# Architecture Options for a WASM-Based Deterministic RTOS

## Introduction

Building a WebAssembly-based real-time operating system (RTOS) on commodity hardware running Linux requires balancing **microsecond-level determinism**, **strong isolation**, and **security**. The target system should achieve P99.99 latency on the order of tens of microseconds, provide fault isolation between components, and enforce a capability-based security model. Recent advancements in the Linux kernel and WebAssembly ecosystem make this feasible: the PREEMPT\_RT patch (now in mainline) enables worst-case scheduling latencies around 100–125 µs on Linux[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22), and modern WASM runtimes like **Wasmtime** can ahead-of-time (AOT) compile modules to near-native speed with only ~10 µs of runtime overhead[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22). At the same time, WebAssembly’s sandbox and the WebAssembly System Interface (WASI) use a **capability-based security** model where modules are only given access to specific resources via handles, enforcing least privilege by default[[2]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20foundation%20of%20the%20security,nano).

This section presents **multiple architecture options** for such a WASM-RTOS, ranging from a conventional host-process design to more novel microVM-isolated and bare-metal approaches. Each option is described with its architectural layering, runtime environment, I/O strategy, memory management for determinism, scheduling approach, and security mechanisms. We analyze the **trade-offs** (latency vs. isolation vs. complexity, etc.) for each design. A comparative table at the end summarizes key differences in latency, modularity, complexity, security, and composability.

## Option 1: **Direct Host Process RTOS (Linux Process + WASM Runtime)**

**Architecture Overview:** The first design runs the WebAssembly runtime *directly on the Linux host* as a real-time process, without an intervening VM layer. The Linux host is tuned with a real-time kernel (PREEMPT\_RT) and isolcpus to dedicate one or more CPU cores exclusively to the RTOS process. The WASM runtime (e.g. Wasmtime or WasmEdge) executes as a single process (or a small number of processes/threads) pinned to isolated core(s) using a real-time scheduling policy. The *host OS acts as the scheduler and resource manager*, but non-RT background activity is minimized on those cores. This yields a simple stack: **[Linux Kernel (PREEMPT\_RT) → WASM Runtime Process → WASM Modules]**. Each real-time WebAssembly module runs in the same process, isolated from each other by the WASM sandbox’s linear memory and type safety, rather than by separate OS processes.

**Runtime and Layering:** In this approach, the WASM runtime is essentially a *library OS* within a process. It can load multiple WASM modules (possibly as WASI components) into the same address space, facilitating direct function calls between modules. This takes full advantage of the emerging WASM **Component Model** for in-process composition. Modules from different languages can be linked via WASI interfaces and the canonical ABI. Because all components share the runtime, cross-component calls incur only in-memory data copying (which, however, can introduce a performance cost if large data are passed – up to a 3× slowdown due to the canonical ABI overhead[[3]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,time%20deadlines.%20%5Bmult) until upcoming zero-copy flat<T> types resolve this). The benefit is **high composability**: real-time-critical logic can be written as separate modules (e.g. a control algorithm in C++ and a rules engine in Rust) and linked together securely in the same process. Care must be taken to isolate any module with a *garbage-collected language* (like Go or C#) if it could pause unpredictably – such modules might be kept out of the real-time core or run on a separate thread, since GC pauses of >5 ms would violate hard deadlines[[4]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,Model%20to%20manage%20the%20boundary).

**Scheduling and Preemption:** The host Linux kernel is responsible for scheduling the WASM runtime process on dedicated core(s), using real-time policies. **SCHED\_FIFO** or **SCHED\_DEADLINE** can be used to ensure the WASM process gets CPU time with minimal interference. In particular, SCHED\_DEADLINE (with an appropriately set period and runtime) provides hard scheduling guarantees by deadline admission control, essentially treating the WASM process as a periodic real-time task[[5]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=4,Guarantees). Inside the process, the WASM runtime can enforce fine-grained scheduling of guest tasks using **cooperative yield or fuel-based preemption**. For example, Wasmtime supports a *fuel mechanism* where the engine counts instructions and can interrupt a module’s execution when a fuel budget is exhausted. This provides deterministic intra-process scheduling: e.g. a module can be preempted after N instructions to yield to another, with a known context-switch cost <1 µs[[6]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AWASI,budgets%20on%20sandboxed%20WASM%20code). By aligning these fuel-based yields with host scheduler ticks or deadlines, the system can ensure no single WASM module hogs the CPU beyond its allotted time. This design achieves end-to-end *preemption latency* on the order of a few microseconds for switching between WASM tasks, leveraging the combination of a fully preemptible kernel and the lightweight user-space task switching in the WASM runtime[[6]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AWASI,budgets%20on%20sandboxed%20WASM%20code).

**Memory Determinism:** Running directly on Linux allows the use of strict memory locking and huge pages to eliminate paging jitter. The process should mlockall() its memory at startup, and use pre-allocated **static hugepages** for its heap to avoid any runtime page faults. This ensures that no major page faults occur during operation – **page faults can otherwise introduce 10–40 ms latency if memory gets swapped or demand-paged**, which is unacceptable[[7]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2ADeterministic%20memory%20wins%2A%2A%3A%20Non,0%20%C2%B5s). By locking all pages in RAM and using 2 MB or 1 GB hugepages, the TLB misses are reduced and no Linux memory management activity (like collapse of transparent hugepages or swap) will interrupt execution. The result is worst-case memory access latency that is constant and **page-fault free (0 µs page fault delays)**[**[8]**](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2ADeterministic%20memory%20wins%2A%2A%3A%20Non,based%20interruption%20provides). The memory allocator inside the WASM runtime should also be a real-time allocator (e.g. TLSF or a slab/pool allocator) to guarantee bounded malloc/free times[[9]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=7,Purpose%20Allocator%20Slowdowns)[[10]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,69). In this design, **all dynamic memory usage is made deterministic** by design: no swapping, no overcommit, and constant-time allocation.

