Remote Direct Code Execution

Yibo Huang, Yiming Qiu[†], Daqian Ding[‡], Patrick Tser Jern Kon, Yiwen Zhang
Yuzhou Mao, Archit Bhatnagar, Mosharaf Chowdhury
Srinivas Devadas[⋆], Jiarong Xing[§], Ang Chen

University of Michigan [†]*The University of Hong Kong* [‡]*Shanghai Jiao Tong University*
[⋆]*Massachusetts Institute of Technology* [§]*Rice University*

Abstract

We propose *remote direct code execution (RDX)*, which elevates the power of RDMA from memory access to code execution. We target runtime extension frameworks such as Wasm filters, BPF programs, and UDF functions, where RDX enables an *agentless architecture* that unlocks capabilities such as fast extension injection, update consistency guarantees, and minimal resource contention. We outline the roadmap for RDX around a new *CodeFlow* abstraction, encompassing programming remote extensions, exposing management stubs, remotely validating and JIT compiling code, seamlessly linking code to local context, managing remote extension state, and synchronizing code to targets. The case studies and initial results demonstrate the feasibility of RDX and its potential to spark the next wave of RDMA innovations.

CCS Concepts

• **Hardware → Networking hardware**; • **Networks → Data center networks**; **Programmable networks**; **Programming interfaces**.

Keywords

RDMA, runtime extensions, agentless, cloud infrastructure, eBPF, service mesh

ACM Reference Format:
Yibo Huang, Yiming Qiu[†], Daqian Ding[‡], Patrick Tser Jern Kon, Yiwen Zhang, Yuzhou Mao, Archit Bhatnagar, Mosharaf Chowdhury, Srinivas Devadas[⋆], Jiarong Xing[§], Ang Chen . 2025. Remote Direct Code Execution. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772397

1 Introduction

Remote Direct Memory Access (RDMA), initially developed for high-performance computing, has since expanded well beyond the boundaries of HPC and revolutionized modern datacenters. Bypassing the OS kernel and CPU, RDMA allows networked hosts to access each other's memory at hardware speeds, while riding upon conventional Ethernet substrates (i.e., RoCE) with high compatibility. Our community now has over a decade of innovation in the use of RDMA within datacenters: accelerating memory-intensive workloads [41, 47] and storage services [20, 39], disaggregated remote memory [42] and database systems [38, 64], efficient network telemetry [48] and system monitoring [58], and memory forensics for malware detection [49, 53, 54]. The ability to directly operate on data in a remote node's memory, without its software involvement, has proven to be a powerful primitive.

However, despite such tremendous progress, we believe that RDMA still holds vastly-untapped potential that could unlock the next wave of innovation. In this paper, we make a case for *elevating the power of RDMA from memory access to code execution*, whereby one node can inject and execute code on another remote node via RDMA. The intuition is that *code is data*, so it should be possible to inject a compiled binary to a machine's memory over RDMA, and attach the code to some software entity (e.g., a predefined sandbox) for execution. Like classic RDMA operations, this new task is performed by the RNIC, with negligible overhead. It does not require any modification to RDMA stack (e.g., OS, libraries, or driver), and is transparent to the remote node. We call this new primitive Remote Direct Code Execution (RDX).

We envision RDX to enable a wide variety of use cases, and as a start, we demonstrate one such application: executing *runtime extensions*, such as service mesh Wasm filters [44, 67], kernel- or user-level BPF [31, 65], or user-defined functions (UDFs) [23, 28, 51]. These frameworks allow custom logic to be expressed in some constrained language, then validated and compiled just-in-time to execute in a sandboxed environment. Compared to ahead-of-time programming with static compilation, runtime extensions can be loaded and unloaded on-the-fly based on workload characteristics, and they can be applied in a modular manner without application recompilation or service downtime. Requests are routed to dedicated sandboxes, where the extension logic is executed safely. Runtime extensions provide diverse benefits for security defenses [29, 45, 57], performance acceleration [13, 26, 37, 66], as well as monitoring and telemetry [15, 55].

