# CIVIC (Central Intelligence Virtualization Instruction Cluster)

A capstone project report submitted in partial fulfillment of the requirements for the degree of

## Masters of Engineering

In the Department of Computer Science Graduate Program,

College of Engineering & Applied Science

04/11/2025

Margeson, Jack

# 1 Abstract

Distributed computing is a design methodology that enables digital storage and data processing to be completed across multiple devices, often on separate networks. CIVIC (Central Intelligence Virtualization Instruction Cluster) allows organizations (labs, campuses, businesses, etc.) to deploy their own instance of a cooperative computational system. CIVIC's platform streamlines program development for organizations by leveraging virtualization technologies, and the user-friendly and flexible workflow allows participants to easily volunteer computational resources to CIVIC instances.

The overall goal of this project is to address the limitations of current volunteer distributed computing programs and open new possibilities for research and development. Unlike existing frameworks that require specific programming languages or hardware architectures when writing applications, CIVIC provides a flexible and language/architecture-independent design.

Researchers can deploy their existing code to a CIVIC instance with minimal changes, while participants can easily contribute computational resources. This is achieved through the creation of models, a set of instructions that can be used by client machines to create virtualizations. These virtualizations, referred to as citizens, are worker containers capable of receiving fractions of input data and executing code to generate output. Each fraction of data generated by the server is called a duty, and each duty can be assigned to one or more citizens depending on the result verification needs of the project.

The CIVIC Server manages model distribution and communication with citizens, as well as processing, validating, and assimilating incoming duty results–ensuring efficient and scalable distributed computing by minimizing required human interaction from both organization and participant perspectives.

# 2 Introduction

In the field of distributed computing, one subset of applications is volunteer computing. Services that fall under this category, including SETI@home, Folding@home, and BOINC, allow users to donate their idle computing resources to scientific research projects. These projects typically require large amounts of computational power to process data, and volunteer computing allows researchers to leverage the collective power of many individual computers. For example, one of the most popular volunteer computing applications, Folding@home, is used

to simulate protein folding to contribute to scientific research in the field of biology through the study of diseases such as COVID-19 and Alzheimer's disease. The protein folding process is a complicated mathematical simulation that requires significant computational resources to run. For problems like these, volunteer computing is a powerful methodology to leverage the computing power of several machines to independently work on fractions of the problem. Utilizing volunteer computing in a scenario like this reduces the overall time that it would take one machine to complete the task.

However, there are limitations to some of the existing volunteer computing frameworks. For example, we can examine BOINC, the the Berkley Open Infrastructure for Network Computing. BOINC is a framework that allows researchers to create their own volunteer computing projects and allows volunteers to donate computational resources to any of these projects by downloading the client and selecting from a list of available causes. One of the big issues when it comes to creating a research project on BOINC is the fact that programs that are to be distributed to volunteers must be packaged in a very specific way. The BOINC framework provides three options for application developers:

- Native applications, in which the program must be written in a language that supports the official BOINC API and compiled for a target architecture.
- Wrapper applications, in which a program is written in a language that does not support the official BOINC API but can be run as a sub-process of a wrapper application that handles input and output.
- Virtual applications, in which a program is run in a virtual machine that is managed by the BOINC framework through external scripts and programs such as VirtualBox.

Each of these options have their own limitations. With native applications, the developer of the program must write in a programming language that either officially supports (C/C++) or unofficially supports (FORTRAN, Python) the BOINC API interface. For both native and wrapper applications, the program that is written must be compiled for a target architecture before distributed to a client. The implication of this is that if a researcher would like to allow participants to contribute to the project, they must first compile any programs written for the project for each architecture that they would like to support, i.e. x86 Windows, x86 Linux, ARM Linux, Android, etc. This can be a time consuming process, especially when program changes

are made and all of the architectures must be recompiled. Additionally, code changes must be made to programs when dealing with certain functionality, such as differences in how system commands are handled between Windows and Unix based operating systems. The third option, virtual applications, mitigates some of the problems with native and wrapper applications. These types of BOINC applications are able to execute in a virtual machine, which allows the program to be written in any language and run on any architecture. However, this option entails significant additional setup and configuration with an external program, VirtualBox, which can introduce additional complexity, runtime overhead, and download size for volunteers.

These limitations are not exclusive to BOINC—several other volunteer computing frameworks have similar restrictions. In this capstone report, a new alternative to existing volunteer computing frameworks is proposed. CIVIC (Central Intelligence Virtualization Instruction Cluster) is a framework that utilizes virtualization technology to allow researchers to easily create and deploy volunteer computing projects. By utilizing virtualization, CIVIC allows for the development of language and architecture independent applications. In combination with a binary application, researchers can use the framework to parse and create datasets to create models for distribution. Volunteers looking to contribute to research are able to donate their computational resources to a project by creating a "citizen"—a virtual client program that runs in it's own isolated container environment. These citizens automatically download and execute model instructions given from the server with a subset of the input dataset and return the results to the server. The primary goal of CIVIC is to provide a framework that allows easy configuration and deployment of volunteer computing projects for researchers, while also providing a simple and user-friendly experience for volunteers looking to contribute to research. The framework is designed to be flexible and extensible, allowing researchers to easily adapt it to their specific needs.

