THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
KOREA

서울

Building Event-Driven WebAssembly on Kubernetes
Runtime, Observability, and Security

Brandon Kang, Principal Technical Solutions Architect
Akamai Technologies

Nam Hai Nguyen, Software Engineer
Hylatek Co.ltd

SPEAKERS

Who is behind this benchmark

SPEAKER 01

Brandon Kang

Akamai Technologies

Cloud-native runtime specialist working on the intersection of **WebAssembly, edge compute, and Kubernetes**. Focus areas: SpinKube adoption, eBPF observability, event-driven platforms.

SPEAKER 02

Nam Hai

Hylatek JSC

Systems engineer building the shared Rust `compute-core` crate and the two-clock dispatcher that powers this benchmark. Focus: reproducible measurement discipline on shared VMs.

AGENDA

Four acts, one live demo

01
THE PROBLEM

Why containers hurt on event-driven workloads

· Cold start tax
· Instance density ceiling
· WebAssembly as a complement, not a replacement

02
THE BENCHMARK

SpinKube vs Knative on real Kubernetes

· Two-clock discipline
· Shared `compute-core` crate
· Live sweep — 6 workloads · 30–50 samples

03
SECURITY

Wasm as a sandbox primitive

· Capability-based isolation
· Reduced attack surface
· Consistent execution across environments

04
ADOPTION

A practical production guide

· When to pick Wasm
· When to stay on containers
· Migration path & operational tips

ACT 01 · THE PROBLEM

Containers were built for long-running services.

Event-driven workloads look nothing like that. Short. Bursty. Idle-heavy.
And they still pay the **container startup tax** on every cold recovery.

THREE SYMPTOMS

01 · COLD START
Container recovery is measured in **hundreds of milliseconds**, not micros.

02 · DENSITY
~64 MB resident per idle container. The RAM ceiling is real.

03 · OVERHEAD
HTTP framing + scheduler latency dominates the actual compute.

WEBASSEMBLY

WebAssembly (Wasm) is a low-level, binary instruction format designed
As a portable compilation target for high-performance code,
Enabling near-native execution speeds on the web and other environments.

ENTER WEBASSEMBLY

A runtime built for events.

Wasm ships a **binary-portable, capability-sandboxed** execution format that starts in microseconds and holds a resident footprint smaller than a container's DNS resolver.

For **event-driven workloads** — HTTP triggers, queue consumers, edge functions — that changes the arithmetic of what "scale to zero" costs.

01 · FAST STARTUP

From cold to serving in ~1 ms

No container image pull. No runtime cold path. Just load a Wasm module into a live process and go.

02 · STRONG ISOLATION

Capability-based, deny by default

Modules see only what the host explicitly grants. No ambient filesystem. No ambient network. WASI as the contract.

03 · PORTABILITY

One binary — every host

Same `.wasm` runs on Kubernetes, on the edge, on a laptop. Deterministic execution across environments.

