Khronos: A Type II Exovisor Architecture with WebAssembly

John Abrahams, Ajax Li
University of Pennsylvania

***Abstract*— We present *Khronos*, a runtime that combines WebAssembly's language-level sandbox with exokernel-style capability gating and scheduler-mediated user-level parallelism. We call this combination a *Type II exovisor*: an ordinary host-OS process that multiplexes operating-system resources among mutually untrusted WebAssembly tenants through a small, tiered, capability-gated import surface. Khronos exposes only ∼74 named host entry points to a tenant module, in contrast to the ∼300 syscalls reachable from a Linux container, and runs each invocation in a fresh sandboxed store with fuel-based instruction accounting and epoch-based wall-clock deadlines. To support the compute-heavy workloads typical of regulated-data flows, Khronos additionally provides a multi-tenant-safe data-parallelism path built on the WebAssembly threads proposal: tenant code uses `pthread_create` unchanged, and a custom `wasi:thread-spawn` handler dispatches workers onto a bounded Rayon pool with per-isolate trap containment. Across matrix multiply, k-means, and SVD power-iteration workloads, Khronos achieves up to $6.7\times$ speedup on four cores while remaining ∼0.08 MiB per tenant isolate—an order of magnitude denser than vanilla Wasmtime and roughly $20\times$ denser than Docker. We argue that this combination—WebAssembly sandboxing, exokernel-style capabilities, and shared-memory data parallelism—defines a usable cloud architecture for short, bursty, multi-tenant compute over regulated data.**

I. Introduction

The architecture of multi-tenant cloud compute is a series of compromises between three goals: *isolation strength* (misbehaviour in one tenant does not affect another), *tenant density* (how many independent guests fit on one host), and *compute throughput* (how much real work each guest can do). Containers [9] chose density at the price of isolation, sharing a single Linux kernel with a ∼300-syscall attack surface. Microvirtual machines such as Firecracker [6], [7] chose isolation at the price of density, paying a full guest-kernel image per tenant. Browser-style isolate runtimes such as Cloudflare Workers [8] chose density and isolation at the cost of throughput, ruling out compiled native-style code for languages other than JavaScript.

We argue that WebAssembly [10], [11] now makes a fourth point in this space available. WebAssembly is structurally a sandbox: linear memory is bounds-checked at every load and store, control flow is structured, and host-reachable functionality enters only through declared imports. The host owns the import table. Therefore, the host owns the abstraction surface.

This paper develops that observation into a system. Khronos is a *Type II exovisor*: an ordinary user-mode process—no kernel modules, no virtualization extensions—that exposes a small, tiered, capability-gated set of imports to WebAssembly tenants and runs each invocation in a fresh sandboxed store with quota-bounded resources. We call it a Type II *exovisor* rather than a Type II hypervisor because the surface is not a virtualized machine. Following the exokernel philosophy [1], the host exposes primitive resources rather than fixed operating-system abstractions, and tenants compose those primitives through their own (compiler-emitted) library OS. Following the Type II hypervisor convention, we run as an ordinary process and delegate hardware multiplexing to the host OS.

Khronos additionally treats *data parallelism* as a first-class part of the import surface. We provide a multi-tenant-safe adaptation of the WebAssembly threads proposal: tenants compile their C++ kernels with `-pthread` and `-shared-memory`, `pthread_create` flows through a custom `wasi:thread-spawn` handler, and that handler dispatches workers onto a bounded `Rayon` [19] pool with per-isolate trap containment. The result is that a tenant's single-source C++ kernel compiles to either a sequential or a data-parallel WebAssembly module by build flag, and the parallel build wins by up to $6.7\times$ on dense matrix multiply on four cores while preserving the multi-tenant property: a worker trap in tenant A is invisible to tenant B.

a) Contributions: This paper makes three contributions.

1) **An exovisor design.** We articulate the Type II exovisor as an architectural class distinct from containers, microvirtual machines, and isolate-only runtimes, and we show how WebAssembly's sandbox makes the design economically viable (Sections II–IV).
2) **A tiered, capability-gated import surface.** We classify every host-reachable name into one of five tiers and gate Tier 3 syscalls on per-tenant capabilities checked at module *instantiation* time, before any tenant code runs (Section IV).
3) **A multi-tenant-safe data-parallelism path.** We adapt the WebAssembly threads proposal to a multi-tenant runtime by replacing the upstream `wasi:thread-spawn` handler with one that catches worker traps, scopes them to the offending isolate, and never calls `process::exit` (Section V).

We evaluate Khronos along three axes—isolation cost, multi-tenant throughput, and parallel speedup—in Section VI.

II. Background and Motivation

A. Operating-system architectures

Prior to the dot-com bubble, researchers debated how to improve application speed, flexibility, extensibility, and fault-tolerance through different operating-system architectures. The original monolithic kernels provided every abstraction in kernel space: scheduling, file systems, networking, device drivers, memory management, paging, and more. Implementing all of these in the kernel came with tradeoffs. A faulty subsystem could crash the entire machine; modularity and separation of powers were essentially impossible; the resulting system surface introduced unanticipated security vulnerabilities.

Microkernels were proposed to address these issues by moving most abstractions into user space, preserving modularity and enabling fault-tolerant designs. The price was severe inter-process communication (IPC) overhead, which hindered adoption on general purpose computers. The L4 family of μ-kernels eventually solved the IPC problem and became standard in embedded, IoT, and high-security systems.

