ground between low-level, efficient solutions like Arduino or ESP-IDF, and high-level solutions like Toit and MicroPython, which offer more functionalities but with decreased performance. Moreover, the benchmark results of the base Wasm3 interpreter on top of ESP-IDF suggest that the modifications made in WASMICO do not add a significant overhead to the baseline WebAssembly performance.

5.2.4. Dynamic test

The objective of the dynamic test (DT) experiments is to demonstrate the behavior of the two platforms when performing a series of

Table 4
Statistics generated from the execution time results, in milliseconds, on all six platforms, when executing a single task computing the first 1500 Fibonacci numbers. (∗) More than one mode thus the smallest value is presented.

Platform	Min	Max	$\sum$	$\tilde{X}$	Mo	$\bar{X}$	σ	σ^2
Arduino	28	62	4433	45	45	44.33	6.81	46.41
ESP-IDF	30	62	4808	47	44	48.08	7.01	49.06
WASM	1096	1135	111753	1122	1125	1117.53	9.74	94.78
WASMICO	1255	1332	130072	1304	1305	1300.72	11.74	137.76
Toit	4200	4461	429379	4258	4255	4293.79	64.58	4170.98
MicroPython	27 697	27 752	2 773 391	27 735	27736*	27 733.91	11.39	129.76

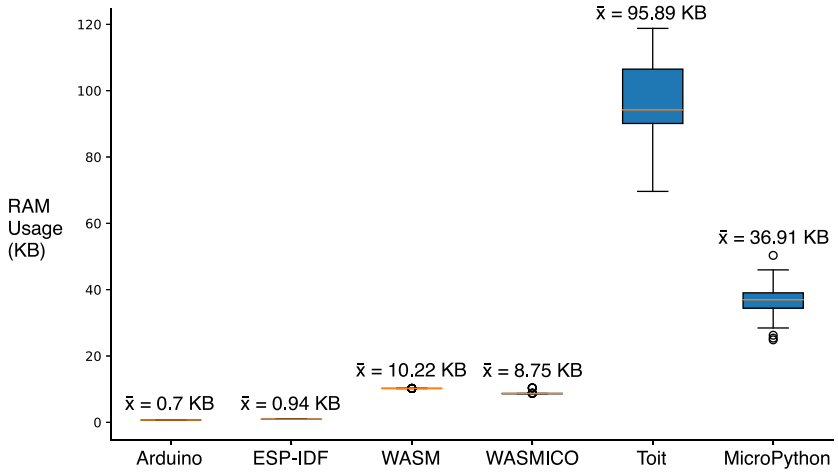

Fig. 6. RAM usage results on all six platforms, when executing a single task computing the first 1500 Fibonacci numbers.

Table 5
Statistics generated from the RAM usage results, in KBs, on all six platforms, when executing a single task computing the first 1500 Fibonacci numbers. (∗) More than one mode thus the smallest value is presented.

Platform	Min	Max	$\sum$	$\tilde{X}$	Mo	$\bar{X}$	σ	σ^2
Arduino	0.700	0.700	70.000	0.700	0.700	0.700	0.000	0.000
ESP-IDF	0.940	0.940	94.000	0.940	0.940	0.940	0.000	0.000
WASM	10.212	10.220	1021.568	10.216	10.216	10.216	$1.469 \cdot 10^{-3}$	$2.158 \cdot 10^{-6}$
WASMICO	8.648	10.388	874.764	8.676	8.672	8.748	0.335	0.112
Toit	69.632	118.784	9588.736	94.208	90.112	95.887	11.499	132.227
MicroPython	24.864	50.336	3691.328	36.960	33.408∗	36.913	4.264	18.182

operations requested by a client, such as creating and deleting multiple tasks. Various timestamps were recorded to capture the timing of the events, including when the client made the request, when the server received it, when the tool completed the requested operation, and when the client received a response from the server.

