Product Design

Team CTRL 5

Ololade Awoyemi, Shivam Sakthivel Pandi, Biraj Sharma, Benjamin Wilson, Sritan Reddy Gangidi

This document presents the product design for our capstone project being put together by Team CTRL 5. It describes the overall system depiction, design rationale, and technical choices that help in developing our solution. The purpose of this document is to indicate that the designs are well-structured, traceable to the requirements and open to changes as we go through development.

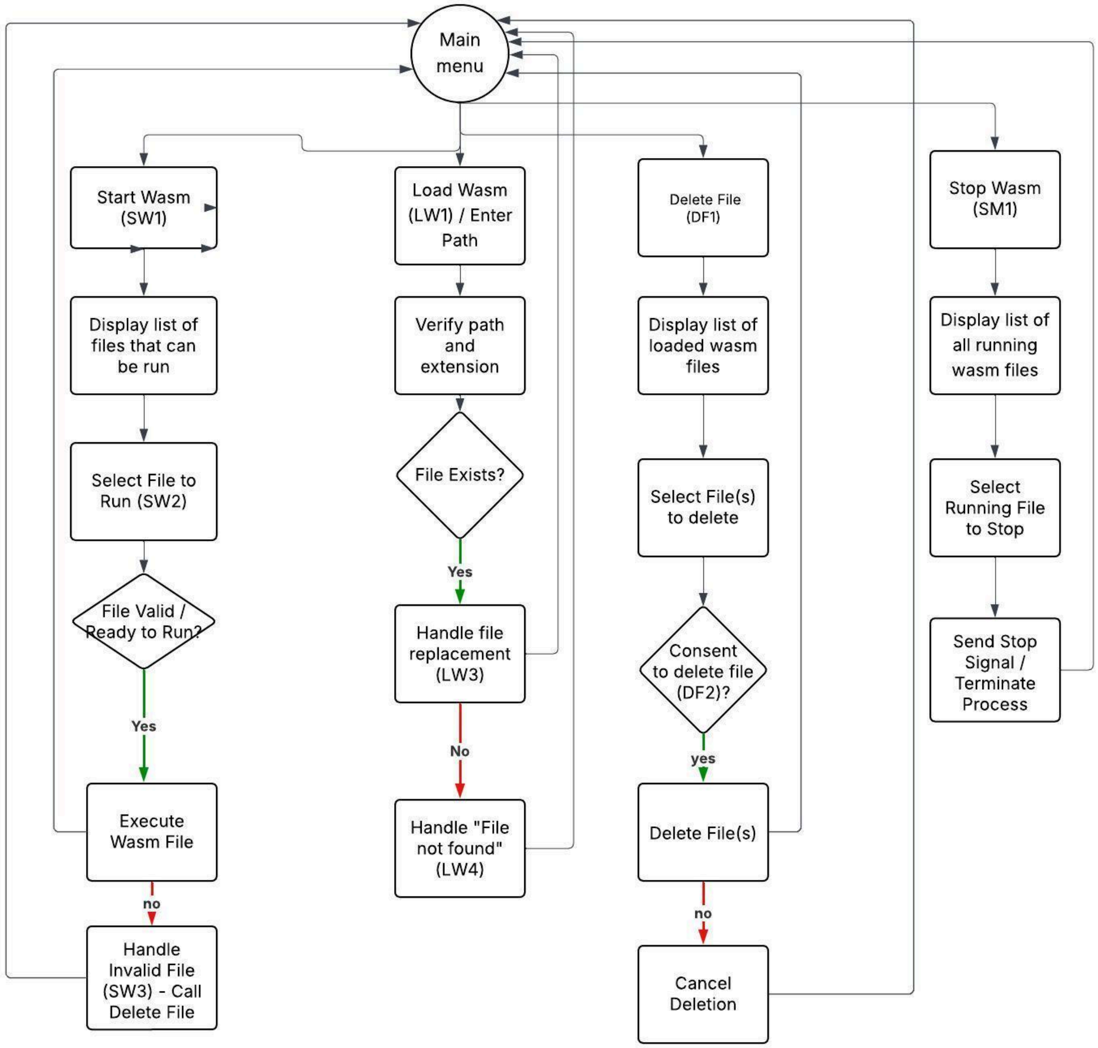
Our project involves the building of a secure and modular WebAssembly runtime with a Rust-based back-end and a React-based front-end. The design is concentrated upon performance, scalability, and safety so that untrusted WASM modules may execute themselves in an environment characterized by control. The present introduction lays the groundwork for the class diagrams, ER diagrams, UI wireframes and rationales which follow in this document. As development progresses, this document too will progress in describing the refinements and improvements of our system.

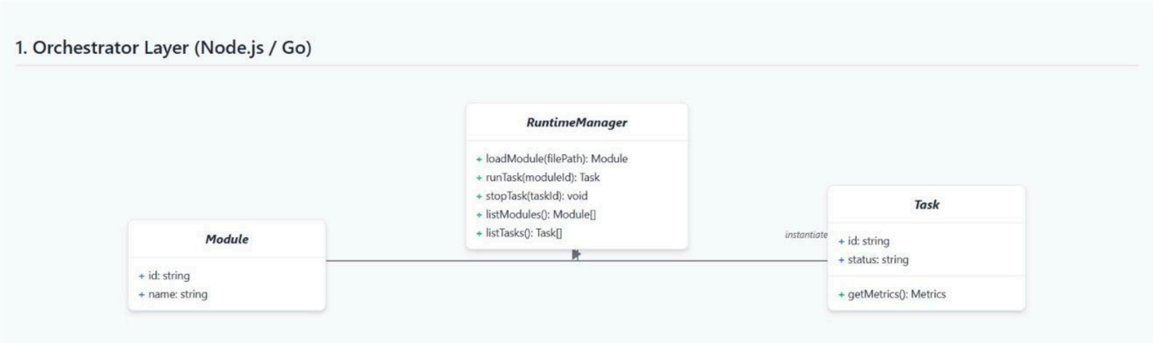
Revision Number	*Revision Date*	*Summary of Changes*	*Author(s)*
0.1	10/08/2025	Initial designs and diagrams.	Ololade Awoyemi, Shivam Sakthivel Pandi, Biraj Sharma, Benjamin Wilson, Sritan Reddy Gangidi