Exokernels [1] took a different posture: rather than fixing a particular set of abstractions, the kernel should *export hardware resources* through a low-level interface and let untrusted library operating systems implement whatever abstractions a particular workload needs. The argument was twofold: the right abstraction is workload-specific, and the kernel-fixed abstraction is rarely the right one. Closely related, the SPIN [2] system used type-safe Modula-3 to allow user-supplied extensions to run safely inside the kernel. Both ideas were intellectually compelling but rarely realised in deployed systems, because asking application developers to ship a library OS or kernel extension is a tall order.

The contemporary descendants of these ideas are eBPF [4] (user-supplied verified policy code in the Linux kernel) and language-level safety projects (e.g. Rust-based operating systems), but the general posture—a small, primitive-only substrate above which applications choose their own abstractions—never reached the cloud.

B. Cloud-compute architectures

If one squints, every major cloud-compute platform reflects an operating-system philosophy. Xen and EC2 are monolithic in spirit: each tenant gets a full OS image. Firecracker and AWS Lambda are microkernel-shaped: a minimal hypervisor multiplexes lightweight guests [6]. Cloudflare Workers run V8 isolates in-process, much like SPIN's idea of safe extensions inside a shared substrate [8]. Dune [5] let processes treat virtualization extensions as a primitive, an exokernel-shaped move that gave applications direct access to ring-zero hardware features in concert with the host OS.

A concrete instance of this design space is multi-tenant compute over regulated data: third-party applications run on a shared substrate that brokers access to a sensitive backing source (financial, healthcare, or other compliance-bound data) and must guarantee that raw credentials, sockets, and files never leave the host. The constraints become concrete:

- *Multi-tenant by construction.* Each third-party app is its own tenant. A bug, OOM, or trap in tenant A must not affect tenant B.
- *Density per host.* The unit economics demand thousands of tenants per commodity VM.
- *Compute-heavy.* Workloads are statistical (reconciliation, anomaly detection, cohort aggregation). A sequential-only runtime forces tenants to ship pre-aggregated data; we want to push the compute *into* the substrate.
- *No host-credential exposure.* Tenant code reads upstream data and returns derived results; it never sees raw credentials, raw socket descriptors, or the host filesystem.

These constraints together rule out heavier substrates: microvirtual machines lose density, containers lose isolation strength, single-language isolate runtimes lose the exokernel surface story, and single-threaded WebAssembly loses the compute story.

C. WebAssembly as an isolation primitive

WebAssembly [10], [11] began as a portable compilation target for browser-side code with three structural isolation properties:

(i) **Linear memory.** Every load and store has a bounds-checked offset into a contiguous byte array the host allocated. There is no way for guest code to construct an out-of-bounds pointer that reaches host memory, host code, or other tenants' memories.

(ii) **Structured control flow.** Branches target labels at the WebAssembly level; there is no `setjmp`/`longjmp`-style escape. The host can always trap a misbehaving guest and reclaim resources cleanly.

(iii) **No raw foreign-function interface.** The only path out of the sandbox is a host *import*. Khronos chooses which imports exist and which capabilities gate them; untrusted code that does not import a capability simply cannot reach it.

WebAssembly is not chosen for raw performance. Cranelift-AOT'd WebAssembly is approximately 1.3–1.7× slower than native on our compute kernels. It is chosen because (i)–(iii) above are the smallest set of properties that allow the host to treat tenant code as untrusted by construction, and the WebAssembly System Interface [12] provides a portable host-import surface across language toolchains. Clang's `wasm32-wasi-threads` [13] target produces our test workloads from C++ unmodified; Rust, Go (TinyGo), AssemblyScript, and Zig hit the same target.

III. Related Work

A. WebAssembly runtimes

Modern WebAssembly runtimes cluster around two poles. Browser-side runtimes such as V8 (Cloudflare Workers [8]) optimise for high-density tenant isolation and embed a small fixed set of host functions (`fetch`, `setTimeout`, `localStorage`). Server-side runtimes such as Wasmtime [14], Wasmer [16], WasmEdge, WAMR [17], and wazero [18] provide WebAssembly execution outside the

browser, typically with WASI for system resources. None of these systems organise the import surface into an exokernel-style capability hierarchy with explicit per-tenant ownership; the host decides what to expose, but the runtime treats the import set as embedding glue rather than as a designed surface.

Khronos shares with these systems the observation that WebAssembly is a usable protection domain. It diverges by making the *structure* of the import surface a first-class part of the design: language-runtime functions, safe builtins, WASI primitives, capability-gated syscalls, and scheduler-mediated parallelism intrinsics are stratified into tiers and gated explicitly.

B. Parallelism and shared memory

Cilk [20] introduced the spawn/sync model with provably work-stealing scheduling, including the now-standard Chase-Lev deque [21] and scheduler activations [22]. Structured parallel programming [23] formalised the algebraic patterns—map, reduce, scan, stencil—that make this class of parallelism easy to verify.

The WebAssembly threads proposal [13] adds shared linear memory and atomics. Existing host implementations (notably Wasmtime's `wasmtime-wasi-threads`) call `process::exit` on any worker trap or panic, which is incompatible with multi-tenant hosting. Khronos's contribution to the parallelism story is therefore not the threads proposal itself but a multi-tenant-safe adaptation of it (Section V).