# 3 Methods

The architecture of the framework can be split into two main components: the server and the client. The server component is responsible for managing the distribution of models, duties, and other instructions to connected clients. The client component, or citizen, is a container that receives inputs and commands from the server and returns outputs of executed duties.

Two scripts are provided to facilitate the installation and management of the server and

client components—`server_manager.py` and `client.py`. In addition to server installation, the server manager script is also used to create new models for the server to distribute and parse/generate datasets for those models. The client script is used to create new citizens that connect to the server and execute duties.

To achieve the server-client architecture outlined, several technologies are used. The first of these technologies is Docker, a containerization technology that allows for programs to be run in an isolated environment. Docker and the Docker Engine API are used in both the server and client components of CIVIC. The installation of a CIVIC server is handled through the server manager script which integrates with the Docker SDK to build and manage Docker images and containers that power the server. With the use of Docker, the script builds and installs four services: the ”internal CIVIC server”, a Python Flask middleware application, a PostgreSQL database, and Adminer, an external database management tool. In addition to handling image builds and container management, the installation process also creates volumes (persistent data) on the server machine for database use, as well as a virtual network to facilitate communication between the four services.

The first of the four services, the ”internal CIVIC server”, is a Python script that is in charge of handling connections with clients, downloading and distributing model files and datasets, and generating duties for clients to execute. The application utilizes `curses`, a Python extension module that enables finer control over terminal interfaces to create a console interface for management. This console can be accessed through the use of the server manager script by attaching to the Docker container. Within the console, the user can execute several commands to facilitate the operations outlined above.

Python Flask is used to create the second core services of the server, enabling intra-service communication via RESTful API calls. The Flask application contains several endpoints that allow the internal CIVIC server to talk to the PostgreSQL database. This middleware directly connects to the PostgreSQL database through the use of the `psycopg2` Python library, which enables SQL queries to be execute when connected to remote databases. Since all communication between client and server is done through the internal CIVIC server, clients do not directly interact with the Flask application.

PostgreSQL, the database software that makes up the third core service, is used to store information about models, duties, and clients. It is queried through use of RESTful API endpoints exposed in the Flask service to retrieve and store information about the state of the

CIVIC network. Several tables are created on installation of the server—and two additional tables are created upon the addition of a new model to store both the model's dataset and the results of duties executed by clients pertaining to that model.

The final service, Adminer, is an external database management tool that allows for the viewing and editing of the PostgreSQL database. Adminer is a web-based application that can be accessed through a web browser on the server machine. The tool is used to view the contents of the database, as well as to make changes to the database if necessary. Adminer is not required for the operation of the CIVIC server, but is included as a convenience for users who may want to view the contents of the database without needing to use the command line.

Combined, these four services create a server that is capable of managing the distribution of models and duties to clients and storing results for future analysis. The server is designed to be scalable, allowing for the addition of new clients and models as needed. The server is also designed to be fault-tolerant, with the ability to recover from crashes and other failures by utilizing the persistent data stored in the PostgreSQL database volume and Docker monitoring and container restart capabilities.

On the client side, the Docker SDK is used once again with the `client.py` script to build a single Docker image—the `internal_client` service, which, when run as a container, acts as a citizen in the CIVIC network. The citizen containers run on Alpine Linux, a lightweight and simple Linux distribution often used in combination with Docker due to the operating system's small file size and resource overhead. This container is "dumb" in the sense that when created, it no longer needs to communicate with the client machine that initializes it. Citizen containers retry their connection to the IP and port provided during creation until they are able to connect. Once connected, citizens will wait for instructions given by the central server. When prompted, the citizen will download model binaries and datasets from the server, execute the model with the given dataset, and return the results to the server. The citizen will then wait for the next instruction from the server, repeating the process until the server has no more duties to distribute. This process is repeated for each citizen in the network, allowing for the distributed execution of models across multiple instances of citizens on the same machine, citizens on multiple machines, and citizens on multiple external networks.

# 4 Results

The results of the CIVIC framework are promising. The server component is able to successfully manage multiple citizens connected from multiple different machines and distribute instructions, models, and datasets to those citizens. As stated previously, two Python scripts, `server_manager.py` and `client.py`, have been bundled with the framework to facilitate the installation and management of the server and client components. These scripts have been designed with simple terminal console-like interfaces to equip both researchers and volunteers with easy to understand

In addition to the creation of the CIVIC framework, this project has also created a proof of concept model that demonstrates the capabilities of the framework. Included in the project's repository is a simple model called Alphabet. The Alphabet model is a simple program that takes a single letter as input and returns the next letter in the alphabet as output, followed by the burning of CPU cycles for a short amount of time to simulate a heavier workload, which is typical in demonstrations of distributed computing. The model binary is written in C and compiled for Linux x86 architecture and is designed to be simple and easy to understand, making it a good starting point for researchers looking to create their own models for the CIVIC framework.

# 5 Discussion

# 6 Bibliography