Product Requirements

WASM as OS

Team CTRL 5

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

1.Brief problem statement

The aim of the WASM as OS project is to develop a web based sandbox environment to test suspicious code. With the completion of this project users will be able to test code without installing a virtual machine, saving storage on their device. Our project will be developed without the use of existing .wasm engines and instead we will be developing our own environment to better suit our clients needs. The proprietary wasm environment will be accessible through a web UI and will boast features such as scheduling, multitasking, and reportable runtime metrics.

2.System requirements

- The project shall use Go for its concurrency model and high-level orchestration components.
- The project shall use Rust for performance-critical and security-sensitive components, such as the WASM execution engine and sandbox implementation.
- The runtime shall provide both a Command-Line Interface (CLI) for automation and a Web-based User Interface (UI) for visualization and manual control.
- The project shall be a custom implementation from scratch, without using external WASM engines like wasmtime or wasmer.
- The system shall be compatible with modern Linux, macOS, and Windows operating systems.

3.Users profile

The platform is designed for technically savvy users who need a protected and isolated environment to run and analyze untrusted or test WebAssembly code. Target users can be classified as follows:

Cybersecurity Researchers and Analysts: They are the primary audience. They are the people who are tasked with examining potentially malicious code. They require a sandboxing environment where they could execute suspicious WASM modules without affecting their local machine. They must be strong in system security, possess malware analysis knowledge, and have knowledge of low-level code execution. They must be searching for the way the code executes, the way its system calls interact (F2), and the way its resources are being utilized (F6).

Software Developers: This group includes developers writing applications using WebAssembly or incorporating third-party WASM modules into their application. They will use the system to run their code in a deterministic, resource-constrained environment that emulates a bare OS. These individuals must be able to write in languages for WASM (e.g., Rust, C/C++, or Go) and have knowledge of software development and debugging (F7).

Students and Teachers: At educational institutions, the system is a valuable learning tool for learning about operating system concepts, sandboxing, and secure computing. The users need to have a university-level of computer science knowledge.

In general, all users need to have a high level of technical reading, be familiar with developer tools or command-line interfaces (F5), and be familiar with some programming and systems architecture.

4.List of Features

F1. WASM Execution Engine
Ability to load and run WebAssembly modules by parsing bytecode and managing linear memory safely.

F2. Custom ABI and Syscalls
A minimal set of built-in functions (e.g., logging, sensor access, sending alerts) that modules can call, controlled through a custom ABI.

F3. Sandboxing and Isolation
Strong restrictions on what modules can access. No direct file system, network, or OS calls unless specifically allowed. Resource limits such as memory caps and syscall quotas are enforced.

F4. Multi-Task Scheduling
Support for running more than one WASM module at a time, with simple scheduling (cooperative or round-robin) to share resources fairly.

F5. Runtime Management Interface
A command-line or web interface that lets the user load, start, pause, and stop modules.

F6. Runtime Metrics
Tracking and reporting of execution statistics such as operations per second, memory use, time spent, and number of syscalls.

F7. Debugging Support
A lightweight debugger that allows stepping through instructions and tracing system calls.

F8. Static Analysis Tool
A scanner that checks a WASM file's imports and exports before it runs, warning about risky or unexpected behaviors.

5.Functional requirements (user stories)

No.	User Story Name	Description	Priority
R1	Task Scheduling	The system will feature a scheduler to assign cpu resources to WASM modules based on need	1
R2	Runtime Report	The system shall report runtime information to the user such as memory usage	1
R3	Sandboxed ABI	The system shall provide a custom, capability-based Application Binary Interface (ABI) that allows WASM tasks to interact with the host system only through predefined functions (e.g., read_sensor, log, send_alert).	
R4	Run multiple nodes	The system should allow the user to load and run more than one WASM module at the same time. When this option is used, each module will run in its own sandbox, and the scheduler will handle switching between them. This makes it possible for users to test or compare different modules without restarting the runtime.	1
R5	Metrics Reporting	The system shall provide a custom, capability-based Application Binary Interface (ABI) that allows WASM tasks to interact with the host system only through predefined functions (e.g., read_sensor, log, send_alert).	2
R6	Runtime Control	The system shall provide a custom, capability-based Application Binary Interface (ABI) that allows WASM tasks to interact with the host system only through predefined functions (e.g., read_sensor, log, send_alert).	2

Use Case Description

Use Case Number: UC-01
Use Case Name: Load WASM Module

Overview: A user uploads or points the runtime at a WebAssembly module; the system validates, stores, and prepares the module for execution (including static analysis).
Actor(s): Researcher/Analyst, Developer, Student
Pre-condition(s): User is authenticated and has permission to upload modules; runtime service is running.
Main (success) Flow:

1. User selects “Load Module” in the Web UI or invokes `cli load <path/URL>`.
2. The system receives the WASM binary and computes a unique module ID and checksum.
3. The system runs a static analysis scanner on the module (imports/exports, suspicious syscalls).
4. If the scanner returns pass/warnings, the system stores the module in the sandbox registry and presents the results.
5. Users may tag the module with metadata (description, tags, capability requirements).
6. The system marks the module as *Ready* for execution.
   Alternate Flows:

- If static analysis finds severe issues, the system rejects the module and returns a detailed report.
- If upload fails (network/size), the system returns an error and partial cleanup.
  Post Condition: Module is stored in registry with status Ready or Rejected; static analysis report saved.