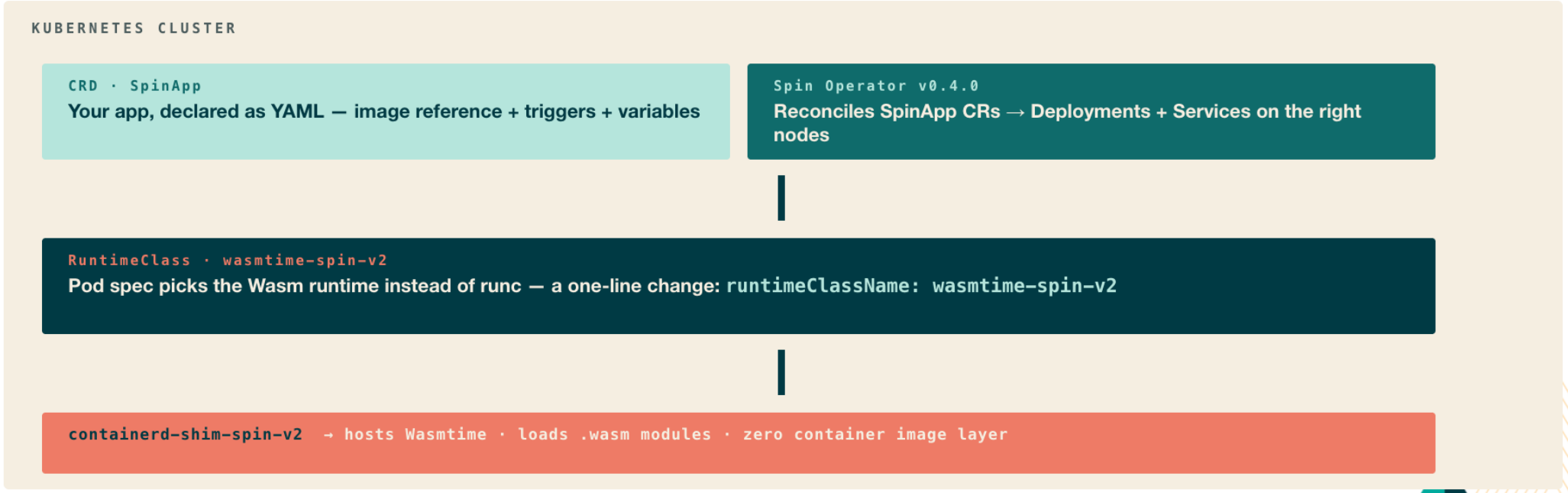
SpinKube

SpinKube is an open source, Kubernetes native project that streamlines developing, deploying, and operating Wasm workloads in Kubernetes
- resulting in delivering smaller, more portable applications with exciting compute performance benefits.

SPINKUBE

Wasm inside your Kubernetes cluster.

SpinKube slides in as a **runtime class** — no sidecar, no proxy, no separate control plane. Your existing kubectl workflow keeps working.



THE CONTENDERS

WaaS vs FaaS — where they differ.

WAAS

SpinKube

`containerd-shim-spin-v2`

P50 **0.9 ms** latency — sub-millisecond floor

RAM **~6 MB** per module resident

DEN **170 instances / GB** RAM

CLD **~18 ms** scale-from-zero

> Stays warm at negligible cost. Pays cold-start once **per node**, not per instance.

FAAS

Knative Serving

`native container + Kourier`

P50 Higher steady-state latency

RAM **~64 MB** per container resident

DEN **16 instances / GB** RAM

CLD **~480 ms** scale-from-zero

> Scales to zero after 30s idle. Full SIMD access — wins on dense compute.

ACT 02 · THE BENCHMARK

No mocks.
No simulated runtimes.
Real Kubernetes.

SpinKube and Knative Serving deployed side-by-side on a single Docker Desktop cluster. **Both runtimes execute the identical Rust** `compute-core` **crate** — only the HTTP wrapper differs.

PRIME DIRECTIVES

§0.1
Single shared codebase across all runtimes

§0.2
Two-clock discipline — exec vs roundtrip, never mixed

§0.3
Fairness — every Wasm win paired with matmul loss

§0.5
Control plane isolated from compute pods

§0.6
Discard any block with pod eviction or OOMKill

MEASUREMENT DISCIPLINE

Two clocks. Never mixed.

DISPATCHER

Rust · Axum

```
▶ start = Instant::now()
  http.post(adapter, event).await
■ roundtrip_ms = elapsed
```

HTTP POST

ADAPTER (in pod)

Spin / Knative handler

```
▶ t0 = Instant::now()
  compute_core::run(wl, n)
■ exec_ms = elapsed
  return { exec_ms, cold, hash }
```

RESPONSE

DERIVED

Platform overhead

```
overhead_ms
  = roundtrip_ms
  - exec_ms
```

*The **only** permitted combination.*

WHY IT MATTERS

Averaging `roundtrip` and `exec` together conflates **platform framing** with **pure compute**. A runtime can look fast because its transport is thin, or its compute is fast — never both dimensions from a single number.