A total of three sub-experiments, each executing a different set of operations, were conducted on the platforms: **DT1** (shown in Fig. 7), **DT2** (shown in Fig. 8), and **DT3** (shown in Fig. 9). Each figure displays a list of all operations, the order in which they were requested, and the state of each task at every moment during the experiment. Notably, all experiments were performed on a single physical device.

To ensure that the pause and resume operations on WASMICO were correctly implemented, we utilized the Fibonacci task from our previous experiments and kept track of an iteration counter that was updated throughout the program's execution. This counter was saved when the task was paused and restored when the resume operation was called. For the WASMICO platform, the tasks were compiled to WebAssembly using a stack size of 2048 bytes, while each thread that executed the WASM routine had a stack size of 6000 bytes.

Results. In **DT1**, we uploaded five copies of the specified task, started them, and then deleted them in order. The operation requests were made through the CLI, with an interval of approximately 2 s between them.

Tables 6 and 7 display the recorded delay in milliseconds, compared to the time of the initial request, for each phase of each operation for both WASMICO and Toit. For WASMICO, delays were recorded when each request was received, when the operation started and ended, and when the client received a response. For Toit, as we cannot determine when events happen internally, such as when the request was received, only two events were recorded: (1) when the operation finished and (2) when the response was received. It is worth noting that, when using Toit, the upload and task creation operations are not separate since the task starts automatically when sent to the device. However, these operations are distinct when using WASMICO and were thus represented separately in the experiments with this tool.

DT2 involved executing start and stop operations, but unlike in **DT1**, we executed them randomly.

Tables 8 and 9 show the recorded delays for each event of each operation in **DT2**, for WASMICO and Toit, respectively.

Regarding **DT3**, since the set of operations included pausing and resuming tasks, we only executed it on WASMICO, as Toit does not support such operations. We used three copies of the task instead of the five in the other experiments. The tasks were uploaded to the device before executing the operations, and we ignored the upload process

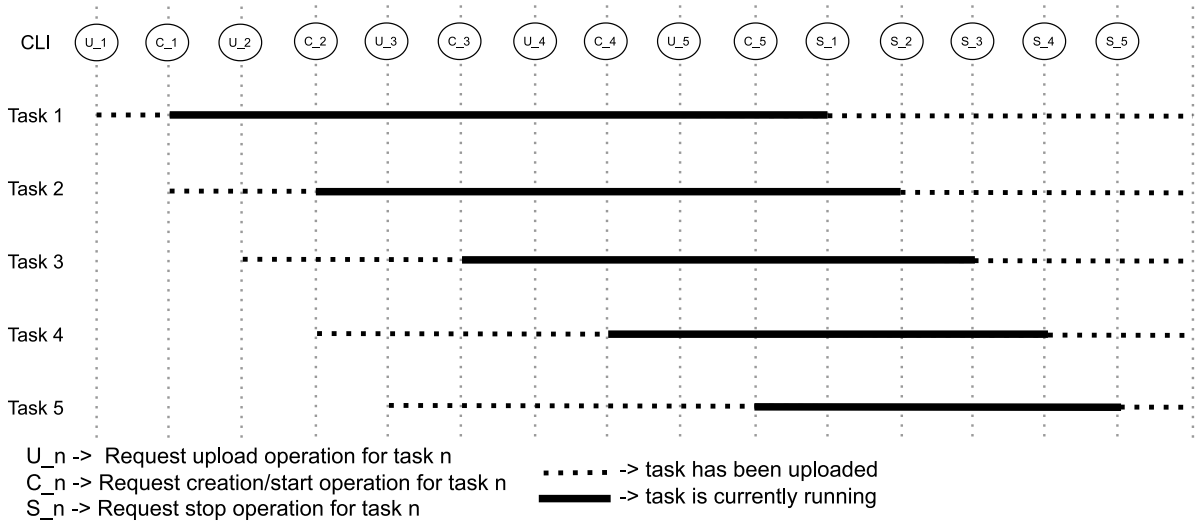

Fig. 7. Set of operations used and the status of all tasks through the lifetime of the first dynamic experiment (**DT1**).