Use Case Number: UC-02
Use Case Name: Execute (Run) WASM Module
Overview: Start execution of a loaded WASM module inside a sandboxed VM instance.
Actor(s): Researcher/Analyst, Developer
Pre-condition(s): Module is in Ready state; scheduler and execution engine are healthy; user has execute permission.
Main (success) Flow:

1. The user selects a Ready module and clicks “Run” (UI) or `cli run <module-id> [flags]`.
2. The system creates a sandbox instance with configured resource limits (memory cap, syscall quota, time slice).
3. Scheduler assigns CPU/time-slice and dispatches the module to the Rust WASM engine.

4. The engine maps linear memory, initializes modules, and begins instruction execution.
5. Runtime collects metrics and streams logs to the UI/CLI.
6. On completion, the sandbox returns exit status and final metrics to the user.
   Alternate Flows:

- If the module exceeds resource limits, the runtime terminates execution and flags a violation.
- If the module attempts disallowed syscalls, the kernel trap handler blocks the operation and logs an alert.
  Post Condition: Execution results, logs, and metrics are stored in the run history.

Use Case Number: UC-03
Use Case Name: Pause / Resume / Stop Module
Overview: Allow the user to control a running module (pause, resume, force-stop). Useful for investigation and resource control.
Actor(s): Researcher/Analyst, Admin
Pre-condition(s): A module instance is running and the user has control permissions.
Main (success) Flow:

1. User selects the running instance and clicks Pause (or `cli pause <instance-id>`).
2. Scheduler saves the module execution context (registers, memory snapshot if required) and stops giving it CPU.
3. User can inspect metrics/logs while paused.
4. User issues Resume; scheduler re-queues instance and execution continues.
5. User may issue Stop to terminate the instance and persist a final report.
   Alternate Flows:

- If saving state fails, the system warns and may still force-stop.
- Admin may forcibly stop a runaway module even if the user does not request it.
  Post Condition: Instance is paused/resumed/stopped; appropriate state and logs are stored.

Use Case Number: UC-04
Use Case Name: Multi-Task Scheduling / Run Multiple Nodes
Overview: Load and run multiple WASM modules concurrently; scheduler enforces fairness and isolation.
Actor(s): Researcher/Analyst, Developer, Admin
Pre-condition(s): Multiple modules are Ready; scheduler is active; system

resources available.
Main (success) Flow:

1. The user submits multiple modules for concurrent execution or starts a batch.
2. Scheduler creates separate sandbox instances for each module.
3. Scheduler applies configured policy (round-robin, cooperative, priority-based) to allot CPU slices.
4. Each instance runs in isolation; runtime aggregates per-instance metrics.
5. System enforces cross-instance resource limits and preemptively throttles if needed.
   Alternate Flows:

- If resources are insufficient, scheduler queues instances and reports ETA.
- If a module misbehaves, scheduler isolates and optionally migrates resources to healthy modules.
  Post Condition: All instances either complete, are queued, or are terminated; scheduling logs and metrics saved.

Use Case Number: UC-05
Use Case Name: View Runtime Metrics & Reports
Overview: User inspects live and historical runtime metrics (ops/sec, memory use, syscall counts) and retrieves execution reports.
Actor(s): Researcher/Analyst, Developer, Student
Pre-condition(s): Module runs or has run; metrics subsystem is enabled.
Main (success) Flow:

1. The user opens the Metrics dashboard for a selected instance or module.
2. The system displays live metrics and historical charts (or table for CLI).
3. Users can filter by time range, metric types, or instance ID.
4. Users can export a report (JSON/CSV/PDF) with metrics and static-analysis summary.
   Alternate Flows:

- If metric collection was disabled for instance, the system returns only partial data.
- If export fails due to size, the system suggests smaller ranges or async export (with caveat: you cannot promise async — present immediate alternatives like chunked export).
  Post Condition: Metrics are displayed/exported; users may save/export reports.

