h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences

– Faculty of Computer Science –

WebArgo: A Dynamic and Heterogeneous Volunteer Computing Platform Leveraging Modern Web Technologies

Submitted in partial fulfilment of the requirements for the degree of

Master of Science (M.Sc.)

by

Philipp Christian Otto Zimmermann

Matriculation number: 755647

First Examiner	:	Prof. Dr. Ronald Moore
Second Examiner	:	Prof. Dr. Lars-Olof Burchard

DECLARATION

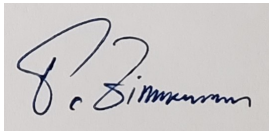
I hereby certify that I have written this thesis independently and have not used any sources other than those listed in the bibliography.

All passages taken verbatim or in spirit from published or as yet unpublished sources are identified as such.

The figures or illustrations in this work have been created by myself or have been provided with a corresponding source reference.

This paper has not been submitted in the same or a similar form to any other examining authority.

Darmstadt, 10. December 2024



Philipp Christian Otto Zimmermann

ABSTRACT

This work presents WebArgo, a novel volunteer computing platform that leverages modern web technologies to create an accessible and efficient distributed computing solution. The platform addresses the growing need for computational resources for research and development projects, while providing an alternative to traditional high-performance computing infrastructure, particularly beneficial for organizations with limited resources.

WebArgo's implementation utilizes three key web technologies: WebAssembly for platform-independent code execution, WebSockets for efficient real-time communication, and WebWorkers for potential browser-based parallel processing. The platform is designed to support heterogeneous client devices, requiring only access to a web browser for volunteers to participate in WebArgo and contribute their computational resources. Additionally, WebArgo allows dynamic worker participation, because the volunteer computing environment is expected to consist of potentially unreliable and fluctuating volunteer workers.

The evaluation experiments demonstrate WebArgo's capability to effectively distribute computational workloads across diverse consumer devices while maintaining system stability in environments with fluctuating worker participation. Empirical evaluation using a benchmark project to visualize the Mandelbrot set in high resolution achieved a 59% reduction of the total computation time compared to native code execution in a cloud environment when distributing the workload across four heterogeneous common household devices with WebArgo. The platform successfully supported various client devices, including smartphones, tablets, laptops, desktop Personal Computer (PC), and single-board computers, while maintaining a reliable task distribution and providing a complete result collection.

This work contributes to the field of volunteer computing by providing a web-based solution that reduces the barrier of entry for volunteer participation while addressing security concerns through the browser's inherent security model and WebAssembly's sandbox environment. WebArgo's implementation enables a straightforward deployment, management and development of custom jobs for independent usage without a deep understanding of the underlying architecture, making distributed computing more accessible in general.

CONTENTS

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

API Application Programming Interface
BOINC Berkeley Open Infrastructure for Network Computing
CGI Common Gateway Interface
CPU Central Processing Unit
CRUD Create, Read, Update und Delete
DOM Document Object Model
DTO Data Transfer Object
EGEE Enabling Grids for E-science in Europe
FCGI Fast Common Gateway Interface
FIFO First In - First Out
FLOPS Floating Point Operations Per Second
GPU Graphics Processing Unit
GUI Graphical User Interface
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HPC High-Performance Computing
h_da Darmstadt University of Applied Sciences
IQI Internet Quality Index
JSON JavaScript Object Notation
JWT JSON Web Token
LTS Long Term Support
OS Operating System
PC Personal Computer
PNG Portable Network Graphics
POC Proof of Concept
REST Representational State Transfer
RPC Remote Procedure Call
SETI Search for Extra-Terrestrial Intelligence
TCP Transmission Control Protocol
UI User Interface
UML Unified Modeling Language
URI Uniform Resource Identifier
URL Uniform Resource Locator

Part I

THESIS

1 INTRODUCTION

WebArgo is named after the legendary ship in Greek mythology *Argo*, known for its speed and efficiency in ancient tales. The *Argo* carried a diverse crew of Greek heroes, each with their unique abilities, known as the *Argonauts*. This group of *Argonauts* worked together to complete a shared goal on their quest. Likewise, the WebArgo platform efficiently coordinates a heterogeneous group of volunteer devices, each diverse in hardware or operating system, to collaboratively solve computational workloads.

This paper proposes a new approach to implement a volunteer computing platform. The implementation of WebArgo leverages the infrastructure of the continuously evolving web ecosystem. While the modern web provides multiple technologies to develop a robust and high-performance volunteer computing platform, the continuous development of hardware for consumer devices has led to a significant increase of the computational capabilities of devices that are connected to the internet like laptops, smartphones, tablets and desktop computers [Hab+17; HH22]. The key web technologies utilized in the implementation of WebArgo include:

- **WebAssembly** [Gro19], a binary format that can be executed in a web browser. Multiple programming languages support WebAssembly as a compilation target and it is promising near-native performance while providing platform independent code execution [Haa+17; Gro19].
- **WebSockets** [Cor24a], a bidirectional communication channel that establishes persistent connections between clients and servers. This technology enables data transmission in both directions while maintaining a minimal overhead. Therefore, WebSockets significantly reduce the latency compared to traditional HTTP polling methods [PN12].
- **WebWorker** [Cor24b], enabling parallel processing in the browser by executing JavaScript code in isolated background threads. These WebWorker threads are independent of the main browser User Interface (UI) thread and prevent computationally intensive tasks from blocking the UI rendering process [Cor24b].

Consumer devices that are used to participate in volunteer computing projects are usually very diverse in hardware and operating system [AR09]. Furthermore, these clients are typically unreliable [And20], as they fluctuate in availability and may disconnect during active task execution. To manage such an inconsistent crowd of heterogeneous clients, WebArgo implements the following two main features:

1. **Heterogeneity:** The platform supports a diverse range of devices, regardless of their operating system or hardware. The only requirement to participate in a project of WebArgo is access to a web browser, which supports the previously specified web technologies. Supported client devices range from PC and smart-devices to single-board computers like Raspberry Pis.

2. **Dynamic participation:** The number of participating devices is variable and can change dynamically during the runtime of a job. The implemented infrastructure of WebArgo ensures that all active jobs are completed without issues, even as the number of participating devices fluctuates. Hence, clients are able to join an active job or disconnect from the platform at any time without disrupting ongoing processes.

1.1 MOTIVATION

The primary objective of this work is to develop a resource-efficient, accessible and flexible alternative to conventional data centers and High-Performance Computing (HPC) infrastructures. These traditional HPC infrastructures often require significant investments in dedicated hardware and maintenance. This creates barriers for universities, research projects or smaller businesses that do not have access to external supercomputers and do not possess the necessary resources to build a HPC infrastructure themselves. WebArgo aims to provide for such organizations with limited resources a simple and user friendly platform to execute extensive research or development tasks. By leveraging WebArgo, these organizations can establish their own local volunteer computing network that utilizes existing and available computational resources. Furthermore, WebArgo can be expanded to a global volunteer computing platform where volunteering clients are able to support research projects and jobs world wide.

1.2 OBJECTIVES

WebArgo aims to reduce the overhead for volunteering participants, making it more appealing for clients to participate and potentially generating a larger volunteer base. The platform is designed in a way that volunteers require no setup to participate in projects. They connect to the web application through their browser via an URL and automatically participate in the active job of the volunteer computing network, with the platform's structure autonomously handling all processes in the background. No files or executables need to be downloaded on to the device, except for a browser, which already comes pre-installed on most consumer devices. Volunteers can transparently monitor the computing process of their processed workload through the web interface and maintain flexibility in joining or leaving the process at any time. When participants disconnect

from the platform, no persistent files or data are stored on their devices leaving no permanent effect on the device.

A study has revealed that privacy and security concerns significantly impact willingness to participate in volunteer computing projects [TMN11]. The approach to utilize the browser environment offers an advantage in addressing these concerns, as it implements an inherent security model. The application executed in the browser cannot read or write files or access device hardware such as cameras or microphones without explicit user permissions. WebAssembly, the key technology underlying WebArgo, is also restricted to these security constraints. Furthermore, the fact that volunteers are not required to install any third-party tool on their device could enhance the trust and positively affect the sense of security for participants.

Furthermore, WebArgo aims to minimize any overhead for administrators that maintain the platform or develop custom jobs. The process of creating and distributing new custom jobs through the platform is designed to be straightforward and does not require a deep understanding of the underlying architecture of WebArgo. WebAssembly significantly facilitates this objective, since it is designed to be a compilation target for numerous high-level programming languages [Haa+17; Gro19; Cor24c; HH22]. Developers can select a programming language with which they are familiar to create projects, while the overhead required to implement this code into the WebArgo framework remains comparatively low.

In addition, administrators are able to monitor and control the deployed volunteer computing platform through a transparent web application, which implements specific security measures to be only accessible by an administrator.

1.2.1 *Research Questions*

This work investigates the viability of the developed volunteer computing platform - WebArgo. Through empirical testing, the evaluation focuses on addressing three fundamental research questions that examine WebArgo's computational capabilities, dynamic adaptability, and support of heterogeneous devices:

1. **Computational Capability:** Is WebArgo capable of successfully solving large, parallelizable problems?
2. **Dynamic Viability:** Is the WebArgo platform stable in a environment with dynamic clients?
3. **Heterogeneous Viability:** Does WebArgo support a diverse range of client devices without issues?

These research questions are designed to evaluate critical aspects of WebArgo's functionality and performance. The first question examines the platform's fundamental ability to successfully distribute tasks across the

network to complete a parallelizable job. The second question addresses the platform's resilience and stability when dealing with a varying number of connected clients and potentially unreliable connections. The third question investigates the platform's compatibility across different device hardware and operating systems. Through these investigations, this work aims to validate WebArgo's effectiveness as a viable solution for distributed computing in the modern web environment.

1.3 STRUCTURE

This paper presents a comprehensive examination of the developed WebArgo platform. The theoretical foundation is established in Chapter 2, which explores the fundamental concepts of volunteer computing, introduces the enabling key web technology for WebArgo - WebAssembly - and examines related work in the fields of distributed and volunteer computing as well as edge computing with WebAssembly or mobile devices. Chapter 3 defines the terminology used in the following chapters, presents the architectural design of WebArgo and provides the developed theoretical model which serves as the foundation of the WebArgo implementation. Then, Chapter 4 presents the structure of the development process, introduces the utilized frameworks and technologies in the WebArgo implementation and describes the approach to benchmark and evaluate the developed WebArgo platform. The implementation of WebArgo and challenges that occurred during the development are discussed in Chapter 5, providing insights into the developed communication protocol, persistence mechanism, scheduling approach, security measures and technical design of the platform. Chapter 6 describes the experimental setup that has been used to test and benchmark the WebArgo platform and evaluates the corresponding results of these experiments. Finally, Chapter 7 presents the conclusion with a summary of the findings, discussion about limitations of WebArgo and outlines potential future work directions of this platform.

2

BACKGROUND AND RELATED WORK

This chapter covers the theoretical background of this work. At first Section 2.1 introduces the concept of volunteer computing. Then Section 2.2 provides technical information about WebAssembly, which is the key web technology utilized in WebArgo. And finally Section 2.3 presents related works to describe the state of research and development in the field of volunteer computing and edge computing with WebAssembly.

2.1 DISTRIBUTED COMPUTING & VOLUNTEER COMPUTING

Distributed computing describes an architecture where many autonomous computing elements collaborate as a unified system to solve complex computational problems through coordinated sharing of resources. A distributed computing system typically consists of multiple independent nodes, each possessing local memory and computational resources, interconnected through a network infrastructure that facilitates communication and synchronization between processes. This concept is particularly suitable to compute parallelizable workloads.

Volunteer computing is a sub category of distributed computing that focuses on public support. Participants share their computer processing power to support a collective interest. The idea of volunteer computing already emerged in 1996 [And20] and Sarmenta, Hirano, and Ward later characterized the term *volunteer computing* [SHW98]. The first large-scale scientific projects SETI@home and Folding@Home were launched in 1999 [And20; And+02]. These pioneering projects have demonstrated the success and popularity of volunteer computing.

Since the total number of consumer devices like laptops, smartphones, tablets, and desktop computers continues to increase [Sta24b; And20] there is a large pool of devices that can be used for volunteer computing. A study in 2014 has estimated that about 2 Billion computers are actively in use worldwide [Ref14]. However, many of these resources often remain idle or underutilized during regular operation [Hab+17; HH22]. While these untapped resources may appear insignificant individually, they collectively represent substantial potential due to their total number. These available resources located in office buildings, universities or public spaces can be utilized to participate in volunteer computing projects and thereby share their unused computational capabilities.

Furthermore, a study of 2011 revealed significant potential acceptance of volunteer computing in society, with 37% of respondents indicating they were either *"somewhat likely"* or *"very likely"* to participate in volunteer computing projects [TMN11]. However, the study also identified privacy and

security concerns as primary factors that negatively influence willingness to volunteer in such projects [TMN11].

2.2 WEBASSEMBLY

WebAssembly is a low-level binary instruction format designed to serve as a compilation target for high-level programming languages [Haa+17; Gro19; Gro24b], that promises near-native performance execution in web browsers [Haa+17; Gro19; HH22]. It employs a stack-based virtual machine architecture, operates in a memory-safe sandboxed environment, and interacts with JavaScript of the browser environment through a defined API [Haa+17; Gro19; Gro24b; Cor24c]. Utilizing WebAssembly is particularly effective for computationally intensive tasks in the browser [Gro24b; Gro19] like image processing, game engines, or cryptographic operations.

Furthermore, WebAssembly is bound by the browser's inherent security model [Gro19; Gro24b; Cor24c]. This prevents any application to read and write files or to access device hardware such as cameras or microphones without explicit user permissions.

Leveraging WebAssembly offers several advantages for distributed or edge computing [HH22]. It provides near-native performance, while maintaining platform independence [Haa+17; Gro19; HH22]. These flexible features of platform independence and the support of multiple high-level programming languages targeting the WebAssembly binary format [Gro19] was the motivation behind the development of WebArgo. These features make WebAssembly a fantastic choice and enabling technology for the implementation of a platform like WebArgo. The web environment additionally provides the previously mentioned inherent security model. Therefore, WebArgo potentially provides a higher security for worker nodes than other volunteer computing applications, which require the installation of third-party software directly to the machine. This directly addresses the privacy and security concerns revealed by the study of Toth, Mayer, and Nichols [TMN11] to increase the potential pool of participating workers.

2.3 RELATED WORK

This section presents relevant research and implementations in the fields of volunteer computing and edge computing with WebAssembly or mobile devices. Each project discussed has contributed unique insights and approaches to addressing the challenges inherent in distributed computing systems. The first subsection of this section focuses on historic volunteer computing projects or implementations and also introduces BOINC, which is an actively maintained volunteer computing platform. The summaries of these various related works establish a foundation for WebArgo's motivation and validate WebArgo's approach of distributed computation in the modern web ecosystem.

2.3.1 *Volunteer Computing*

The idea of volunteer computing has been established for a long time. The earliest concept of volunteer computing started already in 1996 [And20]. This section introduces relevant projects in this field, which have been used in the past or are currently in active use and maintained.

2.3.1.1 *SETI@home*

Search for Extra-Terrestrial Intelligence (SETI) at home (SETI@home) represents a pioneering project in the field of distributed and volunteer computing. It utilizes the resources of volunteer computers worldwide to analyze radio telescope data in search of extraterrestrial intelligence. Data from the Arecibo radio telescope is divided into smaller chunks which are distributed to volunteer computers for processing. The system is designed to identify several types of patterns in radio signals that could indicate an artificial origin. These anomalies are spikes, Gaussians, pulsed signals, and triplets in the collected radio data. [And+02]

In the year 2002, SETI@home had engaged over 3.8 million participants across 226 countries, achieving an average computing power of 27.36 TeraFLOPS (Floating Point Operations Per Second (FLOPS)) [And+02].

SETI@home demonstrated the viability of large-scale public-resource computing and identified key factors that make tasks suitable for this approach:

- High computing-to-data ratio
- Independent parallelism
- Error tolerance
- Ability to attract public interest

The success of SETI@home not only advanced scientific research but also increased public awareness and involvement in scientific participation. This project laid the groundwork for present volunteer computing projects and frameworks. [And+02]

The SETI@home project is to this day maintained and actively supported by volunteer computing participants. Currently SETI@home is managed through the BOINC platform [And20], introduced in Section 2.3.1.4.

2.3.1.2 *XtremWeb*

XtremWeb is an experimental global computing platform designed to harness idle computing resources connected to the internet for high-throughput computing. The system aims to provide a platform for experimenting with global computing capabilities and addressing issues such as scalability, heterogeneity, availability, and fault tolerance in massively distributed computing environments. [Fed+01]

The architecture of XtremWeb consists of three main components:

- **Workers**: These are volunteer PCs that execute tasks. They are implemented primarily in Java for portability, with native code used for Operating System (OS)-specific functions. Workers monitor resource availability based on user-defined policies and support multiple platforms, including Linux and Windows.
- **Servers**: These manage tasks and applications. The server design is modular, with components for application pool, job pool, accounting, and scheduling. Servers can be clustered for increased throughput and support specialization for tasks such as dedicated result collection.
- **Clients**: These submit tasks to the system.

XtremWeb employs a two-part scheduling system:

1. **Dispatcher**: Selects tasks from the pool based on application priorities.
2. **Scheduler**: Assigns tasks to workers.

The default scheduling policy of XtremWeb is First In - First Out (FIFO) and XtremWeb implements a timeout to enable rescheduling of aborted tasks [Fed+01].

Furthermore, XtremWeb implements a worker-initiated communication protocol, which facilitates easier deployment through firewalls. This protocol consists of the four main requests: *hostRegister*, *workRequest*, *workAlive*, and *workResult*. [Fed+01]

XtremWeb has been successfully applied to various projects in the past. These include the Pierre Auger Observatory for studying high-energy cosmic rays. In this application, XtremWeb was used to run the *Air Showers Extended Simulation* program by partitioning large simulations into smaller subtasks. [Fed+01]

In summary, XtremWeb provides a flexible and robust platform. Its approach and concept has some similarities to WebArgo, but the original XtremWeb project appears to no longer be maintained since its development in the early 2000s.

2.3.1.3 *EGEE*

Initiated on April 1, 2004, Enabling Grids for E-science in Europe (EGEE) was a project planned to be maintained over four years, involving 71 partners from Europe, Russia, and the United States [Gag04]. The project's primary objectives were to establish a seamless European grid infrastructure for scientific research, provide production-level grid services, and re-engineer grid middleware for enhanced robustness and scalability [Gag04].

The infrastructure of EGEE was built upon the EU Research Network GÉANT [GÉ24], leveraging expertise from previous initiatives such as EU DataGrid, UK e-Science, INFN Grid, Nordugrid, and US Trillium. The

project aimed to expand its computational capabilities from an initial 3,000 CPUs at 10 sites after the first month to 10,000 CPUs across 50 sites by the end of the second active year [Gag04].

EGEE focused on two primary pilot applications:

- The *Large Hadron Collider Computing Grid* for high-energy physics data analysis.
- *Biomedical Grids* addressing challenges in genomic database mining and medical database indexing

The project focused on re-engineering existing middleware to address issues from first-generation implementations and ensure adherence to *Open Grid Services Architecture* standards. This approach aimed to enhance the reliability and scalability of the grid infrastructure [Gag04].

In conclusion, the EGEE project represented a significant effort to transform grid computing from a research concept into a practical infrastructure to support "e-science" across Europe.

2.3.1.4 *BOINC*

Berkeley Open Infrastructure for Network Computing (BOINC) is an open-source middleware system for volunteer computing that enables scientists to create and operate volunteer computing projects while allowing volunteers to participate in these projects [And20]. It addresses key challenges inherent in distributed computing systems, such as dealing with untrusted, unreliable, and heterogeneous computing resources, validating results from potentially malicious hosts, and supporting diverse applications and computing environments.

The BOINC architectural design implements two components that interact through HTTP-based Remote Procedure Call (RPC) interfaces [And20]. These components are the *BOINC Server* and multiple *BOINC Client* components.

At its core, the *BOINC Server* infrastructure consists of: a relational database (typically MySQL or MariaDB) that maintains tables for volunteer accounts, hosts, applications, versions, jobs, and job instances; a scheduler implemented as a Common Gateway Interface (CGI) or Fast Common Gateway Interface (FCGI) program that handles RPCs from *BOINC Clients*; and a shared-memory job cache that typically holds thousands of jobs for efficient dispatch [And20]. The scheduler's design is particularly noteworthy - rather than directly querying the database for each job dispatch, it utilizes a shared-memory segment containing a cache of available job instances, which is continuously replenished by a feeder process [And20]. This architectural decision allows a single *BOINC Server* to efficiently dispatch hundreds of jobs per second. The *BOINC Server* additionally employs multiple components that work together to manage the system:

- **Validator**: compares the output files of results and determines the validity of a result

- **Assimilator**: processes completed and validated jobs and handles their results according to project-specific requirements
- **Transitioner**: manages job state transitions in a finite-state model
- **File deleter**: removes input and output files of completed jobs
- **Database purger**: maintains database efficiency by removing records of completed jobs

The software of the BOINC *Clients* component consists of three main programs that communicate via RPCs over TCP connections: the core client that manages job execution and file transfers, a Graphical User Interface (GUI) (the so-called BOINC *Manager*) that enables user control and monitoring, and an optional screensaver for displaying application graphics [And20]. The BOINC *Clients* implements scheduling policies to manage multiple projects and handle various resource types including CPUs and GPUs [And20]. It also maintains job queues and implements checkpoint/restart capabilities to handle interrupted computations [And20].

Furthermore, BOINC implements a result validation system to handle potential incorrect results caused by hardware errors or malicious behavior of volunteering clients [And20]. At its core, the system employs replication-based validation where each job is processed on multiple unrelated computers. Initially, when a quorum of successful instances is achieved (typically two or more), their outputs are compared. If the outputs agree, one instance is designated as the canonical result and considered correct [And20]. However, due to variations in floating-point hardware and math libraries across different platforms, bitwise-identical results cannot always be expected. Therefore, BOINC allows projects to supply application-specific validator functions that determine result equivalence based on specified tolerances [And20]. For applications with inherent numerical instability, such as physical simulations, BOINC provides a mechanism called homogeneous redundancy, which groups computers into equivalence classes based on their hardware and software configurations, ensuring that subsequent instances of a job are only dispatched to computers within the same equivalence class [And20]. To optimize computational efficiency, BOINC employs adaptive replication of jobs. This adaptive replication reduces the replication factor of a job close to one by identifying clients that consistently return correct results and therefore are trustworthy clients [And20]. The system maintains this "reputation" at the granularity of host-application version pairs, as some computers may be reliable for CPU jobs but unreliable for GPU jobs. This adaptive approach achieves a low error rate while minimizing the computational overhead typically associated with result validation.