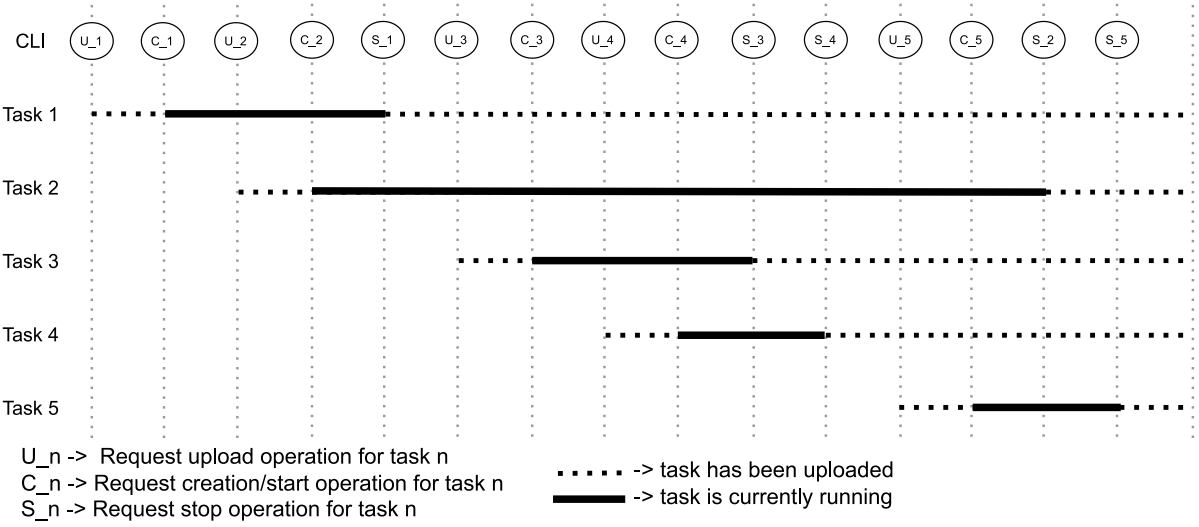

Fig. 8. Set of operations used and the status of all tasks through the lifetime of the second dynamic experiment (**DT2**).

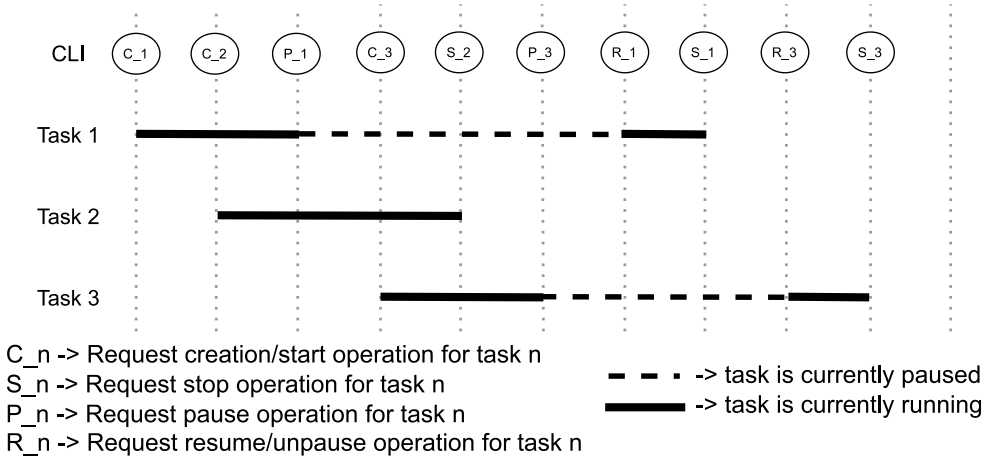

Fig. 9. Set of operations used and the status of all tasks through the lifetime of the third dynamic experiment (**DT3**).