THE CLOCK RULE

Both use `CLOCK_MONOTONIC` via Rust's `std::time::Instant`, timed **inside Linux pods** — never on the Windows host, whose default timer granularity is ~15 ms and would corrupt sub-ms exec measurements.

SHARED CODEBASE INVARIANT

One Rust crate. Six workloads. Two targets.

crate

compute-core

Pure Rust · No I/O · No framework
One function per workload · `fn run(wl, n) → hash`

COMPILES TO

`wasm32-wasi` → SpinKube adapter

`x86_64-linux` → Knative container adapter

FAIRNESS CHECK

Both compile targets must return an **identical** `result_hash` for every workload/N pair — this is how we prove the compute path didn't diverge across runtimes.

THE 6 WORKLOADS

WORKLOAD	BUCKET	N SWEEP	WHAT IT SURFACES
JSON Validate	`Wasm-favor`	`1 – 100 items`	Per-invocation overhead
CRC32 / RLE	`Wasm-favor`	`64 – 4 096 B`	Scalar throughput
Fibonacci (recursive)	`Wasm-favor`	`10 – 30`	Crossover point
Prime Sieve	`Neutral`	`100 K – 5 M`	Linear-memory model
SHA-256 Loop	`Neutral`	`1 K – 100 K`	Pure scalar CPU
Dense Matmul	`Native-favor`	`64 – 320`	SIMD gap · guardrail

*The unfavorable bucket is **retained on purpose** — every Wasm-favorable win is paired with the dense-matmul counterexample. That's the credibility guardrail.*

HOW SAMPLES ARE COLLECTED

Randomized. Serialized. Percentiled.

STEADY-STATE PATH · SERIALIZED

```
for sample in 1..=SAMPLES:      # 30 – 50, default 45

  order = shuffle([wasm, knative])   # anti-drift

  for runtime in order:
    warm_guard(runtime)              # discard 1–2 throwaway calls
    row = measure(runtime, workload, n)
    record(row)                      # INSERT → pg_notify
    settle(50..100 ms)              # cache quiesce
```

Randomized order per sample spreads each runtime's samples evenly across the whole time window — thermal and cache drift hit both runtimes equally instead of confounding one with "*ran while VM was cool.*"

01 · SAMPLE COUNT

30–50 samples per data point

Enough for stable p95/p99. Fewer, and outliers dominate.

02 · REPORT PERCENTILES

p50 / p95 / p99 — never means

A single-VM outlier wrecks averages. Percentiles survive.

03 · TIMEOUT CEILING

~1 s per invocation

One slow cold start can't hang the demo. Timeouts logged as their own category.

04 · SANITY GATE

Null block before every session

Fire a trivial payload — if RTTs diverge, the dispatcher itself is biased. Fix before trusting anything.

LIVE

Full sweep — starting now.

http://wasm.brandonkang.org:9999/

https://youtu.be/gB-yjezNFho

WHAT TO WATCH

▼ **OVERHEAD BAND**

Wasm line stays flat. Knative rises with rate.