0.2	11/05/2025	Updated Design Rational, Added Wasm as Os CLI Screenshots, Added Wasm as OS class diagram	Benjamin Wilson
3.0	12-4-2025	Updated Design Rational, added Wasm Engine class Diagram	Benjamin Wilson
4.0	02-14-2026	Updated Design Rational and Summary	Benjamin Wilson

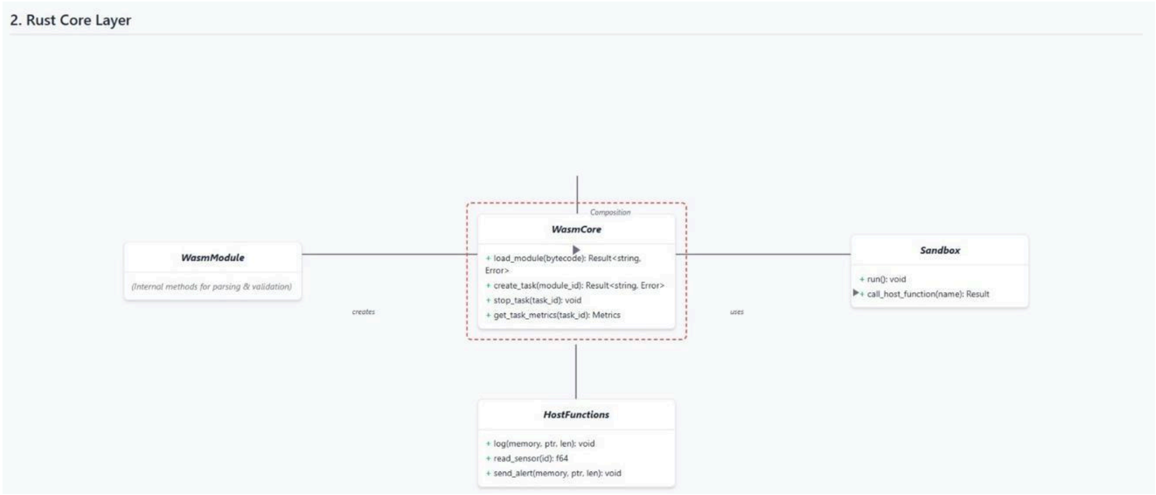
Class Diagram(s)

Menu Diagram

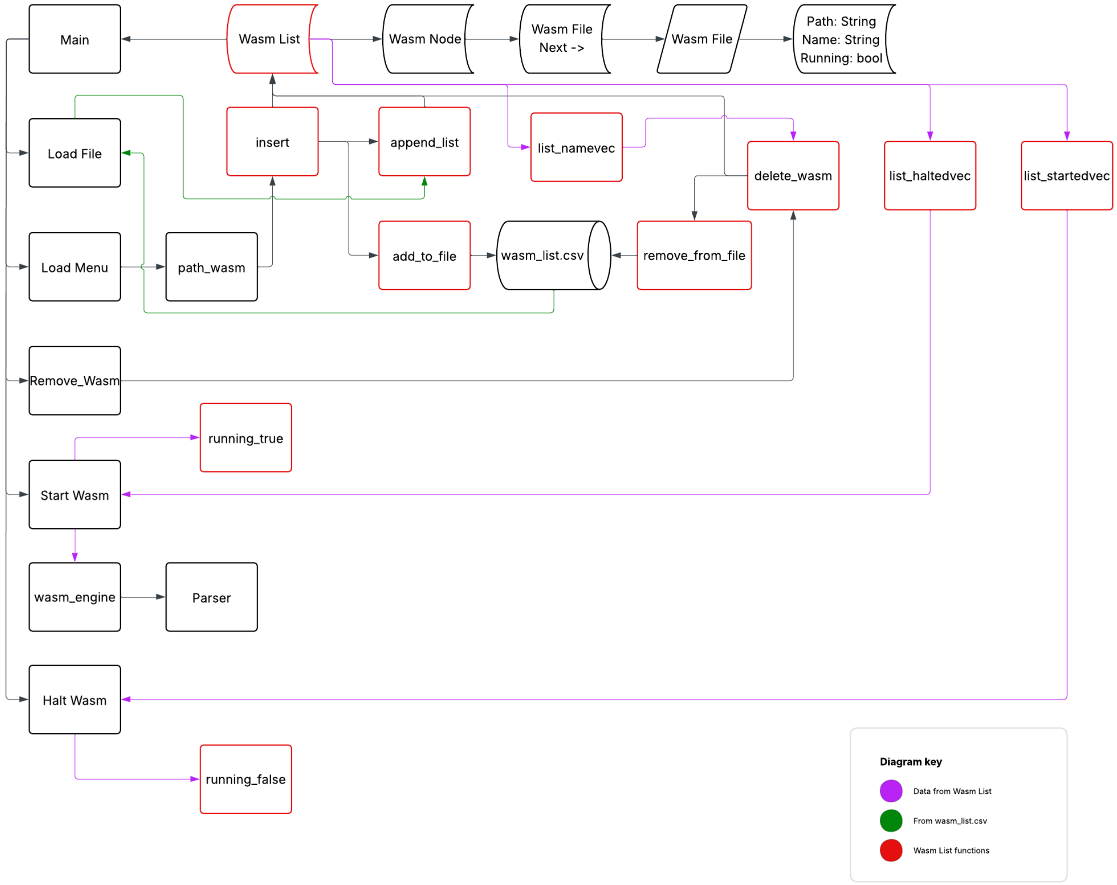

Main menu
Start Wasm (SW1)
Display list of files that can be run
Select File to Run (SW2)
File Valid / Ready to Run?
Yes
Execute Wasm File
no
Handle Invalid File (SW3) - Call Delete File
Load Wasm (LW1) / Enter Path
Verify path and extension
File Exists?
Yes
Handle file replacement (LW3)
No
Handle "File not found" (LW4)
Delete File (DF1)
Display list of loaded wasm files
Select File(s) to delete
Consent to delete file (DF2)?
yes
Delete File(s)
no
Cancel Deletion
Stop Wasm (SM1)
Display list of all running wasm files
Select Running File to Stop
Send Stop Signal / Terminate Process