Use Case Number: UC-06
Use Case Name: Debug Module (Step / Trace)

Overview: Provide lightweight debugging: step through instructions, set breakpoints, and trace syscalls.
Actor(s): Researcher/Analyst, Developer
Pre-condition(s): Module is running or paused; debug capability is allowed; module compiled with debug info (if needed).
Main (success) Flow:

1. The user attaches the debugger through UI or `cli debug <instance-id>`.
2. Users set breakpoints or requests single-step.
3. Runtime halts at breakpoint and presents current instruction pointer, local memory state, and recent syscalls.
4. User inspects state, resumes, steps over/into functions, or rewinds if snapshot supported.
5. Debug session ends and a debug trace gets stored.
   Alternate Flows:

- If stepping is disabled by policy, system denies and logs the attempt.
- If breakpoints produce high overhead, the system warns about performance impact.
  Post Condition: Debug trace and session metadata saved (if user opted to save).

Use Case Number: UC-07
Use Case Name: Static Analysis Scan (Pre-Run)
Overview: Analyze a WASM binary for imports/exports, suspicious patterns, and capability requirements before executing.
Actor(s): Researcher/Analyst, Developer, Admin
Pre-condition(s): Module uploaded or referenced.
Main (success) Flow:

1. User triggers “Static Scan” in UI or `cli scan <module-id>`.
2. Analyzer parses WASM sections, lists imports/exports, identifies syscall-like imports, and checks for known suspicious byte patterns.
3. The system produces a graded report (OK / Warning / Severe) with recommended capability constraints.
4. The user reviews the report and either approves or rejects running the module.
   Alternate Flows:

- If the analyzer cannot parse the module (corrupt binary), it returns a parse error.
  Post Condition: Static analysis report stored with module metadata; run permission may be gated on severity.

Use Case Number: UC-08
Use Case Name: Manage ABI Capabilities & Policy
Overview: Admin configures which ABI functions (capabilities) are available to modules and maps capability tokens to modules.
Actor(s): Admin, Researcher (with elevated rights)
Pre-condition(s): Admin authenticated; policy engine running.
Main (success) Flow:

1. Admin opens Capability Manager (UI/CLI).
2. Admin defines or edits capabilities and associated quotas/constraints.
3. Admin assigns a capability profile to a module or a module tag.
4. System enforces these capabilities at runtime; any attempt outside granted capabilities is trapped and logged.
   Alternate Flows:

- If capability assignment conflicts with security policies, the system rejects and suggests safe defaults.
  Post Condition: Capability profiles updated and enforced on subsequent runs.

Use Case Number: UC-09
Use Case Name: Save / Restore Machine State (Snapshot)
Overview: Create and restore full or partial sandbox snapshots for reproducible testing and rollback.
Actor(s): Researcher/Analyst, Admin
Pre-condition(s): Runtime supports snapshotting; user has snapshot permissions.
Main (success) Flow:

1. While the instance is paused or at a safe checkpoint, the user requests “Save snapshot.”
2. The system serializes module memory, register state, and selected runtime metadata into a snapshot artifact.
3. Snapshots can be named/tagged and stored in the snapshot registry.
4. Users can restore a snapshot to a new sandbox instance via UI or cli restore <snapshot-id>
   Alternate Flows:

- If snapshot storage is full, the system denies and suggests cleanup or partial snapshot options.
- If the module uses non-serializable host resources, the system warns and provides a partial snapshot option.
  Post Condition: Snapshot created or restoration performed; snapshot metadata available.

Use Case Number: UC-10
Use Case Name: Authentication, Authorization & Audit Logging
Overview: Manage user login, role-based access control (RBAC), and maintain audit logs of actions (load, run, kill, change policy).
Actor(s): User (Researcher/Dev/Student), Admin
Pre-condition(s): Authentication backend configured (LDAP/OAuth/local); roles defined.
Main (success) Flow:

1. User authenticates via Web UI or CLI (OAuth / username-password / EUID mapping).
2. The system validates credentials and issues session tokens.
3. Based on role, system enables/disables UI/CLI commands (e.g., only Admin sees Capability Manager).
4. All privileged actions are logged to the audit store with timestamps, actor ID, and parameters.
   Alternate Flows:

- If authentication fails, the system denies access and logs attempts.
- If a token expires mid-session, the system prompts re-authentication.
  Post Condition: User session established with appropriate permissions; audit logs updated for actions taken.

6.Non-Functional Requirements

NF1: Tested code shall not read user data

NF2: Tested code shall not escape the sandbox environment

NF3: Test code shall not Access the internet

NF4: The WebAssembly runtime shall provide crash reports

Usability

NF5: The web UI shall feature Colorblindness features

NF6: The web UI will feature text to speech

NF7: The WebAssembly runtime shall allow users to save runtime data

NF8: The WebAssembly runtime shall allow users to save machine states

NF9: The WebAssembly runtime shall be installable for offline use

Cross-Platform Compatibility

NF10: WASM as OS shall function on Windows and Linux OS

NF11: The Web UI shall function on common browsers

Accuracy

NF12: The WebAssembly runtime shall report accurate runtime data

Sponsor Requirements:

I have read and approve the material in this document. If there is no external sponsor, the TA or instructor will sign it for accuracy/scope.

_____	_____	_____
Print Name	Signature	Date