WebAssembly Ecosystem

WASM, WASI, and WASIX

Technical Presentation

Modern Web Technologies

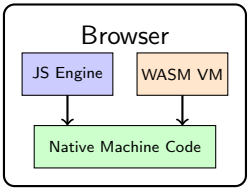
July 9, 2025

Outline

1. Introduction to WebAssembly (WASM)
2. WASI - WebAssembly System Interface
3. WASIX - Extended WASI
4. Architecture and Design Principles
5. Use Cases and Applications
6. Performance Characteristics
7. Future Developments

What is WebAssembly?

- Binary instruction format for a stack-based virtual machine
- Designed as a portable compilation target
- W3C standard since 2019
- Runs in modern web browsers
- Near-native performance



Key Features of WebAssembly

- **Fast**: Near-native execution speed
- **Safe**: Memory-safe, sandboxed execution environment
- **Open**: Open web standard, platform-independent
- **Portable**: Single .wasm file runs anywhere
- **Compact**: Binary format, smaller than JavaScript
- **Polyglot**: Compile from C/C++, Rust, Go, etc.

WebAssembly Text Format (WAT)

```
(module
  (func $add (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add)
  (export "add" (func $add))
)
```

Compiles to binary format:

```
00 61 73 6d 01 00 00 00 01 07 01 60 02 7f 7f 01
7f 03 02 01 00 07 07 01 03 61 64 64 00 00 0a 09
01 07 00 20 00 20 01 6a 0b
```

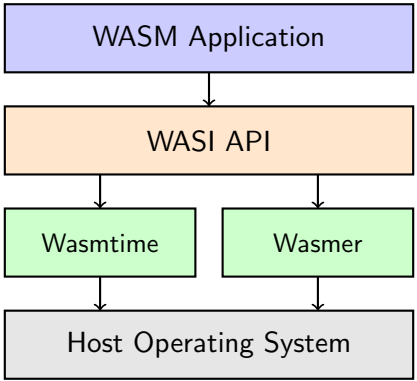
What is WASI?

Definition

WebAssembly System Interface (WASI) is a modular system interface for WebAssembly that enables WASM modules to interact with the operating system.

- Standardized API for system calls
- Platform-agnostic interface
- Capability-based security model
- Write once, run anywhere (beyond browsers)

WASI Architecture



WASI Capabilities

Core WASI APIs:

- File system access
- Environment variables
- Clock/Time functions
- Random number generation
- Process exit codes
- Standard I/O streams

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // WASI file system
    FILE *f = fopen("data.txt", "r");

    // WASI environment
    char *path = getenv("PATH");

    // WASI random
    int r = rand();

    return 0;
}
```

What is WASIX?

WASIX Definition

WASIX is a superset of WASI that extends the standard with additional POSIX-compatible system calls, enabling more complex applications to run in WebAssembly.

Key additions:

- Full POSIX threading support
- Network sockets (TCP/UDP)
- Process forking and execution
- Shared memory
- Signal handling
- Extended file system operations

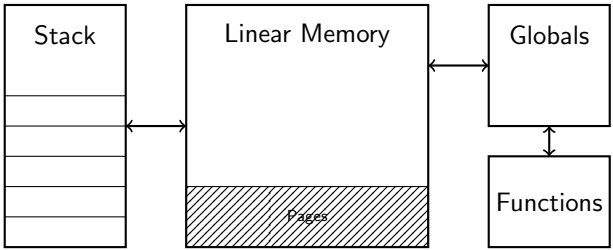
WASI vs WASIX Comparison

Feature	WASI	WASIX
File I/O	✓	✓
Environment Variables	✓	✓
Random Numbers	✓	✓
Clocks/Time	✓	✓
Threading	Limited	✓Full
Networking	×	✓
Fork/Exec	×	✓
Signals	×	✓
Shared Memory	×	✓
Futex	×	✓

WASIX Example: Threading

```c
#include <pthread.h>
#include <stdio.h>

void* worker(void* arg) {
    int id = *(int*)arg;
    printf("Thread %d running\n", id);
    return NULL;
}

int main() {
    pthread_t threads[4];
    int thread_ids[4];

    for (int i = 0; i < 4; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL,
                       worker, &thread_ids[i]);
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

WebAssembly Architecture



Key Components:

- Stack-based virtual machine
- Linear memory model (pages of 64KB)
- Global variables
- Function imports/exports

