next-rc: A Multi-Runtime Architecture for Full-Stack Agentic AI Deployment

Samuel Joseph Troyer
Rizome Labs, Inc.
sam@rizome.dev

Abstract

We present **next-rc** [1] (Next.js Runtime Controller), an open-source multi-runtime execution architecture that enables full-stack deployment of agentic AI systems requiring dynamic code execution. Unlike traditional container-based approaches that impose prohibitive latency penalties, next-rc provides a unified abstraction over four specialized runtime environments: eBPF for kernel-level operations, WebAssembly for sandboxed computation, V8 Isolates for JavaScript execution, and a Python runtime for ML workloads.

The architecture's core innovation lies in its modular design that allows runtime selection based on workload requirements, a unified security model with capability-based permissions, and seamless integration with Next.js applications. By providing a consistent API across heterogeneous execution environments, next-rc enables developers to build agentic applications that were previously impractical due to execution overhead. The system represents a fundamental shift in how we approach code execution for AI agents, moving from monolithic containers to purpose-built, composable runtimes.

1 Introduction

The emergence of sophisticated, code-generating AI agents marks a paradigm shift in software development and automation. These agents, capable of complex reasoning and autonomous action, require a new class of *agent infrastructure* to mediate their interactions with the digital world. Current infrastructure, largely reliant on containerization technologies like Docker, imposes prohibitive latency and resource overhead, fundamentally limiting the potential of agentic AI. The full-stack deployment of these systems, from user interface to backend logic, demands a more dynamic, efficient, and secure approach to code execution.

We introduce **next-rc**, a multi-runtime execution controller that provides the foundational infrastructure for full-stack agentic AI systems. Our work is predicated on the understanding that agent-generated code is not monolithic; it is a diverse collection of tasks, each with unique performance, security, and resource requirements. By providing a unified abstraction over four specialized runtimes—eBPF, WebAssembly, V8 Isolates, and a dedicated Python environment—next-rc enables developers to seamlessly execute agent-generated code in the most appropriate context. This approach moves beyond the one-size-fits-all model of containerization, offering a purpose-built, composable, and deeply integrated solution for the next generation of AI-powered applications.

2 Background and Related Work

2.1 Evolution of Agentic AI Systems

Recent advances in LLMs have enabled increasingly sophisticated agent architectures [2]. These systems require not just generation of code, but its immediate execution within appropriate security boundaries. The infrastructure requirements for such systems are fundamentally different from traditional software deployment [3].

2.2 Execution Environments

Several approaches to lightweight code execution have emerged:

V8 Isolates: Cloudflare Workers [4] demonstrated that V8's isolation capabilities could provide sub-millisecond startup times for JavaScript workloads. This approach trades language flexibility for performance.

WebAssembly: WASI [5] provides a capability-based security model for executing compiled code. While more flexible than V8 Isolates, WASM requires ahead-of-time compilation.

eBPF: Originally designed for packet filtering, eBPF has evolved into a general-purpose in-kernel virtual machine. Its verifier ensures safety but limits expressiveness.

Containers: Technologies like Docker [6] and Firecracker [7] provide strong isolation but with significant startup overhead.

Our work synthesizes these approaches into a unified architecture specifically designed for agentic AI workloads.

3 System Architecture

3.1 Runtime Environments

Each runtime serves a specific purpose within the ecosystem:

3.1.1 eBPF Runtime

The eBPF runtime executes verified bytecode within kernel space. It is designed for:

- High-frequency, low-latency operations
- Data filtering and transformation
- System monitoring and observability

The runtime leverages eBPF's built-in verifier to ensure safety without runtime overhead.

3.1.2 WebAssembly Runtime

Built on Wasmtime, this runtime executes compiled WASM modules. It targets:

- Compute-intensive algorithms
- Multi-language support (C/C++, Rust, Go)
- Deterministic execution requirements

The WASI interface provides capability-based security for system resources.

3.1.3 V8 Isolate Runtime

This runtime provides isolated JavaScript execution contexts. It excels at:

- Dynamic JavaScript/TypeScript code
- Web API compatibility
- Rapid prototyping and iteration

Memory sharing between isolates reduces overhead for concurrent executions.

3.1.4 Python Runtime

A hybrid approach combining PyO3 bindings with selective WASM compilation. Designed for:

- Data science and ML workloads
- Library ecosystem compatibility
- Gradual performance optimization

Critical paths can be compiled to WASM while maintaining Python compatibility.