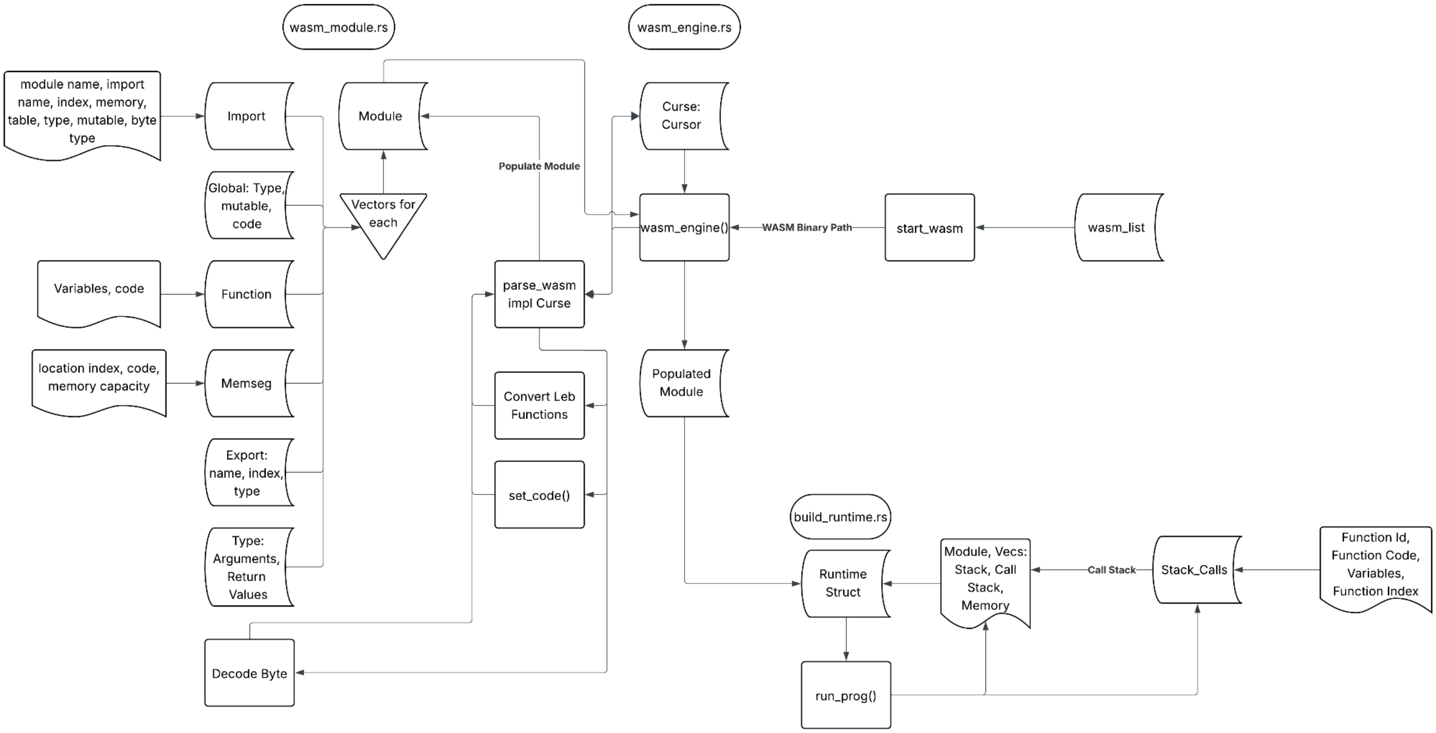
1. Orchestrator Layer (Node.js / Go)


2. Rust Core Layer
Composition
WasmCore
+ load_module(bytecode): Result<string, Error>
+ create_task(module_id): Result<string, Error>
+ stop_task(task_id): void
+ get_task_metrics(task_id): Metrics
WasmModule
(Internal methods for parsing & validation)
creates
uses
Sandbox
+ run(): void
+ call_host_function(name): Result
HostFunctions
+ log(memory, ptr, len): void
+ read_sensor(id): f64
+ send_alert(memory, ptr, len): void