**I/O Strategy:** As a user-space process, the RTOS can choose between standard kernel I/O (with real-time tuning) or kernel-bypass methods for networking and storage. For networking, a **kernel-integrated approach** using technologies like io\_uring or **AF\_XDP** might be preferred for predictable latency under bursty loads. The Linux networking stack with io\_uring can handle sporadic traffic gracefully, capping tail latency around ~200 µs in worst-case bursts[[11]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AHidden%20tail,io_uring). Alternatively, for maximum throughput at the cost of potential jitter, the process could use **DPDK** (user-space NIC drivers) or raw AF\_XDP sockets to get packets directly from the NIC with minimal copies – achieving base round-trip latencies as low as ~6.5 µs[[12]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,%5Bhybrid_ebpf_and_wasm_architectures.zero_copy_data_path%5B0%5D%5D%5B23)[[13]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20efficiency%20of%20this%20hybrid,between%20kernel%20and%20user%20space). However, DPDK’s **tail latency under load** can spike to ~1100 µs[[11]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AHidden%20tail,io_uring), so for workloads where traffic patterns are unpredictable, sticking with io\_uring + SO\_RXQ\_OVFL or AF\_XDP with careful pacing might yield more stable P99.99 behavior. Similarly for disk I/O, the process can use **io\_uring** with fixed buffers for async disk access, or even **SPDK** for NVMe access if needed (though SPDK bypasses the kernel, it provides highest throughput but must be carefully managed to avoid starvation of other tasks[[14]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,workloads%20with%20unpredictable%20traffic%20bursts)[[15]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=performance%2C%20but%20with%20high%20CPU,resourced.)). In essence, the host-based design has the *flexibility to use the rich Linux I/O ecosystem* – standard drivers or bypass libraries – depending on the latency requirements of each I/O channel. Linux features like SO\_BUSY\_POLL and tuned interrupt affinities (isolating NIC interrupts to specific cores, disabling interrupt coalescing) should be applied to minimize latency noise[[16]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,as%20there%20are%20CPUs%20handling)[[17]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,BQL%29%20to%20prevent%20bufferbloat).

**Security and Isolation:** In the pure host-process model, isolation between the real-time workload and the rest of the system is achieved through Linux mechanisms and the WASM sandbox, rather than hardware virtualization. The **WASI capability model** ensures the WASM modules only access whitelisted files, sockets, or devices – for example, the runtime might open specific /dev devices or IPC channels and pass those descriptors into the WASM modules, which otherwise cannot open arbitrary resources[[2]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20foundation%20of%20the%20security,nano). This fine-grained permission model drastically reduces attack surface for code running within the WASM runtime. Additionally, the process can be locked down with Linux security features: **seccomp** filters to allow only syscalls needed for WASI, **cgroups** to limit resource usage, and **namespaces** (or running inside a container) to drop ambient privileges. Still, compared to stronger isolation options, this design shares the host kernel with other processes – a bug or malicious module that escapes the WASM sandbox (e.g. via a zero-day in the runtime or syscalls) could potentially affect the host. The *attack surface* includes the full Linux kernel (though mitigated by using only specific syscalls) and the WASM runtime engine. Consequently, this option is best suited for scenarios where all loaded modules are relatively trusted or single-tenant deployments. It provides **fault isolation** at the module level (memory safety and limited capability scope), but not a hard separation of kernel contexts. For additional safety, one can disable hyper-threading (SMT) to avoid side-channel leakage between the real-time core and any other thread[[18]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=guarantees%2C%20hardware%20features%20are%20used%3A,on%20the%20same%20physical%20core), and use techniques like Intel CAT to partition cache if running alongside other workloads[[19]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=In%20multi,to%20prevent%20two%20workloads%20from).

**Observability and Debugging:** Since everything runs in one Linux process, observability can leverage existing low-overhead tracing tools. For instance, **LTTng in snapshot mode** can be attached to the process to capture trace events with <200 ns overhead per event[[20]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,time%20systems), storing them in memory and dumping to disk only if a latency violation is detected (to avoid perturbing the runtime). This “flight recorder” approach makes it possible to capture rare P99.99 latency spikes for post-mortem analysis without constantly writing to slow storage[[20]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,time%20systems). Standard Linux perf and ftrace can also be used on the process (with precaution to keep overhead minimal). Because the system is single-process, correlating events is straightforward, and one can even utilize **Wasm-level record/replay** tools (as mentioned, e.g. Wasm-R3 for deterministic replay of module execution[[21]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=9.2%20Wasm,and%20rr%20as%20a%20Fallback)). The downside is that adding instrumentation in the process, if not done carefully, can impact the very timing one is trying to measure – hence the emphasis on low-intrusion tracing. Formal verification hooks are mostly applied at the software level: e.g. using model-checking on the scheduling algorithm used by the runtime (fuel mechanism and any internal task scheduler) to ensure no deadlocks or missed deadlines. The logic being part of the application, it could even be verified with tools like TLA+ and checked against the implementation, similar to verifying an application-specific scheduler (inspired by seL4 verification methods)[[22]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,78)[[23]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,83). However, the **Linux kernel itself is not formally verified**, so this design relies on Linux’s proven (but complex) codebase for low-level scheduling and drivers.

**Summary – Pros & Cons:** This host-based approach is **minimal in layering**, which yields **excellent raw latency** (only a single context switch from kernel to app) and simplicity in deployment (just run the binary on an RT-patched Linux). It can achieve tail latencies well under 150 µs on commodity hardware[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22), given that PREEMPT\_RT reduces kernel scheduling delays by ~294× compared to stock kernels[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22). It also enables rich in-process composition of components and direct use of Linux’s features. The trade-off is weaker isolation: the **fault domain is the Linux process** (or container) – a crash or compromise in the WASM runtime could affect the host. Security is mitigated but not as foolproof as hardware isolation. Thus, Option 1 shines for **single-tenant or tightly controlled deployments** where maximum performance and low overhead are needed, and where the convenience of Linux’s full environment is a positive. For multi-tenant or highly safety-critical uses, one might require the stronger isolation of the next design.

## Option 2: **MicroVM-Isolated RTOS (Lightweight VM + WASM Runtime)**

**Architecture Overview:** This design introduces a lightweight **microVM layer** (such as Amazon Firecracker or Cloud Hypervisor) between the Linux host and the WASM runtime. The WebAssembly application runs inside a **microVM**, which is essentially a minimalist virtual machine providing a dedicated kernel and userspace just for the RTOS workload. The host Linux acts as a “hypervisor” (using KVM) to launch the microVM, and then largely stays out of the way on the isolated core(s) dedicated to that VM. The stack becomes **[Linux (PREEMPT\_RT) → MicroVM (guest OS) → WASM Runtime → WASM Modules]**. Each microVM typically runs a stripped-down guest OS (could be a small Linux with only necessary drivers, or even a unikernel) that immediately starts the WASM runtime process. The **microVM is pinned** to specific CPU cores on the host, just like in Option 1, and given a fixed chunk of memory (locked in host RAM). Essentially, the microVM carves out a **hardware-enforced fault domain** for the RTOS: it has its *own kernel* separate from the host’s, providing a hard isolation boundary at the cost of a thin layer of virtualization overhead.