▼ **EXEC vs N**

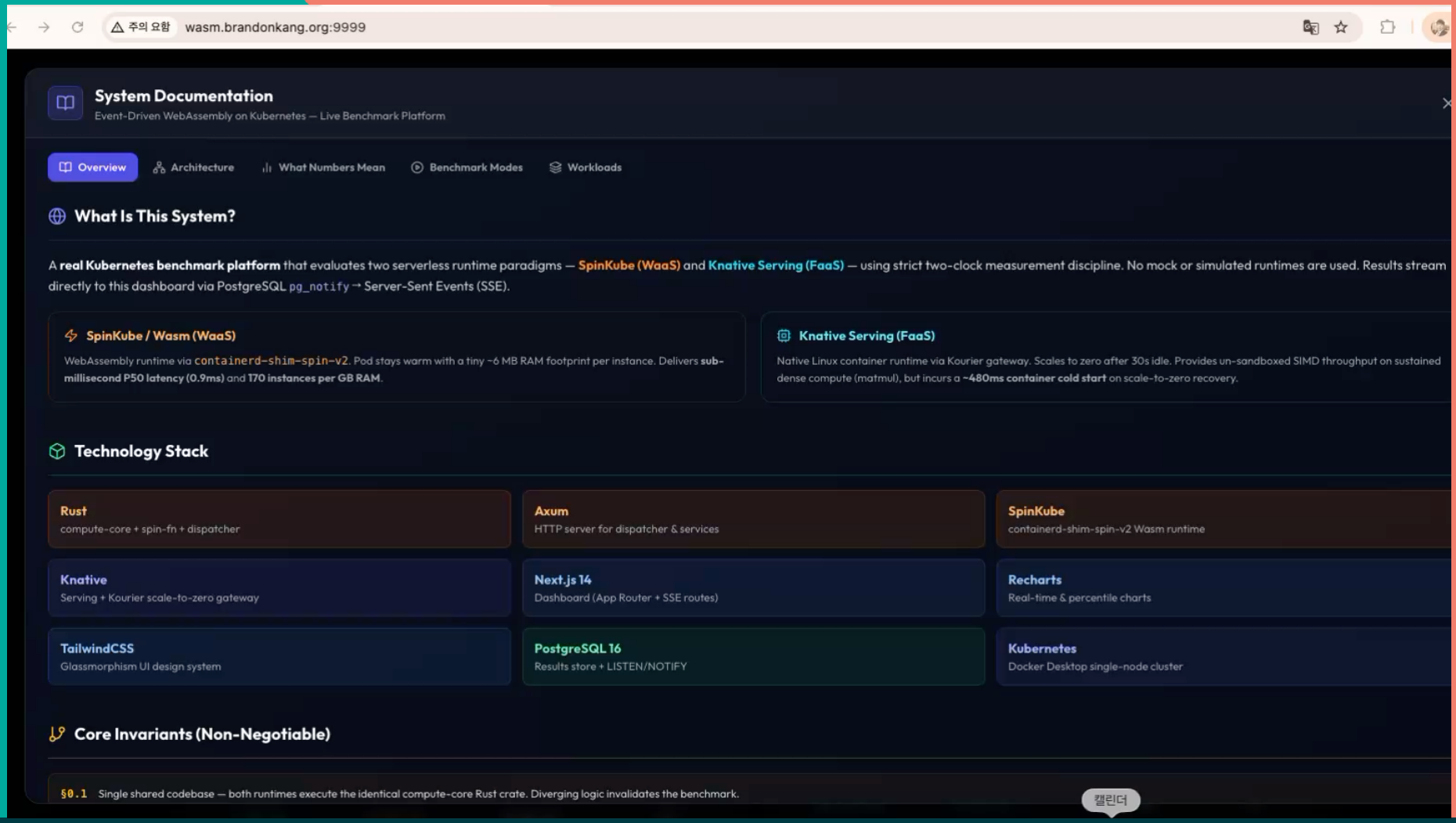
Matmul crossover — Knative overtakes at high N.

▼ **P99 COLD START**

Fired simultaneously after a 35 s idle window.

▼ **SSE STREAM**

Every row lands in `postgres → pg_notify → Next.js` in under 30 ms.