The BOINC platform has demonstrated significant success in supporting scientific computing, with approximately 700,000 devices actively participating across various projects [And20]. These devices collectively

provide an average throughput of 93 PetaFLOPS, utilizing about 4 million CPU cores and 560,000 GPUs [And20]. BOINC's versatility is evident in its support for various scientific applications, including standard programs like Autodock, Gromacs, Rosetta, LAMMPS, and BLAST [And20]. The platform can handle applications that use GPUs (through CUDA and OpenCL), multiple CPUs (via OpenMP or OpenCL), and applications running in virtual machines or Docker containers [And20].

Furthermore, BOINC implements a robust security model where applications run under an unprivileged user account, provides checkpointing capabilities to handle interrupted computations, and includes features for monitoring and controlling resource usage [And20]. It is estimated that BOINC projects can achieve approximately 2 PetaFLOPS of computing power at an annual cost of around $100K, which is assumed to be significantly less expensive than equivalent commercial cloud computing solutions [And20].

While both BOINC and WebArgo are volunteer computing platforms, their architectural approaches and implementation strategies differ significantly. BOINC requires volunteers to install a dedicated client application that handles job execution and project management [And20], whereas WebArgo takes a simplified web approach that only requires client devices to have access to an internet connection and a web browser, which supports WebAssembly, WebSockets, and WebWorker. This architectural difference significantly impacts the barrier of entry for volunteers - WebArgo's browser-based approach eliminates installation requirements and potential security concerns associated with executing third-party applications on a personal device. However, BOINC's architecture includes multiple advanced features like adaptive result validation and homogeneous redundancy or the execution of multiple jobs simultaneously, whereas WebArgo's implementation is currently limited in these functionalities. Both systems address the challenge of heterogeneous computing environments, but through different means - BOINC through its complex platform-specific application versions and plan classes, and WebArgo through WebAssembly's inherent platform independence.

Furthermore, while BOINC requires project-specific servers, a significant setup overhead and organizations to pay a significant amount to distribute their projects, WebArgo focuses on simplicity and accessibility for these organizations. WebArgo aims to minimize administrative overhead that every organization can host their own volunteer computing platform to distribute their research projects.

2.3.2 *Edge Computing*

Edge computing represents a paradigm shift in distributed computing architectures by moving computational resources closer to where data is

generated and consumed. This approach is expected to reduce latency, since the data and computing device are physically closer.

The following subsections examine research leveraging WebAssembly or mobile devices for edge computing scenarios.

2.3.2.1 *WebAssembly for Edge Computing*

The work from Hoque and Harras [HH22] investigates WebAssembly as a promising solution for edge computing challenges, particularly addressing the crucial requirements of portability and migratability for such applications [HH22]. In this work WebAssembly is compared to virtual machines, containers, Java, JavaScript, and a native environment as edge computing environment. This work builds upon previous research in edge computing virtualization while introducing novel perspectives on using WebAssembly as an enabling technology for portable and migratable edge applications [HH22]. Their comprehensive analysis demonstrates WebAssembly's capabilities in achieving near-native performance while maintaining platform independence, which proposes a significant advancement compared to traditional virtualization approaches [HH22]. In their work the authors present a systematic evaluation of different WebAssembly execution environments, including browser-based implementations and standalone runtimes, focusing on the performance to portability trade-off.

Their findings highlight WebAssembly's potential as a lightweight alternative to conventional virtual machines or container-based approaches in edge computing scenarios [HH22]. However, they identified that WebAssembly is currently lacking migratability features, compared to virtual machines and containers, to make it a complete solution for edge computing environments [HH22].

Depending on the migration scenario they provide ideas for further actions or development to utilize WebAssembly in edge computing. These four drafts of migration methods as proposed in their work are:

1. **Cold Migration**: The simplest approach where the WebAssembly binary is completely restarted on another new device and only persistent data like disk writes is migrated to the new device. This method can be enhanced to "semi-live" migration by keeping the JavaScript runtime state while restarting only the WebAssembly modules. This method is suited for small, standalone workloads where occasional restarts are acceptable. [HH22]

2. **Interpreter-Based**: Runs WebAssembly code through an interpreter instead of compiling it, making it easier to capture and transfer program state since the interpreter has full visibility into execution. While this enables straightforward live migration, it comes with significant performance overhead, running much slower than a compiled WebAssembly binary. [HH22]

3. **WebAssembly Instrumentation**: Modifies the WebAssembly code to track its own state during runtime. When migration is needed, the instrumented code dumps its state, which is then used to resume execution on the target device. While this maintains better performance than interpretation, it faces challenges with low-level state management and increases code size with each migration. [HH22]

4. **Binary Instrumentation**: Works at the native binary level, where the WebAssembly compiler produces architecture-specific binaries with built-in migration capabilities. The compiler ensures equivalent migration points across different architectures. This requires careful coordination to maintain uniform data and code layout across different platforms but can potentially offer the best performance among migration options. [HH22]

The investigation of this work validates, that the WebAssembly technology can be leveraged to develop a volunteer computing platform like WebArgo. Additionally, WebArgo has implemented a similar approach to the proposed "Cold Migration" method to distribute and reschedule workloads among volunteer participants.

2.3.2.2 *Dynamic Mobile Device Clusters in Edge Femtoclouds*

The work from Habak et al. [Hab+17] presents an enhanced architecture for Femtoclouds that enables mobile device clusters to provide edge computing services while using the cloud for control and management. As described by Habak et al., a Femtocloud is a cluster of co-located mobile devices in places such as public transit, classrooms, or coffee shops that work together to provide edge computing services, similar to a traditional cloud but on a smaller scale [Hab+17]. The key innovation is that these everyday mobile devices can collectively serve as a computational resource when properly managed, offering advantages like lower latency and reduced network congestion compared to traditional cloud services. Habak et al. explored the concept of these edge femtoclouds, which have demonstrated a significant performance improvements through collaborative mobile edge computing in their work [Hab+17]. In their experiment they compared the execution time of a matrix multiplication job on a regular cloud environment compared to their Femtocloud, consisting of 6 mobile devices with the Android OS. The tested prototype was implemented as an Android application. Their experimental results show that a their edge Femtocloud can reduce the job completion time of their matrix benchmark by up to 26% compared to traditional cloud computing approaches, with their implemented checkpointing mechanism further improving efficiency by an additional 31% [Hab+17].

The authors developed a Femtocloud architecture to address the challenges of device churn and system stability. This Femtocloud architecture consists of the following three main components:

- **Femtocloud Controller**: A cloud-based controller that handles registration, job admission, and resource management, serving as a stable interface between job originators and helper devices
- **Femtocloud Helpers**: Mobile devices, which run the client application that executes computation tasks and shares their resources based on user-defined policies
- **Job Managers**: An instance, which is spawned for each accepted job to handle task assignments, monitor progress, and manage task checkpointing, which can run either in the cloud or be migrated to Femtocloud Helpers or job originators for better efficiency

The system uses a hybrid approach where the cloud handles control and management while the actual computation happens at the edge through the mobile device clusters, with all components working together to handle job distribution, resource allocation, and fault tolerance in a dynamic environment where devices can join and leave the system. This system receives a number of external jobs, defined by the so-called "job arrival rate". These received jobs are then distributed among available Femtocloud Helpers.

Their prototype demonstrated robust performance across varying job arrival rates [Hab+17]. While highlighting the benefits of reduced latency and network congestion, their findings also identified key challenges including the need for sufficient device participation and robust security measures [Hab+17]. The work presents edge Femtoclouds as a cost-effective alternative to dedicated edge servers.

Furthermore, the authors mention that implementing an effective user incentive mechanism is one of the open issues that need to be addressed before these Femtoclouds are fully deployable. The paper includes a pilot study on user incentives that surveyed about 50 students to understand what would motivate users to share their resources. They found two main motivators:

- Financial compensation (when supporting for-profit companies) [Hab+17]
- Meaningful causes (76% would participate for scientific purposes, 83% for emergency cases like finding a lost child) [Hab+17]

Habak et al. stated that the support of heterogeneous devices is an important feature for such a application, due to the heterogeneous characteristic of the mobile device environment [Hab+17]. However, the implemented prototype that was built in their work is only supporting android devices as a proof of concept.

The work of Habak et al. has a different implementation approach to WebArgo, but has a similar motivation. The implementation of WebArgo additionally takes the development of custom jobs, simple usability for participants, and support for devices with heterogeneous operating systems into consideration compared to Femtoclouds. However, their experimental results validate that ubiquitous mobile devices can be utilized to perform distributed computation and thereby achieve a performance improvement compared to traditional cloud methods.

3 CONCEPT

This chapter presents the theoretical concept of WebArgo as a foundation for the following chapters. The first section defines the terminology used to describe the various actors and entities of WebArgo. Following this, the next section presents WebArgo's architectural design and its key components. The last section of this chapter introduces the underlying theoretical model of WebArgo, which describes the constraints and conditions necessary to achieve potential performance improvement by utilizing the WebArgo platform.

3.1 TERMINOLOGY

This section defines the fundamental terms used to describe all entities and actors of WebArgo. These precise definitions establish all terms used to describe WebArgo's processes in the following chapters of this work.

3.1.1 *Job*

A job represents a computational problem or project to be processed or solved by participating volunteers of WebArgo. Jobs are required to be parallelizable, hence a job can be divided into multiple tasks for parallel execution of tasks.

3.1.2 *Task*

Each job is divided into multiple tasks and each task ideally has an equal computational workload. Therefore, each task represents a unique, atomic unit of computational workload from its assigned job. Additionally, each task is required to operate independently and therefore requires no communication to other tasks during the execution. These tasks are distributed among the volunteer participants of WebArgo to achieve parallel processing of the entire job.

3.1.2.1 *Batch*

A batch is defined to be a subset of tasks, which are all assigned to the same job.

3.1.3 *Worker*

Workers are volunteers who contribute the computational resources of their consumer device to participate in WebArgo and therefore support jobs computed by the WebArgo platform. They receive tasks from the current batch, compute them using their local resources, and transmit the task results to the WebArgo platform.

3.1.4 *Administrator*

Administrators manage the WebArgo platform. Therefore, they have the ability to control job execution, monitor overall job progress and monitor the connected workers. Additionally, administrators can develop custom jobs to distribute their project through the WebArgo platform.

3.1.5 *Backend*

The backend is used as the central component to maintain the network. Each client (worker & administrator) intending to connect to WebArgo must establish a WebSocket connection to the backend. Additionally, the backend is responsible for scheduling and distributing the tasks of a job among all participating workers. Furthermore, the backend is responsible to manage information about all jobs and to persistently store the results of all completed tasks.

3.1.6 *Frontend*

The frontend enables the connection between workers or administrators and the backend. Furthermore, it represents a interactive UI for the connected clients and is responsible to provide the WebAssembly environment for the workers.

3.2 ARCHITECTURE

The WebArgo platform consists of the following three components:

- Backend (Section 3.1.5) handling the distribution of tasks
- Frontend (Section 3.1.6) providing the interface of the web application for all clients
- Database used to persist job and job progress data

Figure 3.1 illustrates the architecture of WebArgo. The web server that hosts the WebArgo platform serves the backend and the frontend on independent ports. Any heterogeneous clients, diverse in hardware and operating system, are able to connect to WebArgo, if they use a web browser

that supports WebAssembly, WebSockets and WebWorker. These clients access the platform by connecting to the frontend, which then establishes the WebSocket communication between the client and the backend. Each client (worker & administrator) always maintains a bidirectional WebSocket connection to the backend. The database is used to persistently store data about jobs and the progress of each job, and is exclusively accessible by the backend.

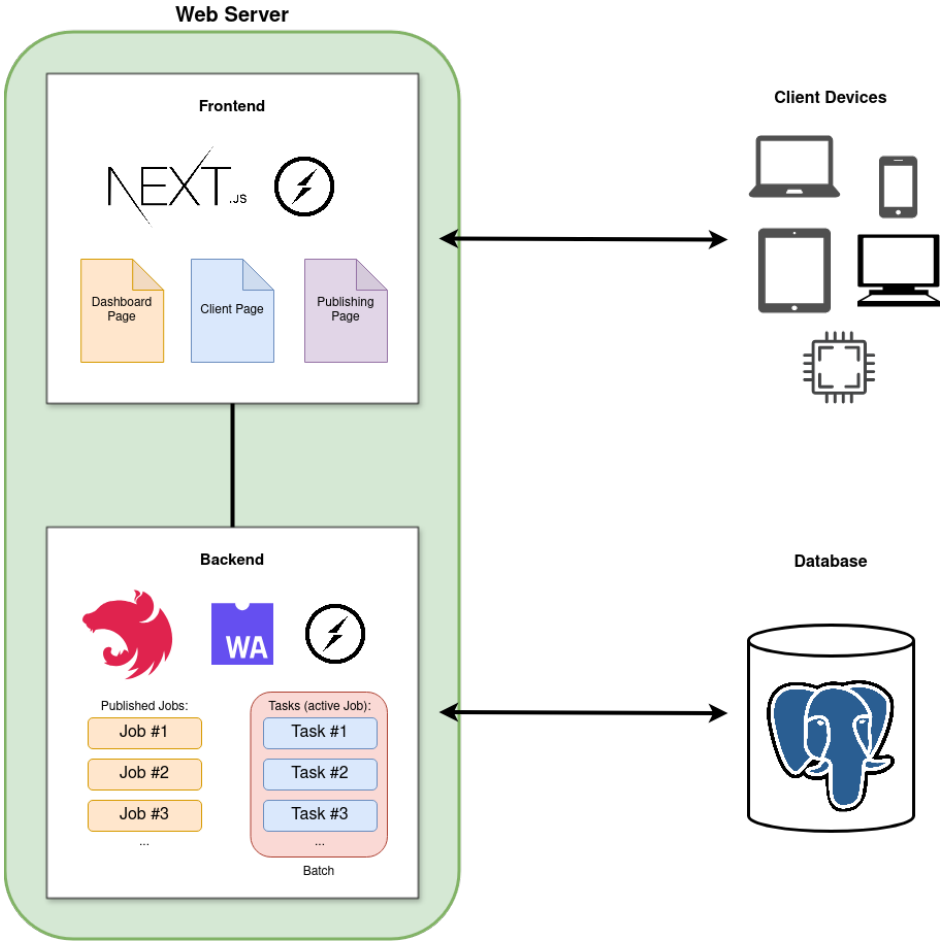


Figure 3.1: Architecture Modell of the Platform

3.3 THEORY

This following section presents the mathematical and theoretical background of the developed volunteer computing platform WebArgo. The objective is to identify the constraints that must be satisfied to ensure that the use of WebArgo offers advantages over regular native computing on a single device.

The primary constraint is that the program code, which is distributed across multiple devices, must be capable of parallel execution. This allows all participating devices to each compute a portion of the overall workload

simultaneously. In this context, a parallelizable job is characterized by the following properties:

- The problem can be divided into multiple tasks
- Each task executes the same program code
- Each task can receive specific input parameters to process a distinct part of the problem
- Each task operates independently, with no need for communication between tasks and no dependencies on other sequentially preceding tasks

The total estimated execution time of a job that is divided into T tasks which are computing an equal amount of workload and are executed sequentially is denoted as t_{Seq}. It is assumed that, on average, each of these tasks requires a computing time of t_{Native} when executed in the native environment. Consequently, the total computation time for the sequential execution of a job on a single device is expressed as:

$$t_{Seq} = T \cdot t_{Native} \tag{3.1}$$

When T equal tasks are distributed among W independent workers, the total computation time is expected to be proportionally reduced, because the workload is being parallelized. This reducing factor I represents the amount of iterations needed to distribute all tasks T over W workers and is described by the fraction of T divided by W. Since T and W are part of the natural numbers $\mathbb{N}$ and a single task can not be split into smaller chunks, I is also part of the natural numbers $\mathbb{N}$. In order to esure that the result of the fraction is always a natural number the Gaussian ceiling function is applied here. Hence, I is given by:

$$I = \left\lceil \frac{T}{W} \right\rceil \tag{3.2}$$

Furthermore, when a problem is distributed across W workers over a network connection, the networking overhead must be taken into account in the computation of the total execution time t_{Dist}. The networking overhead, denoted as t_O, consists of the time required to send a task to a single worker and the time needed to transmit the result from a single worker to the backend. It is assumed that the WebAssembly binary file, necessary to execute a task, is loaded on all workers in advance and therefore, the additional time required for this initialization process is excluded from the calculation. This networking overhead time can be calculated if the network latency L, the network bandwidth B and the file size of the task and result

to be transmitted is known. Assuming the average file size of a single task is F_T and of a single result is F_R the networking overhead can be calculated by:

$$\begin{aligned} t_O &= L + \frac{F_T}{B} + L + \frac{F_R}{B} \\ t_O &= 2L + \frac{F_T + F_R}{B} \end{aligned} \tag{3.3}$$

Additionally, it is assumed that the computation time $t_{Virtual}$ of each task executed in the virtualized environment is equally long across all independent workers W. Thus, the total execution time t_{Dist} for a parallelizable job distributed across W workers can be expressed as:

$$\begin{aligned} t_{Dist} &= I \cdot (t_{Virtual} + t_O) \\ t_{Dist} &= \left\lceil \frac{T}{W} \right\rceil \cdot (t_{Virtual} + t_O) \end{aligned} \tag{3.4}$$

The main objective of this approach is to reduce the total computation time of a job by distributing the workload across multiple workers in parallel. This means that the distributed execution time on multiple workers t_{Dist}, as described by (3.4), must be shorter than the sequential execution time on a single device t_{Seq}, as described by (3.1). This leads to the following inequality expression:

$$\begin{aligned} t_{Seq} &> t_{Dist} \\ T \cdot t_{Native} &> \left\lceil \frac{T}{W} \right\rceil \cdot (t_{Virtual} + t_O) \end{aligned} \tag{3.5}$$

In order to transform this inequality, the term needs to be simplified by substituting the Gaussian bracket. To remove the ceiling function of the Gaussian bracket, the value of I can be overestimated by the following term:

$$I = \left\lceil \frac{T}{W} \right\rceil \quad < \quad \frac{T}{W} + 1 \tag{3.6}$$

This overestimate is used to substitude the value of I in function (3.4) to create the following inequality:

$$T \cdot t_{Native} > \left(\frac{T}{W} + 1 \right) \cdot (t_{Virtual} + t_O) \tag{3.7}$$

This allows the transformation of the inequality in (3.7). The inequality in (3.7) can be transformed as follows, to estimate a maximum threshold value for $t_{Virtual}$:

$$\begin{aligned} T \cdot t_{Native} &> \left(\frac{T}{W} + 1\right) \cdot (t_{Virtual} + t_O) \\ T \cdot t_{Native} &> \left(\frac{T+W}{W}\right) \cdot (t_{Virtual} + t_O) \\ \frac{T \cdot W \cdot t_{Native}}{T+W} &> t_{Virtual} + t_O \\ \frac{T \cdot W \cdot t_{Native}}{T+W} - t_O &> t_{Virtual} \end{aligned} \tag{3.8}$$

In the case of the computation time $t_{Virtual}$ being shorter than this threshold value $\frac{T \cdot W \cdot t_{Native}}{T+W} - t_O$ from (3.8) with a given amount of tasks T and available workers W, the execution time t_{Dist} will be faster than t_{Seq}, resulting in a performance improvement by distributing the workload T among W workers.
Furthermore, to estimate a minimum threshold value for the amount of available workers W, the inequality in (3.7) can also be transformed as follows:

$$\begin{aligned} T \cdot t_{Native} &> \left(\frac{T}{W} + 1\right) \cdot (t_{Virtual} + t_O) \\ \frac{T \cdot t_{Native}}{t_{Virtual} + t_O} &> \frac{T}{W} + 1 \\ \frac{T \cdot t_{Native}}{t_{Virtual} + t_O} - 1 &> \frac{T}{W} \\ \frac{T \cdot t_{Native} - t_{Virtual} - t_O}{t_{Virtual} + t_O} &> \frac{T}{W} \\ W &> \frac{T \cdot (t_{Virtual} + t_O)}{T \cdot t_{Native} - t_{Virtual} - t_O} \end{aligned} \tag{3.9}$$

The last expression through the transformation in (3.9) results in following constraint: Distributing a parallelizable job across multiple workers is only beneficial if the amount of workers W is larger than the estimated threshold value of $\frac{T \cdot (t_{Virtual} + t_O)}{T \cdot t_{Native} - t_{Virtual} - t_O}$ with a given amount of tasks to execute T.

Additionally, the expression from (3.2) can be used to determine the optimal amount of workers W for a job divided in T tasks of equally sized computational workload. The number of iterations I reaches its minimum of 1 if W and T are equal. This outcome is intuitive, as in this ideal scenario each worker is rersponsible for a single task and therefore all tasks T are executed simultaneously. It can be concluded from (3.2) that the amount of workers W should ideally be a divisor of T to ensure a efficient utilization of all workers. However, in practice, it should be considered to utilize additional backup workers to account for potential failures or disruptions.

3.3.1 *Theoretical Example Case*

To illustrate the performance improvement achieved by the volunteer computing platform WebArgo, both expressions t_{Seq} (3.1) and t_{Dist} (3.4) are plotted in Figure 3.2. To calculate these graphs, an average computation time t_{Native} for a single task was assumed to be 34.97 s, based on the measured average run time of the source code in Section A.2.1 on the system specified in Section A.1.1. This measurement was repeated with a WebAssembly binary, generated with the source code in Section A.2.2. The average computation time $t_{Virtual}$ of the same task in a browser environment (Mozilla Firefox 132.0 [Moz24a]) on the system specified in Section A.1.1 was measured at 71.17 seconds, being about 2.04 times slower than the native computation time.