**Isolation and Layering:** Compared to Option 1, here the WebAssembly modules are one step further removed from the host. The microVM provides **hardware-enforced isolation** through virtualization extensions – any fault or crash in the guest will not directly crash the host, and a compromised module would have to break out of both the WASM sandbox *and* the guest OS to affect other systems. This significantly reduces risk in multi-tenant scenarios. In fact, giving each tenant or critical application its own microVM can eliminate **95% of the host kernel’s attack surface** for that application[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). The cost is that we now manage two kernels (host and guest). The microVM approach is favored by security-sensitive platforms; e.g. Firecracker was designed for multi-tenant workloads and has ~5 MB memory overhead and ~125 ms boot time[[25]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=As%20described%20in%20the%20architecture%2C,5MB) – extremely low overhead to spin up an isolated environment. Because each microVM has a tiny footprint, dozens of them could run on one physical host if needed, each pinned to dedicated cores or time-sliced with real-time scheduling at the host level.

Within the microVM, the **WASM runtime runs in the guest OS**. You can still use WASI and the component model inside the VM just as in Option 1, but you would typically run one primary WASM application per microVM for simplicity. (It’s possible to host multiple modules in one VM, but if they are truly independent, one may consider separate VMs per module for stronger isolation.) Communication between modules in the same microVM is as fast as in Option 1 (since it’s just intra-process calls if in one runtime), but communication *across* microVMs has to go through an external channel (network sockets, shared memory via the host, etc.). This means **composability is lower** across VM boundaries: the WASM component model cannot natively link modules that are in different VMs. Instead, those modules would interact via IPC or RPC (e.g. using a virtio-vsock, network loopback, or shared memory queue between VMs). Therefore, this architecture often lends itself to a more **microservice-like partitioning**: each microVM runs a self-contained service (one or a group of tightly coupled components), and high-level composition happens via message passing. Within a single microVM, though, you still get the benefits of WASI capability security and can run multi-language components together if needed.

**Scheduling:** The scheduling in this option happens at two layers. The host Linux uses **SCHED\_FIFO or SCHED\_DEADLINE to schedule the entire microVM’s vCPU thread(s)** on dedicated core(s). In an ideal setup, you pin the microVM’s virtual CPU 1:1 to a physical CPU core, giving it full control of that core – effectively no host contention except the minimal hypervisor overhead. The host’s real-time scheduler ensures the VM gets immediate execution when needed (since nothing else of equal priority is on that core). Within the microVM, we have a guest OS (which could also be a Linux with PREEMPT\_RT, or a simplified RTOS). The **guest OS schedules the WASM runtime process**. If the guest is Linux, it can again use SCHED\_DEADLINE or FIFO for the WASM process (now inside the VM). This double scheduling might sound complex, but if the microVM is single-vCPU and pinned, the host scheduling is trivial (always run the VM when it wants to run), and only the guest’s scheduling matters for the workload. Essentially, the real-time guarantees can be configured in the guest almost as if it were running on bare metal, because the host is giving it an entire core. To minimize latency, devices for the microVM (like virtio interrupts) should also be bound to isolated host cores if possible. In practice, experiments have shown that adding a KVM microVM layer adds on the order of **tens of microseconds (<<100 µs) of worst-case latency** if the vCPU is pinned[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). This is a small tax for the benefit of isolation. The WASM runtime inside the VM can also use its fuel-based preemption or internal task scheduler as in Option 1, so *inside* the VM we still get sub-1 µs context switch for WASM tasks[[6]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AWASI,budgets%20on%20sandboxed%20WASM%20code). Overall, the scheduling hierarchy is designed such that the microVM is essentially a fixed real-time partition of the CPU – avoiding complex interactions with host scheduling.

**Memory & Determinism:** When using a microVM, memory determinism has two aspects: host and guest. On the **host side**, the memory assigned to the microVM (say 512 MB or 1 GB, depending on the application) should be **pre-allocated (hugepages) and locked** similarly to Option 1. Most KVM-based VMMs allow backing the guest memory with hugepages on the host for performance. The host will then not swap that memory because it’s mlocked and wired to the guest. On the **guest side**, the guest OS should also disable any swapping and use its own real-time allocators. If the guest is Linux, one would similarly call mlockall() inside the VM for the WASM process and perhaps use a tmpfs for any file I/O to avoid disk access latency. Essentially the same precautions are applied *inside* the VM to avoid page faults. One benefit of the microVM is that the guest has a **smaller memory footprint and simpler workload**, so it’s easier to audit that no other daemon or process will consume memory unexpectedly. By giving the VM a fixed chunk of memory, you eliminate interference from other processes’ memory usage. As a result, worst-case memory latency in the microVM can be just as controlled as in Option 1 (0 µs page fault if locked[[7]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2ADeterministic%20memory%20wins%2A%2A%3A%20Non,0%20%C2%B5s)). One must ensure the microVM’s kernel is configured appropriately (e.g., if using Linux, disable transparent hugepages, use isolcpus, etc., in the guest as well). Another advantage is that memory access within the microVM **cannot be interfered with by host kernel background operations** like Linux’s own paging or housekeeping, since the host sees the VM memory as one large region. The host will not perform operations on it beyond maybe VM exits on page faults (which won’t happen if guest is locked). So the memory determinism arguably improves because of the hard wall – *nothing outside can evict the VM’s cache lines or memory allocation except general hardware effects*. That said, cache is still shared across cores (unless cache partitioning like CAT is used[[26]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=guarantees%2C%20hardware%20features%20are%20used%3A,on%20the%20same%20physical%20core)), so cross-VM interference via cache is possible if other cores are running non-RT tasks – this can be mitigated the same way as Option 1 (pin the VM to one LLC cluster, etc.).

**I/O Strategy:** A microVM introduces an extra layer for I/O: typically I/O will go through **virtio devices** (virtual NIC, virtual disk, etc.) provided by the host. There are two sub-approaches here: - **Paravirtualized I/O:** The microVM uses virtio-net for networking, which means the host Linux is running a vhost backend that ultimately pushes packets to the Linux network stack or to a tap device. This is convenient (no special drivers in guest aside from virtio) but adds some overhead per packet and context switches between guest and host. If the absolute lowest latency networking is needed, this might be suboptimal because each packet traversal involves exiting the VM (VM exit for virtio kick) and being handled by the host’s network stack. - **Direct assigned I/O:** For higher performance, one can dedicate a device to the microVM. For example, using **SR-IOV** (Virtual Functions of a NIC) or **VFIO** to pass a physical NIC (or a slice of it) into the microVM. In this case, the guest can run DPDK or AF\_XDP directly on the NIC hardware without the host in the data path. This effectively moves the kernel-bypass approach inside the VM. It yields near-native I/O performance at the cost of giving the VM more control over hardware. If only one microVM needs network access, the entire NIC can even be assigned to it, and the guest’s DPDK could drive it with 100% focus on low latency. This approach can achieve similar ~6–20 µs network latencies with careful tuning, like an ordinary DPDK app[[12]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,%5Bhybrid_ebpf_and_wasm_architectures.zero_copy_data_path%5B0%5D%5D%5B23). However, assigning devices to VMs can complicate sharing resources among VMs. - For storage, similarly, one could use virtio-blk (paravirt disk) or pass through an NVMe device or use a technology like **vhost-user** to let the guest access an I/O service in the host with minimal overhead.