주의 요함
wasm.brandonkang.org:9999
System Documentation
Event-Driven WebAssembly on Kubernetes — Live Benchmark Platform
Overview
Architecture
What Numbers Mean
Benchmark Modes
Workloads
What Is This System?
A real Kubernetes benchmark platform that evaluates two serverless runtime paradigms — SpinKube (WaaS) and Knative Serving (FaaS) — using strict two-clock measurement discipline. No mock or simulated runtimes are used. Results stream directly to this dashboard via PostgreSQL pg_notify → Server-Sent Events (SSE).
SpinKube / Wasm (WaaS)
WebAssembly runtime via containerd-shim-spin-v2. Pod stays warm with a tiny ~6 MB RAM footprint per instance. Delivers sub-millisecond P50 latency (0.9ms) and 170 instances per GB RAM.
Knative Serving (FaaS)
Native Linux container runtime via Kourier gateway. Scales to zero after 30s idle. Provides un-sandboxed SIMD throughput on sustained dense compute (matmul), but incurs a ~480ms container cold start on scale-to-zero recovery.
Technology Stack
Rust
compute-core + spin-fn + dispatcher
Axum
HTTP server for dispatcher & services
SpinKube
containerd-shim-spin-v2 Wasm runtime
Knative
Serving + Kourier scale-to-zero gateway
Next.js 14
Dashboard (App Router + SSE routes)
Recharts
Real-time & percentile charts
TailwindCSS
Glassmorphism UI design system
PostgreSQL 16
Results store + LISTEN/NOTIFY
Kubernetes
Docker Desktop single-node cluster
Core Invariants (Non-Negotiable)
§0.1 Single shared codebase — both runtimes execute the identical compute-core Rust crate. Diverging logic invalidates the benchmark.
캘린더

RESULT 01 · PLATFORM OVERHEAD

A flat overhead band — Wasm's story.

SPINKUBE · OVERHEAD_MS

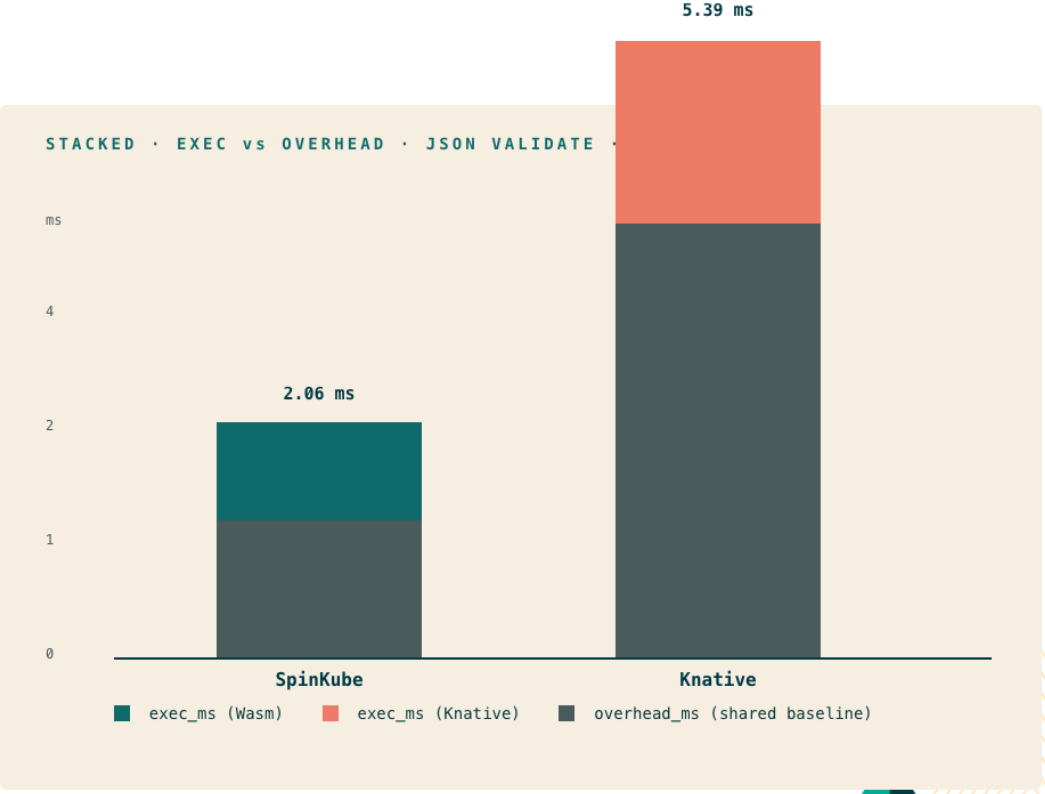
1.2ms

Median platform overhead — the flat baseline that stays flat under load.