This is an unexpected difference in computation duration between native and WebAssembly code execution, since WebAssembly is promised to perform at near-native speed [Haa+17; Gro19]. Jangda et al. stated in their work that WebAssembly shows a significantly worse performance compared to native code, with an average performance gap of 45% (Firefox) to 55% (Chrome) and peak slowdowns of 2.08x (Firefox) and 2.5x (Chrome) across measured SPEC CPU benchmarks [Jan+19]. This matches the unexpected slowdown of 2.04x measured in the previously mentioned experiment of this work. Jangda et al. identified the following performance issues to be the root cause of the experienced performance gap when using WebAssembly [Jan+19]:

1. *Increased Register Pressure*
2. *Extra Branch Instructions*
3. *Increased Code Size*

To calculate the networking overhead t_O an internet latency L of 32 ms [Inc24a] and a network bandwidth B of 29 Mbps (3.625 MBps) [Inc24a] was assumed. These values represent the Internet Quality Index (IQI) estimated round trip time within Europe [Inc24a] and the IQI estimated download speed in Europe [Inc24a]. In this theoretical scenario a task represents only the input arguments required to execute the WebAssembly module, as described in the source code of Section A.2.2. These input arguments consists of seven float values, represented as array of strings with a size of 7. Since this is a comparatively very small file size, the task file size F_T is assumed to be 0. The PNG file generated through the execution of the source code in Section A.2.2, displayed in Figure 4.7, has a size of 699.9 KB, representing the result file size F_R. Using the function in (3.3) and based on these values the networking overhead t_O is estimated to be 0.257 seconds.

Additionally, in the modeling of this theoretical case is an initial offset time considered, which estimates the time needed to initialize the WebAssembly

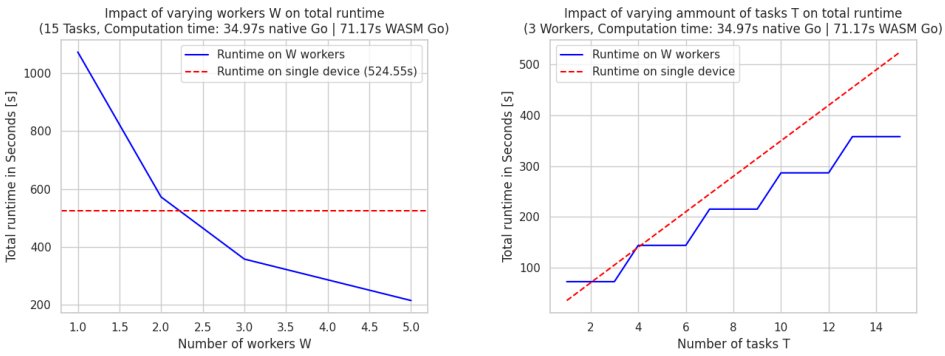
environment for all participating workers of WebArgo. In this process, each worker is required to download all necessary files in advance to be able to execute the workload of a task. Since this setup process occurs in parallel across all workers, this additional offset time matches the time required for the setup of a single worker. This offset value is again calculated using the IQI estimated round trip time and download speed in Europe [Inc24a]. The size of the compiled WebAssembly binary file from the source code in Section A.2.2 is 2.7 MB, and the corresponding *wasm_exec.js* JavaScript glue code file to setup the WebAssembly environment is 16.7 KB, resulting in a total file size of about 2.717 MB. The time required to download both files to a single node can be estimated using the following expression:

$$DownloadTime = Latency + \frac{Filesize}{Bandwith} \tag{3.10}$$

Since the bandwidth is given in megabytes per second, while the latency is provided in milliseconds, the latency value needs to be converted to seconds. Using the calculation in (3.10), the additional offset that is required to set up the platform, in this theoretical case, is estimated to be at least 0.782 seconds.

The amount of all tasks T is set to 15. Using the inequality in (3.9) the threshold of workers W is calcualted to be $W > 2.36$, hence at least three available workers are required to achieve a performance improvement in this theoretical case.

Both graphs in Figure 3.2 have been calculated based on the previously mentioned values and the inequality terms (3.1) and (3.4) of Section 3.3.



(a) Variable amount of workers W

(b) Variable amount of tasks T

Figure 3.2: Total Computation Time for an Estimated Example Case

The graph in Figure 3.2a represents the total runtime, on the Y-axis in seconds, as a function of the number of workers W in the network, on the X-axis. The red dotted line displays the total execution time t_{Seq} on a single machine, while the blue line illustrates the theoretical total execution time t_{Dist} of WebArgo. The total number of all tasks T is set to 15. This plot validates the expectation, that WebArgo achieves a faster total computation time than a single device, if three or more workers are available. The total

runtime t_{Dist} is already by about 35% faster than t_{Seq} when the workload is distributed to three nodes.

In summary, this plots illustrates, that the total computation time t_{Dist} is already almost even to the threshold value of t_{Seq} if two workers actively participate in WebArgo and faster if three or more workers participate. This holds true even if the computation time $t_{Virtual}$ of a single task is more than twice as slow in the WebAssembly environment than the computation time t_{Native} on a native system. This proves that the approach of distributing the workload has a performance improvement in this theoretical example, if multiple workers participate in WebArgo.

However, it needs to be considered, that if only one or two workers participate in WebArgo this approach will always be slower in this example due to the networking overhead, initial offset time and the higher computation time for each individual task.

The other graph in Figure 3.2b displays the total runtime on the Y-axis in seconds as a function of the number of tasks T on the X-axis. The red dotted line represents again t_{Seq} of a single device and the blue line t_{Dist} of the volunteer computing network. The number of workers W is set to three. This plot demonstrates that WebArgo achieves in this theoretical example a faster total computation time than a single device when the workload T consists of two, three or more than four tasks. The performance gain grows exponentially as the number of tasks increases. Furthermore, it can be concluded that the total computation time t_{Dist} of WebArgo increases in steps, depending on the condition if the number of tasks T and the number of workers W have a common divisor.

However, it is crucial to note that the WebArgo platform will have a longer computation time t_{Dist} in every scenario with only one task, as a single task cannot be executed in parallel. Also in the scenario of exactly four tasks T and three workers W the total computation time t_{Dist} is longer than t_{Seq}.

4

METHODOLOGY

The overall process of developing the WebArgo platform presented in this work consisted of the following steps:

1. Theoretical design of WebArgo's architecture
2. Development of a Proof of Concept (POC) prototype
3. Development process from the POC to the WebArgo platform
   - Implementation of API and web interface
   - Usage of generic job and task entities to support any kind of custom job
   - Add WebAssembly Support for multiple programming languages
   - Implement support for multiple datatypes of task results (primitive datatypes, lists, files as binary)
4. Deployment of the web application
   - Orchestrate all components of the WebArgo platform with Docker Compose [Doc24]
   - Host the project on a cloud services (*openstack* from Darmstadt University of Applied Sciences (h_da))
   - Implement user entity with specific user roles to enable security
   - Implement security measures through authentication & authorization
5. Enhance robustness of the platform
   - Implement system recovery measures and persistence of critical data
   - Adjust scheduling algorithm to ensure the completion of jobs, even when workers are unreliable
   - Eliminate bugs through testing the application
6. Implement the visualization of the Mandelbrot set in multiple languages as the benchmark job
7. Evaluation of the platform through multiple experiments

The content of this chapter introduces and describes all the technologies, frameworks, and tools utilized in the development of the WebArgo volunteer computing platform, along with the reasoning for their selection. At first Section 4.1 provides the framework selection for the three components

of WebArgo's architecture, as described in Section 3.2. Then Section 4.2 describes each of the three key web technologies and corresponding tools utilized in the implementation of WebArgo to establish a foundation for the following Chapter 5, which focuses on the implementation. Finally Section 4.3 of this chapter introduces the approach to benchmark WebArgo, which is used in Chapter 6 to evaluate the developed platform.

4.1 FRAMEWORKS

This section introduces the frameworks selected for the development of all components of the WebArgo platform. The following criteria were used to guide the selection of a suitable framework for the backend, frontend as well as the database:

- The framework is well-tested and provides a stable Long Term Support (LTS) version.
- The framework is popular among web developers.

4.1.1 *Backend*

It was crucial for the backend framework to be popular among web developers. Working with a popular framework improves the development process by ensuring the availability of detailed educational resources online as well as various online support forums. Additionally, a widely used framework increases the likelihood that the platform can be maintained or further extended by other programmers.

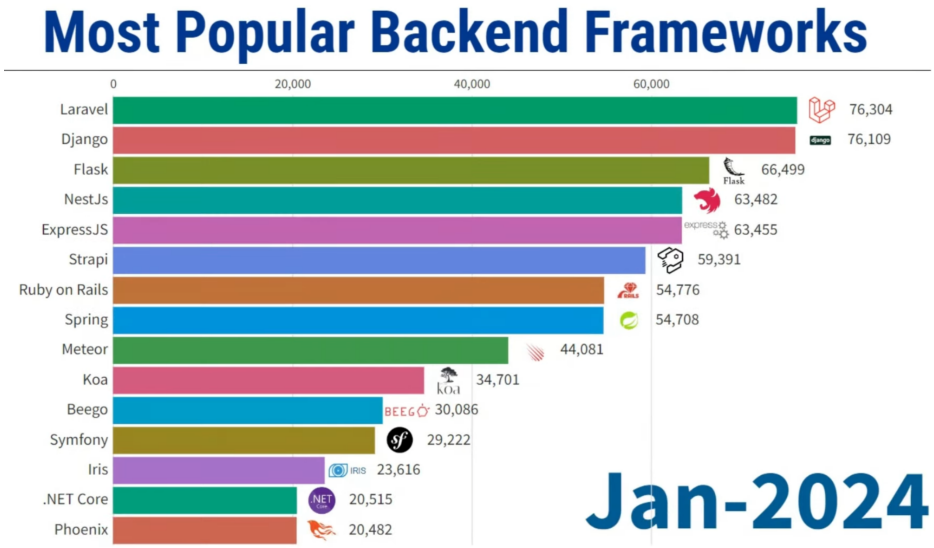
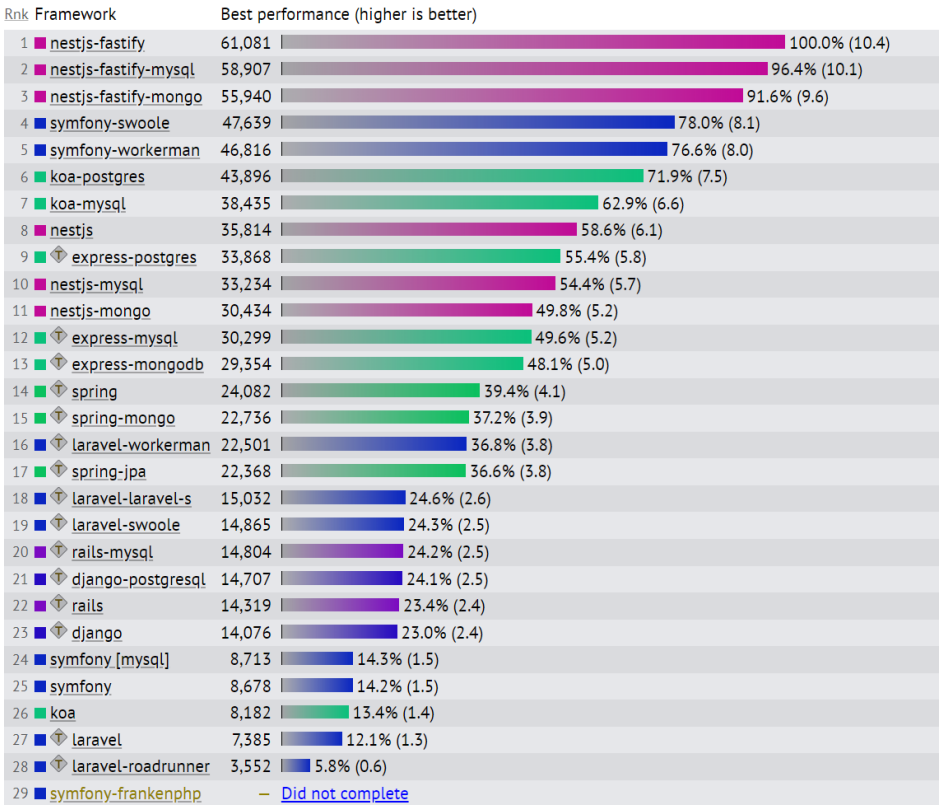


Figure 4.1: Most Popular Backend Frameworks (Jan 2024) by GitHub Stars ["St24]

The 15 most popular backend frameworks of January 2024 are displayed in Figure 4.1. The popularity for each framework of this list is calculated by

the number of GitHub Stars from repositories listed in a GitHub Archive ["St24]. The selection options of the backend framework were based on this popularity list.

In addition to the previously stated criteria, the selected backend framework needed to meet specific performance requirements. It was essential for the framework to efficiently handle multiple connected clients with minimal latency. Furthermore, the framework's internal computation speed was critical, particularly for preparing input arguments for each task and managing task scheduling across all clients. The goal was to reduce overhead as much as possible to ensure high performance.



Rnk	Framework	Best performance (higher is better)	
1	nestjs-fastify	61,081	100.0% (10.4)
2	nestjs-fastify-mysql	58,907	96.4% (10.1)
3	nestjs-fastify-mongo	55,940	91.6% (9.6)
4	symfony-swoole	47,639	78.0% (8.1)
5	symfony-workerman	46,816	76.6% (8.0)
6	koa-postgres	43,896	71.9% (7.5)
7	koa-mysql	38,435	62.9% (6.6)
8	nestjs	35,814	58.6% (6.1)
9	express-postgres	33,868	55.4% (5.8)
10	nestjs-mysql	33,234	54.4% (5.7)
11	nestjs-mongo	30,434	49.8% (5.2)
12	express-mysql	30,299	49.6% (5.2)
13	express-mongodb	29,354	48.1% (5.0)
14	spring	24,082	39.4% (4.1)
15	spring-mongo	22,736	37.2% (3.9)
16	laravel-workerman	22,501	36.8% (3.8)
17	spring-jpa	22,368	36.6% (3.8)
18	laravel-laravel-s	15,032	24.6% (2.6)
19	laravel-swoole	14,865	24.3% (2.5)
20	rails-mysql	14,804	24.2% (2.5)
21	django-postgresql	14,707	24.1% (2.5)
22	rails	14,319	23.4% (2.4)
23	django	14,076	23.0% (2.4)
24	symfony [mysql]	8,713	14.3% (1.5)
25	symfony	8,678	14.2% (1.5)
26	koa	8,182	13.4% (1.4)
27	laravel	7,385	12.1% (1.3)
28	laravel-roadrunner	3,552	5.8% (0.6)
29	symfony-frankenphp	–	Did not complete

(a) Best responses per second (2023-10-17) [Baca]

Figure 4.2: Frameworks from Figure 4.1 Ranked by Performance

To identify a high-performance framework, the most popular backend frameworks listed in Figure 4.1 were compared by using two independent benchmark sources. The results of these benchmarks are presented in Figure 4.2, where the frameworks are ranked from top to bottom based on their performance. It is important to note that some frameworks from the list in Figure 4.1 were not part of these specific benchmarks. Both benchmark sources simulated numerous clients sending requests to each backend and measuring the number of successful responses per second in order to determine the performance [Bacb; Baca].

Language	Framework	▼ Requests / Second (64)	Requests / Second (256)	Requests / Second (512)
go (1.23)	beego (2.3)	266 819	310 719	317 611
javascript (ES2019)	nestjs-fastify (10.4)	195 213	197 940	196 182
elixir (1.17)	phoenix_bandit (1.7)	191 702	195 441	192 252
javascript (ES2019)	koa (2.15)	179 877	184 585	182 558
kotlin (2.1)	spring (3.4)	165 775	160 106	138 873
elixir (1.17)	phoenix_cowboy (1.7)	145 931	144 436	142 188
javascript (ES2019)	express (4.21)	88 060	87 501	85 293
javascript (ES2019)	nestjs-express (10.4)	66 673	65 978	64 531
ruby (3.3)	rails (8)	9 538	9 359	9 094
php (8.3)	laravel (11.34)	7 944	7 860	7 791
php (8.3)	symfony (7)	4 803	4 799	4 735
python (3.13)	flask (3.1)	3 046	2 753	3 935
python (3.13)	django (5.1)	2 804	2 647	3 621

(b) Requests/Second (2024-06-25) [Bacb]

Figure 4.2: Frameworks from Figure 4.1 Ranked by Performance

Finally, NestJS [Mys24] with Fastify was selected as the framework to implement the backend component. It performed exceptionally well in both benchmarks, ranking on first place in Figure 4.2b and second place in Figure 4.2a. Additionally is the NestJS framework a popular choice among developers, as shown in Figure 4.1.

4.1.2 *Frontend*

The criteria stated in Section 4.1 also applied for the selection of a framework for the frontend. Figure 4.3 shows the 36 most used frameworks among web developers in July 2024 [Sta24a]. This statistic was collected by Stack Overflow and is the result of a survey. In this survey web developers have been asked which web frameworks and web technologies they had been working with in the past year, and which do they want to work in over the next year [Sta24a]. The two most popular options are, by far, Node.js and React with each around 40% of votes. While Node.js is mainly used for backend development React is a JavaScript library used for frontend development. This qualifies React as the choice of technology for the frontend development. Ranking fourth in Section 4.1 is Next.js with 17.9% of votes. Next.js is a web framework based on React [Inc24b] and according to the survey the most popular React web framework in the year 2024. Therefore Next.js was selected as the framework for frontend development.

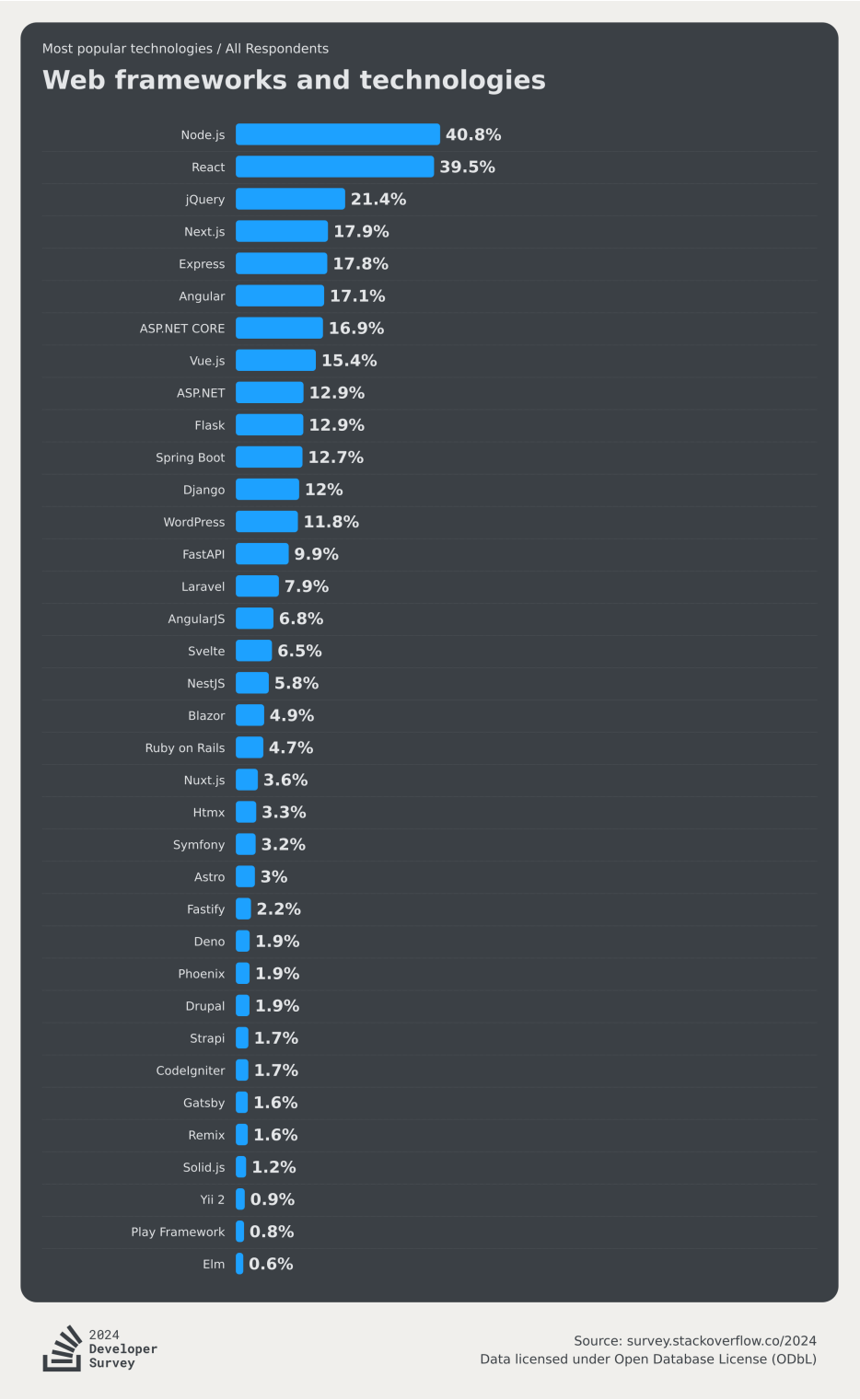


Figure 4.3: Most Used Web Frameworks and Technologies in 2024 (Developer Survey from Stack Overflow) [Sta24a]

4.1.3 *Database*

An object-relational database system fulfills WebArgo's requirements, as these systems typically offer high performance capabilities and the entities described in Section 3.1 can be represented with an object-relational database schema.

PostgreSQL is an open source object-relational database system that has been actively developed for more than 35 years [Gro24a]. Therefore it has gained a strong reputation for reliability, feature robustness, and performance [Gro24a], hence PostgreSQL was selected as the database management system for this implementation.

4.2 WEB TECHNOLOGIES

The implementation of WebArgo leverages three key web technologies that are fundamental to its functionality in the web environment. These web technologies enable the WebArgo platform to efficiently distribute computational workloads across heterogeneous devices while maintaining platform independence and security. The following subsections present each technology and corresponding tools, discussing their specific roles and contributions to WebArgo.

4.2.1 *WebAssembly*

As presented in Section 2.2, WebAssembly serves as the core enabling technology for the development of WebArgo. WebAssembly is a low-level binary instruction format designed to serve as a compilation target for multiple high-level programming languages, promising near-native performance execution in supporting web browsers [Haa+17; Gro19; Gro24b].

The characteristics of WebAssembly make it particularly suitable for WebArgo's requirements. Its platform independence provides the flexibility needed in a volunteer computing environment with heterogeneous consumer devices. Additionally, WebAssembly's support for multiple high-level programming languages as compilation targets enhances the usability for administrator users to develop custom WebArgo jobs in their preferred programming language. Furthermore, WebAssembly maintains the browser's inherent security model, that prevents any application from accessing the hardware or the file system of the client device without explicit user permissions. This security feature directly addresses privacy concerns identified by Toth, Mayer, and Nichols in their study about participation in volunteer computing [TMN11], potentially increasing participant trust and willingness to actively support a WebArgo project.