In summary, **Option 2’s I/O** can be made as fast as Option 1, but it requires more configuration. If using virtio, the tail latency might increase slightly due to the extra copy and VM exit: e.g. virtio-net might add tens of microseconds in worst case. But because the microVM is pinned and not oversubscribed, and virtio interrupts can be handled on an isolated core, the jitter remains bounded. Empirically, Firecracker has been used to run networking functions with only ~30 µs added latency vs. bare metal in exchange for strong isolation[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). If extreme performance is needed, direct device assignment lets the microVM bypass the host for I/O, essentially achieving parity with host-based I/O (just with the device under VM control).

**Security and Fault Isolation:** This is where the microVM design excels. Each microVM is a **true kernel boundary**. Even if a WASM module somehow breaks out of the runtime and compromises the guest OS, it’s still trapped inside a virtual machine. The guest kernel is separate from the host kernel, so host security is maintained (barring vulnerabilities in the hypervisor/KVM). Firecracker, for instance, is designed with a very limited device model and attack surface, making it difficult to attack the host from inside[[27]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=As%20described%20in%20the%20architecture%2C,%5Bsecurity_and_isolation_model.hardware_enforced_isolation%5B0%5D%5D%5B27). The network between host and guest can be restricted (e.g. using vsock or TAP with no external routing). From a **fault tolerance** perspective, if a bug causes a crash, it will ideally only reboot the microVM’s kernel or kill the app inside it, but the host remains up and other microVMs are unaffected. This strong isolation is **essential for multi-tenant scenarios or mixed-criticality systems** – one can even run a safety-critical control loop in one microVM and a non-critical monitoring or logging service in another, on separate cores, without fear that the non-critical code will perturb the critical one. The cost in security is mainly complexity: you have to maintain a minimal OS inside the VM and ensure *its* security as well (patching the guest kernel, etc.). But since the guest OS can be extremely minimal (for example, no open SSH ports, no extraneous services), the attack surface inside the VM is also small.

**Capability Security:** The microVM approach still leverages the WASI capability model within each VM. Each WASM module in a VM gets only handles that the guest OS provides (and the guest OS itself might only have access to certain host resources). One could further combine layers: for instance, run the microVM inside an unprivileged Linux *cgroup* and network namespace on the host to restrict what the VM can reach (like not letting it see the host file system at all, only a virtio-fs mount of a specific directory). This layered sandboxing (WASM sandbox → guest OS restrictions → host cgroup/selinux) yields defense-in-depth. **Side-channel attacks** (like Spectre/Meltdown) are mitigated because the microVM acts as a strong barrier – e.g., flushing or partitioning CPU caches between VMs[[28]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=In%20multi,on%20the%20same%20physical%20core) can be done if needed for high assurance, and disabling SMT on host ensures no two VMs share a core simultaneously[[18]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=guarantees%2C%20hardware%20features%20are%20used%3A,on%20the%20same%20physical%20core).

**Observability and Debugging:** With microVMs, observability becomes a bit more involved. You have to consider debugging inside the VM and the host-VM interaction. The host can still run tracing tools on the **vCPU thread** to see when the VM is scheduled or if there are any stalls. But to trace application events, one might use tracing inside the guest. A strategy is to use something like **LTTng** *within the guest OS* (since it can run on a Linux guest too) for internal events, and maybe use **virtio serial or shared memory buffers** to get data out to the host if needed. Firecracker provides metrics about VM execution that can be gathered without interfering with the guest. Moreover, time-stamping events relative to host and guest clocks needs synchronization (perhaps using host clock reference via a paravirtual clock). Despite these complexities, one can still implement a “flight recorder” approach: for example, let the guest OS trace to a ring buffer in shared memory, and if a deadline miss occurs, the host is signaled (via an exit or an explicit heartbeat mechanism) and can extract the trace buffer for analysis. In terms of formal verification and safety certification, the microVM approach splits the problem into two parts: verifying the *WASM application and runtime* (same as Option 1) and trusting the separation kernel/hypervisor. The KVM hypervisor and Firecracker are not formally verified (though Firecracker’s design is simple), but one could choose to use a formally verified separation kernel (in theory) or at least leverage the fact that hypervisor code is much smaller than an entire OS. The scheduling algorithm can be verified in the guest just as before, and the **safety certification** process can benefit from the strong isolation (for instance, you could certify one microVM to a certain standard independent of others, since interactions are only via well-defined channels). There is precedent in using VMs for mixed-criticality separation in aerospace and automotive. The main drawback is that debugging real-time issues might require looking at two systems (host and guest) simultaneously – e.g. a hiccup might come from a host VM-exit event or from inside the guest; tooling and expertise need to cover both layers.

**Summary – Pros & Cons:** The microVM-based architecture offers **much stronger security and fault isolation** at the cost of a slight increase in latency and complexity. It **trades ~50–100 µs of additional worst-case latency for an almost complete isolation of faults**[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). Tail latencies can still be kept around sub-200 µs range with tuning, which meets the P99.99 requirements for many hard real-time tasks. The approach is highly modular at the system level – each microVM is a module that can be deployed, updated, or restarted independently, which improves fault tolerance (one can hot-restart a crashed RTOS microVM without impacting others or the host). **Deployment complexity** is higher: one needs to build or snapshot a guest OS image for the WASM app and manage the lifecycle of VMs (though with tools like Firecracker API, launching a VM is automated, and a 125 ms boot is usually acceptable for initialization[[25]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=As%20described%20in%20the%20architecture%2C,5MB)). For developers, debugging across VM boundary is harder than a single process. Nonetheless, for **multi-tenant cloud or safety-critical industrial systems**, this approach provides peace of mind that a runaway or malicious module cannot escape its sandbox to harm others – hardware enforces the separation. It hits a sweet spot where each application gets its “own kernel” with minimal overhead, marrying the determinism of RT Linux with the security of VM isolation.