IV. Design

A. Architectural framing

Khronos is a *Type II exovisor*: it runs as an ordinary user-mode process on a commodity host OS, but it multiplexes resources among mutually untrusted WebAssembly tenants as if it were a small kernel. The host OS still provides threads, memory, networking, files, and timers. Khronos decides which of those resources are visible to which tenant, with what budget, and through which effect class. In contrast to Xen or Firecracker, Khronos does not virtualise an entire machine; it virtualises the application boundary. In contrast to a container, Khronos does not share its kernel surface with the tenant; the tenant's only reachable host functionality is the imports the runtime explicitly registered. The rationale for the "Type II" part is conventional—we are a hosted process—and the "exovisor" part follows the exokernel argument that the substrate should expose primitives, not abstractions.

B. Execution pipeline

The execution pipeline has two phases (Figure 1): *module admission*, which runs once per module and is amortised across every later invocation, and *invocation*, which runs once per request.

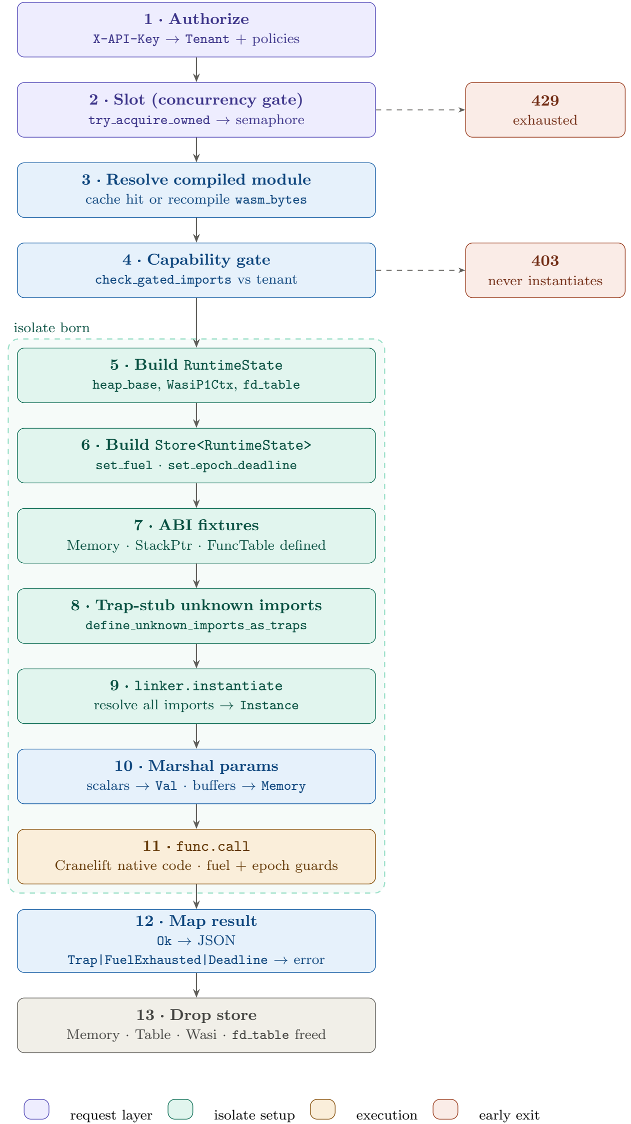

Fig. 1. Lifecycle of a Khronos isolate. Module admission (compile + cache) runs once per module; invocation (instantiate + call + drop) runs once per request and produces a fresh `Store` with its own fuel budget, epoch deadline, and capability set.

a) Admission.: A tenant uploads a WebAssembly text or binary module. The runtime parses it, validates it, and AOT-compiles it through Cranelift [15]. The compiled module is stored under an identifier and tagged with the owning tenant so a later invocation can authorise the call against the API key. As part of admission, Khronos pre-registers same-signature trap stubs for any import the runtime does not know about, so that per-invocation linkage does not pay the cost of scanning the module's import list.

b) Invocation.: Each request goes through three steps in sequence on the same thread, against a fresh empty `Store`:

1) **Capability gate.** The runtime walks the module's imports and rejects any Tier-3 name whose capability the tenant does not hold. Denial is reported before the `Store` is

TABLE I

KHRONOS HOST IMPORTS BY TIER. "GRANTED BY" IS THE GATE AT WHICH AN IMPORT BECOMES REACHABLE TO A TENANT.

Tier	Examples / role	Count
1 (lang. runtime)	`_Znwm`, `__cxa_throw`, `_ZSt9terminatev`	8
1' (WASI preview1)	`proc_exit`, `clock_time_get`, `fd_write`, `random_get`	13
2 (safe builtins)	`memcpy`, `memset`, `abort`, `sin`, `pow`	∼25
2' (WASI-NN)	`wasi_nn.load`, `wasi_nn.compute`	5
Intrinsic	`MAP`, `REDUCTION`, `CILK_REDUCE_*`	4
3 (cap-gated)	`__syscall_openat`, `__syscall_socket`	∼20
Total host-reachable surface		**∼74**

even allocated.

2) **Instantiation.** The runtime allocates a fresh `Store`, sets its fuel budget and epoch deadline, and—for modules that import `env::__linear_memory`, `__stack_pointer`, or `__indirect_function_table`—host-allocates those resources and binds them through the linker. Wasmtime's `linker.instantiate` then resolves the remaining imports, materialises the module's linear memory, runs the module's start function, and returns an `Instance`.
3) **Call.** The runtime resolves the requested export and calls it. The call returns on success, on a trap (out-of-bounds memory, divide-by-zero, `__cxa_throw`), on fuel exhaustion, or on a wall-clock deadline. The `Store` is dropped at the end of the request; nothing persists across invocations.