VS KNATIVE

3.2× lower

Knative's overhead band sits above and **rises with request rate**. SpinKube's stays flat.



RESULT 02 · COLD START RACE

Both scaled to zero — then fired at once.

t=0 ms

t=500 ms

SPINKUBE / WASM

18 ms
P99 cold start

KNATIVE / CONTAINER

480 ms
Container startup

THE ARITHMETIC

Knative pays cold-start per instance.
SpinKube pays cold-start once per node.

TEST PROTOCOL

Cluster idle for **35 s** → both runtimes scaled to zero → simultaneous `tokio::join!` fire → measured **p99**, never mean.

RESULT 03 · INSTANCE DENSITY

What one GB of RAM can hold.

SPINKUBE · INSTANCES PER GB

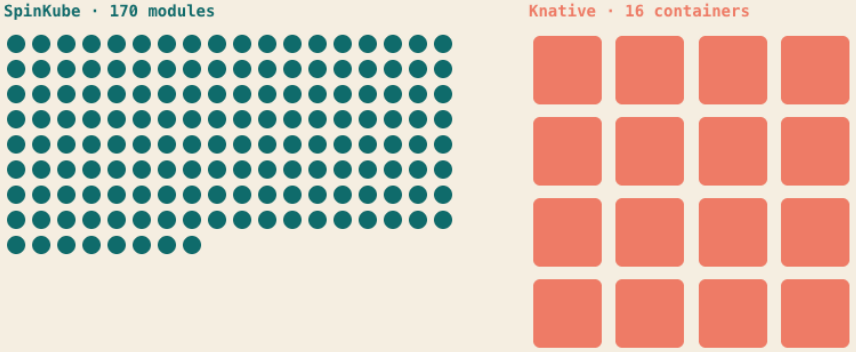
170

~6 MB resident per Wasm module

KNATIVE · INSTANCES PER GB

16 · ~64 MB per container

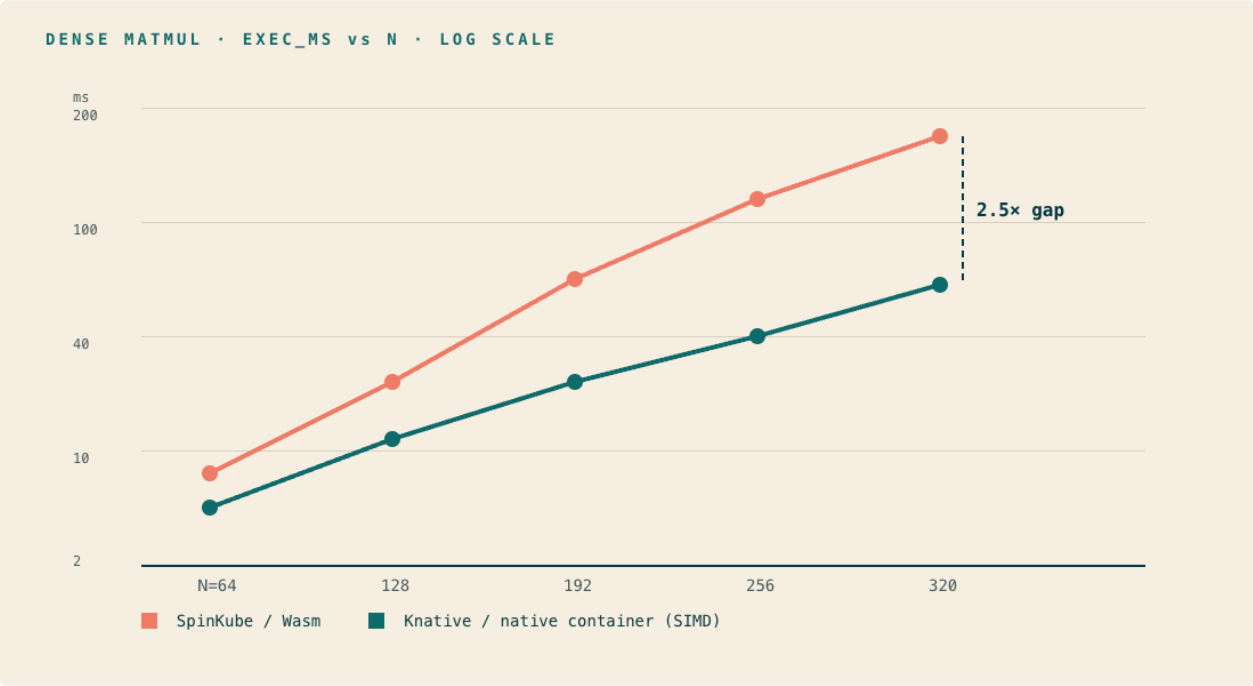
1 GB RAM · VISUAL



~15× instance density on the same GB — architectural, not a config trick.

RESULT 04 · THE CREDIBILITY GUARDRAIL

Where the container wins — and it does.



WHY THE GAP WIDENS

SIMD is the container's home turf

Native containers get full un-sandboxed access to AVX2/AVX–512 vector units. Wasm's SIMD is available but pays sandbox verification cost per lane operation.

On **dense f64 matmul** at N ≥ 128, that gap widens with N — it doesn't close.

THE HONEST FRAMING

"No runtime wins outright. That's the credible — and more interesting — result."

ACT 03 · SECURITY

A smaller attack surface, by construction.

Wasm isn't "a lighter container." It's a different security model — one where every capability is **opt-in, not opt-out**.

01 · CAPABILITY-BASED

Deny by default

No ambient filesystem. No ambient network. WASI hands the module **only the file descriptors and hosts** the operator explicitly grants — the CVE class of "container escaped and read /etc/shadow" doesn't apply.

02 · MEMORY MODEL

Linear memory, bounds-checked