## Option 3: **Bare-Metal or Unikernel WASM RTOS (Minimal OS on Hardware)**

**Architecture Overview:** The third architecture pushes the concept to an extreme: run the WebAssembly runtime *on a minimal OS or directly on hardware*, rather than as a process under a general-purpose OS like Linux. In effect, the WASM runtime **becomes the operating system** for the application, achieving a “bare-metal” deployment. This could be implemented in two ways: (a) as a **unikernel** – a highly specialized kernel that includes the WASM runtime and necessary drivers compiled into a single bootable image, or (b) as a **library on a microkernel** – using a tiny proven microkernel (like seL4 or a minimal RTOS kernel) to provide low-level scheduling and interrupt handling, and running the WASM engine in user space on top of that. In both cases, Linux is removed from the execution path (though Linux might still be present on the side for non-RT tasks or just used during development). The stack thus might look like **[Bare-metal Hardware → Minimal RTOS Kernel (or Hypervisor) → WASM Runtime → Modules]**, with the “kernel” being either the runtime itself or a very thin layer under it.

**Runtime Layering:** In a pure bare-metal approach, one could imagine the **WASM runtime running in supervisor mode** on the CPU, scheduling WASM tasks itself without an underlying OS. This would require the runtime to include or be linked with basic drivers for timers, interrupts, and I/O devices (or to run on top of firmware/BIOS services for some initialization). For example, a Rust-based runtime could use no\_std and access hardware registers directly for, say, an on-board NIC or serial port. This is analogous to how an embedded RTOS works, but here the “user application” is a WASM module. The alternative approach using a microkernel is perhaps more realistic: use a very small, **certified microkernel** (for instance, seL4, which has a formally verified core[[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this)[[30]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=)) to handle memory, threads, and interrupts. Then port the WASM runtime to run as a user process on that microkernel, with the microkernel scheduling it and managing isolation between processes (if any). In that case, one could even run each WASM module as a separate isolated process, and the microkernel mediates their communication. This would provide strong isolation (comparable to microVM, but using a microkernel instead of a hypervisor) with the potential benefit of formal verification of the kernel and simpler worst-case timing analysis. The trade-off is that Linux’s rich features are gone – everything needed must be provided by the new kernel or the runtime itself.

**Scheduling:** If the WASM runtime itself is effectively the OS, it would implement a scheduling strategy to run multiple modules or tasks. This could be a simple cyclic executive or a fixed-priority scheduler tailored to the application. Because no general-purpose OS is in the way, **ultra-low interrupt latency** is achievable: the only scheduling latency is in the minimal kernel (on the order of a few microseconds or less for a well-designed RTOS). For instance, a proven RTOS or microkernel might have a worst-case interrupt latency below 10 µs on typical hardware, and since there’s no Linux scheduler at all, jitter sources are drastically reduced. If the design uses a microkernel like seL4, one could assign time slots (budget) to the WASM tasks using its scheduling policy (seL4 supports fixed-priority or can be extended for EDF). In a unikernel style, one might integrate something like a **Rate-Monotonic scheduler or EDF** directly into the WASM runtime’s loop, possibly using a high-precision hardware timer to trigger task switches. The absence of Linux’s scheduling overhead means **determinism can be extremely high** – theoretically, worst-case scheduling latency could be limited only by hardware IRQ dispatch times and the runtime’s own scheduling algorithm, which can be kept very simple and analyzable. This is attractive for **formal verification**: one could formally prove the correctness and timing bounds of the scheduler because it’s a small codebase (indeed, seL4’s scheduling and interrupt handling has proofs of correctness[[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this)). In summary, scheduling in this architecture can be as optimized as needed: for example, implementing a static cyclic schedule for periodic tasks that guarantees each task a precise start time each cycle (common in high-assurance systems), or using a priority-based scheme with **Interrupt handler threads** that immediately preempt when external events occur.

**Memory Management:** By forgoing Linux, we also forgo virtual memory complexities (unless the microkernel uses them). A bare-metal WASM RTOS might run everything in a single address space, or use an MMU only for isolating components if needed. Memory determinism is easier to assure: one could allocate all needed memory at system startup. If using a microkernel, you would give each module a chunk of physical memory; the microkernel will not swap or overcommit it. In fact, many microkernels (including seL4) have no concept of overcommit – all memory management is explicit, so no unexpected paging can occur. In a unikernel design, you might simply manage memory with a custom allocator that is *non-blocking and constant-time*. For instance, integrating a **TLSF (Two-Level Segregated Fit) allocator** for any dynamic allocation ensures O(1) allocation/free[[31]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=General,189%20cycles)[[10]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,69). You might also lock caches or use scratchpad RAM if available for absolute predictability. Essentially, *the system designer has full control over memory*: you can decide to run everything out of SRAM/locked DRAM, use hugepages if the MMU is on, or even disable the MMU and run purely physical memory addresses (common in microcontrollers). The elimination of a general-purpose OS means **no background memory daemons, no page fault interrupts, no disk swapping** – so worst-case memory latency is just memory access latency of the hardware. The challenge, however, is that one must re-implement or incorporate device drivers that don’t rely on Linux’s paging or dynamic allocations. For example, a network driver on bare metal must handle its own buffers carefully to avoid unpredictable delays (no Linux driver to offload this to). This is doable but raises development effort.

**I/O Strategy:** Without Linux, handling I/O for networking, storage, etc., must be done via either direct hardware access or a simplified I/O framework. One approach is to leverage existing *kernel-bypass libraries in user-space* by porting them to run on the bare-metal environment. For example, **DPDK** could potentially be adapted to run on top of a bare-metal environment (DPDK primarily depends on UIO or VFIO drivers in Linux, but one could provide equivalent functionality if running with full control of hardware). Alternatively, one could use a small dedicated I/O core or an FPGA for network handling, but that goes beyond our scope. If using a microkernel, you might dedicate one process as an “I/O driver” that interacts with hardware and then shares data with the WASM runtime process via shared memory. For instance, on seL4 you could have a NIC driver process and a WASM app process, and they communicate through a shared ring buffer for packets – similar to how user-space drivers work. This is akin to implementing your own minimal version of AF\_XDP: the NIC writes to memory buffers and an interrupt notifies the WASM app process. The latency in such a design can be extremely low – potentially a few microseconds from NIC interrupt to the app seeing the packet, since the path is minimal (no kernel network stack, just a direct memory copy or zero-copy share). Indeed, achieving **sub-10 µs network round-trip** is feasible, as evidenced by specialized RTOS networking stacks or even the hybrid eBPF+WASM approach on Linux that hit 6.5 µs RTT[[12]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,%5Bhybrid_ebpf_and_wasm_architectures.zero_copy_data_path%5B0%5D%5D%5B23). The difference here is our system *is* that specialized stack. For storage, one might use polled-mode drivers or simple asynchronous IO routines that operate on raw disks or flash, ensuring no unpredictable blocking. The lack of io\_uring is made up by the fact we can perform DMA directly and know exactly when completion interrupts fire. That said, implementing these drivers is a **significant engineering task**, and using existing libraries (like reusing portions of RTEMS, Zephyr drivers, or DPDK) is advisable. Another option is to restrict I/O to simpler interfaces (for example, only use UART or simple CAN bus etc., if that’s the application’s need) which are easier to handle with tight timing.