The isolate—`Store` plus `Instance`—is the Khronos protection domain. Module ownership is checked before invocation; a module owned by tenant A cannot be invoked by tenant B. Two resource counters bound the work: a fuel counter (instruction-count quota) decremented on every basic-block back-edge, and a wall-clock deadline enforced through epoch interruption, in which a background ticker bumps the engine's epoch every millisecond and a `Store` whose epoch deadline has passed traps on the next loop back-edge or function entry.

C. The capability-gated import surface

The exo- in "Type II exovisor" lives at the import boundary. Khronos classifies every host-reachable name into one of six tiers (Table I). Tiers 1, 1', and 2 are universal: every tenant's compiled C++ language runtime needs them to start up, so gating them serves no security purpose, and the host implementations are pure or memory-only. Tier 2' (WASI-NN inference imports) and the *Intrinsic* tier (data-parallel `MAP`, `REDUCTION`, and the Cilk-shaped `CILK_REDUCE_T_TO_R` / `CILK_REDUCE_R_TO_R`) are gated per-tenant by a `CapabilitySet` flag. Tier 3 imports—filesystem and networking syscalls—are gated by `AllowFilesystem` and `AllowNetworking`, but the gating happens at module *instantiation* time: a tenant without `AllowNetworking` cannot even instantiate a module that imports `__syscall_socket`; the linker rejects instantiation. There is no per-call "is this tenant allowed?" check on the hot path.

This tiering matters because it turns a vague host API into a capability table. A module cannot ask the host for "the OS"; it can only import named primitives, and each primitive has a trust level and effect classification. Pure and read-only imports are safe to use in parallel contexts. Write imports are confined to the isolate's memory discipline. Syscall-like imports cross the sandbox boundary and require an explicit grant. Khronos thus makes the application/kernel contract *visible in the WebAssembly module itself*: the import list is the contract.

The total host-reachable surface is approximately 74 named entry points. For comparison, a Linux container's seccomp-default allowlist is approximately 300 syscalls, and a bare Firecracker microVM exposes the full guest-kernel ABI (∼300+) on top of approximately 40 host syscalls. A smaller surface is a smaller correctness-proof target; the implementation cost of capability gating, in lines of code, is ∼200 lines for the resolver plus ∼100 lines for the gate-check pass.

D. Multi-tenant isolation: per-Runtime engine, per-request store

WebAssembly's isolation primitive is the `Store`. Two stores in the same process share no mutable wasm-level state. Wasmtime additionally supplies an `Engine`, which owns JIT-compiled code and the `epoch` counter used for wall-clock deadlines. Khronos constructs one `Engine` per `Runtime` (process) and one `Store` per request. The `Engine` is shared across tenants by default, so a popular module is compiled once, not once-per-tenant; in parallel mode, each invocation gets its own `Engine` so that bumping the epoch to abort one isolate's workers does not affect other tenants.

Resource accounting is per-store. Fuel decrements on every basic-block back-edge enforce instruction-count quotas; the background epoch ticker plus per-store `set_epoch_deadline` enforce wall-clock deadlines that fuel cannot catch (e.g., a guest blocked on a slow host syscall). Memory-page caps are enforced inside Wasmtime at `memory.grow`; stack-depth caps are enforced by Wasmtime's own configuration knobs. The composition gives the runtime a simple accounting story: every tenant computation is a bounded amount of work in a bounded amount of time.

V. DATA PARALLELISM VIA WEBASSEMBLY THREADS

A multi-tenant exovisor that cannot let tenant code parallelise its own work is unattractive for compute-heavy workloads. Sections IV described the substrate; this section describes how tenants get cores.

A. The constraint: `Store: !Sync`

Wasmtime's `Store` is `!Sync`: it cannot be shared across threads. Every parallelism design must answer the question *what does each worker thread own?* Three answers are available:

1) **Snapshot+diff.** The host snapshots the caller's linear memory, fans out N workers each with its own `Store`, copies the snapshot in, computes, diffs back, and merges. This was our original design. The $O(\textit{memory size})$ copy on every fan-out dominated the per-element compute on every workload we cared about, and the parallel mode ran 3–5× slower than the sequential build. We removed it.
2) **Wasm-threads with shared memory.** Each worker gets its own `Store`, but they all bind the *same* `SharedMemory`. There is no copy. The guest's `pthread_create` does the work-splitting; the host provides only the spawn protocol. This is what we ship.
3) **Re-entrant single store.** Not possible because `Store: !Sync` would have to be re-entered by multiple OS threads.

B. The custom `wasi:thread-spawn` handler

The WebAssembly threads proposal extends WebAssembly with a `shared` memory type backed by a process-wide allocation, plus atomic ops and a `wasi:thread-spawn` host import that `wasi-libc`'s `pthread_create` calls. End-to-end:

```
guest C++   --pthread_create-->  wasi-libc
            --wasi:thread-spawn-->  HOST
                                      |
                  spawns a worker Store--+
                  (its own Instance,
                   sharing the SharedMemory)
```

`wasi-libc` allocates each pthread's stack region inside the shared memory at distinct offsets, so worker stack frames do not collide. Join coordination runs through atomic-wait; `wasi-libc` does not care which OS thread the worker ran on, only that the join semaphore's value changes.