Table 6
Time delays of WASMICO when processing the set of operations of **DT1**.

Operation	Received request	Started operation	Finished operation	Received response
Upload #1	250	250	605	605
Start #1	90	90	90	90
Upload #2	135	135	446	446
Start #2	175	175	175	175
Upload #3	200	200	584	584
Start #3	255	255	255	259
Upload #4	89	89	393	393
Start #4	140	140	140	140
Upload #5	187	187	615	624
Start #5	230	230	230	235
Stop #1	272	272	272	273
Stop #2	97	97	97	97
Stop #3	153	153	153	153
Stop #4	184	184	184	185
Stop #5	232	232	232	236

Table 7
Time delays of Toit when processing the set of operations of the **DT1**.

Operation	Finished processing operation	Received response
Upload & Start #1	3875	2171
Upload & Start #2	4377	2248
Upload & Start #3	4214	2170
Upload & Start #4	5390	2253
Upload & Start #5	4911	2234
Stop #1	1631	760
Stop #2	1827	687
Stop #3	1694	670
Stop #4	1495	876
Stop #5	1266	701

Table 8
Time delays of WASMICO when processing the set of operations of the **DT2**.

Operation	Received request	Started operation	Finished operation	Received response
Upload #1	322	322	442	442
Start #1	154	154	154	154
Upload #2	87	87	385	378
Start #2	134	134	134	134
Stop #1	422	422	422	663
Upload #3	109	109	652	654
Start #3	156	156	156	158
Upload #4	86	86	389	389
Start #4	134	134	134	134
Stop #3	93	93	93	93
Stop #4	121	121	121	121
Upload #5	159	159	448	448
Start #5	100	100	100	102
Stop #2	138	138	138	138
Stop #5	309	309	309	616

Table 9
Time delays of Toit when processing the operations of **DT2**.

Operation	Finished operation	Received response
Upload & Start #1	4222	2371
Upload & Start #2	4322	2326
Stop #1	1504	916
Upload & Start #3	4106	2291
Upload & Start #4	5099	2351
Stop #3	1778	898
Stop #4	1427	781
Upload & Start #5	4649	2331
Stop #2	1554	794
Stop #5	1728	1190

Table 10
Time delays of WASMICO when processing the set of operations of the **DT3**.

Operation	Received request	Started operation	Finished operation	Received response
Start #1	83	143	143	149
Start #2	129	129	129	137
Pause #1	73	73	73	75
Start #3	110	110	110	113
Stop #2	159	159	159	169
Pause #3	99	99	99	107
Resume #1	132	132	253	162
Stop #1	76	76	76	83
Resume #3	112	112	112	121
Stop #3	153	153	153	153

as the emphasis was on the remaining operations. The delays for the requested operations in **DT3** are shown in Table 10.

In general, we can conclude that WASMICO has shorter delays for all operations compared to Toit.[10] For Toit, we observed that the delay in receiving a response is often smaller than the delay for the operation itself, which indicates that the operations are likely processed separately in a different thread while the response is sent to the client. For WASMICO, most operations typically take between 100 to 300 ms to complete, except for the upload operation, which takes slightly longer due to the file being sent and saved on the device's filesystem. For the specific task used in the experiment, this operation took around 400 to 650 ms to complete, but this time may vary depending on the file size. Similarly, for the Toit platform, the upload operation (done at the start of the task) usually takes 4 to 5 s, and the stop operation takes 1 to 2 s. However, as mentioned earlier, the response is generally received earlier, with the response time for the upload and start operation taking around 2 s and the response time for the stop operation taking less than 1 s.