**Security and Isolation:** In a bare-metal or unikernel scenario, we have to consider security carefully. We lose the security blanket of Linux’s process isolation or hypervisor isolation. If the entire system is one address space (for performance), a memory safety bug in the WASM runtime could corrupt other parts of the system. However, WebAssembly’s safety guarantees still apply to modules – a WASM module cannot arbitrarily read/write outside its linear memory, *unless the runtime itself or the embedding code has a flaw*. Given the runtime is likely written in Rust, many classes of memory bugs in the runtime can be avoided, but low-level hardware interfacing might require unsafe code. We can also still leverage hardware features: for instance, even on a microkernel, we could run each WASM module in a separate protection domain (like separate processes with the microkernel ensuring memory isolation between them). seL4, for example, can isolate address spaces and enforce capability-based access even at kernel level; one could map each module’s linear memory into its own address space and use the kernel’s IPC mechanism for them to interact, effectively implementing capability security in a very literal sense. This would mirror the WASI capability model but backed by hardware memory protection. The **capability model** would then be enforced at two levels: WASM-level checks and kernel-level permission grants for resources. This approach can yield **extreme security**: a compromise of one module’s memory doesn’t give access to others or the kernel. The cost is more context switches on communication (which might still be acceptable if carefully managed and if the microkernel is fast).

In the unikernel variant (no microkernel, just one binary), security relies on language safety and possibly MPUs (Memory Protection Units) if available to carve out regions. Some CPU architectures allow setting memory domains or using segmentation to protect certain memory regions from writes, which a unikernel could use to guard critical structures. But generally, a pure unikernel is more vulnerable if it has a bug, since everything is in one trust domain.

Another security aspect is that with no Linux, you have a smaller attack surface (no large syscalls interface, no userland aside from the app). This can be an advantage in certification: an **ultra small trusted computing base**. For example, the seL4 microkernel is ~10k lines and proved secure[[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this), and if the WASM runtime can be kept reasonably sized and verified/tested, the overall system might be easier to certify for things like ISO 26262 or DO-178C safety standards. In fact, this option could be ideal for an **embedded appliance**: the RTOS is packaged as a single binary that boots on the hardware and *only* does the one application. No ability for an attacker to even run other processes or inject code at runtime (since there’s no shell or dynamic loader unless you add one). The system could be locked down to only accept new modules via secure update mechanisms. The flip side is, of course, that updating or patching requires rebuilding the image or having a custom update pipeline. There’s also no inherent support from a big OS for things like ASLR, user accounts, etc., so you’d have to bake in any desired security features.

**Observability and Verification:** Without Linux, standard tooling goes away. We cannot simply attach gdb or run strace. Instead, one might integrate a **serial console logger** or a lightweight tracing system that toggles a GPIO or writes to a memory buffer. Some microkernels have tracing frameworks, or one could port something like LTTng to run on the microkernel if it supports user programs. But often, bare-metal RTOS debugging relies on JTAG hardware debuggers or custom logging. To minimize interference, one might run the system with a periodically dumping high-resolution timer log that records timestamps of important events (like context switches, I/O events) to memory and then post-mortem examine it. This is similar to flight recorder but now implemented from scratch or using a minimalist library. On the formal verification side, **Option 3 is the most amenable to full verification**. Because this design can be made very simple and closed-world, one can construct formal models of the scheduler, the communication, and even use existing proofs (if using seL4, many proofs of isolation and correctness are provided[[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this)[[30]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=)). One could envision proving that the scheduling of WASM tasks meets deadlines under certain assumptions (using e.g. RTA - Response Time Analysis formulas or model checking as was done for seL4’s scheduling). The complexity of Linux made full verification infeasible, but a custom RTOS of limited scope might achieve at least a *proof of absence of certain runtime errors* and verified worst-case execution time for critical sections. There is precedent: high-assurance systems in aerospace often use custom kernels or seL4 for exactly this reason[[32]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=support%2C%20creating%20vendor%20lock,purpose%20operating%20systems%20like%20Linux)[[33]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%28higher%20rate%20%3D%20higher%20priority%29,Time%20Analysis%20%28RTA%29.%20%5Breal_time_scheduling_and_synchronization.suitable_scheduling_algorithms%5B0%5D%5D%5B62). In terms of tail latency measurement, lacking perf or similar, one could build in periodic checks or use hardware performance counters if available to sample latency. Another idea is to run the system under simulation (like QEMU or gem5) with instrumentation to gather detailed timing information as part of verification/testing.

**Trade-offs:** The bare-metal approach is **highly novel and custom**. Its primary benefit is *maximum determinism and minimal overhead*. There is no Linux scheduler noise, no virtualization overhead – the only sources of jitter are hardware and the code we write. Tail latencies could potentially drop to the tens of microseconds reliably, and jitter might be reduced such that even P99.999 is within a tight bound (because we removed many unpredictable elements). It also yields a small trusted base which is good for security in theory (less code to audit) and great for **specialized deployments** (e.g. an industrial robot controller delivered as a standalone box that boots directly into the control loop code). However, the **development and maintenance cost is highest**: every OS service we need (drivers, file systems, networking stack, etc.) has to be provided in some form. We lose the huge ecosystem of Linux – meaning fewer ready-made tools for things like USB, complex networking protocols, etc. Another downside is **reduced flexibility**: while Linux can run other tasks in parallel (logging, management, etc.) on other cores, a bare-metal system running the RTOS can’t easily have those luxuries unless we dedicate another core to a separate environment (which starts to resemble a hybrid with Linux anyway, like Xenomai co-kernel approaches). In pure bare-metal, adding non-real-time functionality can interfere unless carefully partitioned. For composability, this approach can still use the WASM component model, but likely at *build time* – you might link multiple modules into the image or load them from a simple flash filesystem at boot. Dynamic loading is possible if you include a loader, but often static linking is chosen for simplicity and safety. So, updating one component might require rebuilding the whole image, unless the runtime supports loading new WASM modules from, say, a network update. **In summary, Option 3 sacrifices general-purpose convenience for *ultimate predictability***. It could be considered for the most demanding use-cases (e.g. a *WASM-based motor controller* that must never exceed a 50 µs loop jitter, or a military-grade system where Linux is not certifiable). It is an **innovative architecture** that aligns with the trend of unikernels and mixed-criticality separation (akin to running the critical part on a separate bare-metal core). But it implies re-inventing portions of the OS stack in Rust/WASM context. This option would likely be pursued only when Options 1 or 2 cannot meet the requirements for latency or certification.