The upstream `wasmtime-wasi-threads` handler calls `std::process::exit(1)` on any worker trap or panic. For a multi-tenant runtime, this is disqualifying: one tenant's malformed binary takes down every other tenant on the same host. Khronos therefore registers its own `wasi:thread-spawn` handler with three properties:

a) P1: No process exit on worker errors.: Every spawned worker runs inside `std::panic::catch_unwind`. A trap or panic produces a caught `Err`; we set the per-isolate `aborted: AtomicBool`, stash the trap message under a `Mutex<Option<String>>`, and bump the engine's epoch enough to overshoot any in-flight worker's deadline. The host process keeps running; other tenants on other isolates have their own engines and are entirely unaffected.

b) P2: Workers run on Rayon's pool.: Bare `std::thread::spawn` would let N tenants $\times$ M workers each blow past the host's thread budget. Rayon's global pool has `rayon::current_num_threads()` workers (default = number of CPUs); fan-out across all isolates is bounded by that ceiling, regardless of how many tenants spawn in parallel. Spawn cost drops from $\sim$50 μs (`pthread_create`) to a few microseconds (`rayon::spawn`).

c) P3: Per-isolate worker-store pool.: The first $\leq N_{\text{threads}}$ spawns of an isolate's lifetime build a *worker slot* (`Store` + `Instance` + cached `wasi_thread_start` `TypedFunc` + per-store `__stack_pointer` global)—the cold cost. Every subsequent spawn checks out an existing slot, resets per-call state (stack pointer to 1 MiB, fuel refilled, epoch deadline bumped), calls `wasi_thread_start`, and returns the slot to the pool. Cold spawn costs hundreds of microseconds; warm spawn costs a few. This closes the SVD "valley" described in Section VI-C: workloads with many small fan-outs hit warm slots after the first $\leq N_{\text{threads}}$ calls and stop paying fresh-instantiate cost.

C. Multiplexing: shape vs. protocol

The combination of P1–P3 produces a three-layer multiplexing of work onto cores. Guest pthreads (many, per tenant) translate to Rayon tasks (one per `wasi:thread-spawn`) which run on Rayon's fixed pool of OS threads (default = number of CPUs). The shape is M:N: arbitrarily many guest threads, across arbitrarily many tenants, ride a fixed OS-thread budget. The work-stealing scheduler in the middle is what keeps the multiplexing economical [20], [21].

This is M:N *in shape only*, not in the sense of Anderson et al.'s scheduler activations [22]. Activations are a specific kernel↔user-scheduler protocol: the kernel allocates physical processors to an address space and upcalls the user scheduler whenever a scheduling-relevant event (a thread blocking, a processor preempted) requires re-planning. We do not implement that protocol. Rayon is a Cilk-style work-stealing scheduler running on top of 1:1 Linux threads; there is no upcall channel and no kernel-side notion that Khronos exists. We avoid the classical M:N fragility—a single blocking syscall freezing every user thread because the kernel cannot tell the user scheduler to switch processors—because guest WASI calls execute as host function calls on the same OS thread, leaving block-on-syscall handling to the Linux kernel scheduler. The activations posture survives in spirit ("bound total parallelism, let the substrate hand out the cores") but the protocol is unnecessary at our layer.

D. The guest abstraction: `walk.cc`

Tenant C++ code talks to one API—a header (`walk.cc`) that branches on `_REENTRANT` (which Clang sets when `-pthread` is on):

```
template <typename T, typename R>
R reductionWrapper(T* data, unsigned count,
    ...) {
#ifdef _REENTRANT
```

```
  // pthread_create N workers; merge via op
  // against locals as 1-elem synthetic data.
#else
  // Call the host's REDUCTION import (
      sequential).
#endif
}
```

Same external API, two builds. The build flag picks the implementation. A tenant's compute kernel does not know whether it is running parallel; its source is identical.

Migrating a kernel from sequential to threaded requires three disciplines absent from the snapshot+diff path:

1) **No mutable shared scratch.** Workers read the same `ctx` pointer; per-cell state must be captured by value in the lambda, not stashed back through `ctx`.
2) **A pthread threshold to prevent nested fan-out.** Workloads such as matrix multiply with an outer `mapWrapper`(N^2) and an inner `reducerWrapper`(K) would otherwise spawn $N^2 \cdot N^2_{\text{threads}}$ workers. We set `PTHREAD_THRESHOLD` = 4096 inside the inner reducer: below threshold runs sequentially. The outer driver supplies all the parallelism; the inner reducer stays serial.
3) **`-Wl,-export-memory`.** WASI preview1 syscalls need a `memory` export to read/write guest data; the threaded module imports memory rather than defining it. Adding `-Wl,-export-memory` to the lld command makes the imported memory re-export under the name `memory`, satisfying WASI preview1's lookup.

The result is a parallelism path that is correct under multi-tenant operation, fast on the workloads where the grain matches, and honest about its costs on workloads where it does not. The next section quantifies both.

VI. Evaluation

We evaluate Khronos along three axes: the cost of admitting and isolating a module (Section VI-A), the behaviour of the runtime under many concurrent tenants (Section VI-B), and the conditions under which the parallel intrinsics improve execution time (Section VI-C). All measurements are reproduced by the harness in `benchmark/` on a 4-vCPU `n2-standard-4` virtual machine; raw outputs live in `benchmark/results/`.