Figure 4.4 displays the browser compatibility of WebAssembly with multiple major browsers [Cor24c]. The check mark indicates WebAssembly support

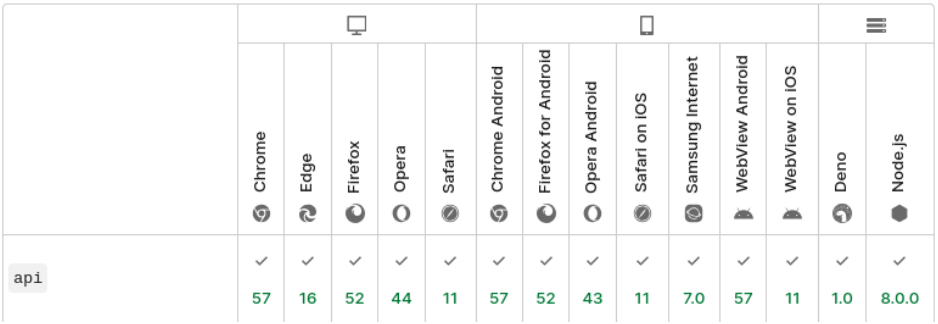


Figure 4.4: Browser Compatibility: WebAssembly [Cor24c]

and the green number below marks the version since WebAssembly support was implemented. All browsers are sorted by desktop and mobile version.

4.2.1.1 *emscriptren for C and C++*

The toolchain, used in this work, to compile C or C++ code into the WebAssembly binary format is emscripten. The distributors of emscripten already focused on the compilation of C and C++ code to *asm.js* - a predecessor of WebAssembly - in the past [Con24c]. This compiler infrastructure has evolved to target the WebAssembly format, enabling the execution of native C/C++ applications in web browsers. Furthermore, emscripten provides the necessary JavaScript glue code to initialize the WebAssembly environment according to the compiled source code [Con24c].

4.2.1.2 *Go*

The standard Go compiler inherently provides a option to directly target the WebAssembly format during the compilation process [Con24a]. This was utilized to support Go applications in WebArgo. Unlike emscripten, the Go compiler is not additionally generating a specific JavaScript glue code file for each Go application. However, Go provides a general JavaScript glue code file suitable for all generated WebAssembly binaries generated by the Go compiler [Con24a]. The file size of this general Go JavaScript glue code is surprisingly small compared to the JavaScript glue code generated by emscripten.

4.2.1.3 *Pyodie for Python*

Pyodide is described to be a port from CPython to WebAssembly/Emscripten [cM24]. It is a JavaScript library that provides a robust foreign function interface to compile python code to WebAssembly and execute it inside the browser environment [cM24]. Additionally it enables the installation and execution of Python packages inside the browser using micropip [cM24]. Also Pyodide promises, that the executed Python code has full access to the Web APIs [cM24], which is not an default feature of emscripten or Go.

Unfortunately, Pyodide performed relatively slow compared to emscripten or Go during the testing of this work. Section 6.4 further investigates the performance of these WebAssembly environments.

4.2.2 *WebSockets*

WebSockets play a crucial role for the communication infrastructure of WebArgo. Different from standard HTTP requests enable WebSockets a faster, bidirectional and full-duplex communication between clients and servers [Cor24a; PN12; Con24b]. Unlike traditional HTTP request-response patterns, WebSocket are establishing a persistent connections between client and server [PN12], allowing real-time data transmission in both directions with minimal overhead after the initial handshake [PN12]. This protocol utilizes the ws:// or wss:// (secure) Uniform Resource Identifier (URI) scheme and efficiently handles scenarios requiring live updates such as financial trading platforms, multiplayer games, or chat applications, with significantly reduced latency compared to polling mechanisms [PN12].

The real-time capability is leveraged for efficient task distribution, progress monitoring, and result collection in the implementation of the WebArgo platform. Additionally, is the bidirectional and full-duplex communication model extremely useful to handle the transmission of task and results between the backend and workers. This replaces inefficient polling strategies and therefore enhances the performance of the communication process between backend and workers.

Figure 4.5: Browser Compatibility: WebSocket [Cor24a]

Figure 4.5 displays the browser compatibility of WebSocket with multiple major browsers [Cor24a]. The check mark indicates WebSocket support and the green number below marks the version since WebSocket support was implemented. All browsers are sorted by desktop and mobile version.

4.2.2.1 *Socket.IO*

Socket.IO [Con24b] is a popular JavaScript library that implements the usage of WebSockets on both the server and client side. NestJS [Mys24], the selected framework to implement the backend, has an inherent support

of Socket.IO, by utilizing the *NestJS-Gateways* library [Mys24]. Therefore the Socket.IO library was selected to implement WebSocket inside the WebArgo platform.

4.2.3 *WebWorker*

WebWorkers can be accessed through a specific JavaScript API, enabling concurrent execution of scripts in background threads separate from the main browser UI thread [Cor24b]. These WebWorkers operate in an isolated context, communicating with the main thread through a message-passing interface, and therefore cannot directly access the HTML DOM of the web page [Cor24b].

This implementation of parallel processing through WebWorkers prevents the computationally intensive WebAssembly execution from blocking the user interface of a worker. This enables the worker to monitor the currently executed task and prevents an unpleasant user experience caused by an unexpectedly frozen screen. Furthermore, the WebWorker API allows to create multiple WebWorkers in parallel. This feature could be used in the future to implement a browser-based multi-threading solution for WebArgo.

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS	Deno	Node.js
Worker	✓ 2	✓ 12	✓ 3.5	✓ 10.6	✓ 4	✓ 18	✓ 4	✓ 11	✓ 5	✓ 1.0	✓ 4.4	✓ 5	✓ 1.0	12.17.0 …

Figure 4.6: Browser Compatibility: WebWorker [Cor24b]

Figure 4.6 displays the browser compatibility of WebWorker with multiple major browsers [Cor24b]. The check mark indicates WebWorker support and the green number below marks the version since WebWorker support was implemented. All browsers are sorted by desktop and mobile version.

4.3 BENCHMARK: VISUALIZING THE MANDELBROT SET

To benchmark the platform's performance, a computationally intensive job is implemented and executed across multiple connected workers. The total execution time is compared to the execution time of the same job on a single machine with native source code. The visualization of the Mandelbrot set represents all characteristics previously described in section 3.3, making it suitable as a job for this benchmark.

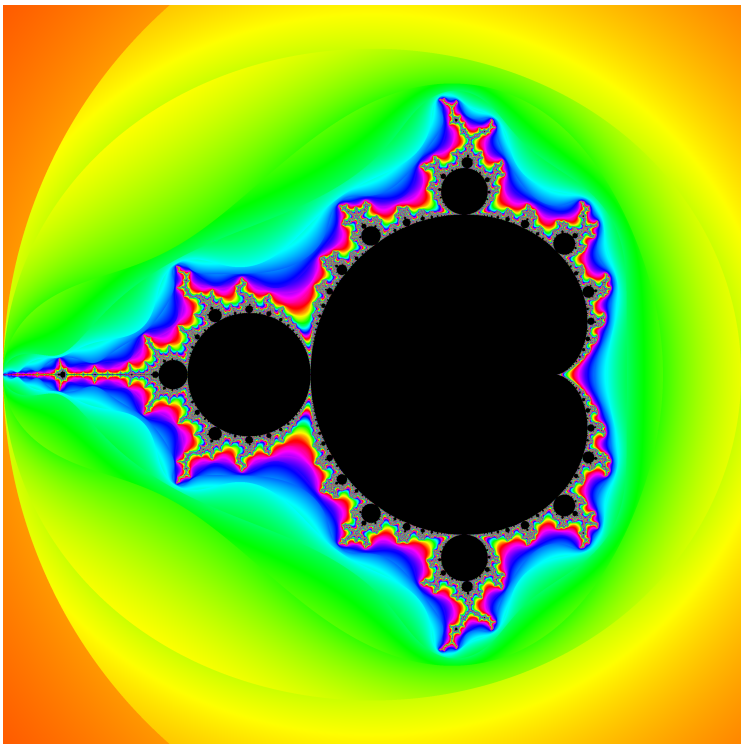


Figure 4.7: Mandelbrot Set (Generated with Code in Section A.2.1)

The Mandelbrot set is a famous subset of the complex numbers $\mathbb{C}$. Figure 4.7 displays a colorized visualization of the set in the plane of complex numbers. To determine if a complex number c is part of the Mandelbrot set, c is applied to the function (4.1) with $z_0 = 0$.

$$z_{n+1} = z_n^2 + c \tag{4.1}$$

If the value of z_{n+1} does not diverge over n iterations, c belongs to the Mandelbrot set. In Figure 4.7, all complex numbers c, for which z_{n+1} remains bounded over n iterations are colored black. All other c are colored based on the number of iterations required for z_{n+1} to diverge, with the color spectrum ranging from red (low iteration count) over to blue (high iteration count) indicating increasing iteration counts.

The calculation and visualization of the Mandelbrot set can be partitioned into multiple tasks. Each task executes the same source code but with different input parameters, which define a unique two-dimensional area in the complex plane. These tasks can be executed in parallel due to the independence of calculations between different areas. The implementation for this benchmark is provided in Appendix A. The source code for the native Go variant can be found in Section A.2.1 and its Go-to-WebAssembly counterpart in Section A.2.2 as well as the C++-to-WebAssembly version in Section A.2.4.

5 IMPLEMENTATION

This chapter presents the technical implementation of the dynamic and heterogeneous volunteer computing platform WebArgo in detail. The implementation leverages modern web technologies introduced in Section 4.2 to create a high-performance distributed computing solution for the web. At its core, the system utilizes WebAssembly for cross-platform compatibility, high-performance, and language independence (Section 4.2.1), WebSockets for efficient bidirectional communication (Section 4.2.2) and WebWorkers to enhance user experience and to enable potential parallel Task execution (Section 4.2.3).

The following sections describe WebArgo's communication protocol, persistence mechanisms, scheduling strategies, security measures and the challenges encountered during the implementation process. Section 5.5 and Section 5.6 describe the implementation and usage of WebArgo's API and web application interface.

5.1 WEBARGO'S COMMUNICATION PROTOCOL

When workers or administrators establish a connection to the platform by accessing the frontend application, a WebSocket connection to the backend is automatically initiated. This enables real-time and bidirectional communication between the backend and these clients. Therefore, this connection is used to send tasks from the backend to the workers as well to send the result of each task from the workers back to the backend. Furthermore, administrators receive real-time data about all connected workers and the current status of each job through their WebSocket connection.

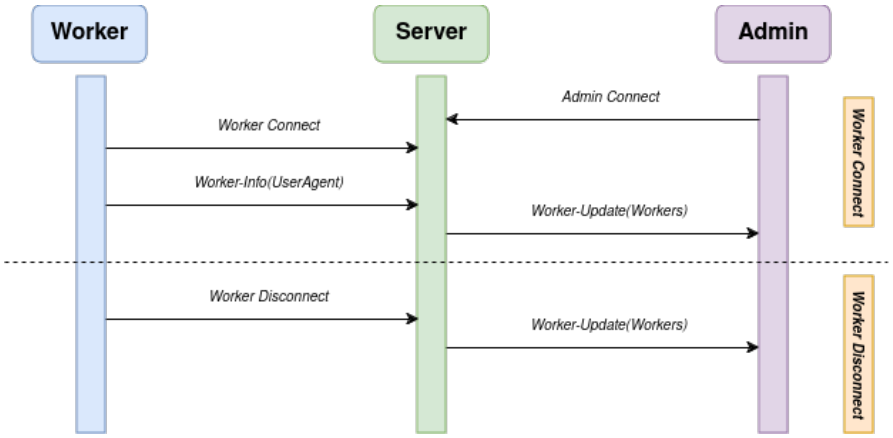


Figure 5.1: Communication: Connection of Worker & Real-time Update for Administrator

The first sequence in Figure 5.1 illustrates the process of a worker establishing a connecting to the backend in WebArgo. Upon successful connection, the worker transmits all available information regarding its hardware and operating system in form of the browser user agent to the backend. After this initialization of the worker, all previously connected administrators automatically receive an updated list of all connected workers. Similarly, if a Worker disconnects a automatic update is send to all administrators, as described by the second sequence in Figure 5.1.

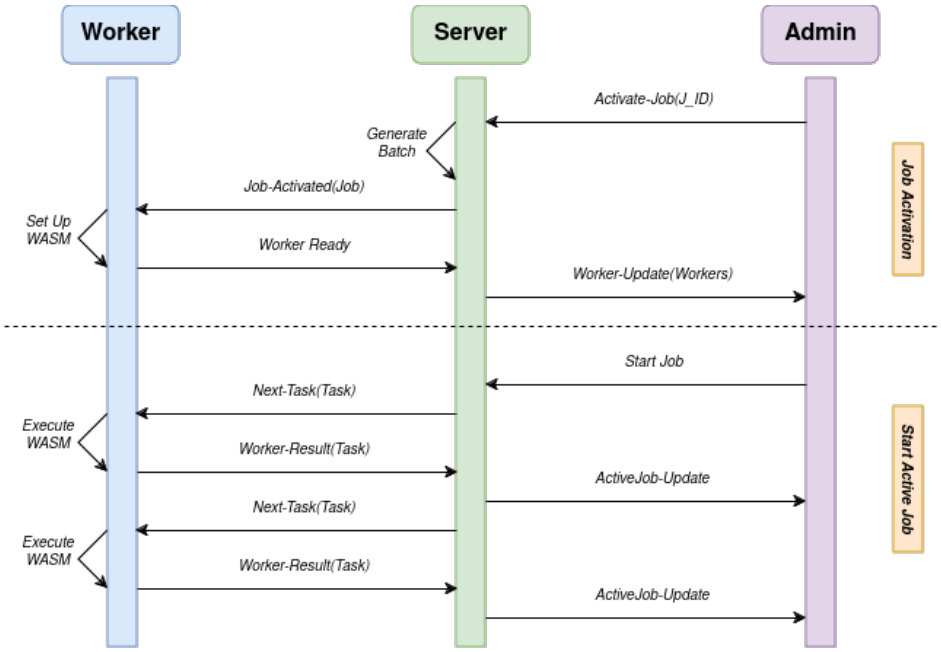


Figure 5.2: Communication: Administrator Starts Job & Worker Executes Tasks

Figure 5.2 illustrates the job initialization and execution process. The first sequence describes the job activation process and is initiated when an administrator changes a jobs status to *ACTIVE*. The backend then generates a batch of tasks for this specific job and notifies all connected workers by transmitting a object of this activated job to them. Upon receiving this notification, each worker fetches the corresponding WebAssembly binary and JavaScript glue code file from the backend via HTTP request to initializes a WebWorker with the specific WebAssembly environment for this currently active job. When this step is successfully completed the worker notifies the backend, which then marks this workers as *ready*. This update is also forwarded to all administrators with an active WebSocket connection to the backend.

The second sequence in Figure 5.2 illustrates th process of starting an active job. This process is initiated when an administrator changes the status of an active job to *RUNNING*. After that begins the backend to distribute unique tasks from the current batch to all workers who have been marked as *ready* by transmitting the *Worker Ready* message and therefore already have successfully completed the initialization process of the currently active job. Each worker executes its assigned task, then appends the corresponding

result to the task object and transmits the completed task back to the backend application. If unprocessed or unscheduled tasks remain in the batch at the moment of receiving the completed task, the backend responds by assigning the next pending task to this worker. Additionally, the backend notifies all administrators after each successfully completed task about the current progress of the *RUNNING* job. This enables monitoring of the jobs progression in real-time for all administrators.

If a Worker is connecting to WebArgo while there is already an active or running job, the communication sequence is automatically executed identically. Accordingly, the worker starts to initialize this job and then proceeds to process tasks of the corresponding batch, hence enabling dynamic participation in ongoing jobs for workers.

Each job has a timeout attribute for its tasks. Scheduled tasks that remain incomplete after their allocated timeout period can be redistributed to a different worker. This mechanism enables the rescheduling of tasks, which have been assigned to a malfunctioning worker or a worker that has disconnected before completing its assigned task. Additionally, this approach allows rescheduling of tasks that have been assigned to slower workers, known as stragglers, to more efficient workers. Hence, this mechanism can optimize the overall execution time of a job.

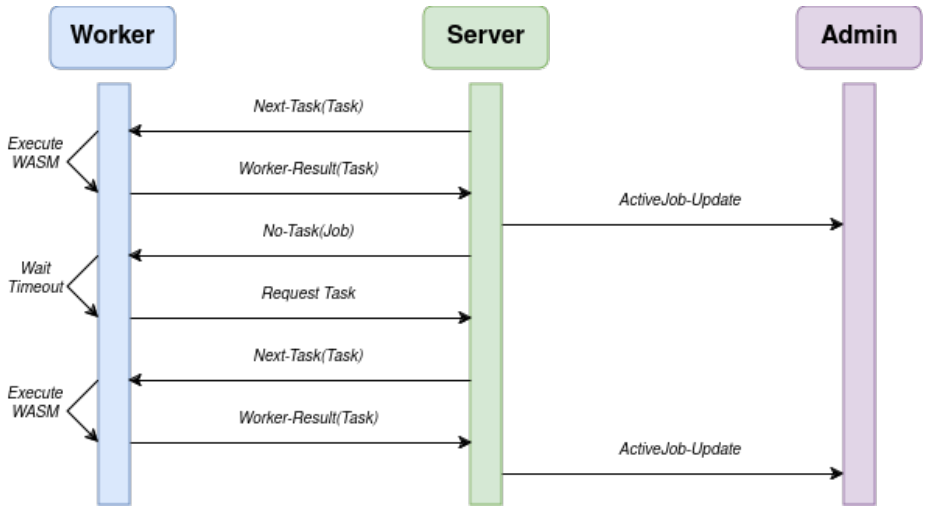


Figure 5.3: Communication: Rescheduling of Tasks after Timeout

Figure 5.3 illustrates the case, if all tasks of the batch already have been scheduled, but one or more are not yet completed. When a worker transmits a completed task to the backend - and therby is expecting to receive a new task to compute from the backend - while the timeout period of all other scheduled tasks has not expired, the backend responds with the *No-Task* message accompanied by the job object of the currently active job. The worker extracts the status and the timeout value from this job objects and initiates a waiting period equivalent to the timeout value plus a randomized overhead, if the jobs status is still *RUNNING*. This additional randomized

overhead time prevents simultaneous task requests, if multiple workers are waiting for a task at the same time. After the completion of the waiting period, the worker transmits a *Request Task* message to the backend. The backend responds either with a newly available task or another *No-Task* message, repeating this sequence until the current batch is fully processed.

5.2 PERSISTENCE

This section describes the persistent data storage implementation of the platform. The primary adressed objective is to ensure system recovery capabilities in cases of system failures, unexpected system crashes, or scheduled system restarts. The following data objects have been identified as critical for a system recovery:

- Job data
- Task input arguments
- Task results
- User credentials

Information regarding workers, their hardware specifications and operating systems is intentionally excluded from permanent storage.

WebArgo implements a robust persistence mechanism for job progression. When a running job is stopped, the current progress and all task results are automatically saved, hence actively stopping a running job initiates the persistence process for this job's current state. This functionality should be utilized if the platform is scheduled to be restarted or terminated.

Additionally, the system persists the progress of a currently running job after each batch completion. This mechanism enables periodic job backups, as each batch completion establishes a save point. Consequently, the batch size determines the job's fallback tolerance in the event of an unexpected system failure.

It is noteworthy that from a worker's perspective, task progress can not be persisted. Since WebAssembly is executed in a save sandbox environment the code has usually no access to local files. In the event of an unexpected crash of the worker's WebAssembly process or browser, the progress of the affected task has not been persisted on the workers device and therefore will be lost.

5.2.1 *Data in Files*

The system utilizes text files for persistent storage of task input arguments and task results. Each job's critical information about its tasks is stored in a separate text file for task input arguments and the task results on the web server.

The method to generate a new batch of the active job reads the corresponding task input arguments text file line by line, where each line represents an input argument. For each line, the method creates a unique task and adds it in sequence to the batch.

When the progress of a job is persisted, the backend writes the results of all completed tasks within the batch to the corresponding task results text file. Each result is written in a new line, and the results are stored in sequential order.

Additionally, the backend implements a special mechanism to handle tasks that produce files as results. When a worker transmits such a completed task, the backend immediately stores the result file in a result directory corresponding to the running job. Subsequently, when later a jobs progress is saved, the backend stores the file path to each result in the corresponding task results text file instead of the actual result data.

5.2.2 *Database*

The PostgreSQL [Gro24a] database is used to store the user credentials as well as each job object. Each job object contains a progress attribute that represents a pointer indexing the last completed task in the task sequence. During job recovery, this progress value determines the starting point for the first task in the new batch.

Furthermore, the following subsections describe the implementation of all entities that are used in WebArgo, as introduced in Section 3.1.

5.2.2.1 *Job*

The job entity represents a problem to be processed or solved using the WebArgo platform. The platform is designed to handle multiple jobs while each job is a unique object. The State of a job can have one of the following values:

- PENDING
- ACTIVE
- RUNNING
- STOPPED
- DONE

However, at the current state of the WebArgo implementation only one job at a time can have the *ACTIVE* or *RUNNING* state.

Figure 5.4 illustrates the Unified Modeling Language (UML) class diagram for the job entity on the left side, lisitng all important attributes and methods of a job. The language attribute defines the programming langugae of the source code that has been compiled to a WebAssembly binary file.

Job entities are stored on the database and can be loaded from the backend. Additionally, the backend provides a compromised Data Transfer Object (DTO) of each job object for workers or administrator users.