3.2 Runtime Selection

The scheduler selects runtimes based on multiple factors:

1. **Language Detection**: Automatic identification of source language
2. **Performance Requirements**: Latency constraints and throughput needs
3. **Security Context**: Trust level and required isolation
4. **Resource Availability**: Current system load and runtime availability

3.3 Security Model

Security is enforced at multiple levels:

3.3.1 Capability-Based Permissions

Each execution context is granted specific capabilities:

```
{
  filesystem: { read: ["./data/*"] },
  network: { allowedHosts: ["api.example.com"]
    },
  memory: { limit: "128MB" },
  execution: { timeout: 5000 }
}
```

3.3.2 Runtime Isolation

Each runtime provides its own isolation mechanism:

- eBPF: Kernel verification
- WASM: Memory sandboxing
- V8: Process isolation
- Python: Namespace separation

3.3.3 Trust Levels

Three trust levels determine available capabilities:

- **Untrusted**: Minimal permissions for unknown code
- **Verified**: Extended permissions for validated sources
- **Trusted**: Full access with audit logging

4 System Interactions

The power of next-rc lies in the fluid interaction between its core components. The following describes the lifecycle of a typical execution request:

1. **Request Initiation:** An execution request is initiated from the application layer, typically a Next.js application. The request contains the code to be executed, the language it's written in, and any specific execution requirements (e.g., performance constraints, security context).
2. **Intelligent Scheduling:** The **Runtime Controller** receives the request and passes it to the **Intelligent Scheduler**. The scheduler analyzes the request, considering factors like language, code complexity, and resource requirements. It also consults its historical performance data to make an informed decision.
3. **Runtime Selection:** Based on its analysis, the scheduler selects the optimal runtime for the task. For example, a simple, low-latency task might be routed to the eBPF runtime, while a complex data analysis script would be directed to the Python runtime.
4. **Security Context Establishment:** The **Security Manager** creates a secure execution environment for the request. This involves setting up the appropriate isolation level (e.g., V8 Isolate, WASM sandbox), and applying capability-based permissions to restrict access to system resources.
5. **Code Execution:** The selected runtime executes the code within the secure context. The **Resource Manager** monitors the execution, ensuring it doesn't exceed its allocated resources.
6. **Result and Teardown:** Upon completion, the execution result is returned to the application layer. The **Security Manager** then tears down the execution environment, ensuring no resources are leaked.

This entire process is designed to be transparent to the developer, who interacts with a simple, unified API. The complexity of runtime selection, security management, and resource allocation is handled automatically by the system.

5 Design Decisions and Trade-offs

5.1 Runtime Diversity vs. Complexity

Supporting multiple runtimes increases system complexity but provides significant performance benefits. We manage this through:

- Clear abstraction boundaries
- Comprehensive testing across runtimes

- Consistent error handling and reporting

5.2 Performance Characteristics

Each runtime exhibits a distinct performance profile. The choice of runtime has a significant impact on both latency and cost, especially in a serverless environment. We consider two key metrics: *cold start* time, the time it takes to initialize a new execution environment, and *warm start* time, the time it takes to execute code in a pre-warmed environment.

- **eBPF:** Offers the lowest cold start times, measured in nanoseconds. This makes it ideal for high-frequency, low-latency tasks.
- **WebAssembly:** Provides fast cold starts, typically in the microsecond range. It's well-suited for compute-intensive tasks that can be compiled ahead of time.
- **V8 Isolates:** Have cold start times in the millisecond range. While slower than eBPF and WebAssembly, they offer excellent performance for dynamic languages like JavaScript.
- **Python Runtime:** Has the highest cold start times, measured in tens of milliseconds. However, it provides access to a rich ecosystem of libraries for data science and machine learning.

Performance can also vary significantly between identical runs of the same function. Our intelligent scheduler mitigates this by learning from historical performance data and selecting the most reliable runtime for a given task.

6 Conclusion

next-rc represents a fundamental rethinking of code execution infrastructure for agentic AI systems. By providing purpose-built runtimes accessible through a unified interface, we enable a new generation of full-stack AI applications that seamlessly integrate code generation with execution.

The modular architecture allows the system to evolve with the rapidly changing landscape of AI agents while maintaining a stable developer experience. As agentic AI systems become more prevalent, we believe specialized execution infrastructure will become a critical component of the AI stack.

The open-source release of next-rc aims to accelerate innovation in full-stack agentic AI deployment. We invite the community to build upon this foundation and explore new possibilities for AI-powered applications.

Acknowledgments

We thank the Cloudflare Workers team for pioneering V8 Isolate technology, the Wasmtime maintainers for their excellent WebAssembly runtime, and the eBPF community for expanding the boundaries of safe kernel programming.

References

[1] Rizome Labs. *next-rc: Multi-Runtime Execution Controller*. `https://github.com/rizome-dev/next-rc`, 2025.

[2] K. Chan et al. *Agentic AI Needs a Systems Theory*. arXiv:2503.00237, 2025.

[3] S. Marro et al. *Infrastructure for AI Agents*. arXiv:2501.10114, 2025.

[4] Cloudflare. *How Workers Works*. `https://developers.cloudflare.com/workers/learning/how-workers-works/`, 2024.

[5] Bytecode Alliance. *Wasmtime: A fast and secure WebAssembly runtime*. `https://wasmtime.dev`, 2024.

[6] Docker, Inc. *Docker: Accelerate how you build, share, and run applications*. `https://docker.com`, 2024.

[7] A. Agache et al. *Firecracker: Lightweight Virtualization for Serverless Applications*. NSDI 2020.

[8] Linux Foundation. *eBPF: Dynamically program the kernel*. `https://ebpf.io`, 2024.

[9] PyO3 Project. *PyO3: Rust bindings for Python*. `https://pyo3.rs`, 2024.