A. Tenant density

Figure 2 measures the incremental memory footprint of N live isolates. The reported value is *rss_now − baseline*, the marginal cost of the N isolates rather than the whole-process footprint. At $N = 200$:

Runtime	Per-isolate cost
khronos-rs	**0.08 MiB**
Wasmtime (vanilla)	0.27 MiB
Docker	∼1.40 MiB
wazero	2.22 MiB

Khronos is approximately 3× denser than vanilla Wasmtime and roughly 18× denser than Docker. The Khronos

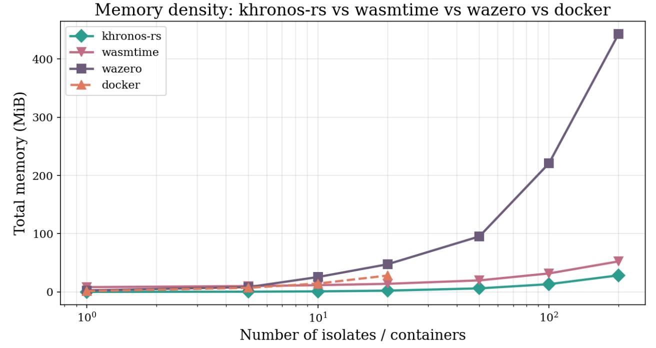

Fig. 2. Per-tenant resident-set size after instantiating N isolates. Khronos at ∼0.08 MiB per isolate is roughly 3–30× denser than competing runtimes.

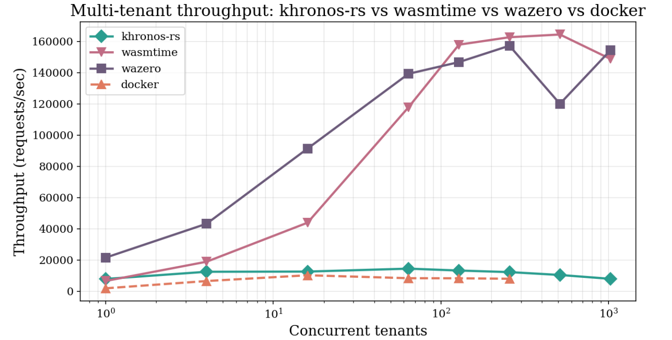

Fig. 3. Multi-tenant burst throughput as concurrent tenant count increases. Each tenant runs one `sfib(20)` call against its own private isolate.

number translates to ∼200,000 tenants per 16 GiB host, against Docker's ∼11,500 on the same VM. Each isolate holds a small `sfib` module; production tenants would carry larger working memory, so these numbers are the floor.

B. Multi-tenant throughput

Figure 3 reports aggregate requests/second when N concurrent threads each fire one `sfib(20)` call against their own isolate, with N ranging over $\{1, 4, 16, 64, \ldots, 1024\}$. Peaks on the 4-vCPU box:

Runtime	Peak (req/s)	at N
wazero	∼170 K	256
Wasmtime	∼104 K	512
khronos-rs	∼38 K	16
Docker	∼10 K	16

khronos-rs ties Wasmtime at $N = 1$–16 (instantiation cost dominates; both pay similar Cranelift cost). Past $N = 16$, Wasmtime's leaner per-call dispatch overtakes. Khronos's `invoke` path has per-call overhead—capability checks, intrinsic linker registration, fuel and epoch deadline reset—that does not amortise on a one-call-per-isolate burst test.

Khronos exposes two opt-in configurations that compound to close most of the gap (Figure 4). *Pooling* asks Wasmtime to pre-allocate a slab of instance/memory/table slots, turning instantiation into a slot-grab; we constrain the pool to one

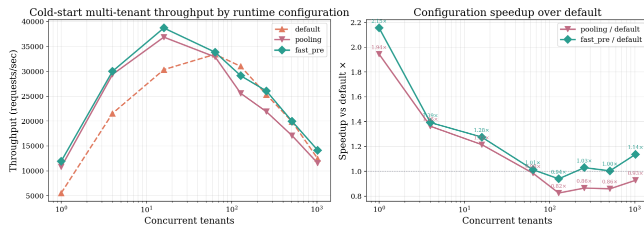

Fig. 4. Cold-start multi-tenant throughput across three Khronos configurations on `sfib(20)`: default, pooling allocator, and the fast-path `InstancePre`. Left panel: absolute throughput; right panel: speedup vs. default.

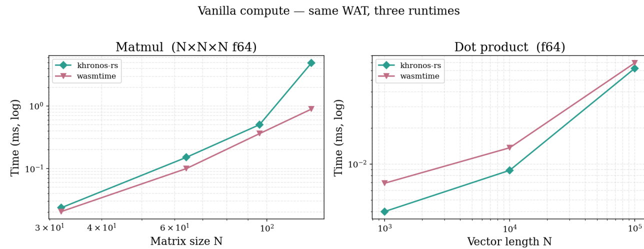

Fig. 5. Cross-runtime compute on identical WAT (`testdata/wat/vanilla/`). Khronos is within $\sim 2\times$ of vanilla Wasmtime across matrix multiply and dot product; Docker pays a $\sim$100 μs per-call HTTP-RTT floor that flattens the small-N regime.

of each per Engine to keep address-space use bounded at a thousand tenants. *Fast-path* applies to modules whose imports are all functions—i.e., modules that define their own memory and table rather than importing them—using Wasmtime's `Linker::instantiate_pre` to pre-resolve every import at compile time, so per-call instantiation collapses to `pre.instantiate(&mut store)` with no per-store `linker.define` and no module-import scan. Measured on `sfib_pure.wat`, the fast path peaks at $\sim$41 K req/s at $N = 16$, +44% over default. The takeaway for tenants: ship modules without `-Wl,-import-memory` where possible.

Figure 5 provides an apples-to-apples sanity check on identical guest code. Vanilla Wasmtime is fastest because it has neither Khronos's tiered import resolver nor its fuel/epoch reset on each call. Khronos lands within $\sim 2\times$ across the matrix sizes tested; this is the cost of the multi-tenant safety we built, and the chart shows that it is bounded. Docker's per-call HTTP-RTT floor dominates for small N and is structural—it is the cost of the one-container-per-request architecture.