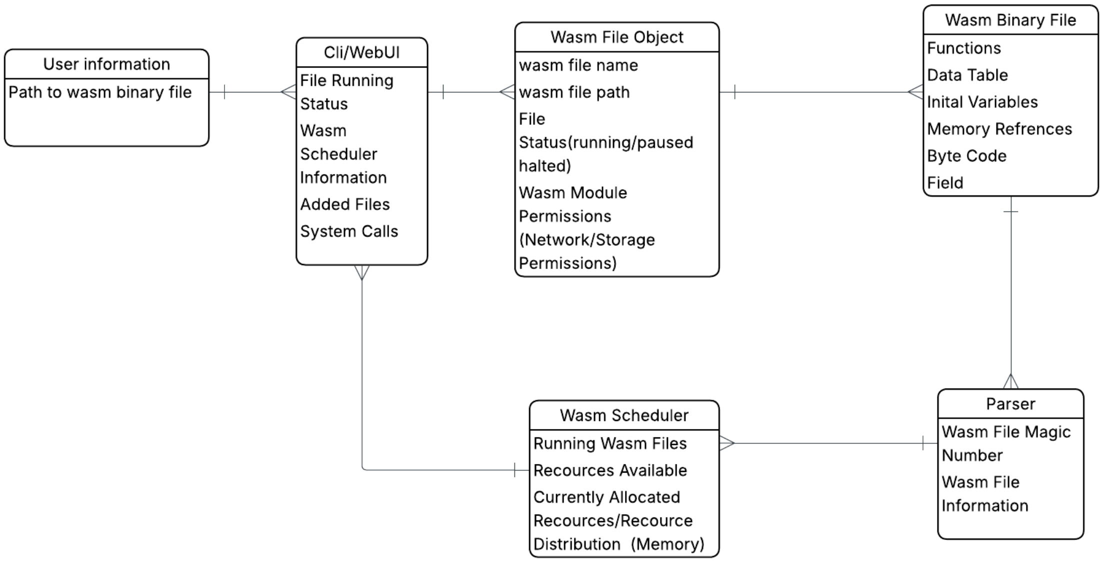
Wasm as Os Class Diagram



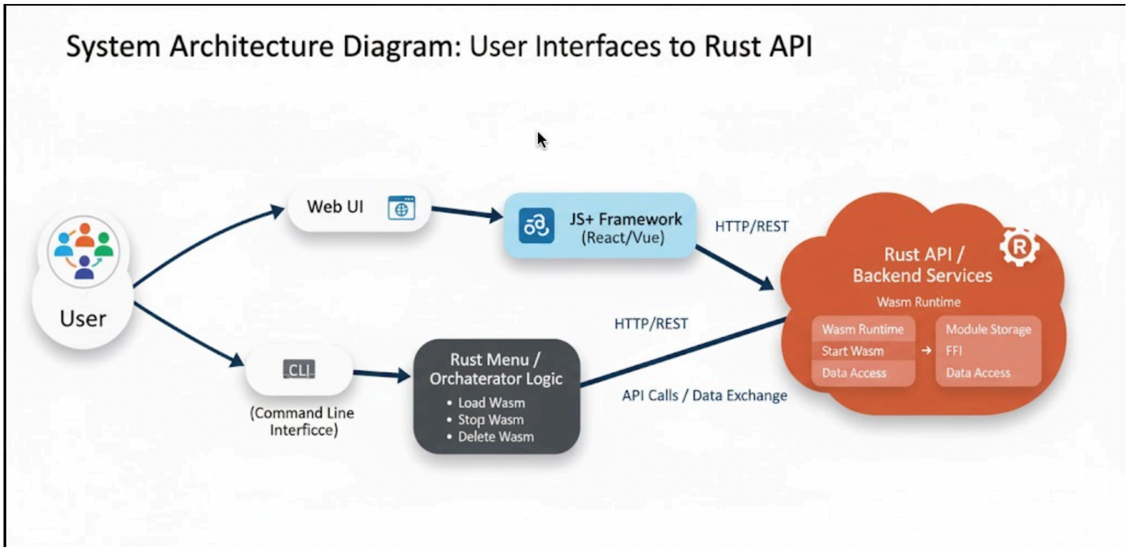
Wasm Engine Class Diagram



ER Diagram

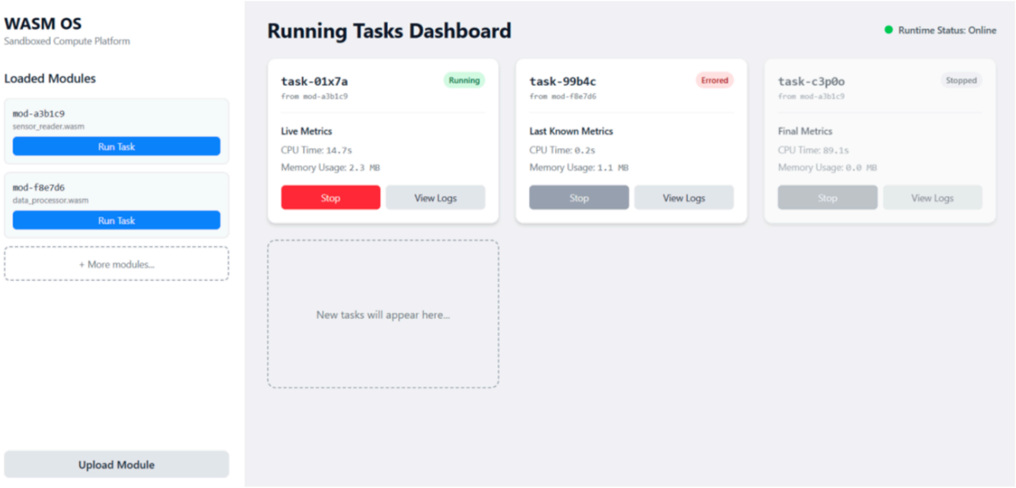


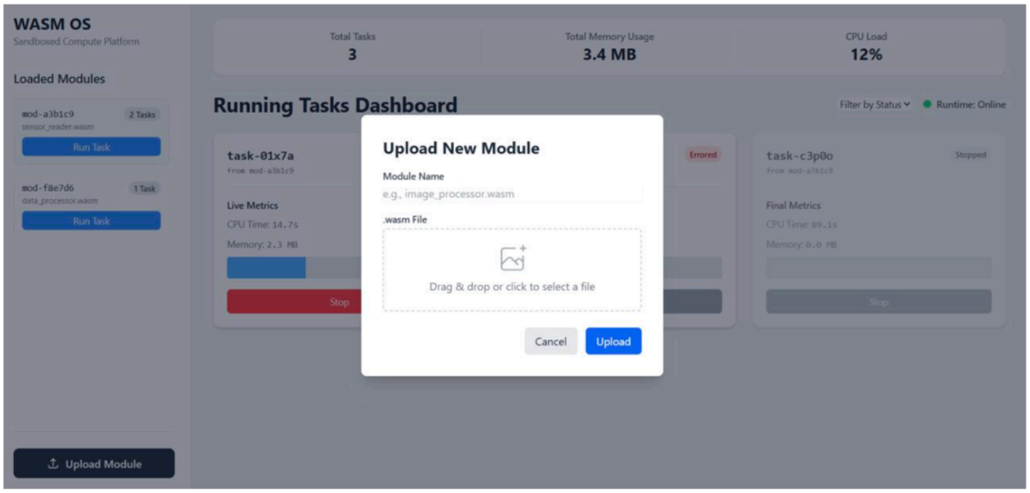
Information Architecture

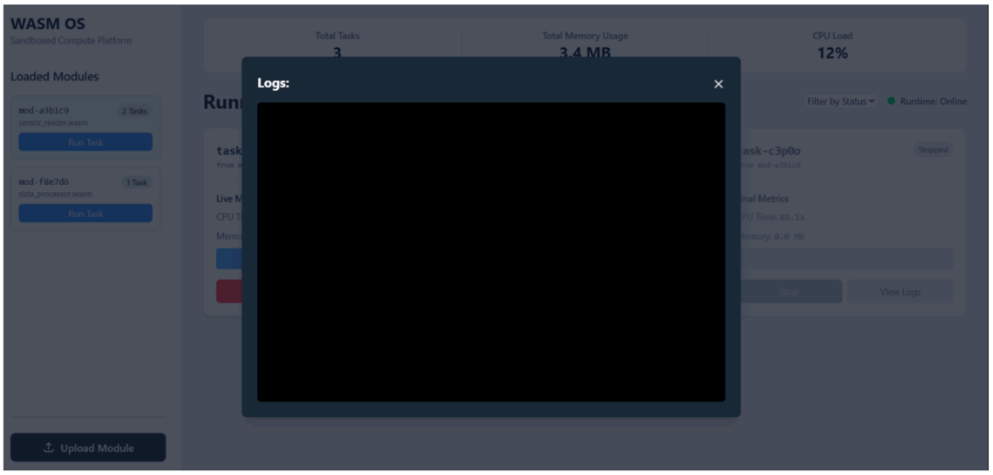


System Architecture Diagram: User Interfaces to Rust API

User Interface Wireframe(s)/Screenshot(s)