Every load and store is bounds-checked against the module's linear memory. Buffer overflows into host memory are **structurally impossible** — not a runtime configuration, a language guarantee.

03 · IMAGE SUPPLY CHAIN

No base image, no CVE churn

A Wasm module is a **single self-contained artifact** — no glibc, no OpenSSL layer, no distro. The container-image CVE stream (thousands of monthly advisories) collapses to your app dependencies only.

OPERATIONAL EFFICIENCY

Same binary. Every environment.

DIMENSION	SPINKUBE (WASM)	KNATIVE (CONTAINER)
Build artifact	Single `.wasm` file (~1–5 MB)	Multi-layer OCI image (~50–500 MB)
Registry cost	Trivial — bandwidth & storage vanish	Real — base image + app layers + churn
Cross-arch portability	**One binary** — arm64, x86_64, edge, laptop	Per-arch build; multi-arch manifest lists
Cold-start economics	Once per **node** — modules load into a live process	Once per **instance** — full container startup
CVE surface	App dependencies only — no distro layer	App + base image + package feed
K8s integration	RuntimeClass — `runtimeClassName: wasmtime-spin-v2`	Native — no change
Where it wins	Bursty, event-driven, high-density workloads	Sustained dense compute · SIMD-heavy paths

ACT 04 · ADOPTION GUIDE

A practical decision tree.

PICK SPINKUBE · WASM

Bursty. Short-lived. Cold-start-sensitive.

- ✓ HTTP triggers, webhooks, edge functions
- ✓ Queue consumers with idle periods
- ✓ Multi-tenant per-customer isolates
- ✓ High instance density on a fixed RAM budget
- ✓ Scalar compute — JSON, CRC, hash, string

STAY ON CONTAINERS

Sustained. Dense compute. Legacy.

- ✓ SIMD-heavy workloads — ML inference, matmul, codecs
- ✓ Long-running services with steady traffic
- ✓ Complex dependency graphs (JVM, Python + native libs)
- ✓ Queue workers with persistent state → Deployments + KEDA
- ✓ Anything already running well — no forced migration

서울

Thank You!!
감사합니다.

Questions Welcome.

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
KOREA