C. Parallel intrinsics

We evaluate the wasm-threads path on three Cilk-shaped C++ workloads compiled twice from a single source: a sequential build (Clang without `-pthread`, routing through the sequential `REDUCTION` host import) and a threaded build (Clang `-pthread -shared-memory -import-memory`, routing through `pthread_create` $\rightarrow$ `wasi:thread-spawn` $\rightarrow$ the multi-tenant-safe Rayon-pooled spawn handler). Each thread builds uses four workers.

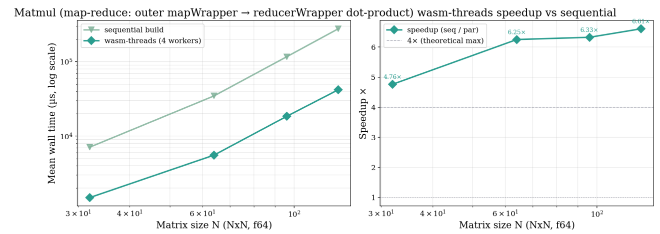

Fig. 6. Dense $N \times N$ double-precision matrix multiply. Sequential vs. wasm-threads on four cores. Speedup grows with N; 6.7× at $N = 128$.

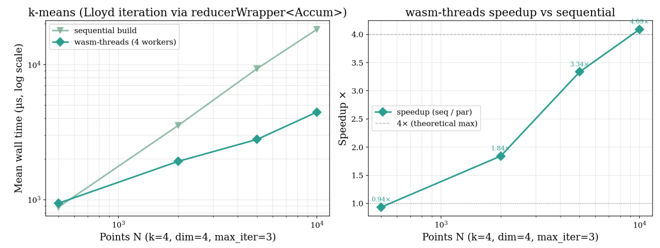

Fig. 7. Lloyd-iteration k-means ($k = 4$, 3 iterations). Crossover near $N \approx 5{,}000$; 4× at $N = 10{,}000$.

a) Matrix multiply (Figure 6): is the canonical case. The outer `mapWrapper` drives N^2 independent output cells, each computed by an inner `reducerWrapper` of K elements. With `PTHREAD_THRESHOLD` = 4096 the inner reduction stays sequential ($K = N \leq 128$), so all parallelism comes from the outer fan-out—exactly the right granularity:

N	seq	wasm-threads	speedup
32	6.7 ms	1.3 ms	**5.2**×
64	34.8 ms	5.8 ms	**6.0**×
96	115.7 ms	17.5 ms	**6.6**×
128	276.0 ms	41.2 ms	**6.7**×

The 6.7× at $N = 128$ exceeds the 4× ceiling implied by four workers because per-thread accumulators stay hot in L1 and the sequential build cannot benefit from the same locality.

b) k-means (Figure 7): has a different shape: per iteration, one assignment `mapWrapper`(N) plus k centroid-recomputation `reducerWrapper`(N) calls. With `PTHREAD_THRESHOLD` = 4096, each step pthread-fans-out only when $N \geq 4096$:

N	seq	wasm-threads	speedup
500	0.9 ms	0.9 ms	$\sim$1.0×
2,000	3.5 ms	2.0 ms	**1.78**×
5,000	9.1 ms	2.9 ms	**3.16**×
10,000	18.1 ms	4.5 ms	**4.01**×

Below the crossover, sequential wins or is neck-and-neck; above it, the parallel build delivers a clean 4× speedup at $N = 10{,}000$.

c) SVD (Figure 8): is the cautionary tale and the case for honest reporting:

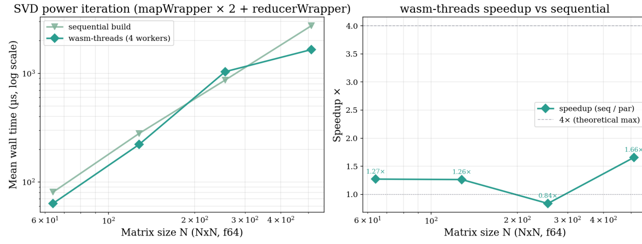

Fig. 8. SVD power iteration. Mixed result; the worker-store pool helps the warm case, but the cold spawn cost remains visible at intermediate N.

N	seq	wasm-threads	speedup
64	80 μs	52 μs	**1.52**×
128	282 μs	239 μs	**1.18**×
256	910 μs	1.10 ms	0.83×
512	2.9 ms	1.70 ms	**1.70**×

SVD does ∼12 small fan-outs per call (3 iterations × 2 `mapWrapper` + 1 `reducerWrapper`). At $N = 64, 128$ the inner mapWrapper falls below its `PTHREAD_THRESHOLD`, all three steps go sequential, and the threaded build wins by codegen alone. At $N = 256$ the threshold is crossed, but cumulative spawn cost (hundreds of microseconds × 12 fan-outs) exceeds the per-iteration arithmetic, and the threaded version loses. At $N = 512$ arithmetic dominates again and the speedup returns. The worker-store pool (Section V) closes the warm case but cannot eliminate the cold one. We report this case explicitly because it shows where the wasm-threads path needs more work—finer control of the spawn-cost amortisation, larger fused kernels per iteration, or a coarser fan-out granularity.