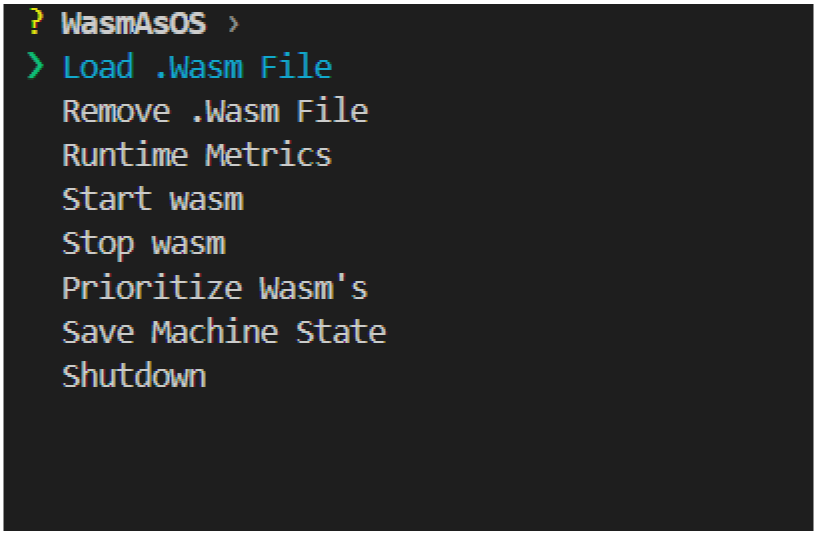






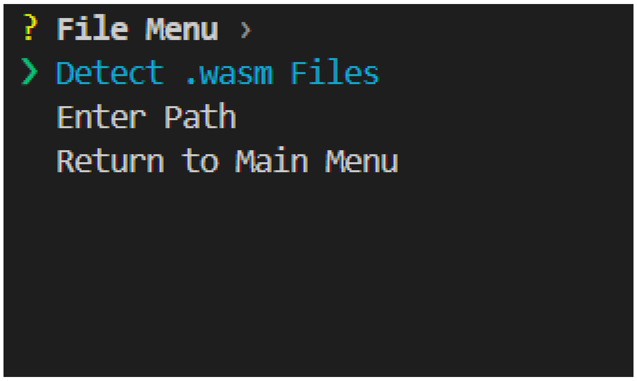
CLI Screenshots

MM1: Main Menu



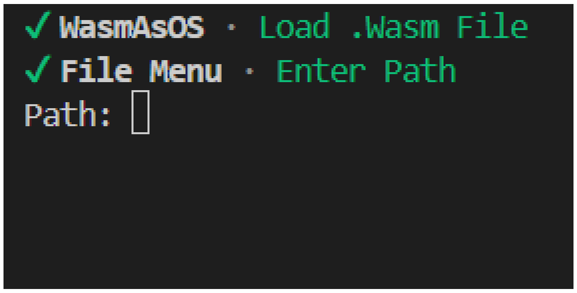
```
? WasmAsOS ›
❯ Load .Wasm File
  Remove .Wasm File
  Runtime Metrics
  Start wasm
  Stop wasm
  Prioritize Wasm's
  Save Machine State
  Shutdown
```

LW1: Load Wasm Menu: Display detect wasm file, Enter path and return to main menu.



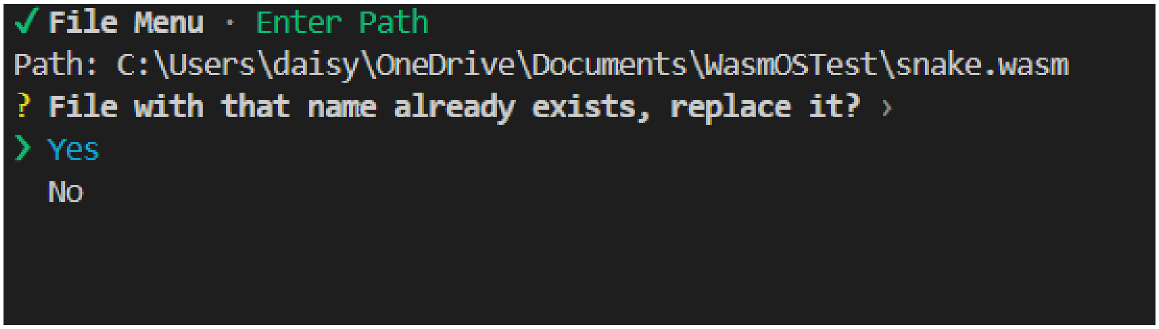
```
? File Menu ›
❯ Detect .wasm Files
  Enter Path
  Return to Main Menu
```

LW2: Enter Path Temporary, Detect Wasm in final product, Verify path is correct, add wasm file to list if path and extension are valid. Reload enter path menu.



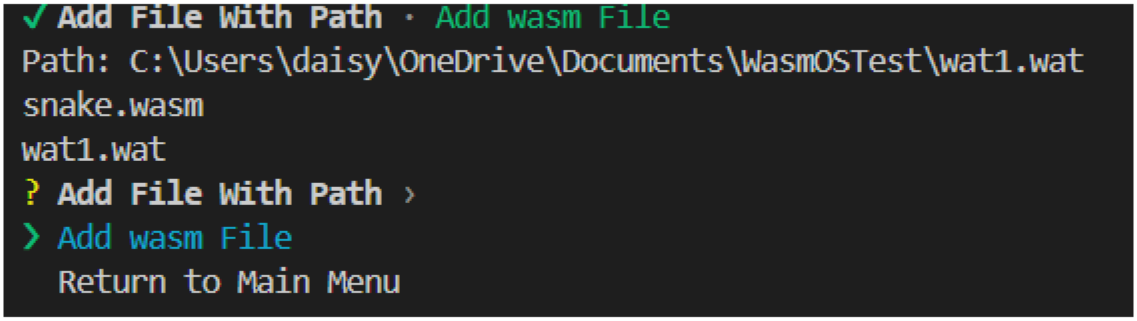
```
✓ WasmAsOS · Load .Wasm File
✓ File Menu · Enter Path
Path: 
```