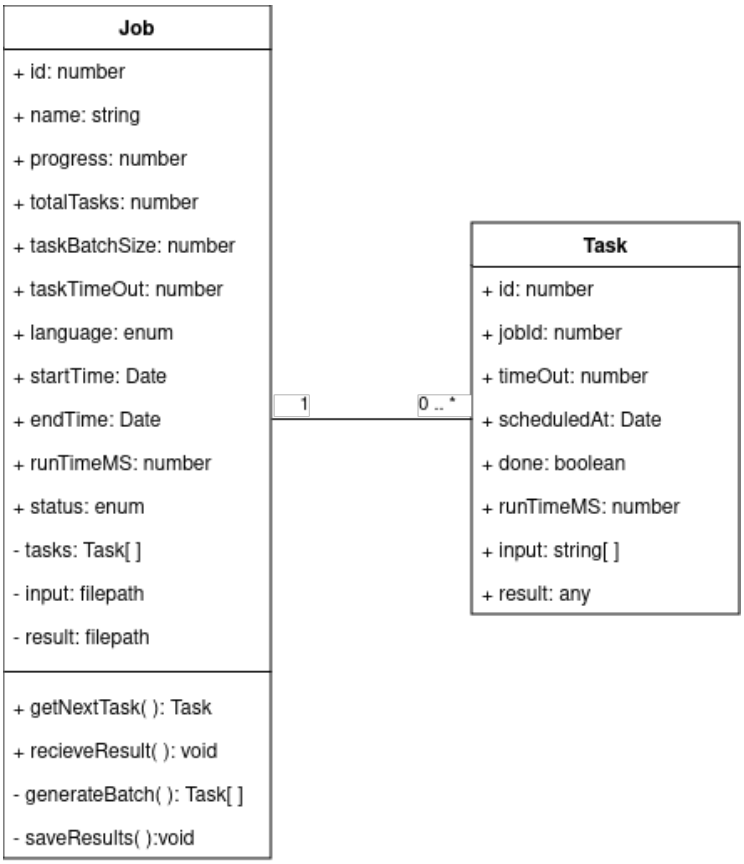


Figure 5.4: UML Class Diagram: Job & Task

5.2.2.2 *Task*

Each job is divided into multiple tasks and ideally has each task an equal computational workload. These tasks are distributed to the participating worker nodes. Tasks are unique objects, each holding the specific input parameters that describe the particular portion of the job to which the task is assigned. Figure 5.4 displays the UML class diagram for the task entity on the right side. The relationship between the job and task entities is defined as One-to-Many, meaning a job can consist of multiple tasks, but each task is always assigned to a single job.

A batch is defined to be a subset of tasks, which are all assigned to the same job. When the state of a job becomes *ACTIVE* the backend enriches the *tasks* attribute of this job object to match the current batch of this job.

Tasks are not stored in the database, but dynamically generated in the backend under utilization of the corresponding task input argument file

stored on the web server. These tasks are then distributed to workers for computation.

5.2.2.3 *User*

A user represents a client that accesses the WebArgo platform through the frontend application. The actions a user can perform are determined by its assigned user role. This user role can either have the value *User* for limited access or *Admin* for full access to all features. Figure 5.5 displays the UML class diagram for the user entity at the top of the illustration. This user information is stored in the database and used for the security measures, later described in Section 5.4.

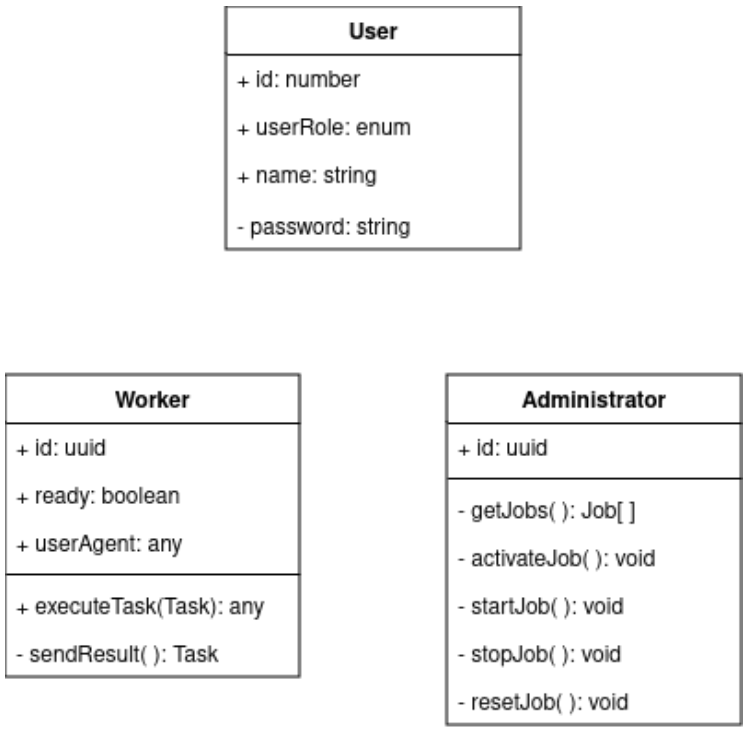


Figure 5.5: UML Class Diagram: User | Worker & Administrator

Workers receive tasks from the current batch of the active job, compute these tasks, and send the corresponding results to the backend. Figure 5.5 presents the UML class diagram of the worker entity on the left side, listing all relevant attributes and methods. As workers connect to the frontend via web browsers, the browser's user agent is utilized to individually characterize each worker. This so-called browser user agent can be accessed inside the clients browser environment and provides information about the client's hardware resources and operating system. A worker object is dynamically generated in the frontend and backend application when a worker client accesses WebArgo.

An administrator user can perform the same actions as a worker, but also has additional capabilities. Only administrators have the ability to set or change the state of jobs and to oversee the progress of all jobs. Figure 5.5 illustrates the UML class diagram of the administrator entity on the right side. When an administrator accesses the WebArgo platform an corresponding administrator object is dynamically generated in the frontend and backend application.

5.3 SCHEDULING

The backend is responsible for distributing tasks from the current batch to participating workers. To maximize performance, the backend keeps track of scheduled tasks to prevent duplicate task distribution among workers. The scheduling of tasks follows the FIFO methodology, scheduling the tasks in sequential order. As described in Section 5.1, each job object has a timeout attribute for its tasks. Only if a task remains incomplete after its designated timeout period has expired, can this task be rescheduled. This mechanism serves to mitigate issues that arise from malfunctioning, disconnected, or straggling workers.

The scheduling mechanism could be enhanced through the implementation of performance-aware distribution, allocating computationally intensive tasks to workers identified to possess superior hardware. Furthermore, these identified stronger workers could execute multiple WebWorker with different tasks, performing parallel task execution on a single worker instance.

5.4 SECURITY THROUGH AUTHENTICATION & AUTHORIZATION

To prevent malicious use, particularly in critical processes managed by administrators, the system implements a authentication mechanism. This is achieved through user credentials combined with a JSON Web Token (JWT).

Authentication and Authorization are two key security concepts that work together to protect systems and data. Authentication is used to verify a clients identity - proving they are who they claim to be - in this platform realized through user credentials. Once a user is authenticated, Authorization determines what they're allowed to do within the system by checking their permissions and access rights. The *userRole* attribute in the user object defines the specific access permissions. Together, these processes ensure that users are verified and these authenticated users can only access the resources and perform the actions appropriate for their role or permission level.

A JWT implements a compact, URL-safe methodology for secure information transmission between parties as a JavaScript Object Notation (JSON) object [JBS15]. The token architecture comprises three dot-separated components: a header that describes the token type and signing algorithm,

a payload containing the encoded data and a signature to verify the token's authenticity [JBS15]. JWTs are commonly used in authentication and authorization processes, where information like user identity and permissions needs to be securely shared between services. The token can be verified by other services using the signature, eliminating the need to repeatedly validate credentials against a database. This makes JWTs particularly useful in modern web applications and microservices architectures where secure, stateless authentication is required. [JBS15]

Upon successful user authentication, when a client connects to the platform, the backend generates and sings a JWT, which is used to authenticate subsequent requests of this user. This generated JWT is stored as a browser cookie on the client side which remains valid during a two-day expiration period. The data stored in the JWT payload is a JSON object containing the attributes *id*, *name*, and *userRole* corresponding to the authenticated user. Clients transmit their specific JWT with every API request or when establishing a WebSocket connection. Hence, the backend can then use this token to authenticate the user and check if the user is authorized for this action.

5.5 BACKEND

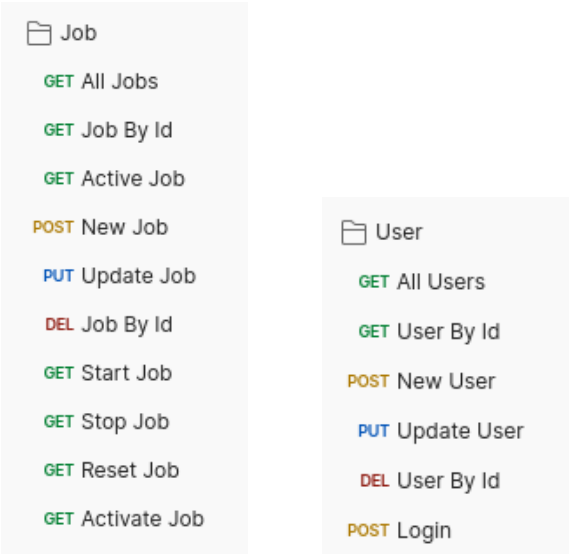
This section describes the usage and features of the backend, which is implemented using the NestJS [Mys24] framework. The backend application serves three primary functions:

- Data management for users, jobs, tasks, and their associated results.
- Handling the WebSockets communication with all connected clients.
- Task distribution to participating workers.

To manage and interact with jobs or users, the backend exposes multiple endpoints through a RESTful API. Figure 5.6 presents a list of all available endpoints and their corresponding HTTP request method. Due to their handling of critical and sensitive data, all endpoints are secured through JWT authorization and always require administrator privileges, granted only for users with the *Admin userRole*. The login endpoint represents the only exception to this security measure, as it serves to generate a JWT in the first place by validating a users credentials, and therefore operates without any JWT authorization.

The backend API implements comprehensive Create, Read, Update und Delete (CRUD) operations for both job and user entities. These endpoints enable the retrieval of individual objects or complete object collections, the creation of new objects, and the modification or removal of existing objects. Furthermore, the API exposes four additional endpoints to change the state of jobs and therefore providing the features to control job execution of WebArgo. Upon job activation, a batch associated to this job is generated and all connected workers are triggered to initiate the initialization of

their corresponding WebAssembly environment. Starting a job initiates the process of distributing tasks from its batch to *ready* participating workers. When a job is stopped, its current state is persisted on the web server as described in Section 5.2. The job reset operation permanently removes all progress associated with a job and generates a new unprocessed batch containing the initial tasks.



(a) Job API endpoints (b) User API endpoints

Figure 5.6: Available Backend API endpoints

Additionally, the backend API serves static files of the compiled WebAssembly binaries and the optional required JavaScript glue code files for each job. These files are required to initializes a WebAssembly environment, and are therefore requested by worker nodes.

Furthermore, the backend implements the WebSocket-based communication protocol as described in Section 5.1 that manages bidirectional data streams between the backend and connected clients. This implementation enables performant task distribution and task result collection as well as real-time comunication for job progress monitoring while minimizing the overall communication overhead.

5.6 FRONTEND

This section describes the features and main web pages served by the frontend. It is implemented utilizing the NextJS [Inc24b] framework in combination with styled components from the React Bootstrap library [Rea24]. The core features implemented in the frontend application are:

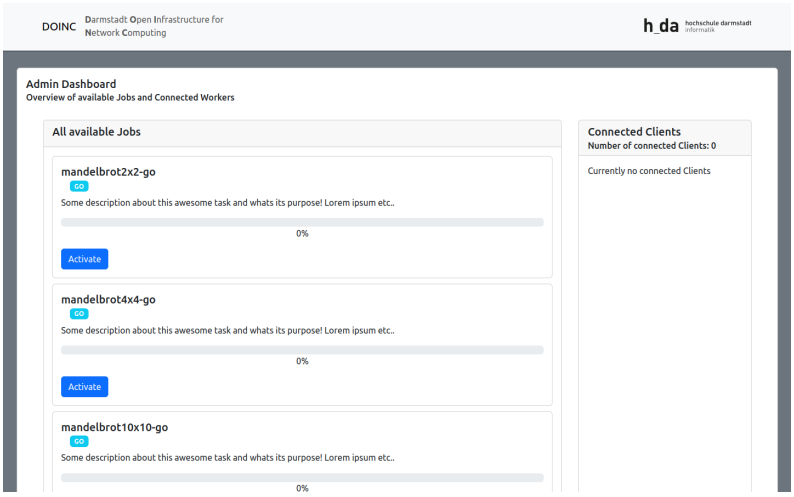
- Providing an easy to use interface to interact with WebArgo
- Establishing a WebSocket connection between clients and the backend
- Computing tasks with WebAssembly in a WebWorker

- Serving a dashboard for administrators to manage jobs

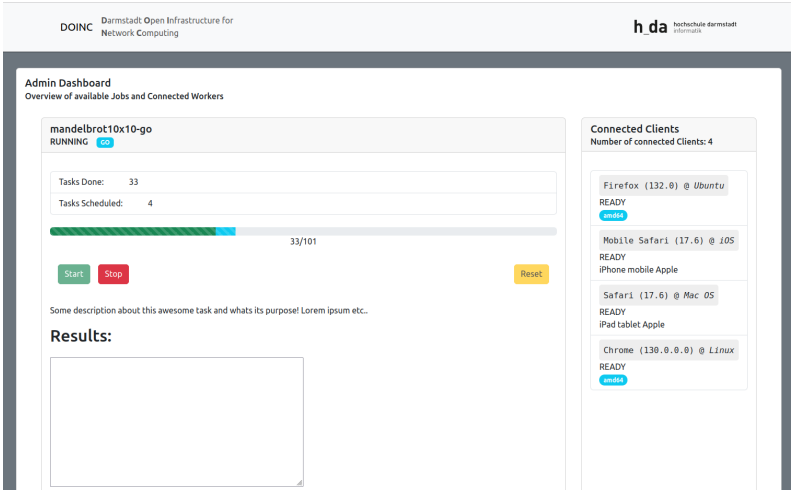
The frontend implements two main web pages. These are the *Dashboard Page* and the *Client Page*.

5.6.1 *Dashboard Page*

The *Dashboard Page*, displayed in Figure 5.7, is only accessible for administrators. This page is used to monitor all connected workers and the progress of jobs in real-time. Furthermore, are administrators able to manage jobs through this web page.



(a) Dashboard Page: Job List



(b) Dashboard Page: Active Job

Figure 5.7: Frontend Dashboard Page

On the left side of the *Dashboard Page* is a component located to manage the jobs of the platform. It holds a list of all existing jobs and their current progress, displayed in Figure 5.7a. Above this list of jobs appears a new component when a job is activated by an administrator.

Figure 5.7b displays a view of this active job component on the left side. This component visualizes the progress of an active job and displays the gathered results of complete tasks after each batch completion in real-time. Additionally it provides an interface to interact with this job. When a button of this interface is clicked, a corresponding HTTP request, containing the administrator's JWT, is transmitted to the backend. After all tasks are completed and the state of the job is *DONE*, this component presents information about the total run time of the job.

On the right side is another component located that displays a list of all connected workers, also in real-time. This list contains information about the initialization progress as well as the hardware and operating system for each worker with an active WebSocket connection to the backend. A view of this list is displayed in Figure 5.7b on the right side.

5.6.2 *Client Page*

The *Client Page* represents the logic for workers and can be accessed by workers or administrators. When a client is accessing this web page the browser process becomes a participating worker in the WebArgo volunteer computing platform. First the worker establishes a WebSocket connection to the backend to receive job and task data and to send task results as described in Section 5.1. If this connection is successful the browser user agent of the workers device is extracted and transmitted to the backend. When a job is active or running the connected worker is creating an independent browser thread - the WebWorker - and initializes the corresponding WebAssembly environment inside this WebWorker. Each task that the worker is receiving is forwarded for computation to this WebWorker. The performance and availability of the main browser thread, responsible for the UI and WebSocket connection, is not affected by the processing of incoming tasks, since the intensive WebAssembly computation is handled inside the separate and independent WebWorker.

Figure 5.8 displays the view of the *Client Page*. The main component of this page presents various information about the worker, like the status of its WebSocket connection, the current actively supported job and statistics about all tasks that have been completed by this worker. Therefore workers can always monitor the current status of their device, task computation and what kind of job they are supporting. Additionally, the usage of a WebWorker for the WebAssembly execution prevents the user page from becoming unresponsive or frozen during task computation. This enhances the user experience and also is supposed to prevent workers from potentially disconnecting because they could experience a unexpected frozen UI.

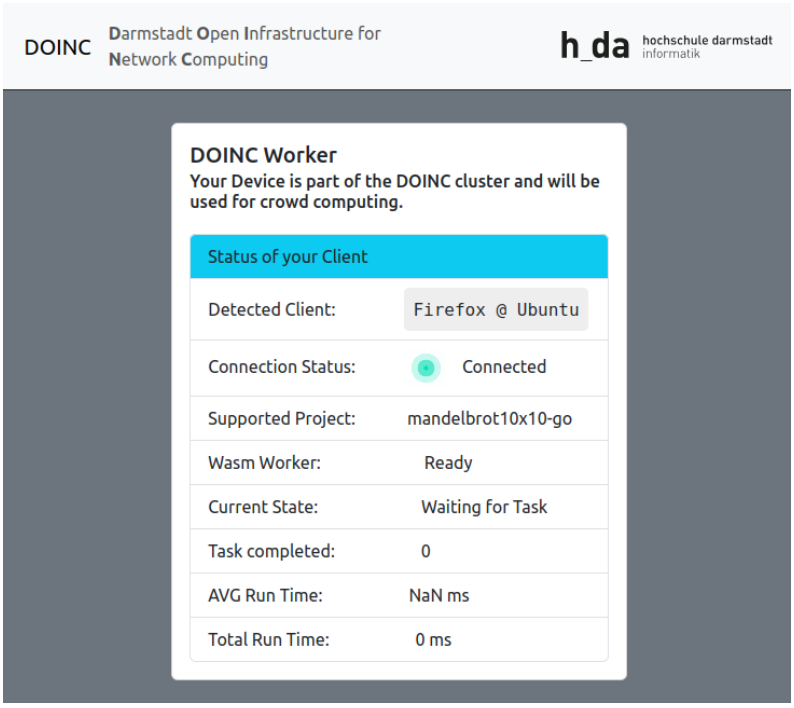


Figure 5.8: Frontend Client Page

In the current state of WebArgo the worker is implemented to initialize and execute WebAssembly binaries complied from either C & C++, Go or Python source code. The support of further WebAssembly targeting languages can be added effortlessly to this work, since the usage of WebWorker scripts is handled generic. To support specific programming languages each WebWorker script requires additional unique glue code during the WebAssembly initialization process.

5.7 BENCHMARK

As described in Section 4.3, the visualization of the Mandelbrot set is used to benchmark the performance of the volunteer computing platform in Chapter 6. To achieve this, this job is distributed among multiple workers through the WebArgo platform and the resulting total execution time of this approach is then compared to the total execution time of the same job on a single device in a native environment.

Since the Mandelbrot set represents a subset of complex numbers, it is visualized in a two-dimensional coordinate system representing the two-dimensional complex plane. The primary region of interest is located in an area bounded by 1 to -2 on the X-axis (real numbers) and 1.5 to -1.5 on the Y-axis (imaginary numbers). This area is partitioned into 100 equally sized sections, forming a 10x10 grid. Figure 5.9 visualizes the Mandelbrot set's primary region of interest divided into 100 equal areas.

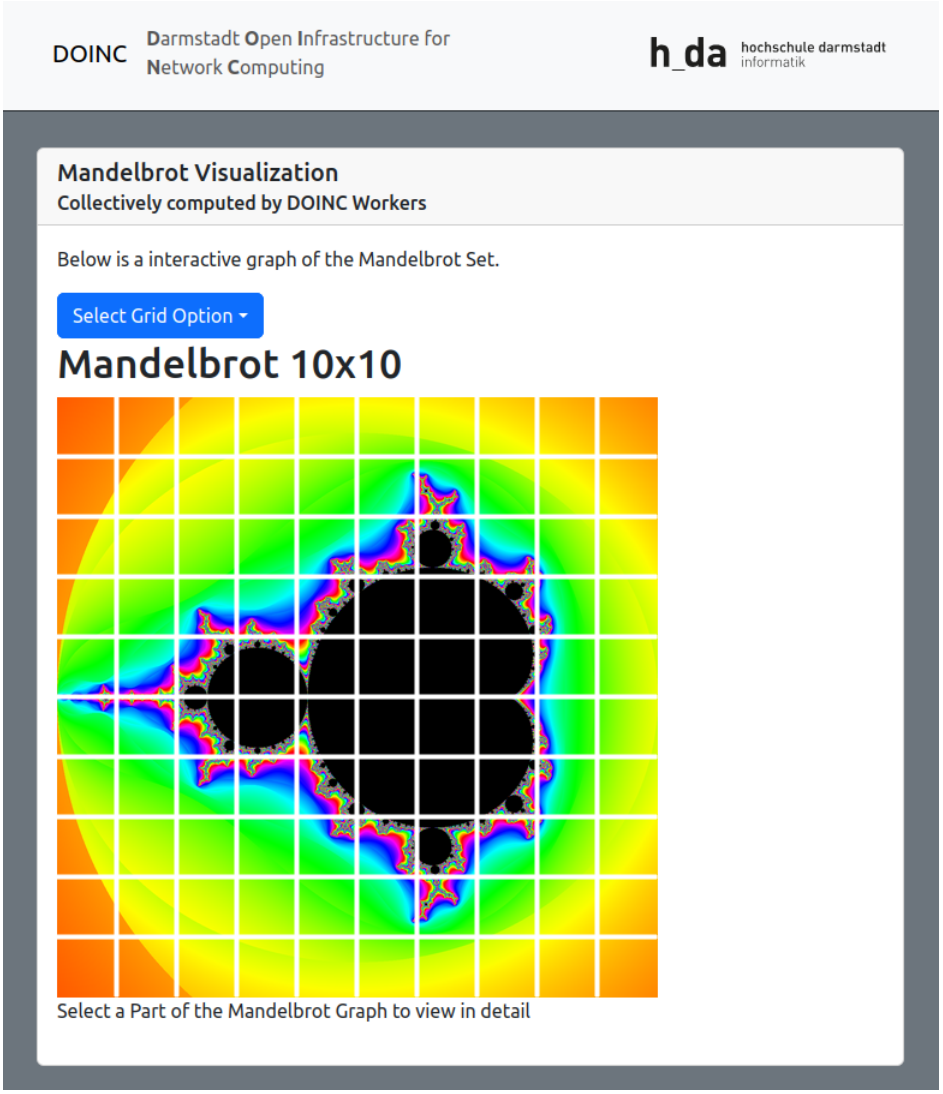


Figure 5.9: Frontend Mandelbrot Page

Each grid tile represents the input arguments of a distinct task. An additional task computes the entire area of the Mandelbrot set at a lower resolution as a thumbnail, resulting in a total of 101 unique tasks for the benchmark job. Each of these tasks is generating a PNG file that visualizes the corresponding Mandelbrot set tile, utilizing the same color scheme as illustrated in Figure 4.7. Each task computes the Mandelbrot condition for 2.25 million complex numbers c within a 1500x1500 square region. Each of these 2.25 million c represents a pixel in the PNG file generated by the task. On task completion the generated PNG file is transmitted to the backend through the WebSocket connection and stored on the web server in a corresponding directory. After the job is successfully completed an interactive visualization of the Mandelbrot set becomes accessible via the *Mandelbrot page*, served by the frontend application. This specific page is displayed in Figure 5.9. Each tile and the thumbnail picture is computed and generated by a worker,

representing the workload of a single task. By clicking on one of these tiles on the thumbnail, the corresponding PNG picture can be viewed in its original resolution underneath the interactive Mandelbrot visualization component.

