← All articles Architecture

WebAssembly on the Server: Running Wasm Workloads Alongside Containers

WebAssembly is not just for browsers. Spin, Wasmtime, and WasmEdge run server workloads with sub-millisecond cold starts, 1MB footprints, and sandboxing that containers cannot match.

Y Yash Pritwani 24 March 2026 11 min read read

The serverless promise has always been compelling — deploy code, pay only for what you use, never think about infrastructure. But traditional serverless platforms carry a dirty secret: cold starts. A Node.js Lambda function spinning up from cold can take 200-400ms before your code even begins executing. WebAssembly on the server is changing that equation entirely, delivering cold starts measured in microseconds, not milliseconds, with a security sandbox that makes container isolation look permissive by comparison.

Why WebAssembly Belongs on the Server

Most developers know WebAssembly as a browser technology — a way to run compiled Rust or C++ in a web tab at near-native speed. The server-side story is different and arguably more interesting.

The Cold Start Problem

Consider what happens when a container-based serverless function receives its first request. The runtime must pull or restore a container image, initialize the OS-level process, load the language runtime, and only then begin executing user code. A WebAssembly module sidesteps nearly all of that. Wasmtime achieves cold start times of 1-5 microseconds for typical modules.

Footprint and Density

A minimal Rust HTTP handler compiled to Wasm might produce a binary under 200 kilobytes. Com… that to a Docker image for a comparable Go service — typically 5-15 megabytes even after multi-… build optimization.

The Sandbox Model

WebAssembly uses a capability-based security model. A Wasm module cannot access the filesystem, network, environment variables, or system clocks unless the host runtime explicitly grants those capabilities. With Wasm, the sandbox is the default.

The WebAssembly System Interface (WASI)

For Wasm to do useful work on the server, it needs access to system resources. WASI provides a standardized, capability-gated API for filesystem access, network sockets, clocks, and more. WASI 0.2 introduces the Component Model — a richer type system and composition mechanism that allows Wasm modules to be assembled like building blocks with well-defined interfaces.

Spin: The Application Framework for Server-Side Wasm

Fermyon Spin is the most complete framework for building server-side WebAssembly applications.

Installing Spin

```
curl -fsSL https://developer.fermyon.com/downloads/install.sh | bash
sudo mv spin /usr/local/bin/
spin --version
```

Creating a Rust HTTP Handler

```
spin new -t http-rust hello-wasm
cd hello-wasm
```

The generated `src/lib.rs`:

```
use spin_sdk::http::{IntoResponse, Request, Response};
use spin_sdk::http_component;

#[http_component]
fn handle_hello_wasm(req: Request) -> anyhow::Result<impl IntoResponse> {
    println!("Handling request to {:?}", req.header("spin-full-url"));
    Ok(Response::builder()
        .status(200)
        .header("content-type", "text/plain")
        .body("Hello from WebAssembly!")
```

```
        .build())
```


Build and run:

```
spin build
spin up
curl http://localhost:3000/
```

Adding Key-Value Storage

```
use spin_sdk::http::{IntoResponse, Request, Response};
use spin_sdk::http_component;
use spin_sdk::key_value::Store;

#[http_component]
fn handle_counter(req: Request) -> anyhow::Result<impl IntoResponse> {
    let store = Store::open_default()?;
    let count: u32 = store.get("visits")?
        .and_then(|bytes| String::from_utf8(bytes).ok())
        .and_then(|s| s.parse().ok())
        .unwrap_or(0);
    let new_count = count + 1;
    store.set("visits", new_count.to_string().as_bytes())?;
    Ok(Response::builder()
        .status(200)
        .header("content-type", "text/plain")
        .body(format!("Visit number: {}", new_count))
        .build())
}
```

You might also like

- → Edge AI Inference: Why the Cloud Is Too Slow and How to Deploy Models at the Edge
  11 min read
- → Proxmox Clustering: High Availability for Your Self-Hosted Infrastructure
  12 min read read
- → Hardening Your Self-Hosted CI/CD Pipeline Against Supply Chain Attacks
  13 min read read

Wasmtime: The Runtime Underneath

Wasmtime is the WebAssembly runtime maintained by the Bytecode Alliance.

```
curl https://wasmtime.dev/install.sh -sSf | bash
```

```
# Compile and run
rustup target add wasm32-wasi
rustc hello.rs --target wasm32-wasi -o hello.wasm
wasmtime hello.wasm
```

Capability Restriction

```
# This fails -- no filesystem access granted
wasmtime my-module.wasm

# This works -- explicitly grant access to /tmp only
wasmtime --dir /tmp:/tmp my-module.wasm
```

WasmEdge: The Cloud-Native Alternative

WasmEdge ships with extensions for HTTP clients, MySQL, Redis, Kafka, TensorFlow, and OpenCV.

```
curl -sSf https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/utils/install.sh | bash
wasmedge --dir .:. my-app.wasm
```

Running Wasm Alongside Docker Containers

The containerd-wasm-shim

Kubernetes can schedule Wasm pods using the containerd-wasm-shim:


```
apiVersion: v1
kind: Pod
metadata:
  name: wasm-hello
spec:
  runtimeClassName: wasmtime-spin
  containers:
  - name: hello
    image: ghcr.io/deislabs/containerd-wasm-shims/examples/spin-rust-hello:v0.11.1
    resources:
      requests:
        cpu: 10m
        memory: 10Mi
```

Docker and Wasm

```
docker run --runtime=io.containerd.spin.v2 \
  --platform=wasi/wasm \
  ghcr.io/fermyon/spin-hello-world:latest
```

Composing Wasm and Container Workloads

A realistic deployment uses containers for stateful services and Wasm for stateless compute:

```
services:
  api-handler:
    image: myregistry/api-handler:latest
    platform: wasi/wasm
    runtime: io.containerd.spin.v2
    ports:
      - "3000:3000"
  postgres:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: appdb
  redis:
    image: redis:7-alpine
```

Performance Characteristics

Wasm shines at:

- **Event handlers and webhooks** — sub-millisecond cold starts
- **Edge compute** — small binary size for constrained nodes
- **Plugin systems** — capability sandboxing for untrusted code
- **High-concurrency APIs** — lower memory per instance

Less suitable for:

- Long-running compute jobs
- Workloads requiring direct hardware access (GPU)
- Applications dependent on native libraries without Wasm ports

Summary

WebAssembly on the server is not a replacement for containers — it is a complement that fills the gaps containers handle poorly. Spin provides a developer-friendly framework built on Wasmtime. The containerd-wasm-shim makes Wasm a first-class Kubernetes workload. Start with a single event handler

compiled to Wasm and deployed via Spin. Measure the cold start improvement. Run it alongside your existing containers without changing anything else.

#webassembly #wasm #containers #serverless #spin #wasmtime #cloud-native

RELATED SERVICE

Technical Architecture & Consulting

System design, microservices architecture, and technology strategy for ambitious projects.

Get a Consultation | Chat on WhatsApp

Need help with architecture?

TechSaaS provides expert consulting and managed services for cloud infrastructure, DevOps, and AI/ML operations.

Get a Free Consultation | WhatsApp Us

Related Articles

Architecture

Rate Limiting Patterns: Protecting Your APIs...

11 min read

DevOps

eBPF Beyond Security: Networking, Observability,...

12 min read

Architecture

Durable Execution: Why Your Workflows Should...

7 min read

Stay in the Loop

Get product updates, engineering blog posts, and tech insights. No spam, ever.

you@company.com

Subscribe

TE TechSaaS