LW3: Replace File: If a file with the same name exists, the user is asked if they want to replace it. Either option reloads the path menu



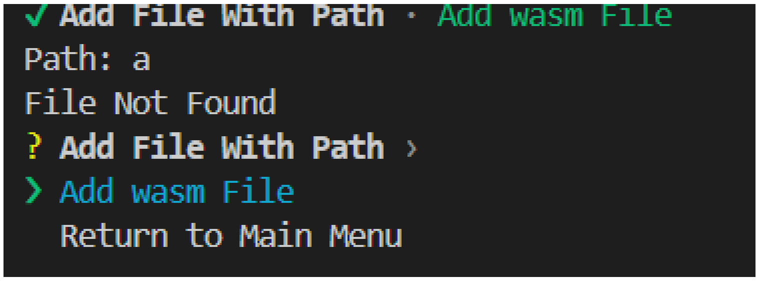
```
✓ File Menu · Enter Path
Path: C:\Users\daisy\OneDrive\Documents\WasmOSTest\snake.wasm
? File with that name already exists, replace it? ›
❯ Yes
  No
```

LW3: New File/file added (file name printouts temporary for debugging) Prompts for new path or return



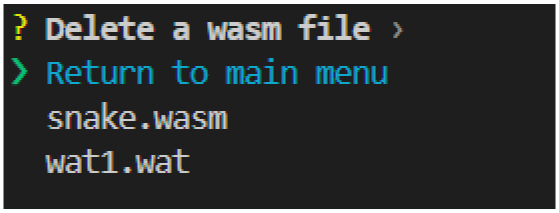
```
✓ Add File With Path · Add wasm File
Path: C:\Users\daisy\OneDrive\Documents\WasmOSTest\wat1.wat
snake.wasm
wat1.wat
? Add File With Path ›
❯ Add wasm File
  Return to Main Menu
```

LW4: File not found: display file not found reload path menu.



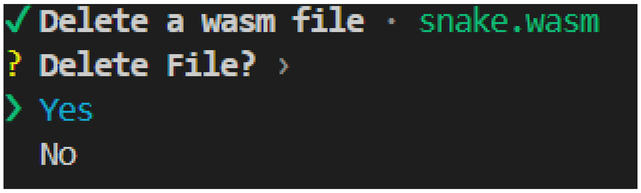
```
✓ Add File With Path · Add wasm File
Path: a
File Not Found
? Add File With Path ›
❯ Add wasm File
  Return to Main Menu
```

DF1: Delete File menu: Displays all loaded wasm files and return to main menu option. User may select multiple files to delete as menu reloads after each deletion



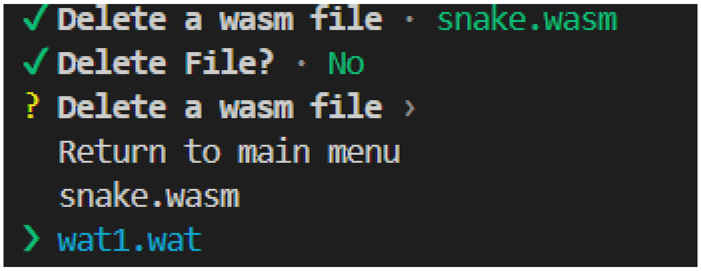
```
? Delete a wasm file ›
❯ Return to main menu
  snake.wasm
  wat1.wat
```

DF2: Consent to delete file: Yes deletes the file from program, both reload delete menu.



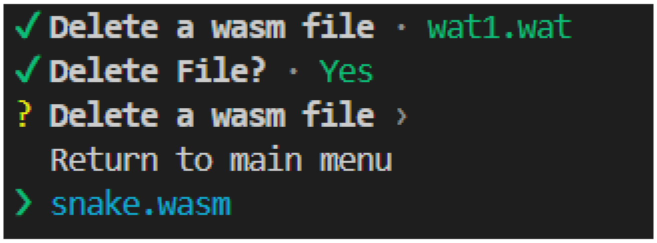
```
✓ Delete a wasm file · snake.wasm
? Delete File? ›
❯ Yes
  No
```

DF3: Menu loops after no



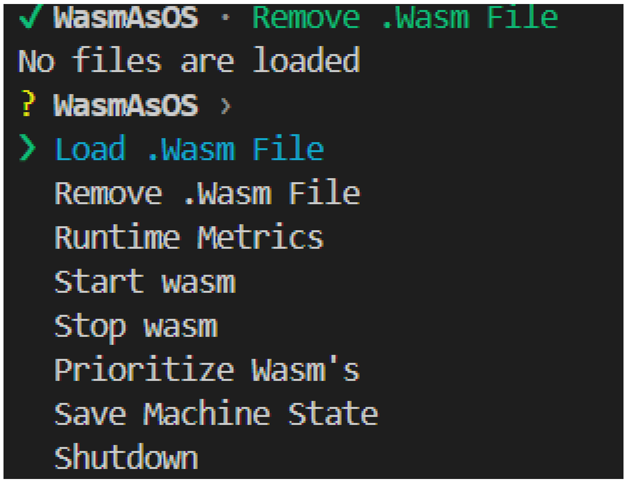
```
✓ Delete a wasm file · snake.wasm
✓ Delete File? · No
? Delete a wasm file ›
  Return to main menu
  snake.wasm
❯ wat1.wat
```

DF4: Menu loops after yes



```
✓ Delete a wasm file · wat1.wat
✓ Delete File? · Yes
? Delete a wasm file ›
  Return to main menu
❯ snake.wasm
```

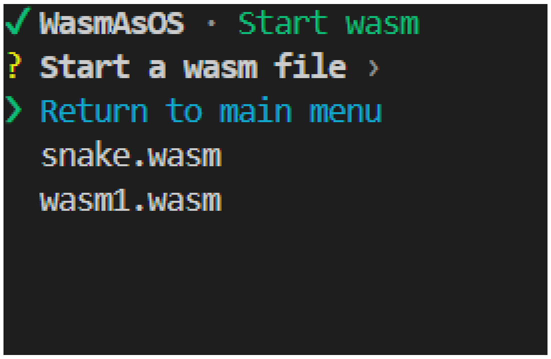
DF5: No wasm file to delete



```
✓ WasmAsOS · Remove .Wasm File
No files are loaded
? WasmAsOS ›
❯ Load .Wasm File
  Remove .Wasm File
  Runtime Metrics
  Start wasm
  Stop wasm
  Prioritize Wasm's
  Save Machine State
  Shutdown
```