## Trade-Off Analysis of the Options

Each of the above architectures meets the core goal of running real-time WASM workloads with microsecond-level precision, but they differ radically in complexity, isolation, and use of existing technology. Option 1 (host process) is essentially **“RTOS in a process”**, leveraging Linux heavily. It is easiest to build and integrate with (since Linux provides scheduling, drivers, and standard WASI support), but it relies on soft isolation. Option 2 (microVM) introduces a **hybrid approach** that still uses Linux for low-level management but puts each RTOS instance in its own mini-VM – a balance of strong isolation with moderate overhead. Option 3 (bare-metal) is a **clean-slate approach**, optimizing absolutely for determinism at the cost of reimplementing OS functionality.

The trade-offs can be viewed along key axes:

* **Latency:** All options strive for low latency, but Option 3 has the least overhead beyond hardware limitations, whereas Option 2 adds a small overhead for virtualization. Option 1 can be nearly as fast as Option 3 in many cases on a tuned system, but may have slightly more jitter due to sharing the kernel with other activities (e.g., a host daemon unexpectedly running on an “isolated” core, if misconfigured).
* **Security/Isolation:** Option 2 is the strongest for isolation – effectively each workload is as isolated as if on separate machines[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). Option 3 can also isolate modules if using a microkernel, but if implemented as a single address space unikernel, it’s the weakest in isolation (everything runs together, aside from WASM sandboxing). Option 1 is moderate: processes and threads on Linux with WASM sandboxing – good against well-behaved bugs, but not if the threat model includes kernel exploits or hostile co-tenants.
* **Complexity:** Option 1 wins on simplicity and development speed. It uses familiar Linux interfaces and one can utilize standard containers or systemd to manage the RTOS process. Option 2 adds complexity in packaging the application into a VM image and coordinating host/guest configuration (two OS environments to tune). Option 3 is most complex, essentially building a special-purpose OS. It likely requires the most engineering effort and has the least existing support, though frameworks like seL4 or some unikernel toolkits could bootstrap the process.
* **Modularity & Composability:** If the goal is to mix and match many small components, Option 1 is very appealing because the WASM component model works best in-process (low overhead calls, memory sharing possibilities). Option 2 forces a coarser modularity – components in different VMs are more isolated but much harder to connect (you’d treat them as services). Option 3 can be designed to allow in-process components (like Option 1, if we run multiple modules on one kernel) or separate processes (like Option 2, if using a microkernel), but dynamic composability is generally lower because adding a new component might require system recompilation or complex bootstrapping. In summary, Option 1 offers **high composability** in software, Option 2 offers **high modularity** in deployment (each service in its own box), and Option 3 offers **high integration** (everything in one tight loop, which is almost the opposite of modular – good for single-function systems).
* **Deployment and Ecosystem:** Option 1 and 2 both build atop Linux – meaning you can take advantage of existing drivers, networking, and tooling. Option 3 throws much of that away; deploying Option 3 might involve custom hardware images and no compatibility with standard user-space apps. For example, in Option 2 you could still log in to the host or VM (in a debug mode) to troubleshoot, whereas Option 3 might only offer a serial console. From an ecosystem perspective, Option 1 can reuse cloud-native tech (containers, orchestration) albeit with special tuning, Option 2 aligns well with modern virtualization and cloud isolation practices, and Option 3 aligns more with embedded systems practices.

To make these differences concrete, the table below compares the architecture choices across key metrics:

## Comparison of Architecture Options

| **Metric** | **Option 1: Host Process (Linux + WASM Runtime)** | **Option 2: MicroVM Isolation (Linux + Firecracker)** | **Option 3: Bare-Metal/Unikernel WASM** |
| --- | --- | --- | --- |
| **Worst-Case Latency** <br> (Tail Predictability) | **Low latency**, near bare-metal performance. With PREEMPT\_RT, scheduling jitter can be ~124 µs max on host[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22); minimal overhead (~10 µs) for WASM runtime. Tail latency ~150 µs achievable with tuning. | **Moderate latency** (slightly higher). Adds ~ tens of µs VM overhead[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). Still typically <200 µs tail if pinned. Extra virtio I/O overhead for networking (~10–50 µs). | **Lowest latency**. Virtually no OS overhead beyond hardware IRQ handling. Potential to keep P99.99 below 100 µs easily. Jitter sources minimized; latency essentially determined by hardware and minimal kernel (few µs). |
| **Security & Isolation** | **Basic isolation**. WASM sandbox and Linux process isolation protect modules, but shares host kernel. Larger attack surface (host OS) and weaker fault isolation – a bug can crash the process (though not the whole OS). Suitable for single-tenant or trusted code. | **Strong isolation**. Each RTOS runs in its own VM with its own kernel – a crash or compromise is contained[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). Host attack surface reduced 95% for each app. Ideal for multi-tenant or mixed-criticality (faults can’t propagate). Relies on hypervisor security, but Firecracker’s footprint is small. | **Variable isolation**. If using a verified microkernel, can isolate modules strictly (like separate processes with proven memory separation). In a single-binary unikernel, isolation is weakest (all in one address space, trust on Rust/WASM safety). No separate host/guest boundary – smaller overall codebase but no second line of defense if sandbox is breached. |
| **Modularity & Composability** | **High software composability**. Multiple WASM modules can run in one process and use the WASI Component Model for function calls. Easy in-process communication (function calls, shared memory) – good for complex apps composed of many components. Adding modules is straightforward (load new WASM). However, all modules run together, requiring cooperative scheduling. | **High system modularity, lower coupling**. Encourages splitting application into services per VM. Components in different VMs communicate via IPC (network, shared mem), which is higher latency and complexity. Composition is at system level (microservice style) rather than in-process. Within a VM, can still compose modules freely, but often one module per VM for isolation. | **Limited / static composability**. Designed more for a single-purpose image. Modules can be linked in at build time or loaded at boot, but dynamic runtime composition is harder. If microkernel used, components can be separate processes with message-passing (similar to microservices, but on a tiny kernel). In a pure unikernel, everything is linked into one program (monolithic but very optimized). Overall, less flexible in adding/swapping modules on the fly. |
| **Deployment Complexity** | **Low**. Deploy as a normal Linux application (or container) with special tuning. No custom OS image needed – uses standard Linux distribution with PREEMPT\_RT. Leverages familiar deployment tools (systemd, Docker, etc). Easiest to develop and debug with standard tools. | **Medium**. Need to manage a guest OS image (kernel + rootfs) for the microVM. Deployment involves launching VMs (could be via an API or script). Still runs on Linux host, but must configure both host and guest. More moving parts (coordinate cores, memory assignment, virtio devices). | **High**. Requires building a bespoke runtime OS. Deployment might involve flashing an image to hardware or using a custom bootloader/hypervisor. No standard package manager – the whole system is the app. Updates and debugging require specialized process (rebuild OS or use JTAG/monitor). Essentially deploying an embedded system rather than an app. |
| **Security Features** <br> (and Safety Certification) | Leverages Linux security (SECComp, namespaces, SELinux) and WASI caps for least privilege[[2]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20foundation%20of%20the%20security,nano). Easier to reuse existing certifications of Linux (e.g. progress via ELISA for ISO 26262). Larger TCB (Linux ~ millions of LOC). Safety-critical certification is moderate effort due to complexity. | Each VM provides an extra security boundary (hypervisor). Can meet high isolation requirements (equivalent to separate physical machines in many respects). AWS Nitro/Firecracker has seen real-world hardening. Certification could compartmentalize: certify critical VMs independently. Slightly smaller TCB per VM (dedicated small guest OS + hypervisor). | Minimal TCB, which can be rigorously audited. Possibly use formally verified kernels (seL4)[[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this) to achieve high assurance. Suitable for systems that need provable guarantees. However, missing POSIX/Linux means new certification effort for the custom OS. If done, offers highest confidence in correctness (down to proven scheduler, memory access, etc.). |