5.3. Results discussion

In addition to Arduino and ESP-IDF, which served as the basis for our tool and offer close-to-native performance, WASMICO outperforms its main competitor, Toit, regarding task execution times, RAM usage, and request processing times. Tasks executed in WASMICO generally have faster execution times than in Toit, regardless of the number of concurrent tasks. Moreover, by enabling users to manually compile code to WebAssembly and adjust the stack size for each task, RAM usage can be customized for each program, ensuring that each task has precisely the memory required. This capability can ultimately support a higher number of tasks running concurrently than Toit, which automatically assigns a 4 KB stack to each task that can expand as needed and does not require user input on task memory usage. However, Toit's higher RAM usage is reasonable and expected, given that it is a more sophisticated solution than WASMICO with more features and background processes, making it a more comprehensive environment for development and operation. This can also justify Toit's longer response delays compared to WASMICO, as observed in DT3.

Compared to a WebAssembly interpreter based on ESP-IDF, such as the one used in **PT**, WASMICO adds minimal overhead regarding execution time and memory requirements. On average, WASMICO added approximately 183 ms of overhead to the execution time of a single task in **PT**. Regarding RAM usage, WASMICO performed better than the WASM interpreter, using approximately 1.4 KB less when running a single task. This improvement is likely due to using the `xTaskCreate` function, which creates a FreeRTOS task with a specified stack size, dynamically and automatically allocated from the FreeRTOS heap. It

[10] It is important to note that delays in Toit can be affected by network delays when communicating with the cloud management platform.

is possible that `xTaskCreate` allocates more memory from the heap than needed, depending on other task information and tasks running at that time.

In contrast, when running WASMICO, any WebAssembly task runs concurrently with at least one other FreeRTOS thread – the HTTP server continuously receiving operation requests from the client – and possibly other active WASM tasks. Therefore, fewer resources are available when calling `xTaskCreate`, leading to lower RAM usage. Additionally, the Wasm3 interpreter used in WASMICO may have optimizations contributing to its efficient memory usage. To further optimize RAM usage, we could potentially use the `xTaskCreateStatic` function, which requires the developer to statically declare the task stack size instead of dynamically allocating it from the heap. However, we have no empirical evidence to suggest that this would lead to a significant improvement.

Compared to MicroPython, WASMICO has significantly better performance in terms of execution time, particularly when multiple tasks are executed concurrently. Although MicroPython supports multi-threading through libraries like `uasyncio`, these libraries are primarily designed for a limited number of small and simple routines; they are unsuitable for use cases with numerous complex tasks running concurrently. While it does allow the creation of many threads simultaneously, the halting and context-switching process involved in switching tasks from a blocked state to an active state (and vice versa) is inefficient. MicroPython is the worst scaling platform among all the benchmarked platforms, as adding the execution of one extra task can severely impact the system's overall performance.

In conclusion, our experiments and tests demonstrated that WASMICO generally outperformed other state-of-the-art platforms, including Toit and MicroPython, but was slightly less performant than lower-level environments like ESP-IDF. Nevertheless, WASMICO offers distinct advantages, such as defining the RAM usage for each program *a priori*, and providing a local-first way to monitor and operate a fleet of IoT devices (both Golioth and Toit are cloud-based solutions). Moreover, WASMICO consistently outperformed Toit regarding task execution times, RAM usage, and request processing times, irrespective of the number of concurrently executed tasks. Therefore, while there may be some performance trade-offs compared to lower-level environments, WASMICO offers a robust solution that can satisfy the demands of numerous IoT applications while providing the flexibility and ease of development associated with higher-level languages.

6. Validation threats

To ensure the scientific soundness of our experiments, we must consider the limitations of the metrics used to measure performance. In particular, we must acknowledge the limitations of the available functions and APIs in each platform and the accuracy of the extracted values.

When measuring performance, we must consider that the metrics used on different platforms may vary, and the accuracy of these metrics may not correspond to the actual values. For instance, on platforms such as Arduino, ESP-IDF, and WASMICO, we have access to the execution environment. We can modify it to extract desired information, such as the free heap size value before and after a task execution or print operations to the serial port. However, we cannot access the inner workings of platforms like Toit, and all metrics must be extracted using the platform's API. Thus, we may not have precise information on when a request was received, and the values calculated may not correspond to the actual values. We attempted to mitigate this threat using recommended and available methods to collect such metrics.