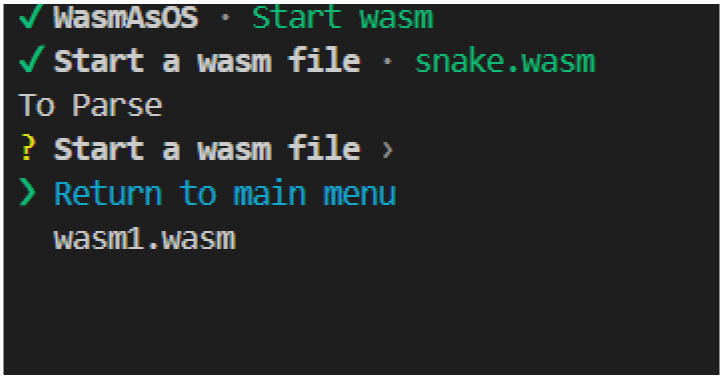
RT0: Runtime Metrics not functional at this time

SW1: Start Wasm Menu: Display Menu of files that can be run and return to main menu option. After a file is run loop updated menu.



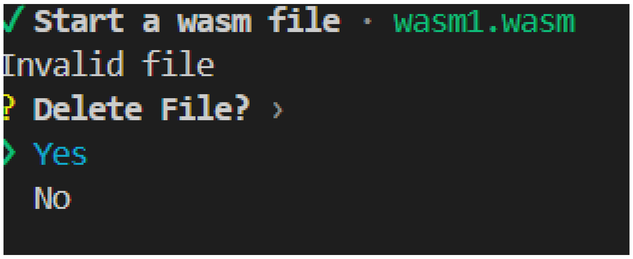
```
✓ WasmAsOS · Start wasm
? Start a wasm file ›
❯ Return to main menu
  snake.wasm
  wasm1.wasm
```

SW2: Looped menu: After starting a module (only UI functional currently, To parse temporary debug statement meaning file reached the parser)



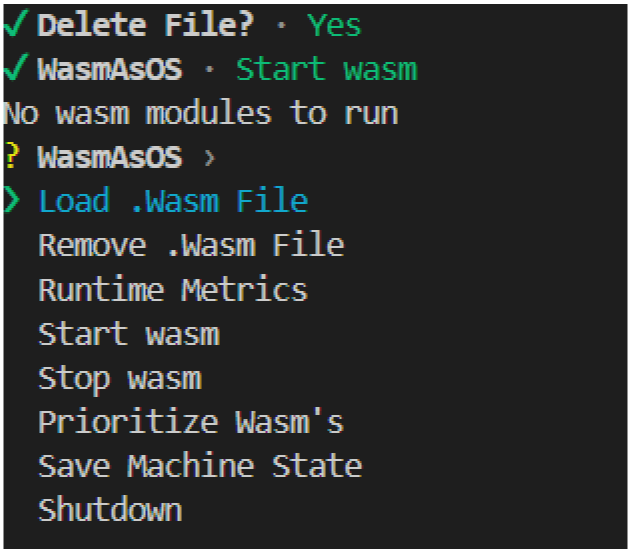
```
✓ WasmAsOS · Start wasm
✓ Start a wasm file · snake.wasm
To Parse
? Start a wasm file ›
❯ Return to main menu
  wasm1.wasm
```

SW3: Invalid File (calls delete file function)



```
✓ Start a wasm file · wasm1.wasm
Invalid file
? Delete File? ›
❯ Yes
  No
```

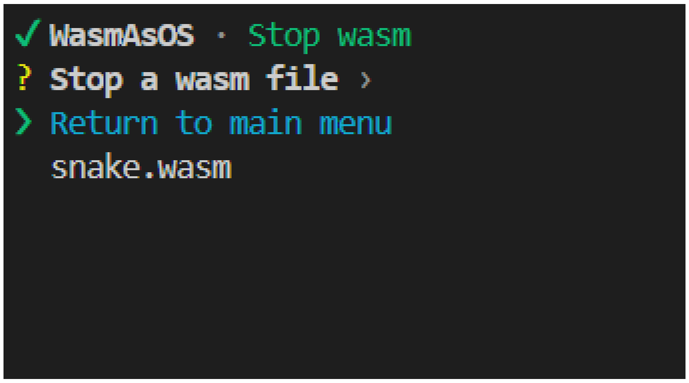
SW4: No files to run: If no files to run return to main menu



```
✓ Delete File? · Yes
✓ WasmAsOS · Start wasm
No wasm modules to run
? WasmAsOS ›
❯ Load .Wasm File
  Remove .Wasm File
  Runtime Metrics
  Start wasm
  Stop wasm
  Prioritize Wasm's
  Save Machine State
  Shutdown
```

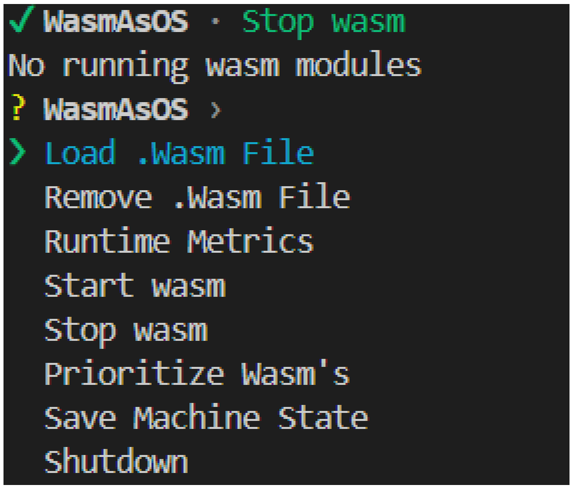
PW0: Pause Wasm menu coming soon

SM1: Stop Wasm Menu: Display a menu of all running wasm files and option to return to the main menu. Function loops displaying updated menu after a wasm is stopped.



```
✓ WasmAsOS · Stop wasm
? Stop a wasm file ›
❯ Return to main menu
  snake.wasm
```

SM2: No wasm files to stop: if no wasm files are found display a message and return to main menu



```
✓ WasmAsOS · Stop wasm
No running wasm modules
? WasmAsOS ›
❯ Load .Wasm File
  Remove .Wasm File
  Runtime Metrics
  Start wasm
  Stop wasm
  Prioritize Wasm's
  Save Machine State
  Shutdown
```

PW0: Prioritize Wasm’s (Scheduler not functional)

S0: Save Machine State (Not required by our client but may be a neat feature if there's time)

Design Summary

Our design focuses on building a secure and modular WebAssembly runtime using Rust for the backend and React for the web interface. The runtime acts as the core execution environment, responsible for parsing, validating, and executing .wasm modules within strict sandbox limits. It manages tasks, memory, and system calls through a custom ABI to ensure safe and isolated execution.