The WebAssembly-compatible Go source code, which is executed by workers during Task computation, is presented in Section A.2.2. The corresponding native Go implementation, executed in comparision on a single device, can be found in Section A.2.3.

5.8 CHALLENGES

This section lists and describes the challenges that occurred during the development process of the volunteer computing platform.

Generic implementation of Jobs:
The platform is designed to allow the execution of any kind of custom job. To ensure a flexible development of new jobs, the job entity and the handling of tasks, WebAssembly binaries, and input arguments are implemented using generic patterns throughout the architecture in all components. This enables the distribution of jobs across the platform, written in any supported programming language and with any amount of tasks, effortlessly with a minimal programming overhead.

Loading glue code and WebAssembly files from external source (Backend) inside WebWorker:
To maintain simplicity, all WebAssembly binaries and their associated glue code files are centrally stored in their corresponding job directory on the web server, hence ensuring that all files specific to a job remain in a single location. However, the initialization of the WebAssembly environment in the *Client Page* of the frontend requires fetching and incorporating files from the backend as a third-party source in this design. This behavior presented a challenge during the implementation, as the glue code scripts expect its corresponding WebAssembly binary to be located in the same local directory instead of being externaly loaded. It was not trivial to reproduce this behavior inside the WebWorker.

Handling the Input and Output of WebAssembly code:
A string array format, similar to conventional command-line input arguments, has been selected as the uniform input type across all tasks. This standardized format enables simple parsing of arguments from the input text files and can also be effortlessly forwarded to WebAssembly functions within the JavaScript runtime environment.

The implementation of a generic output format across all tasks and programming languages presented significant challenges. Especially for C and C++ source code, where the *main* function is constrained to return always a single numeric value (*int*). However, the developed solution

supports the output of any primitive datatypes, lists of primitive datatypes, objects and also binaries, even for C and C++ applications. This behaviour applies to all kinds of tasks throughout all supported programming languages.

To enable binaries as task output the WebSocket connection between workers and the backend had to be adjusted. The default message size limit (*maxHttpBufferSize*) of the Socket.IO [Con24b] library has been raised to $100MB$ per message to ensure the transmission of larger files.

Implement system recovery measures:
The design and implementation of system recovery mechanisms, as described in Section 5.2, has represented a complex enhancement to the platform.

Prevent duplicate Task execution & ensure Job completion in case of malfunctioning Workers:
The timeout mechanism described in Section 5.3, comparable but not identical to the timeout implementation of XtremWeb [Fed+01], was a crucial enhancement that had to be integrated into the scheduling process. This timeout mechanism is used to prevent duplicate task execution and ensures rescheduling of aborted tasks.

6 EVALUATION

This chapter presents the evaluation of the implemented WebArgo platform through a series of experiments designed to address the research questions defined in Section 1.2.1. These three research questions focus on WebArgo's computational capabilities when handling complex parallelizable tasks, its dynamic viability in managing fluctuating worker participation, and its ability to support heterogeneous devices as workers. Each of the following sections in this chapter corresponds to one of the research questions and presents the experimental setup, the expected outcome, and a analysis of these results. Additionally, Section 6.4 compares the performance of the various in WebArgo implemented WebAssembly environments, each supporting the execution of source code from different programming languages compiled to a WebAssembly binary.

The experiments of the evaluation utilize the implemented visualization of the Mandelbrot set, described in Section 5.7, as a benchmark job. This benchmark job represents a computationally intensive test case, which can be used to demonstrate and test WebArgo's capabilities. Each task of this benchmark job covers a unique 1500x1500 area of the Mandelbrot set and all generated PNG files have the same resolution. However, the computation time varies significantly among each of these tasks. According to the Mandelbrot function in (4.1), complex numberers that are part of the Mandelbrot set require more iterations of the calculation than complex numbers that are not part of the Mandelbrot set. Therefore, the execution time of a task depends on the amount of calculated points that belong to the Mandelbrot set.

Furthermore, the networking overhead t_O can be estimated for this benchmark job, assuming an internet latency L of 32 ms [Inc24a] and a network bandwidth B of 29 Mbps (3.625 MBps) [Inc24a]. The average result file size F_R of a PNG file generated by a task of the Mandelbrot benchmark job is measured to be 290 KB. The task file size F_T is estimated to be comparatively very smal and therefore estimate to be 0. Using this values to calculate the networking overhead with function (3.3) results in a estimated networking overhead $t_O = 0.144$ seconds for the benchamrk job.

6.1 COMPUTATIONAL CAPABILITY

Is WebArgo capable of successfully solving large, parallelizable problems?

The objective of the following experiment is to evaluate WebArgo's ability to successfully execute computationally intensive, parallelizable tasks across a distributed network of volunteer workers. This empirical experiment

compares the total execution time between distributed computation across multiple workers and native execution on a single computer.

6.1.1 *Experimental Setup*

The batch size of the benchmark job was set to 101. Therefore, only a single batch is generated and processed during all experiments and the job progress is only persisted once upon completion of all 101 tasks.

At first the benchmark job was computed on the system specified in Section A.1.3 in a native Go environment using the source code of Section A.2.3. The resulting computation time serves as a baseline for the following experiments, because this system is later also used to host the WebArgo platform. Since this system is already required to serve WebArgo in the first place, the following experiments additionally investigate whether distributing the workload to external clients provides a performance advantage compared to utilizing the existing host system for computation. Throughout the experiments, all workers maintained available and executed only the WebArgo browser process. The benchmark job was evaluated across the following four distinct scenarios:

- Two homogeneous and independent smartphones with the hardware specified in Section A.1.5 as workers, executing the client page in a Apple Safari 18.1 [App24a] browser
- Three homogeneous and independent smartphones with the hardware specified in Section A.1.5 as workers, executing the client page in a Apple Safari 18.1 [App24a] browser
- Three parallel Mozilla Firefox 132.0 [Moz24a] browser tabs of the client page on a single laptop, specified in Section A.1.1
- Three parallel Microsoft Edge 131.0.0.0 [Mic24] browser tabs of the client page on a single desktop PC, specified in Section A.1.2

6.1.2 *Expectations*

It is expected that all tasks of the benchmark job will be scheduled as intended and distributed over all participating workers and each task result is successfully received by the server and saved on the server. To verify if this process was successful, each worker is monitored through the interface of the client page during the experiment and after the experiment is the implemented Mandelbrot page (Section 5.7) utilized to examine the generated task results.

Additionally, all experiments are carried out using the interactive dashboard page (Section 5.6.1) to start and monitor the execution of the benchmark job. It is expected that all features of this application work as intended and the job progress as well as all participating workers can be monitored in real-time.

Furthermore, based on the theoretical model presented in Section 3.3, distributing the benchmark job across multiple workers should reduce the total execution time of a job when the number of workers W exceeds the threshold value represented in the inequality term of (3.9). To estimate this threshold value of W for all previously listed experiment scenarios the amount of tasks T was set to 101. Since the computation times t_{Native} and $t_{Virtual}$ are not equal for all 101 tasks, a computationally intensive task - handling the center of the Mandelbrot set - was selected to represent these computation times. This single task has been executed and measured independently on the native Go environment on the server system as well as on the WebAssembly browser environment through the WebArgo platform for each device participating in the experiment as worker. The computation time t_{Native} of this computationally intensive task was measured to be 34.60 seconds on the server system specified in Section A.1.3. The corresponding computation time $t_{Virtual}$ of the same task was measured to be 52.56 seconds using the Apple Safari 18.1 [App24a] browser on a smartphone specified in Section A.1.5, 1 minute and 34.8 seconds using the Mozilla Firefox 132.0 [Moz24a] browser on the laptop specified in Section A.1.1 and 1 minute and 28.2 seconds using the Microsoft Edge 131.0.0.0 [Mic24] browser on the desktop PC specified in Section A.1.2 With this information and the previously calculated networking overhead t_O of 0.144 seconds the amount of workers W - expected to provide a performance advantage compared to the native code execution - was calculated with the inequality term of (3.9) for each scenario:

- **Smartphones:** $W > 1.5$, meaning two or more smartphones are expected to achieve a performance improvement
- **Laptops:** $W > 2.8$, meaning three or more laptops are expected to achieve a performance improvement
- **Desktop PCs:** $W > 2.6$, meaning three or more desktop PCs are expected to achieve a performance improvement

6.1.3 *Results*

The benchmark job was successfully computed in every experiment and all features of the web application behaved as intended. Figure 6.1 compares the measured execution times of the different experiments to the baseline time of the server.

The native execution on the server system completed all 101 tasks in an average execution time t_{Seq} of 10 minutes and 45 seconds across three runs. Distributing the benchmark job across the two smartphone workers resulted in a total execution time t_{Dist} of 10 minutes and 12 seconds, and therefore faster than the baseline as predicted. With three smartphone workers, the execution time further decreased to 6 minutes and 39 seconds, representing a performance improvement of 38% compared to the native execution on the server.

Distributing the benchmark job across three browser tabs on the laptop took 12 minutes and 56.4 seconds to complete the total workload. This approach did not achieve a performance improvement compared to the native code execution on the server. However, distributing the benchmark job across three browser tabs on the desktop PC complete the total workload in only 9 minutes and 52.8 seconds and therefore faster than the baseline of the server. These two experiments show that a single device can be successfully utilized to participate in WebArgo with running multiple worker processes in parallel, and therefore effectively allows to expand the job progress computed by a single device. But it can not be expected that a single browser tab in this scenario will behave in the same way as an independent device. The behavior of multiple worker tabs on a single device most likely depends on the devices hardware, the amount of available CPU cores, and the browser and operating system used.

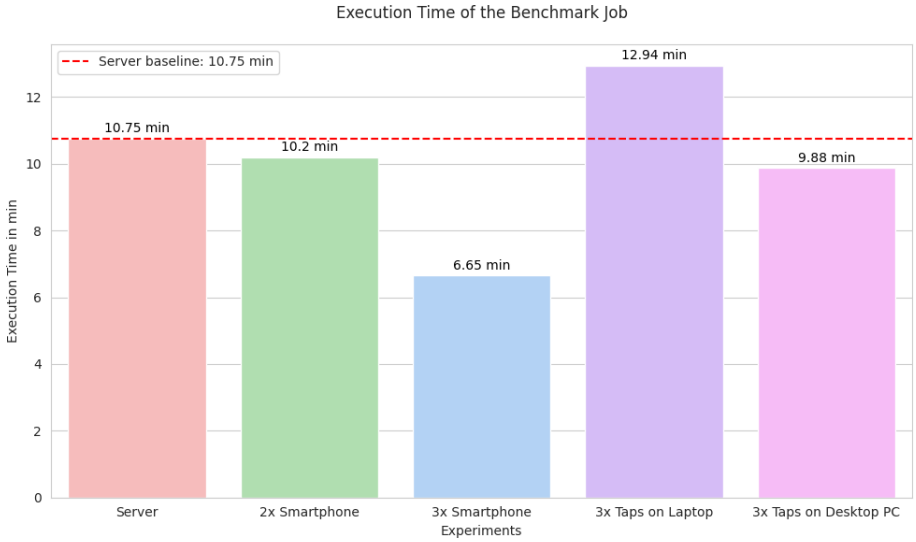


Figure 6.1: WebArgo Execution Time Compared to Native Execution Time

These results validate the predictions of the theoretical model presented in Section 3.3. Using the calculation of the inequality term in (3.9) allowed to predict that two or more smartphone workers would achieve a faster execution time compared to the native baseline, which was confirmed by the experiment. This improvement further scaled with three smartphone workers, reducing the execution time by a total of 38% compared to the server baseline.

The experiments with parallel worker browser tabs on single devices revealed, that a single worker is able to enhance its computational participation by executing multiple worker instances simultaneously. However, the measured performance difference between the laptop and desktop PC is suggesting, that the execution of multiple worker instances is limited by the devices hardware.

Furthermore, these experimental results support the theoretical assumption from Section 3.3 that the performance gain through distributed

computation on multiple workers is able to outweigh the additional overhead of communicating through the internet and the longer computation time $t_{Virtual}$ in the WebAssembly environment. This validates that even with these performance constraints, WebArgo can effectively leverage parallel processing through volunteer computing to reduce the total computation time of a job. The achieved performance improvement scales with the number of participating workers W.

6.2 DYNAMIC VIABILITY

Is the WebArgo platform stable in a environment with dynamic clients? The second research question investigates whether WebArgo maintains stability in an environment with dynamic clients, addressing a fundamental challenge in volunteer computing where worker participation is inherently unpredictable and therefore dynamic. This evaluation is crucial, as a key feature of WebArgo is to maintain operational despite workers joining or leaving at any time. Hence, the following experiments in this section evaluate WebArgo's ability to handle fluctuating worker participation.

6.2.1 *Experimental Setup*

To evaluate WebArgo's dynamic viability, the platform was tested in a controlled environment with manually connecting or disconnecting workers. The experiments utilized 32 single-board computers specified in Section A.1.4, each running a headless Mozilla Firefox 133.0 [Moz24b] browser to participate in WebArgo as a worker through the client web application. Again, the Mandelbrot benchmark job was utilized for these experiments. All single-board computers used in this experiment are part of the *Pi-lab* from h_da, providing a practical infrastructure to interact with multiple Raspberry Pi computers. The task timeout to enable rescheduling of aborted tasks was set to be 60 seconds throughout the benchmark job. The following two experiments were conducted to simulate a dynamic environment where workers join or leave the network during the execution of an active job:

1. Disconnecting of 8 workers (25% of all conneted workers) after about 50% of total job completion
2. Starting with 8 connected workers and gradually connecting more devices until 32 workers are connected

Additionally, an experiment with all 32 single-board computers computing the benchmark job while maintaining a stable connection to the platform was performed. The total execution time of this experiment is used as a baseline time to compare the results of the two experiments mentioned above.

6.2.2 *Expectations*

It is expected that all 101 tasks of the benchmark job are successfully completed, regardless of the fluctuating behaviour of workers. Hence, aborted task from disconnected workers are expected to be rescheduled to other available workers, and workers which are connecting while the active job is already running are expected to automatically setup the corresponding WebAssembly environment and then immediately participate as workers.

However, it is expected that the total execution time of both experiments will be longer than the baseline time, since in these scenarios overall less worker instances are consistently participating throughout the computation of the job.

6.2.3 *Results*

The benchmark job was successfully computed in both experiments and all connecting workers have actively participated in computing the workload of the job. Therefore, these experimental results demonstrate WebArgo's robust handling of dynamic worker participation. Figure 6.2 displays the total computation time of both experiments compared to the baseline computation time of 32 permanently connected single-board computers as workers.

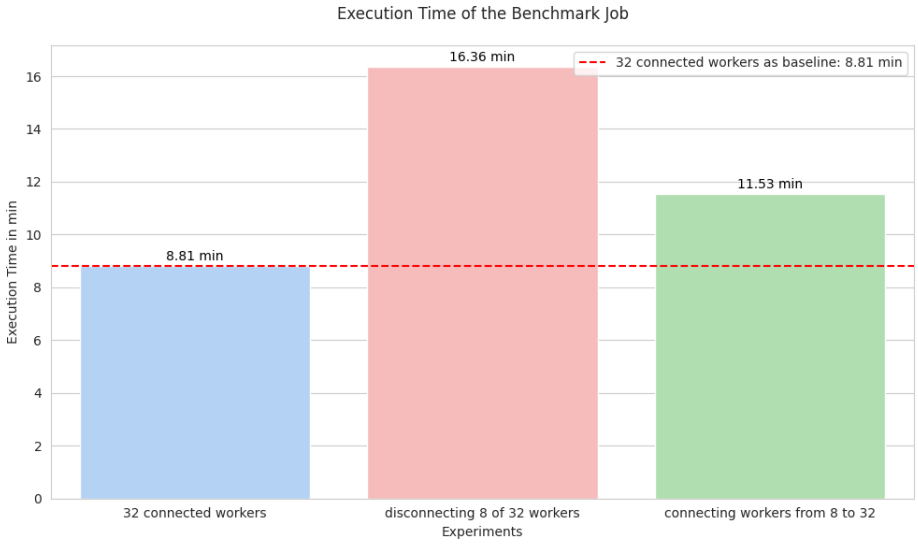


Figure 6.2: WebArgo Execution Time in Dynamic Environment

In experiment 1, the WebArgo platform successfully demonstrated reliable task redistribution when workers are disconnecting during the computation of a task. The implemented timeout mechanism effectively rescheduled all 8 aborted tasks distributed to the 8 manually disconnected workers. The system maintained consistent progress despite losing up 25% of the initial

worker pool, though with an expected increases of the total computation time to 16 minutes and 21.6 seconds.

During experiment 2, the WebArgo platform successfully integrated new workers as they joined, with each additional worker immediately receiving and computing tasks from the current batch after they successful initialized the corresponding WebAssembly environment. Since this experiment started with only 25% of the amount of workers compared to the baseline experiment, it was expected that the total computation time is slower than the baseline. Corresponding, the total execution time of this experiment was 11 minutes and 31.8 seconds.

Both experiments demonstrated that the WebArgo platform can maintain operation continuity despite significant worker pool fluctuations. This validates WebArgo's implementation for dynamic worker participation and confirms, that it is suitable for real-world volunteer computing scenarios where the availability of a worker device is not guaranteed.

6.3 HETEROGENEOUS VIABILITY

Does WebArgo support a diverse range of client devices without issues? The third research question examines WebArgo's capability to effectively support diverse client devices as workers, therefore also addressing a critical requirement for volunteer computing platforms. As the pool of potential worker devices in a real-world environment is expected to be diverse in hardware, software and operating systems [AR09], the following experiment is used to evaluate whether WebArgo can successfully operate with a heterogeneous group of participating workers.

6.3.1 *Experimental Setup*

To evaluate the platform's support for heterogeneous devices, a diverse set of everyday consumer devices was assembled to participate simultaneously in the benchmark job through the WebArgo platform. The following four devices represent different hardware architectures (ARM, x86), various operating systems (iOS, Windows, and Ubuntu), and also different browser environments (Apple Safari 18.1 [App24a], Mozilla Firefox 132.0 [Moz24a], and Microsoft Edge 131.0.0.0 [Mic24]):

- One smartphone as specified in Section A.1.5
- One laptop as specified in Section A.1.1, executing 3 worker tabs in parallel
- One desktop PC as specified in Section A.1.2, executing 3 worker tabs in parallel
- One tablet as specified in Section A.1.6

All of these devices were found in one household to demonstrate the easy accessibility to a potential performance improvement of a computational intensive job by leveraging the WebArgo platform.

6.3.2 *Expectations*

It is expected that the benchmark job is successfully computed and that all participating workers are able to compute their scheduled tasks, regardless of the heterogeneous pool of connected devices. Furthermore, this set of devices - representing a single household - is expected to achieve a significant performance improvement compared to the baseline execution time of the natively computed Mandelbrot visualization on the server.

6.3.3 *Results*

The Mandelbrot visualization job was again successfully computed and all participating workers were able to compute the tasks that have been distributed to them. Therefore, this experiment demonstrated WebArgo's support for heterogeneous devices as workers. All devices successfully connected to the platform, initialized their WebAssembly environments, and computed their assigned tasks without any occuring issues. The WebSocket connections remained stable and the WebWorker-WebAssembly environment was effortlessly established across all corresponding browsers, each being preinstalled on the devices by default. Figure 6.3 displays the total execution time of this experiment compared to the baseline execution time of the native server environment measured in Section 6.1.

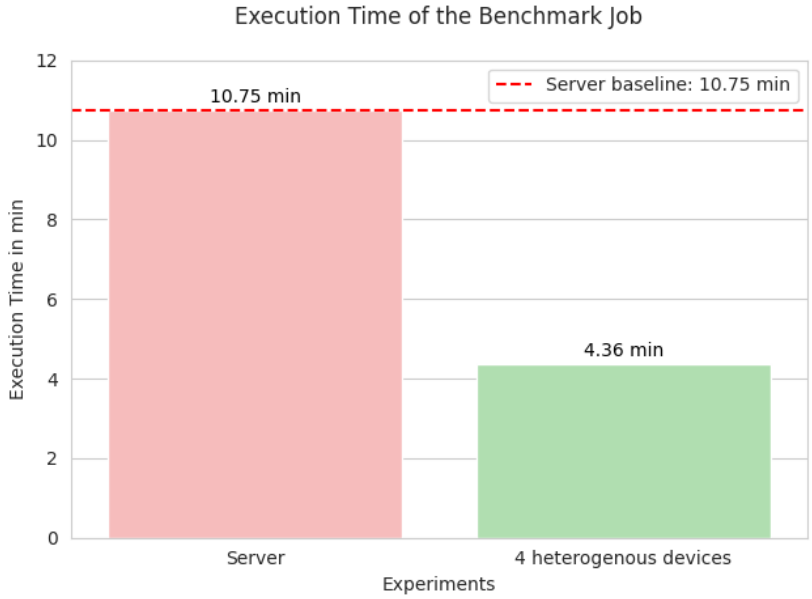


Figure 6.3: WebArgo Execution Time with Heterogeneous Set of Workers

Using the setup of this experiment, the benchmark job was already completed after 4 minutes and 21.6 seconds. This represents a performance improvement of 59% compared to the native execution on the server. The substantial reduction in execution time demonstrates the effectiveness of WebArgo's approach to volunteer computing, even when considering the additional overhead of network communication, task scheduling and computation time $t_{Virtual}$ in a WebAssembly environment. This validates that WebArgo can effectively utilize an environment of heterogeneous devices to achieve significant performance improvements compared to native code execution on a single system.

Furthermore, the implemented web interface of the client page adapted appropriately to all different screen sizes and resolutions, providing a consistent user experience across all devices. Additionally, the WebWorker implementation effectively prevented any freezing effects of the browser UI during the computation of tasks for all participating workers. Therefore, the web application maintained responsive throughout the experiment and provided real-time monitoring of each worker, as intended.

These results validate that WebArgo successfully leverages WebAssembly's platform independence to enable consistent computation across different architectures, while the web-based approach provides a uniform and easy-to-use application, accessible on any device with access to a browser. This confirms that WebArgo achieves its intended design goal.