Similarly, the extraction of RAM usage values was also conditioned by the available functions in each platform, with some being just estimates given the existent information. The available functions on each platform had different limitations, and some functions may provide estimates instead of precise values. For example, the Toit function `serial_print_heap_report` may return a value slightly above the actual one due to the Toit garbage collector's delayed intervention for performance purposes. Different strategies must be used to calculate the RAM usage estimates, considering the methods available on each platform and API. Due to such, these estimates may not correspond to the actual RAM usage values.

Finally, the timestamps extracted in each experiment may not precisely match when the corresponding event occurred. The precision of the timestamps may not be ideal for tasks that take a short time to execute or when print operations are cached. This can delay the timestamp, and consequently, the execution time calculation may not be accurate. To mitigate this threat, we used the same approach and serial port terminal tool in all tests and all platforms to ensure consistency.

Overall, to ensure scientific soundness in our experiments, we acknowledge the metrics' limitations and attempted to mitigate possible threats.

7. Conclusion

This research aimed to develop a micro-containerization tool to bridge the gap between IoT and DevOps paradigms and evaluate its practical usability and potential benefits. The goal was to enable devices to execute various tasks or containers concurrently and receive over-the-air operations without requiring a reboot.

To achieve this, we created the WASMICO tool, built on top of ESP-IDF, the development framework of the ESP32 microcontrollers. WASMICO enables the execution of WebAssembly tasks on-demand on resource-constrained IoT devices and supports all operations required for managing the complete lifecycle of tasks and containers. This includes containerizing applications for the IoT domain, and simplifying the deployment, monitoring, management, and evolution of tasks on IoT devices without requiring device resets for firmware reprogramming. We also created a CLI tool that can be both used locally by developers and easily integrated into Continuous Deployment scripts and can be used as the foundations for device-fleet orchestration system.

Additionally, WASMICO enables HTTP protocol-based over-the-air operations and is an abstraction between the high-level specifications and the device's low-level capabilities. Through middleware, it offers controlled access to device capabilities, such as reading and writing to pins and printing to the serial port. WASMICO executes WebAssembly tasks and uses a Wasm3 interpreter, supporting programs written in programming languages that can be compiled to WASM, such as C++, Go, Rust, and others.

Future research may include additional benchmarking against other approaches presented in the literature, usability experiments to gather developer feedback on the tool's utility, intuitiveness, and ease of use, and using the approach in Decentralized Orchestration Environments as a replacement for solutions such as MicroPython and Lua (Silva et al., 2020).

CRediT authorship contribution statement

Eduardo Ribeiro: Methodology, Software, Validation, Writing – original draft. **André Restivo:** Conceptualization, Data curation, Writing – review & editing, Supervision. **Hugo Sereno Ferreira:** Conceptualization, Visualization, Writing – review & editing, Supervision, Resources. **João Pedro Dias:** Conceptualization, Supervision, Writing – review & editing, Resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The replication package and tool code is linked in the manuscript.

References

Adafruit Industries, 2021. ampy: Adafruit MicroPython tool. Last access: April 15th 2023, URL https://pypi.org/project/adafruit-ampy.

Anon, 2023. WASI: WebAssembly System Interface. Last access: April 15th 2023, URL https://github.com/WebAssembly/WASI.

Arduino, 2023. Arduino. Last access: April 15th 2023, URL https://www.arduino.cc/.

Aslam, F., Fennell, L., Schindelhauer, C., Thiemann, P., Ernst, G., Haussmann, E., Rührup, S., Uzmi, Z.A., 2010. Optimized java binary and virtual machine for tiny motes. In: International Conference on Distributed Computing in Sensor Systems. Springer, pp. 15–30.

Baccelli, E., Doerr, J., Kikuchi, S., Padilla, F.A., Schleiser, K., Thomas, I., 2018. Scripting over-the-air: Towards containers on low-end devices in the Internet of Things. In: 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). IEEE, pp. 504–507.