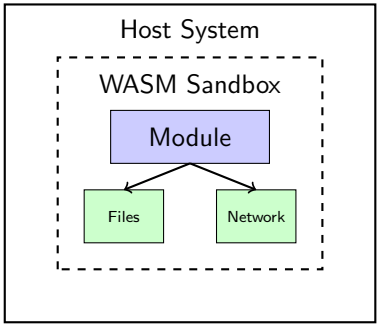
Security Model

Sandboxing

- Memory isolation
- No direct system calls
- Capability-based access
- Explicit imports/exports

WASI Capabilities

- File descriptors
- Directory handles
- Network sockets (WASIX)
- Granular permissions



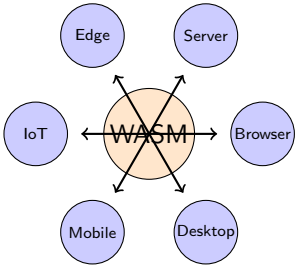
WebAssembly Use Cases

Browser Applications

- Games and graphics
- Video/audio processing
- CAD applications
- Scientific computing
- Cryptocurrency wallets

Server-Side Applications

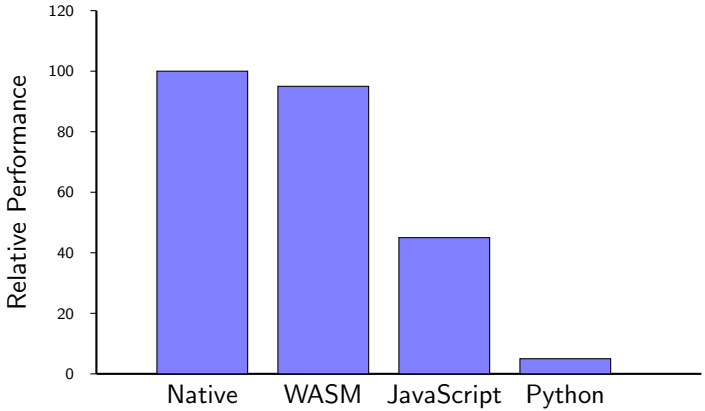
- Edge computing
- Serverless functions
- Plugin systems
- Embedded systems
- Blockchain smart contracts



Real-World Examples

- **Figma**: Design tool running in browser via WASM
- **AutoCAD**: Web version powered by WebAssembly
- **Cloudflare Workers**: Serverless computing with WASM
- **Fastly Compute@Edge**: Edge computing platform
- **Docker Desktop**: Uses WASM for extensions
- **Krustlet**: Kubernetes kubelet for WASM workloads
- **Wasmer**: Universal WASM runtime with WASIX

Performance Comparison



*Benchmark: Computational intensive tasks (approximate values)

Performance Factors

Advantages

- Ahead-of-time compilation
- Predictable performance
- No garbage collection pauses
- Efficient memory usage
- SIMD instructions support

Considerations

- Startup overhead
- Memory copy costs
- JavaScript interop overhead
- Limited threading (WASI)
- Module size

WebAssembly Roadmap

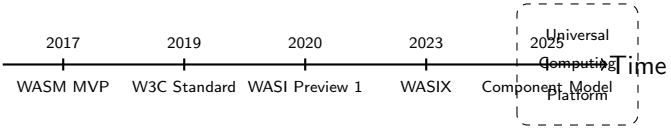
Current Proposals

- **Component Model**: Composable WASM modules
- **Interface Types**: Better language interop
- **GC Support**: Garbage collected languages
- **Exception Handling**: Native exception support
- **Tail Calls**: Functional programming optimization

WASI Evolution

- WASI Preview 2: Component model integration
- Standardized networking APIs
- GPU access proposals
- Improved async/await support

Future Vision



Vision: Write once, run anywhere

- Universal application runtime
- Language-agnostic platform
- Seamless cloud-to-edge deployment
- Native performance everywhere

Conclusion

- **WASM**: Efficient, portable bytecode format
- **WASI**: System interface for non-browser environments
- **WASIX**: Extended POSIX compatibility
- Growing ecosystem with broad industry support
- Promising future for universal computing

Thank You!

Questions?

Resources

- **WebAssembly.org**: `https://webassembly.org/`
- **WASI.dev**: `https://wasi.dev/`
- **Wasmer.io**: `https://wasmer.io/`
- **MDN WebAssembly**: `https://developer.mozilla.org/en-US/docs/WebAssembly`
- **WASM Weekly**: Newsletter for WebAssembly updates
- **Awesome WebAssembly**: Curated list of resources