**Legend:** *TCB = Trusted Computing Base (the code that must be trusted).*

## Conclusion

Each architecture option provides a distinct balance of determinism, security, and complexity for a next-generation WASM-based RTOS. A **direct host-process design** (Option 1) is attractive for its simplicity and raw performance on a real-time Linux kernel, making it a strong choice when the environment is controlled (single tenant, known workload) and quick integration is needed. The **microVM-based design** (Option 2) sacrifices a tiny bit of latency to gain heavy-weight isolation; it shines in cloud or multi-tenant industrial settings where sandboxing each real-time service is worth the overhead for fault containment and defense-in-depth security[[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27). Finally, the **bare-metal/unikernel approach** (Option 3) represents a bleeding-edge path that maximizes predictability and minimizes overhead by stripping away general-purpose OS layers – at the cost of significantly higher development effort and reduced flexibility. This could be justified in niche scenarios like certified control systems or ultra-low-latency devices where every microsecond and every line of code in the TCB counts.

In practice, a hybrid strategy might also emerge: for example, running a few microVM-isolated real-time WASM services on a PREEMPT\_RT Linux host (combining Option 2 and 1), or using Linux for most functionality but offloading a critical real-time loop to a bare-metal core (a dual-OS approach). The decision comes down to the required **latency guarantees, security needs, and engineering resources**. What’s clear is that modern WASM runtimes and the surrounding ecosystem (WASI, real-time Linux, io\_uring, DPDK, etc.) give architects a powerful toolbox. By carefully selecting an architecture, one can indeed achieve **P99.99-grade microsecond determinism** on commodity hardware, while also benefiting from WebAssembly’s safety and portability. Each option outlined above is a viable path, and choosing between them involves trading off ultimate performance vs. robustness vs. time-to-market – a decision that must be informed by the specific application domain and requirements.

[[1]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,22) [[2]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20foundation%20of%20the%20security,nano) [[3]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,time%20deadlines.%20%5Bmult) [[4]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,Model%20to%20manage%20the%20boundary) [[5]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=4,Guarantees) [[6]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AWASI,budgets%20on%20sandboxed%20WASM%20code) [[7]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2ADeterministic%20memory%20wins%2A%2A%3A%20Non,0%20%C2%B5s) [[8]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2ADeterministic%20memory%20wins%2A%2A%3A%20Non,based%20interruption%20provides) [[9]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=7,Purpose%20Allocator%20Slowdowns) [[10]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,69) [[11]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%2A%20%2A%2AHidden%20tail,io_uring) [[12]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,%5Bhybrid_ebpf_and_wasm_architectures.zero_copy_data_path%5B0%5D%5D%5B23) [[13]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=The%20efficiency%20of%20this%20hybrid,between%20kernel%20and%20user%20space) [[14]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,workloads%20with%20unpredictable%20traffic%20bursts) [[15]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=performance%2C%20but%20with%20high%20CPU,resourced.) [[16]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,as%20there%20are%20CPUs%20handling) [[17]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,BQL%29%20to%20prevent%20bufferbloat) [[18]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=guarantees%2C%20hardware%20features%20are%20used%3A,on%20the%20same%20physical%20core) [[19]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=In%20multi,to%20prevent%20two%20workloads%20from) [[20]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,time%20systems) [[21]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=9.2%20Wasm,and%20rr%20as%20a%20Fallback) [[22]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,78) [[23]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,83) [[24]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=,27) [[25]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=As%20described%20in%20the%20architecture%2C,5MB) [[26]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=guarantees%2C%20hardware%20features%20are%20used%3A,on%20the%20same%20physical%20core) [[27]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=As%20described%20in%20the%20architecture%2C,%5Bsecurity_and_isolation_model.hardware_enforced_isolation%5B0%5D%5D%5B27) [[28]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=In%20multi,on%20the%20same%20physical%20core) [[29]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%5Bformal_verification_and_testing.formal_specification%5B7%5D%5D%5B77%5D%20,formally%20verified%20seL4%20microkernel%2C%20this) [[30]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=) [[31]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=General,189%20cycles) [[32]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=support%2C%20creating%20vendor%20lock,purpose%20operating%20systems%20like%20Linux) [[33]](file://file-BudZ41uXU1NFscM2LSKk33#:~:text=%28higher%20rate%20%3D%20higher%20priority%29,Time%20Analysis%20%28RTA%29.%20%5Breal_time_scheduling_and_synchronization.suitable_scheduling_algorithms%5B0%5D%5D%5B62) NEXT-GEN WASM-RTOS\_ How Rust + WASI Can Deliver P99.99-Grade Determinism on Commodity Linux Hardware.md

<file://file-BudZ41uXU1NFscM2LSKk33>