A lightweight command-line interface is implemented first for simplicity and testing, while the web UI (developed in React) provides a more user-friendly dashboard. The dashboard allows users to upload, start, pause, and stop modules, and view real-time runtime metrics such as memory usage, operation speed, and active tasks.

The system's structure includes:

- *A Rust core library handling the execution engine, sandbox environment, and task scheduling.*
- *An optional Node.js/Go orchestrator layer that bridges the Rust core with the web frontend.*
- *A React-based dashboard that serves as the main point of interaction for users.*

Together, these layers form a secure, extensible platform that simulates a lightweight, sandboxed operating system capable of running untrusted WASM code safely and efficiently.

Wasm Runtime

The wasm runtime functions by filling a stack data structure with a series of operational codes(opcode) which are then executed. These opcodes range from allocating memory to simple addition and are what drive the virtual machine.

Virtual Machine Opcode Execution

1. *Last/most recently added Opcode is popped(removed) from the stack.*
2. *Opcode is matched with a set of instructions paired to its name*
3. *Opcode instructions are executed, Instructions may*
    a. *Allocate/deallocate memory*
    b. *Store and load variables*
    c. *Perform arithmetic operations*
    d. *Control the flow of code execution*
        i. *Loop: Execute instructions repeatedly until a condition is met*
        ii. *If: A statement that executes if conditions are met*
        iii. *Break: End execution of a statement or loop*
    e. *Call functions*
    f. *Push or pop to stack*
    g. *End virtual machine execution*
4. *Repeat until stack is empty or terminary opcode is reached*

Design Rationale

- What language should be used for our backend(WebAssembly Engine): Rust/Go => Rust
    - Rust: Memory safe and efficient, large number of resources available for learning but difficult to learn, useful cargo containers for our project, existing webassembly engines to draw inspiration from.
    - Go: Easy to learn, less resources to learn from, less libraries for webassembly, built in garbage collector adds overhead, Memory safe.
- Command line or web interface: => Implement both, cmd interface first Web UI would be nice but our client thinks a command line interface is acceptable.
    - Command line: Simple to implement, would only require us to code in Rust, Not very impressive, our client is ok with this
    - Web UI: Can better display data, more resource consumptive, easier to use, shows extra effort.
- What language should be used for the web interface =>Javascript
    - Rust: Has packages for ui use, would allow us to code the project solely in Rust but may not be the most well suited for our UI since it's more of a backend language and lacks features useful for UI development.
    - Go: Would allow us to study multiple languages but could complicate the development process by increasing the learning load. Also more of a backend language which would make the UI harder to implement.
    - React: Good choice for our web UI, made for developing web UI’s. Would work well as the frontend to our Rust backend.
    - Javascript: Great choice for our web ui, abundant information available on the language.
- CLI interface menu design => Dialoguer_rs
    - Custom made: More flexible, would allow us to create a menu designed just for our product. More time costly
    - Menu_rs: Good crate for creating simple menus, would speed up production but the menu may not be very sophisticated.
    - Dialoguer_rs: Great choice for our project, makes semi complex menus which can use a custom theme. Compatible with other crates to make menus better.
- Rust
    - Add a wasm file: => Implement add by path now replace later with wasm search
        - Add by path: User will have to manually enter wasm file paths to add them to the program. Not very convenient but easy to implement
        - Search for wasm files: Recursively search through user directories to find wasm files, Will be slow if the user has many folders and files but much more practical and convenient.

- Add wasm file: display options to add by path, browse wasms or return to the main menu. Recursively call the menu function after the user selects a choice and adds a wasm module.
- Delete wasm file:
    - Display list of wasm files to delete and return to menu. After the user selects a file, stop the file if it is running and delete from wasm list, recursively call delete wasm file function.
- Start wasm file:
    - Display list of nonrunning wasm files and return to menu option. Start wasm file selected by user and recursively call start wasm file function
    - Changed recursion to a loop so the program doesn't search for nonrunning wasm files additional times
- Halt wasm file:
    - Display list of running wasm files and option to return to menu. Halt user selected wasm file and recursively call function again to display additional files for the user to halt
    - Changed to a loop so the program doesn't use extra resources in the event the user wants to halt multiple files.
- Pause wasm file:
    - Will be implemented very similarly to halt wasm file but will not actually stop the module
- Runtime Statistics
    - Will display memory usage and other data. Is there any other data we should display?
- Wasm Engine:
    - Check for wasm file magic numbers and version so the program doesn't attempt to run an invalid file.
    - Send byte array collected from wasm file to a parser to write into memory
    - Wasm files don't support imports like normal functions but we need them. Should we store a list of known/common functions in a preloaded table?
    - How should imports be handled currently? If a wasm needs them it will throw an exception? Exception handling needs to be implemented asap.
    - What order should opcodes be implemented, there are around 138? Some opcodes have more common use than others? Focus on the most important/common ones for now, loops, arithmetic, etc.
    - There are many possible exceptions when parsing and running a wasm module; how should they be handled? Rust has great exception handling; an exception return policy can be defined at each function which would automatically handle this problem.

        - Not all wasm files will display data the user should be informed about files that don’t and a log should be created for each stack call if the user wants to see how the program runs.
        - Small wasm modules run almost instantly. Should we even spawn a thread or process for these modules?
        - Matchblock for opcodes
            - Functions or match cases: Currently our runtime uses match block cases for running opcodes from the stack but this has proven difficult to test. Instead of having code in the match cases we should implement a function for each match case which should make testing much easier. It will also make the call_indirect opcode much easier since we can use the call opcode directly from there.
    - Scheduler
        - Allow user to specify which module gets how much of the resources allocated
        - Use a thread for each module?
        - Use a process for each module?
        - How should resources be divided among modules? We need to give this option to the user but what would be the best way? Percentage of cpu power and memory? A slider for each? Actual gigabytes and megabyte input?