Bormann, C., Castellani, A.P., Shelby, Z., 2012. Coap: An application protocol for billions of tiny internet nodes. IEEE Internet Comput. 16 (2), 62–67.

Brouwers, N., Langendoen, K., Corke, P., 2009. Darjeeling, a feature-rich VM for the resource poor. In: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. pp. 169–182.

Clang, 2023. Clang: a C language family frontend for LLVM. Last access: April 15th 2023, URL https://clang.llvm.org/.

ContikiOS, 2023. Contiki-NG, the OS for next generation IoT devices. Last access: April 15th 2023, URL https://www.contiki-ng.org/.

Dias, J.P., Ferreira, H.S., Sousa, T.B., 2019. Testing and deployment patterns for the internet-of-things. In: Proceedings of the 24th European Conference on Pattern Languages of Programs. pp. 1–8.

Dias, J.P., Restivo, A., Ferreira, H.S., 2022. Designing and constructing internet-of-Things systems: An overview of the ecosystem. Int. Things 19, 100529.

Erich, F.M.A., Amrit, C., Daneva, M., 2017. A qualitative study of DevOps usage in practice. J. Softw.: Evol. Process 29 (6), e1885. http://dx.doi.org/10.1002/smr.1885, e1885 smr.1885, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1885, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1885.

Espressif Systems, 2023a. ESP-IDF documentation. Last access: April 15th 2023, URL https://docs.espressif.com/projects/esp-idf/en/latest/esp32/.

Espressif Systems, 2023b. SPIFFS filesystem. Last access: April 15th 2023, URL https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html.

Fleming, M., 2017. A thorough introduction to eBPF. Linux Weekly News.

FreeRTOS, 2023. FreeRTOS. Last access: April 15th 2023, URL https://www.freertos.org/.

Gavrin, E., Lee, S.-J., Ayrapetyan, R., Shitov, A., 2015. Ultra lightweight JavaScript engine for internet of things. In: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. pp. 19–20.

Golioth, 2023. Golioth. Last access: April 15th 2023, URL https://golioth.io/.

Gurdeep Singh, R., Scholliers, C., 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. pp. 27–36.

Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017. Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200.

Hall, D., 2011. Python-on-a-chip. Last access: April 15th 2023, URL https://code.google.com/archive/p/python-on-a-chip/.

Hübschmann, I., 2020. ESP32 for IoT: A complete guide. Last access: April 15th 2023, URL https://www.nabto.com/guide-to-iot-esp-32/.

Lwakatare, L.E., Karvonen, T., Sauvola, T., Kuvaja, P., Olsson, H.H., Bosch, J., Oivo, M., 2016. Towards DevOps in the embedded systems domain: Why is it so hard? In: 2016 49th Hawaii International Conference on System Sciences. HICSS, pp. 5437–5446. http://dx.doi.org/10.1109/HICSS.2016.671.

MicroPython, 2023a. MicroPython. Last access: April 15th 2023, URL https://micropython.org/.

MicroPython, 2023b. uasyncio: asynchronous I/O scheduler for MicroPython. Last access: April 15th 2023, URL https://docs.micropython.org/en/latest/library/uasyncio.html.

Ribeiro, E., Restivo, A., Ferreira, H.S., Dias, J.P., 2022. WASMICO: Micro-containers in microcontrollers for the Internet-of-Things. http://dx.doi.org/10.5281/zenodo.6921110.

RIOT OS, 2023. MicroPython RIOT port. Last access: April 15th 2023, URL https://doc.riot-os.org/group_pkg_micropython.html.

RiotOS, 2023. RIOT - The friendly Operating System for the Internet of Things. Last access: April 15th 2023, URL https://www.riot-os.org/.

Silva, M., Dias, J., Restivo, A., Ferreira, H., 2020. Visually-defined real-time orchestration of iot systems. In: MobiQuitous 2020-17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. pp. 225–235.