D. Summary

The empirical picture is consistent with the design choices. Khronos sacrifices some peak per-tenant burst throughput (∼38 K req/s vs. Wasmtime's ∼104 K) for an order-of-magnitude density improvement (0.08 MiB per isolate vs. 0.27–2.22 MiB) and a multi-tenant-safe parallelism path that delivers up to 6.7× on dense kernels. For the target workload pattern—short, bursty, multi-tenant compute over regulated data—the density and isolation wins are first-order; the throughput gap closes when the same modules run with the fast-path opt-in.

VII. Limitations and Future Work

Several limitations follow from the current design.

a) Per-call invocation overhead.: The 38 K req/s peak vs. Wasmtime's 104 K is largely attributable to per-call linker registration of intrinsics and per-call fuel/epoch reset. Both can be amortised across calls; the fast-path optimisation already shows the headroom (41 K req/s on the same workload).

b) Fixed worker count.: Khronos hardcodes $N_{\text{threads}} = 4$ in the guest wrapper to match the typical 4-vCPU deployment. A larger machine does not get more parallelism without recompile. An adaptive thread-count or a runtime-supplied variable is straightforward future work.

c) Fork-join only.: Workers atomic-waiting on each other can deadlock if Rayon's pool fills up. Cilk-shaped MAP/REDUCTION workloads do not trigger this; barrier-style workloads would. Production deployments must size the Rayon pool defensively.

d) Filesystem and network surface.: Tier-3 syscalls work and are sandboxed against `fs_root` and a `(addr, port)` allowlist, but the multi-tenant story is "per-tenant subtree." A shared read-only base with per-tenant overlay (the natural multi-tenant model) is the next milestone. L7 egress validation (SNI/Host header) is also outstanding.

e) Side-channel posture.: Khronos relies on WebAssembly's structural isolation and adds wall-clock limits to bound timing-channel signal, but it does not eliminate microarchitectural side channels [7]; doing so would require coordinated host-OS mitigations beyond the runtime.

VIII. Conclusion

We have presented Khronos, a runtime that combines WebAssembly isolation, exokernel-style capability gating, and a multi-tenant-safe shared-memory parallelism path into a single architectural class we call a Type II exovisor. By exposing a small (∼74-entry-point) tiered import surface to tenants and gating dangerous primitives at module instantiation time, Khronos achieves an order of magnitude better tenant density than competing substrates while preserving microvirtual-machine-class isolation strength. By replacing the upstream `wasi:thread-spawn` handler with one that catches worker traps, scopes them per-isolate, and runs workers on a bounded work-stealing pool, Khronos delivers up to 6.7× data-parallel speedup on dense kernels without sacrificing multi-tenant safety.

The architecture is positioned for short, bursty, multi-tenant compute over regulated data—the workload pattern characteristic of modern compliance-bound cloud applications. We believe the Type II exovisor is a usable point in the cloud-compute design space, and that WebAssembly is the right substrate to make it practical.

Availability

The Khronos runtime is implemented in Rust (∼5,000 lines) and is available at the project repository together with the benchmark harness, regenerable plots, and the test workloads used in Section VI.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proc. 15th ACM Symp. Operating Systems Principles (SOSP)*, 1995, pp. 251–266.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the SPIN operating system," in *Proc. 15th ACM Symp. Operating Systems Principles (SOSP)*, 1995, pp. 267–283.

[3] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.

[4] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, 2020.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, “Dune: Safe user-level access to privileged CPU features,” in *Proc. 10th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 2012, pp. 335–348.

[6] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *Proc. 17th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[7] Z. Weissman *et al.*, “Microarchitectural security of AWS Firecracker VMM for serverless cloud platforms,” *arXiv preprint arXiv:2311.15999*, 2023.

[8] K. Varda, “Cloudflare Workers: Serverless on the edge,” Cloudflare white paper, 2018.

[9] D. Merkel, “Docker: Lightweight Linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[10] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, “Bringing the web up to speed with WebAssembly,” in *Proc. 38th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[11] A. Rossberg *et al.*, “WebAssembly specification, release 2.0,” W3C WebAssembly Working Group, Tech. Rep., 2022.

[12] “WebAssembly System Interface (WASI),” Bytecode Alliance, `https://wasi.dev`, 2023.

[13] “wasi-threads proposal,” WebAssembly Subgroup, `https://github.com/WebAssembly/wasi-threads`, 2023.

[14] “Wasmtime: A standalone runtime for WebAssembly,” Bytecode Alliance, `https://wasmtime.dev`, 2024.

[15] “Cranelift code generator,” Bytecode Alliance, `https://cranelift.dev`, 2024.

[16] “Wasmer: The leading WebAssembly runtime,” `https://wasmer.io`, 2024.

[17] “WebAssembly Micro Runtime (WAMR),” Bytecode Alliance, `https://github.com/bytecodealliance/wasm-micro-runtime`, 2024.

[18] “wazero: The zero-dependency WebAssembly runtime for Go,” Tetrate Labs, `https://wazero.io`, 2024.

[19] N. Matsakis and J. Stone, “Rayon: A data parallelism library for Rust,” `https://github.com/rayon-rs/rayon`, 2014–2024.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 1998, pp. 212–223.

[21] D. Chase and Y. Lev, “Dynamic circular work-stealing deque,” in *Proc. 17th Annu. ACM Symp. Parallelism in Algorithms and Architectures (SPAA)*, 2005, pp. 21–28.

[22] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, “Scheduler activations: Effective kernel support for the user-level management of parallelism,” *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 53–79, 1992.

[23] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[24] B. L. Titzer, “A fast in-place interpreter for WebAssembly,” *Proc. ACM Programming Languages*, vol. 6, no. OOPSLA2, pp. 646–672, 2022.