6.4 COMPARISON OF IMPLEMENTED WEBASSEMBLY ENVIRONMENTS

As WebArgo currently supports three different programming languages as source for jobs, this section compares the performance of these implemented WebAssembly environments. Each of these programming languages utilizes a unique compilation toolchain to generate the executable WebAssembly binary files and unique JavaScript glue code to handle the corresponding WebAssembly binary in a browser environment, as described in Section 4.2.1. A comparison of these implementations can provide valuable insights for potential administrators, which develop new jobs served by a hostet WebArgo platform.

6.4.1 *Prime Numbers*

The three implemented WebAssembly environments were evaluated with three other benchmark jobs, each implemented in either C++, Go or Python. Each of these benchmark jobs finds and lists all prime numbers in the range from 0 to 10,000,000 and is divided in 10 distinct tasks, each processing a unique interval of 1 million numbers. All of these three prime number jobs have been executed through the WebArgo platform by a single worker instance, executed in a Apple Safari 18.1 [App24a] browser on the smartphone specified in Section A.1.5.

Figure 6.4 compares the total execution time of each prime number job executed in this experiment, revealing significant variations in performance across the different environments. The C++ WebAssembly environment, leveraging the toolchain of emscripten [Con24c], demonstrated the best computational performance and successfully completed all 10 tasks in only 5.9 seconds. The Go WebAssembly environment, compiled and initialized with the tools provided by Go [Con24a], achieved similar but slower execution time of 7.2 seconds for the prime number job. In contrast to these results, the Python WebAssembly environment, utilizing the Pyodide library [cM24], took a total of 156.6 seconds to compute the same workload.

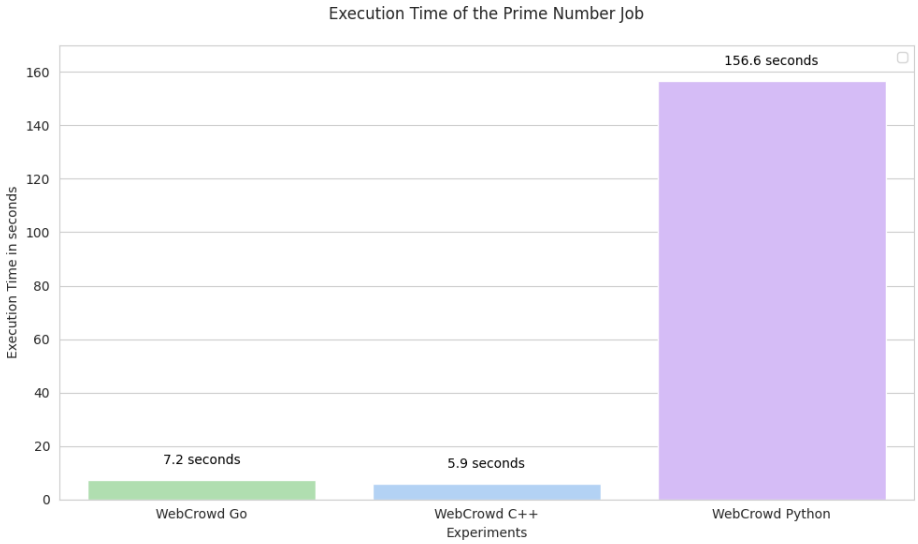


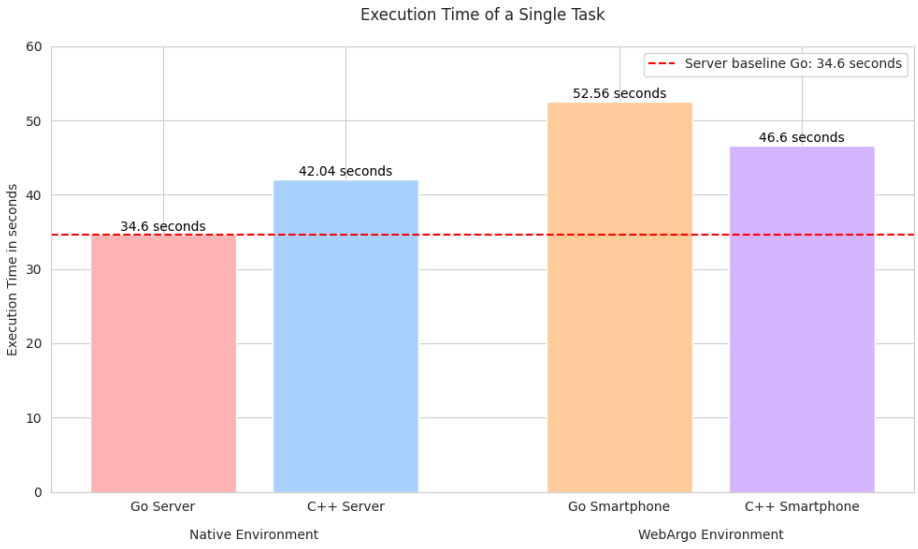
Figure 6.4: WebArgo Execution Time of Different WebAssembly Environments

Despite the fact that all three jobs executed the same workload with a WebAssembly binary on the same system and in the same browser, the execution time of each job varied significantly. The emscripten [Con24c] WebAssembly toolchain demonstrated the best computational performance in this scenario, however it has not been further investigated why this is the case.

6.4.2 *Mandelbrot Set*

Additionally to the previous experiment, the implemented C++ and Go WebAssembly environments are further compared in this section. Both environments have demonstrated a similar performance when computing the prime number job. However, the following experiments again utilize the visualization of the Mandelbrot set described in Section 5.7. The Go source code, compiled with Go [Con24a] to WebAssembly, can be found in Section A.2.2, and the according C++ source code, compiled by the emscripten [Con24c] toolchain, can be found in Section A.2.4.

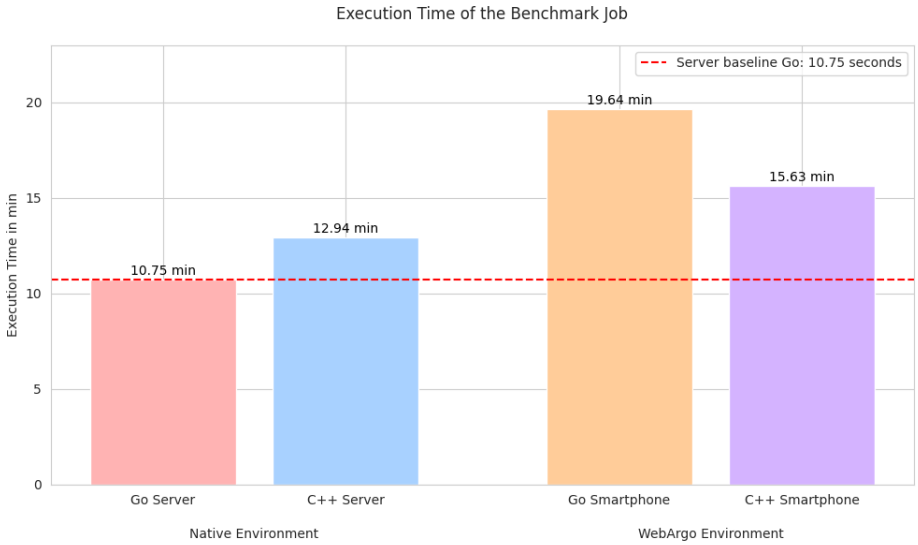
To compare the performance of both environments, first a single computationally intensive task, generating the center of the Mandelbrot set, was computed in each WebArgo environment by a worker instance executed in a Apple Safari 18.1 [App24a] browser on the smartphone specified in Section A.1.5. Then, the same worker instance was used to compute all 101 tasks of the Go Mandelbrot benchmark job and the C++ Mandelbrot benchmark job. Additionally, all of these experiments were repeated with corresponding native environments on the server, specified in Section A.1.3.



(a) Single Computationally Intensive Task

Figure 6.5: Comparison of C++ and Go Execution Time

Figure 6.5 compares the measured total execution time of this experiment compared to the baseline execution time on the native server environment. The C++ WebAssembly environment was able to compute the computationally heavy task in only 46.6 seconds and was therefore again faster than the corresponding Go WebAssembly environment, which took 52.56 seconds to compute the same workload. This represents a performance advantage of 11% for the C++/emscripten approach compared to the Go toolchain, graphically displayed in Figure 6.5a on the right side. This performance advantage increased to 20% when processing the entire benchmark job, as displayed in Figure 6.5b.



(b) Mandelbrot Benchmark Job

Figure 6.5: Comparison of C++ and Go Execution Time

This investigation confirms the performance advantage of the implemented C++ WebAssembly environment over the implemented Go WebAssembly environment.

7 CONCLUSION

This work has introduced WebArgo, a volunteer computing platform that leverages several modern web technologies to create a new, dynamic and, heterogeneous distributed computing solution. Through empirical evaluation, the implemented WebArgo platform has demonstrated its capability to effectively distribute computational workloads across heterogeneous client devices as workers while maintaining robustness in dynamic environments with fluctuating worker participation.

All three research questions, listed in Section 1.2.1, were addressed in Chapter 6 and confirmed by the empirical evidence of the evaluations experiments. Furthermore, the experiments revealed that WebArgo is able to successfully reduce the total computation time of the Mandelbrot benchmark job, described in Section 5.7, by 59% compared to a native code execution, when distributing the corresponding tasks across four consumer devices found in a single household.

Additionally, WebArgo is successfully fulfilling all its objectives outlined in Section 1.2. Hence, WebArgo provides an accessible volunteer computing solution that requires no setup except a browser and an internet connection for participants. WebArgo also limits the overhead for administrators, because the development of new custom jobs does not require a deep understanding of the underlying architecture, and these jobs can be implemented in three different programming languages, however C/C++ in combination with the emscripten [Con24c] toolchain has turned out to be the currently best performing option in the experiments. Additionally, hosting a WebArgo instance has limited requirements and can potentially even be used by a single private individual. A public web server with enough storage to store all incoming task results and capable of running Docker Compose [Doc24] to orchestrate the backend application, frontend application, and database are enough to host a private WebArgo platform.

Furthermore, the web-based approach potentially addressed privacy and security concerns of participants, because installation and execution of third party applications is not required.

In conclusion, WebArgo successfully addresses key challenges in distributed computing while providing an accessible solution for organizations with limited access to traditional HPC computing resources. Also its web-based approach reduces any barriers for participation while additionally maintaining the web security model and WebAssembly sandboxing, potentially enabling broader adoption of volunteer computing.

7.1 LIMITATIONS

While WebArgo demonstrates promising capabilities as a volunteer computing platform, several limitations have been identified during its development and evaluation that warrant acknowledgment. The current implementation of the scheduling algorithm operates on a simple FIFO basis, without consideration of individual worker hardware capabilities. This approach potentially leads to suboptimal task distribution, as computationally intensive tasks may be assigned to less capable devices while more powerful workers remain underutilized. A more sophisticated scheduling mechanism that accounts for worker hardware specifications and historical performance metrics could improve overall system efficiency.

A significant performance limitation is rooted in the performance of WebAssembly. Despite WebAssembly is promised to perform at near native speed Jangda et al. have discovered, that the WebAssembly execution time can potentially be twice as slow as the corresponding native code execution [Jan+19], which is consistent with measurements from experiments of this work. This performance gap impacts WebArgo's computational efficiency. While the parallel processing capabilities of multiple distributed workers can compensate for this limitation, the inherent overhead remains a constraint on individual task performance.

Furthermore, the current implementation is not providing a real multi-threading option for task computation on a single worker device. This limitation prevents full utilization of available computational resources, as workers cannot leverage their potential multi-core capabilities. However, a worker with enough computational capabilities can effectively execute multiple worker instances to process multiple tasks concurrently, each in an independent browser tab, as experiments of this work confirmed.

Currently, WebArgo's operational scope is constrained to support only one active job at a time. This restriction prevents workers to actively choose which project or contributor they support with their participation, and therefore potentially reducing volunteer engagement.

These limitations are significant, but do not fundamentally undermine WebArgo's utility as a volunteer computing platform. In fact, these limitations represent opportunities for future development and enhancement of WebArgo.

7.2 FUTURE WORK

This section lists various drafts for future work options. The previously identified limitations of WebArgo provide a clear roadmap for future development and enhancement of the platform.

Performance-Aware Task Scheduling
A potential focus for future development is the implementation of an

intelligent scheduling algorithm that considers the hardware capabilities of participating workers and assumes the complexity of each task. Therefore, WebArgo could optimize the task distribution across the heterogeneous workers to ensure an optimal utilization of each individual worker.

Multi-Threading Support
Implementing a approach to enable multi-threaded task execution for workers represents a significant opportunity to further improve the performance WebArgo. This can be achieved either at the source code level by leveraging emscripten's Pthreads support [Con24c] for example, or at the browser level through the initialization of more than one WebWorker instances which simultaneous process multiple tasks. This enhancement of WebArgo could be combined with the previously described performance-aware scheduling, hence scheduling multiple tasks for parallel processing to workers with sufficient hardware capabilities.

Multi-Job Support
Expanding WebArgo to support multiple actively running jobs simultaneously can improve the user experience for workers. To achieve this, the server needs to handle multiple batches of active tasks and the implemented communication protocol between server and worker needs to be updated. Furthermore, an additional feature to select preferred jobs can be included in the UI of the client web page.

Extended Programming Language Support
While WebArgo currently supports C++, Go, and Python, future work could focus on expanding WebAssembly compilation support for additional programming languages, since WebAssembly is a compilation target for many more high-level programming languages [Haa+17; Gro19; Gro24b]. This can potentially make the platform more accessible to a broader range of developers and scientific computing applications.

Task Result Validation
Expanding WebArgo to a global volunteer computing platform could significantly increase the pool of participating workers. However, this expansion introduces new challenges regarding result integrity. In a global environment, some participants might intentionally attempt to manipulate jobs by submitting incorrect results. Moreover, even in environments with trusted workers, hardware malfunctions or computational errors could lead to incorrect results. To address these challenges, a robust validation system could be implemented in the architecture of WebArgo, similar to BOINC's replication-based approach. Such a mechanism would distribute each task to multiple independent workers and compare their results to ensure the correctness of a completed task. However, this approach will increase the total computation time of a job, because each task is distributed redundantly and therefore the total amount of tasks that need to be computed increases

significantly. This is only feasible if the pool of volunteer workers is big enough to handle the increased workload.

Furthermore, to implement this approach in WebArgo an additional process is required in the backend, which is able to compare the job-specific task results. For example, the validation function of the Mandelbrot benchmark job, presented in this work, would have to be capable of comparing PNG files. Additionally, a potential validation function should also be able to reliable handle edge cases, as for example the floating-point operations could produce slightly different results across the various browsers and hardware architectures of the participating workers. To achieve this, WebArgo could implement an application-specific validator system that allows job developers to define custom comparison functions with appropriate tolerance levels for their specific use case.

To minimize the performance overhead of replication, an adaptive validation system like it is implemented in BOINC could be developed that tracks worker reliability over time. This system would maintain a reputation score for each worker to identify trustworthy workers. A task completed by a trusted worker would not need any validation and therefore reduce the total redundancy of tasks.

Such a validation system would significantly enhance WebArgo's suitability for scientific computing applications where result accuracy is critical.

Implement the MapReduce model

Implementing the MapReduce [DG08] model in WebArgo is a big opportunity to significantly enhance the platforms capability of providing more complex jobs. Currently the processing module of a job can be assumed to be similar to the mapping step. Each task therefore represents a single map operation. MapReduce could be used to aggregate all gathered task results of a job to a total result of the job. This enables the processing of more complex jobs, for example a single epoch of reinforcement learning, with each task representing an episode and the reduce step performing the corresponding policy updates. This approach can be further investigated to potentially enable reinforcement learning through a pipeline of multiple MapReduce jobs in WebArgo.

Part II

APPENDIX

A

APPENDIX

A.1 SYSTEM SPECIFICATIONS

Experiments and code executions have been made on a systems with the following specifications:

A.1.1 *Local Linux System*

- **CPU:** Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
- **RAM:** 15 Gi (15.4667 GB)
- **Operating System:** Ubuntu 20.04.6 LTS
- **Kernel:** 5.15.0-119-generic
- **GPU:** Intel Corporation UHD Graphics 620 (rev 07)

A.1.2 *Local Windows System*

- **CPU:** Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
- **RAM:** 32 GB
- **Operating System:** Microsoft Windows 10 Home
- **Kernel:** Windows 10.0.19045.5131
- **GPU:** NVIDIA GeForce GTX 1080

A.1.3 *Server Linux System*

- **CPU:** Intel Xeon Processor (Cascadelake)
- **RAM:** 3.8 Gi
- **Operating System:** Ubuntu 24.04.1 LTS
- **Kernel:** 6.8.0-36-generic
- **GPU:** Cirrus Logic GD 5446

A.1.4 *Raspberry Pi Linux System*

- **CPU:** Raspberry Pi 4 Model B Rev 1.4
- **RAM:** 7.7 Gi
- **Operating System:** Ubuntu 22.04.3 LTS
- **Kernel:** 5.15.0-1034-raspi

A.1.5 *Smartphone | iPhone 13*

- **Operating System:** iOS 18.1.1
- **CPU:** A15 Bionic chip
  - 6-core CPU with 2 performance and 4 efficiency cores [App24c]
  - 4-core GPU [App24c]
  - 16-core Neural Engine [App24c]
- **ROM:** 256 GB

A.1.6 *Tablet | iPad Pro 11-inch (3rd generation)*

- **Operating System:** iOS 18.1.1
- **CPU:** Apple M1 chip
  - 8-core CPU with 4 performance and 4 efficiency cores [App24b]
  - 8-core GPU [App24b]
  - 16-core Neural Engine [App24b]
- **RAM:** 8 GB [App24b]
- **ROM:** 128 GB

A.2 SOURCECODE AND OUTPUT

A.2.1 *Calculation of Mandelbrot Set: Go*

Below is the Go source code for generating a colored PNG file of the Mandelbrot set:

Listing A.1: Mandelbrot Set Calculation: Go

```
package main

import (
    "os"
    "fmt"
```

```
    "image"
    "image/color"
    "image/png"
    "math"
    "math/cmplx"
    "time"
)

func main() {
    startTime := time.Now()

    // Set Parameters
    width := 1500
    height := 1500
    maxIterations := 3000
    realMin := -2.0
    realMax := 1.0
    imagMin := -1.5
    imagMax := 1.5

    img := image.NewRGBA(image.Rect(0, 0, width, height))
    fmt.Println("Generating PNG of Mandelbrot set... ")

    // Calculate Mandelbrot for each Pixel
    for py := 0; py < height; py++ {
        y := imagMax - float64(py)/float64(height)*(imagMax-imagMin)
        for px := 0; px < width; px++ {
            x := float64(px)/float64(width)*(realMax-realMin) +
                realMin
            z := complex(x, y)
            c := z
            iteration := mandelbrot(z, c, maxIterations)

            // Paint Pixel depending on Iteration count
            if iteration >= maxIterations {
                img.Set(px, py, color.Black)
            } else {
                img.Set(px, py, colorize(iteration))
            }
        }
    }

    // Save PNG
    f, _ := os.Create("mandelbrot_theory.png")
    png.Encode(f, img)
    f.Close()

    // Calculate and Print Computation Time
    elapsedTime := time.Since(startTime)
    fmt.Printf("Mandelbrot PNG generation completed in %v\n",
        elapsedTime)
}
```

```
// Mandelbrot Algorithm
func mandelbrot(z, c complex128, maxIterations int) (int, float64) {
    var v complex128
    for i := 0; i < maxIterations; i++ {
        if cmplx.Abs(z) > 2 {
            log_zn := math.Log(real(z)*real(z) + imag(z)*imag(z)) /
                2
            nu := math.Log(log_zn / math.Log(2)) / math.Log(2)
            return i, float64(i) + 1 - nu
        }
        z = z*z + c
        if z == v {
            return maxIterations, 0
        }
        v = z
    }
    return maxIterations, 0
}

// Set Color for Pixel
func colorize(t float64) color.Color {
    t = math.Mod(t*0.1, 1.0) // Adjust color cycle frequency
    hue := 6 * t
    saturation := 1.0
    value := 1.0
    if t >= 1.0 {
        hue = 0
        saturation = 0
        value = 0
    }

    hi := math.Floor(hue)
    f := hue - hi
    p := value * (1 - saturation)
    q := value * (1 - saturation*f)
    t = value * (1 - saturation*(1-f))

    var r, g, b float64
    switch int(hi) % 6 {
        case 0:
            r, g, b = value, t, p
        case 1:
            r, g, b = q, value, p
        case 2:
            r, g, b = p, value, t
        case 3:
            r, g, b = p, q, value
        case 4:
            r, g, b = t, p, value
        case 5:
            r, g, b = value, p, q
```

```
        }

        return color.RGBA{
            uint8(r * 255),
            uint8(g * 255),
            uint8(b * 255),
            255,
        }
    }
```

This code has been executed in three isolated runs as follows:

```
go run mandelbrot_native.go
```

The output of this code on the system specified in A.1.1 was:

```
Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 33.592224199s

Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 34.73396341s

Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 36.602893897s
```

A.2.2 *Calculation of Mandelbrot Set: Go (WASM build)*

Below is the Go source code for generating a colored PNG file of the Mandelbrot set with all required additions to be compiled to a WebAssembly binary file and executed in the browser:

Listing A.2: Mandelbrot Set Calculation: Go (WASM build)

```
package main

import (
    "bytes"
    "strconv"
    "fmt"
    "image"
    "image/color"
    "image/png"
    "math"
    "math/cmplx"
    "syscall/js"
    "time"
)

func main() {
    fmt.Println("Hello WebAssembly")
    c := make(chan bool)
    js.Global().Set("wasmMain", js.FuncOf(wasmMain))
    <-c
```

```
}

func wasmMain(this js.Value, args []js.Value) any {
    startTime := time.Now()

    // Get Input Arguments
    width, _ := strconv.Atoi(args[0].String())
    height, _ := strconv.Atoi(args[1].String())
    maxIterations, _ := strconv.Atoi(args[2].String())
    realMin, _ := strconv.ParseFloat(args[3].String(), 64)
    realMax, _ := strconv.ParseFloat(args[4].String(), 64)
    imagMin, _ := strconv.ParseFloat(args[5].String(), 64)
    imagMax, _ := strconv.ParseFloat(args[6].String(), 64)

    img := image.NewRGBA(image.Rect(0, 0, width, height))
    fmt.Println("Generating PNG of Mandelbrot set... ")

    // Calculate Mandelbrot for each Pixel
    for py := 0; py < height; py++ {
        y := imagMax - float64(py)/float64(height)*(imagMax-imagMin)
        for px := 0; px < width; px++ {
            x := float64(px)/float64(width)*(realMax-realMin) +
                realMin
            z := complex(x, y)
            c := z

            iteration := mandelbrot(z, c, maxIterations)

            // Paint Pixel depending on Iteration count
            if iteration >= maxIterations {
                img.Set(px, py, color.Black)
            } else {
                img.Set(px, py, colorize(iteration))
            }
        }
    }

    // Generate PNG BLOB
    var buf bytes.Buffer
    png.Encode(&buf, img)

    elapsedTime := time.Since(startTime)
    fmt.Printf("Mandelbrot PNG generation completed in %v\n",
        elapsedTime)

    // Convert []byte to JS Uint8Array
    uint8Array := js.Global().Get("Uint8Array").New(buf.Len())
    js.CopyBytesToJS(uint8Array, buf.Bytes())

    return uint8Array
}
```

```
// Mandelbrot Algorithm
func mandelbrot(z, c complex128, maxIterations int) (int, float64) {
    var v complex128
    for i := 0; i < maxIterations; i++ {
        if cmplx.Abs(z) > 2 {
            log_zn := math.Log(real(z)*real(z) + imag(z)*imag(z)) /
                2
            nu := math.Log(log_zn / math.Log(2)) / math.Log(2)
            return i, float64(i) + 1 - nu
        }
        z = z*z + c
        if z == v {
            return maxIterations, 0
    }
    v = z
    }
    return maxIterations, 0
}

// Set Color for Pixel
func colorize(t float64) color.Color {
    t = math.Mod(t*0.1, 1.0) // Adjust color cycle frequency
    hue := 6 * t
    saturation := 1.0
    value := 1.0
    if t >= 1.0 {
        hue = 0
        saturation = 0
        value = 0
    }

    hi := math.Floor(hue)
    f := hue - hi
    p := value * (1 - saturation)
    q := value * (1 - saturation*f)
    t = value * (1 - saturation*(1-f))

    var r, g, b float64
    switch int(hi) % 6 {
        case 0:
            r, g, b = value, t, p
        case 1:
            r, g, b = q, value, p
        case 2:
            r, g, b = p, value, t
        case 3:
            r, g, b = p, q, value
        case 4:
            r, g, b = t, p, value
        case 5:
            r, g, b = value, p, q
    }
```

```
        return color.RGBA{
            uint8(r * 255),
            uint8(g * 255),
            uint8(b * 255),
            255,
        }
    }
```

This code was compiled to a WebAssembly binary using the Go [Con24a] compiler as follows:

```
GOOS=js GOARCH=wasm go build -o mandelbrot.wasm mandelbrot.go
```

Listing A.3: Execute *mandelbrot.wasm* in Browser (HTML)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Mandelbrot Set with WebAssembly</title>
    <script src="wasm_exec.js"></script>
</head>
<body>
    <h1>Mandelbrot Set Calculator</h1>
    <button id="calculate">Calculate Mandelbrot Set</button>
    <br><br>
    <img id="mandelbrotImage" alt="Mandelbrot Set">
    <script>
        // Set Up WebAssembly Environment
        const go = new Go();
        let wasmInstance;

        WebAssembly.instantiateStreaming(fetch("mandelbrot.wasm"),
            go.importObject).then((result) => {
            wasmInstance = result.instance;
            go.run(wasmInstance);
        });

        document.getElementById('calculate').addEventListener('click
            ', () => {
            const pngData = wasmMain('1500', '1500', '3000', '-2.0',
                 '1.0', '-1.5', '1.5');

            // Create a Blob from the PNG data
            const blob = new Blob([pngData], {type: 'image/png'});
            const url = URL.createObjectURL(blob);

            // Display the image
            const img = document.getElementById('mandelbrotImage');
            img.src = url;
            img.onload = () => URL.revokeObjectURL(url);  // Clean
                up the object URL
```

```
        });
    </script>
</body>
</html>
```

This code has been executed in three isolated runs by accessing the web application with a Mozilla Firefox 132.0 [Moz24a] browser on the system specified in A.1.1. The resulting output was:

```
Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 1m11.048999936s

Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 1m11.327000064s

Generating PNG of Mandelbrot set...
Mandelbrot PNG generation completed in 1m11.123000064s
```

A.2.3 *Calculation of Mandelbrot Set: Go (native benchmark)*

Below is the Go source code for generating 101 unique colored PNG files of the Mandelbrot set:

Listing A.4: Mandelbrot Set Calculation: Go (native benchamrk)

```
package main

import (
    "os"
    "fmt"
    "image"
    "image/color"
    "image/png"
    "math"
    "math/cmplx"
    "time"
)

func main() {
    startTime := time.Now()

    // Set Parameters
    width := 1500
    height := 1500
    maxIterations := 3000
    realMin := -2.0
    realMax := 1.0
    imagMin := -1.5
    imagMax := 1.5
    gridSize := 10
    counter := 0
    realStep := (realMax - realMin) / float64(gridSize)
```

```
    imagStep := (imagMax - imagMin) / float64(gridSize)

    // Generate Thumbnail
    generatePNG(counter, width, height, maxIterations, realMin,
        realMax, imagMin, imagMax)
    counter++
    // Generating Grid
    for y := 0; y < gridSize; y++ {
        imagLower := imagMax - (float64(y + 1) * imagStep)
        imagUpper := imagMax - (float64(y) * imagStep)
        for x := 0; x < gridSize; x++ {
            realLower := realMin + (float64(x) * realStep)
            realUpper := realMin + (float64(x + 1) * realStep)
            generatePNG(counter, width, height, maxIterations,
                realLower, realUpper, imagLower, imagUpper)
            counter++
        }
    }

    // Calculate and Print Computation Time
    elapsedTime := time.Since(startTime)
    fmt.Printf("Mandelbrot PNG generation completed in %v\n",
        elapsedTime)
}

// Generate Mandelbrot PNG
func generatePNG(counter, width, height, maxIterations int, realMin,
     realMax, imagMin, imagMax float64) {
    img := image.NewRGBA(image.Rect(0, 0, width, height))
    fmt.Printf("Generating PNG #%v of Mandelbrot set...\n", counter)

    // Calculate Mandelbrot for each Pixel
    for py := 0; py < height; py++ {
        y := imagMax - float64(py)/float64(height)*(imagMax-imagMin)
        for px := 0; px < width; px++ {
            x := float64(px)/float64(width)*(realMax-realMin) +
                realMin
            z := complex(x, y)
            c := z
            iteration, smooth := mandelbrot(z, c, maxIterations)

            // Paint Pixel depending on Iteration count
            if iteration >= maxIterations {
                img.Set(px, py, color.Black)
            } else {
                img.Set(px, py, colorize(smooth))
            }
        }
    }

    // Save PNG
    f, _ := os.Create(fmt.Sprintf("mandelbrot_%d.png", counter))
```

```
    png.Encode(f, img)
    f.Close()
}

// Mandelbrot Algorithm
func mandelbrot(z, c complex128, maxIterations int) (int, float64) {
    var v complex128
    for i := 0; i < maxIterations; i++ {
        if cmplx.Abs(z) > 2 {
            log_zn := math.Log(real(z)*real(z) + imag(z)*imag(z)) /
                2
            nu := math.Log(log_zn / math.Log(2)) / math.Log(2)
            return i, float64(i) + 1 - nu
        }
        z = z*z + c
        if z == v {
            return maxIterations, 0
        }
        v = z
    }
    return maxIterations, 0
}

// Set Color for Pixel
func colorize(t float64) color.Color {
    t = math.Mod(t*0.1, 1.0) // Adjust color cycle frequency
    hue := 6 * t
    saturation := 1.0
    value := 1.0
    if t >= 1.0 {
        hue = 0
        saturation = 0
        value = 0
    }

    hi := math.Floor(hue)
    f := hue - hi
    p := value * (1 - saturation)
    q := value * (1 - saturation*f)
    t = value * (1 - saturation*(1-f))

    var r, g, b float64
    switch int(hi) % 6 {
        case 0:
            r, g, b = value, t, p
        case 1:
            r, g, b = q, value, p
        case 2:
            r, g, b = p, value, t
        case 3:
            r, g, b = p, q, value
        case 4:
```

```
            r, g, b = t, p, value
        case 5:
            r, g, b = value, p, q
    }

    return color.RGBA{
        uint8(r * 255),
        uint8(g * 255),
        uint8(b * 255),
        255,
    }
}
```

This code has been executed in three isolated runs as follows:

```
go run mandelbrot_benchmark.go
```

The output of this code on the system specified in A.1.3 was:

```
Generating PNG #0 of Mandelbrot set...
   [...]
Generating PNG #100 of Mandelbrot set...
Mandelbrot PNG generation completed in 10m43.723762025s

Generating PNG #0 of Mandelbrot set...
   [...]
Generating PNG #100 of Mandelbrot set...
Mandelbrot PNG generation completed in 10m48.318590362s

Generating PNG #0 of Mandelbrot set...
   [...]
Generating PNG #100 of Mandelbrot set...
Mandelbrot PNG generation completed in 10m44.884158311s
```

A.2.4 *Calculation of Mandelbrot Set: C++ (WASM build)*

Below is the C++ source code for generating a colored PNG file of the Mandelbrot set with all required additions to be compiled to a WebAssembly binary file and executed in the browser:

Listing A.5: Mandelbrot Set Calculation: C++ (WASM build)

```
#include <emscripten/bind.h>
#include <emscripten/val.h>
#include <complex>
#include <vector>
#include <cmath>
#include <chrono>
#include <iostream>
#include "lodepng.h"

using namespace emscripten;
```

```
// Structure to hold iteration result and smooth coloring value
struct IterationResult {
    int iterations;
    double smooth_value;
};

// Color structure
struct Color {
    uint8_t r;
    uint8_t g;
    uint8_t b;
    uint8_t a;
};

// Mandelbrot calculation function
IterationResult mandelbrot(std::complex<double> z, std::complex<
    double> c, int maxIterations) {
    std::complex<double> v(0, 0);

    for (int i = 0; i < maxIterations; i++) {
        if (std::abs(z) > 2.0) {
            double log_zn = std::log(std::norm(z)) / 2.0;
            double nu = std::log(log_zn / std::log(2.0)) / std::log
                (2.0);
            return {i, static_cast<double>(i) + 1.0 - nu};
        }
        z = z * z + c;
        if (z == v) {
            return {maxIterations, 0.0};
        }
        v = z;
    }
    return {maxIterations, 0.0};
}

// Color calculation function
Color colorize(double t) {
    t = fmod(t * 0.1, 1.0); // Adjust color cycle frequency
    double hue = 6.0 * t;
    double saturation = 1.0;
    double value = 1.0;

    if (t >= 1.0) {
        return {0, 0, 0, 255};
    }

    int hi = static_cast<int>(std::floor(hue));
    double f = hue - hi;
    double p = value * (1.0 - saturation);
    double q = value * (1.0 - saturation * f);
    double tt = value * (1.0 - saturation * (1.0 - f));
```

```
    double r = 0, g = 0, b = 0;
    switch (hi % 6) {
        case 0: r = value; g = tt; b = p; break;
        case 1: r = q; g = value; b = p; break;
        case 2: r = p; g = value; b = tt; break;
        case 3: r = p; g = q; b = value; break;
        case 4: r = tt; g = p; b = value; break;
        case 5: r = value; g = p; b = q; break;
    }

    return {
        static_cast<uint8_t>(r * 255),
        static_cast<uint8_t>(g * 255),
        static_cast<uint8_t>(b * 255),
        255
    };
}

// Main function to generate Mandelbrot set
std::vector<unsigned char> generateMandelbrot(int width, int height,
     int maxIterations,
                                              double realMin, double
                                                  realMax,
                                              double imagMin, double
                                                  imagMax) {
    std::vector<unsigned char> imageData(width * height * 4);

    // Calculate Mandelbrot for each pixel
    for (int py = 0; py < height; py++) {
        double y = imagMax - static_cast<double>(py) / height * (
            imagMax - imagMin);
        for (int px = 0; px < width; px++) {
            double x = static_cast<double>(px) / width * (realMax -
                realMin) + realMin;
            std::complex<double> z(x, y);
            std::complex<double> c = z;

            auto [iteration, smooth] = mandelbrot(z, c,
                maxIterations);
            Color color;

            if (iteration >= maxIterations) {
                color = {0, 0, 0, 255}; // Black
            } else {
                color = colorize(smooth);
            }

            // Set pixel in RGBA format
            size_t idx = (py * width + px) * 4;
            imageData[idx] = color.r;
            imageData[idx + 1] = color.g;
            imageData[idx + 2] = color.b;
```

```
            imageData[idx + 3] = color.a;
        }
    }

    return imageData;
}

// WebAssembly exposed function
val wasmMain(val args) {
    // Input validation
    if (!args.isArray()) {
        std::cerr << "Error: Expected array of arguments" << std::
            endl;
        return val::null();
    }

    // Get array length using proper method
    int argsLength = args["length"].as<int>();

    if (argsLength < 7) {
        std::cerr << "Error: Not enough arguments" << std::endl;
        return val::null();
    }

    // Extract arguments
    int width = args[0].as<int>();
    int height = args[1].as<int>();
    int maxIterations = args[2].as<int>();
    double realMin = args[3].as<double>();
    double realMax = args[4].as<double>();
    double imagMin = args[5].as<double>();
    double imagMax = args[6].as<double>();

    auto startTime = std::chrono::high_resolution_clock::now();
    std::cout << "Generating PNG of Mandelbrot set..." << std::endl;

    // Generate the Mandelbrot image data
    auto imageData = generateMandelbrot(width, height, maxIterations
        ,
                                        realMin, realMax, imagMin,
                                            imagMax);

    // Encode to PNG
    std::vector<unsigned char> pngData;
    unsigned error = lodepng::encode(pngData, imageData, width,
        height);

    if (error) {
        std::cerr << "PNG encoding error: " << error << std::endl;
        return val::null();
    }
```

```
    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::
        milliseconds>(endTime - startTime);
    std::cout << "Mandelbrot PNG generation completed in " <<
        duration.count() << "ms" << std::endl;

    // Create JavaScript Uint8Array from the PNG data
    val uint8Array = val::global("Uint8Array").new_(pngData.size());
    uint8Array.call<void>("set", val(typed_memory_view(pngData.size
        (), pngData.data())));

    return uint8Array;
}

EMSCRIPTEN_BINDINGS(module) {
    function("wasmMain", &wasmMain);
}
```

This code was compiled to a WebAssembly binary using the emscripten [Con24c] toolchain with the corresponding *loadpng.cpp* file as follows:

```
emcc mandelbrot10x10-cpp.cpp lodepng.cpp -o mandelbrot10x10-cpp.js -
    s WASM=1 -s NO_EXIT_RUNTIME=1 -s "EXPORTED_RUNTIME_METHODS=['
    ccall', 'cwrap']" -s EXPORTED_FUNCTIONS="['_malloc', '_free']" -
    O3 -s ALLOW_MEMORY_GROWTH=1 -s EXPORT_NAME='GlueCode' -s
    MODULARIZE=1 -s ENVIRONMENT=worker --bind
```

BIBLIOGRAPHY

[AR09] D.P. Anderson and K. Reed. "Celebrating Diversity in Volunteer Computing." In: *2009 42nd Hawaii International Conference on System Sciences*. 2009, pp. 1–8. DOI: 10.1109/HICSS.2009.105.

[And20] David P. Anderson. "BOINC: A Platform for Volunteer Computing." In: *Journal of Grid Computing* 18 (2020), pp. 99–122. DOI: https://doi.org/10.1007/s10723-019-09497-9.

[And+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@home: an experiment in public-resource computing." In: *Commun. ACM* 45.11 (Nov. 2002), pp. 56–61. ISSN: 0001-0782. DOI: 10.1145/581571.581573.

[App24a] Apple Inc. *Safari 18.1 Release Notes*. 2024. URL: https://developer.apple.com/documentation/safari-release-notes/safari-18_1-release-notes (visited on 12/01/2024).

[App24b] Apple Inc. *iPad Pro, 11-inch (3rd generation) - Technical Specifications*. 2024. URL: https://support.apple.com/en-us/111897 (visited on 12/01/2024).

[App24c] Apple Inc. *iPhone 13 - Technical Specifications*. 2024. URL: https://support.apple.com/en-us/111872 (visited on 12/01/2024).

[Con24a] GO Contributors. *Go Wiki: WebAssembly*. 2024. URL: https://go.dev/wiki/WebAssembly (visited on 10/02/2024).

[Con24b] Socket.IO Contributors. *Socket.IO*. 2024. URL: https://socket.io/ (visited on 10/02/2024).

[Con24c] emscripten Contributors. *emscripten*. 2024. URL: https://emscripten.org/ (visited on 10/02/2024).

[Cor24a] Mozilla Corporation. *The WebSocket API (WebSockets)*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (visited on 10/02/2024).

[Cor24b] Mozilla Corporation. *Web Workers API*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API (visited on 10/02/2024).

[Cor24c] Mozilla Corporation. *WebAssembly*. 2024. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on 10/02/2024).

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[Doc24] Docker Inc. *Docker Compose*. 2024. URL: https://docs.docker.com/compose/ (visited on 12/03/2024).

[Fed+01] G. Fedak, C. Germain, V. Neri, and F. Cappello. "XtremWeb: a generic global computing system." In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 2001, pp. 582–587. DOI: 10.1109/CCGRID.2001.923246.

[GÉ24] GÉANT Contributors. *GÉANT*. 2024. URL: https://geant.org/ (visited on 12/03/2024).

[Gag04] Fabrizio Gagliardi. "The EGEE European Grid Infrastructure Project." In: *High Performance Computing for Computational Science - VECPAR 2004*. Springer. 2004, pp. 194–203. DOI: https://doi.org/10.1007/11403937_16.

[Gro24a] The PostgreSQL Global Development Group. *PostgreSQL*. 2024. URL: https://www.postgresql.org/ (visited on 10/02/2024).

[Gro24b] W3C Group. *WebAssembly*. 2024. URL: https://webassembly.org/ (visited on 10/02/2024).

[Gro19] WebAssembly Working Group. *WebAssembly Core Specification*. 2019. URL: https://www.w3.org/TR/wasm-core-1/ (visited on 11/21/2024).

[Haa+17] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[Hab+17] Karim Habak, Ellen W. Zegura, Mostafa Ammar, and Khaled A. Harras. "Workload management for dynamic mobile device clusters in edge femtoclouds." In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC '17. San Jose, California: Association for Computing Machinery, 2017. ISBN: 9781450350877. DOI: 10.1145/3132211.3134455.

[HH22] Mohammed Nurul Hoque and Khaled A. Harras. "WebAssembly for Edge Computing: Potential and Challenges." In: *IEEE Communications Standards Magazine* 6.4 (2022), pp. 68–73. DOI: 10.1109/MCOMSTD.0001.2000068.

[Inc24a] Cloudflare Inc. *Internet Quality Index*. 2024. URL: https://radar.cloudflare.com/quality (visited on 09/27/2024).

[Inc24b] Vercel Inc. *NextJS*. 2024. URL: https://nextjs.org/ (visited on 10/02/2024).

[Jan+19] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 107–120. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jangda.

[JBS15] M. Jones, J. Bradley, and N. Sakimura. *RFC 7519: JSON Web Token (JWT)*. USA, 2015. URL: https://dl.acm.org/doi/abs/10.17487/RFC7519.

[Mic24] Microsoft Corporation. *Release notes for Microsoft Edge Stable Channel*. 2024. URL: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-relnote-stable-channel#version-1310290351-november-15-2024 (visited on 12/01/2024).

[Moz24a] Mozilla Corporation. *Firefox 132.0 - Releasenotes*. 2024. URL: https://www.mozilla.org/en-US/firefox/132.0/releasenotes/ (visited on 11/19/2024).

[Moz24b] Mozilla Corporation. *Firefox 133.0 - Releasenotes*. 2024. URL: https://www.mozilla.org/en-US/firefox/133.0/releasenotes/ (visited on 11/28/2024).

[Mys24] Kamil Mysliwiec. *NestJS*. 2024. URL: https://nestjs.com/ (visited on 10/02/2024).

[PN12] Victoria Pimentel and Bradford G. Nickerson. "Communicating and Displaying Real-Time Data with WebSocket." In: *IEEE Internet Computing* 16.4 (2012), pp. 45–53. DOI: 10.1109/MIC.2012.64.

[Rea24] React Bootstrap Contributors. *React Bootstrap*. 2024. URL: https://react-bootstrap.netlify.app/ (visited on 11/06/2024).

[Ref14] Referencerelatedwork.com. *How Many Computers Are There in the World?* 2014. URL: https://www.reference.com/world-view/many-computers-world-e2e980daa5e128d0 (visited on 10/02/2024).

[SHW98] Luis F. G. Sarmenta, Satoshi Hirano, and Stephen A. Ward. "Towards Bayanihan: building an extensible framework for volunteer computing using Java." In: *Concurrency: Practice and Experience* 10.11-13 (1998), pp. 1015–1019. DOI: https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<1015::AID-CPE410>3.0.CO;2-C.

[Sta24a] Stack Overflow. *2024 Developer Survey: Web frameworks and technologies*. 2024. URL: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-webframe (visited on 10/06/2024).

[Sta24b] Statista Research Department. *Volume of the consumer electronics market worldwide from 2019 to 2029*. 2024. URL: https://www.statista.com/forecasts/1286681/worldwide-consumer-electronics-market-volume (visited on 11/20/2024).

["St24] "Statistics and Data" Contributors. *Most Popular Backend Frameworks - 2012/2024*. https://www.youtube.com/watch?v=RSNwYPnhoYc and https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2024/. [Online; accessed 05-September-2024]. 2024.

[Baca] *TechEmpower Web Framework Benchmarks - Round 22*. 2023. URL: https://www.techempower.com/benchmarks/#hw=ph&test=fortune&section=data-r22&a=2&f=zhavon-zik073-zik0zj-zik0zj-zijunz-zik0zj-zik0zj-zik0zj-zih7un-zhxjwf-zik0zj-zik0zj-zik0zj-zik0zj-1ekf (visited on 09/05/2024).

[TMN11] David Toth, Russell Mayer, and Wendy Nichols. "Increasing Participation in Volunteer Computing." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1878–1882. DOI: 10.1109/IPDPS.2011.353.

[Bacb] *Web Frameworks Benchmark*. 2024. URL: https://web-frameworks-benchmark.netlify.app/result?asc=0&f=phoenix_bandit,phoenix_cowboy,beego,spring,express,koa,nestjs-express,nestjs-fastify,spring,laravel,symfony,django,flask,rails&metric=totalRequestsPerS&order_by=level64 (visited on 09/05/2024).

[cM24] Pyodide contributors and Mozilla. *Pyodide*. 2024. URL: https://pyodide.org (visited on 10/02/2024).