Toit, 2023a. Toit: Cloud-managed containers on the ESP32. Last access: April 15th 2023, URL https://toit.io/.

Toit, 2023b. Toit programming language. Last access: April 15th 2023, URL https://github.com/toitlang/toit.

Tollervey, N.H., 2017. Programming with MicroPython: embedded programming with microcontrollers and Python. "O'Reilly Media, Inc.".

Tsiftes, N., Voigt, T., 2018. Velox VM: A safe execution environment for resource-constrained IoT applications. J. Netw. Comput. Appl. 118, 61–73.

wasm-opt, 2023. wasm-opt: Optimize WebAssembly binary files. Last access: April 15th 2023, URL https://command-not-found.com/wasm-opt.

wasm-strip, 2021. wasm-strip: Remove sections of a WebAssembly binary file. Last access: April 15th 2023, URL https://webassembly.github.io/wabt/doc/wasm-strip.1.html.

Wasm3, 2023. Wasm3: The fastest WebAssembly interpreter, and the most universal runtime. Last access: April 15th 2023, URL https://github.com/wasm3/wasm3.

Wasm3-arduino, 2023. Wasm3-arduino: The fastest WebAssembly interpreter for Arduino, PlatformIO, Particle. Last access: April 15th 2023, URL https://github.com/wasm3/wasm3-arduino.

WebAssembly, 2023. WebAssembly. Last access: April 15th 2023, URL https://webassembly.org/.

Zandberg, K., Baccelli, E., 2020. Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf. arXiv preprint arXiv:2011.12047.

Zandberg, K., Baccelli, E., 2021. Femto-containers: DevOps on microcontrollers with lightweight virtualization & isolation for IoT software modules. arXiv preprint arXiv:2106.12553.

ZephyrRTOS, 2023. ZephyrRTOS. Last access: April 15th 2023, URL https://www.zephyrproject.org/.

Zhu, L., Bass, L., Champlin-Scharff, G., 2016. DevOps and its practices. IEEE Softw. 33 (3), 32–34. http://dx.doi.org/10.1109/MS.2016.81.

Eduardo Ribeiro holds a Master in Informatics and Computing Engineering by the Faculty of Engineering of the University of Porto, Portugal (FEUP). He has a particular interest in IoT and programming constrained devices and on improving software development tools' experience.

André Restivo has a Ph.D. in Informatics by the University of Porto in Portugal. He is currently an Assistant Professor, lecturing at the Faculty of Engineering, University of Porto (FEUP) since 2003. He has been involved in more than 30 different curricular units, in 7 different degrees. His main research areas (+40 published works) include Aspect Oriented Programming, Unit Testing and Internet-of-Things. He supervised +50 students in these and other topics.

Hugo Sereno Ferreira has a Ph.D. in Informatics by the Universities of Porto, Aveiro, and Minho in Portugal. Senior Researcher at INESC TEC. Assistant Professor at the Faculty of Engineering, University of Porto (FEUP), he was lecturer of more than 30 different curricular units since 2008. His main research areas (+50 published works) include Internet of Things, Large-Scale Software Systems, Design and Architectural Patterns, Machine Learning and Distributed Ledger Technologies (Blockchain). He supervised +60 students in these topics.

João Pedro Dias has a BSc+MSc in Informatics and Computing Engineering by the Faculty of Engineering, University of Porto (FEUP). He earned his Ph.D. in Informatics Engineering from FEUP in 2022, while receiving a grant from the Portuguese Foundation for Science and Technology (FCT). He maintains a Software Engineer position as a day-to-day job. Since 2017, he has been an Invited Assistant Professor FEUP, where he teaches courses in Software Engineering, Operating Systems, and Software Development at Large Scale, among others. He has (co-)supervised +10 M.Sc. dissertations. His research focuses on Internet-of-Things systems, software engineering, security and privacy, and his work has been published in several top-tier conferences and journals.