To enable automation at scale, existing runtime extensions typically adopt an *agent-based architecture*, where a central controller maintains a repository of runtime extensions and

HotNets '25, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772397

transmits the desired extension logic to cluster nodes over the network. Each node is equipped with local agents that verify and validate the logic, execute just-in-time (JIT) compilation, and incorporate it into sandboxes. This design leads to several inherent limitations in practice:

- *Injection delay:* Agent operations (e.g., compiling, validating, and loading the extension) inevitably introduces CPU processing overhead on each local host. Our evaluation shows that this overhead introduces at least millisecond-level injection delay, making it impossible to support use cases that require *microsecond-scale reactions*, such as short-lived per-query UDF extensions [19, 23, 27] and rapid auto-scaling in serverless computing [35, 46, 60].
- *Update inconsistencies:* Rolling out a collection of interdependent extensions across a chain of interconnected hosts [50] introduces lengthy periods of inconsistencies to the data path, where requests experience disruptions due to the mix of old and new extension logic under an eventual consistency model [11, 12, 18, 30]. In practice, developers rely on manual planning and testing to avoid inconsistencies, but this is by itself an error-prone process [3, 6, 8]. Theoretically, we can buffer all requests during the update to eliminate inconsistencies [32], but the long update injection time requires an impratically large buffer.
- *Resource contention:* The sharing of hardware resources between control and data paths further introduce mutual *resource contention* to each other [10, 24, 43, 67]. The control path extension injection operations experience significant slowdown under heavier data path workloads, and the data path also faces service degradation during control operations. We also observe that the contention could lead to safety hazards by hampering the live patching/rollback capabilities of the control path, leading to *lockout effects* (i.e, stalled operations) when the contention level is high.

The key benefit that RDX brings to runtime extensions is a drastically different, *agentless architecture*, enabled by one-sided RDMA operations. By this, we mean that the cluster nodes no longer have to run local software agents. With RDX, we can restructure each remote extension framework into (1) a *remote control plane* that consolidates the functionality of existing local agents, acting as a centralized authority to oversee the entire runtime lifecycle of extensions; and (2) *local data planes* running on each hosts, dedicated to executing extension logic. The control plane transparently manages the data plane through a set of RDMA code manipulation primitives, completely bypassing the CPU of local hosts. This mitigates the limitations of existing approaches by enabling micro-second scale extension injection, offering strong update consistency guarantees, and minimizing resource contention.

The design of this agentless system raises a set of technical challenges, including how to align with the programming model of existing frameworks, how to expose local host information to the remote control without the help of agents, how to compile and validate extension logic out of the local hosts, how to correctly load the extension logic into host contexts, and how to manage extensions with stateful behaviors and external function calls. We outline our technical roadmap to resolve these challenges via a unified programming abstraction called *Codeflow*, along with initial results showing the feasibility and benefits of our approach.

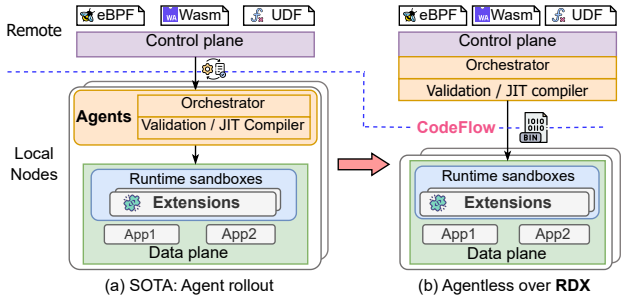


Figure 1: State-of-the-art vs. *agentless* design with RDX.

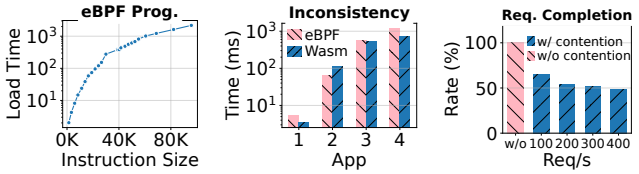
2 Motivation

In this section, we provide more background on runtime extensions, further motivate our problem, and outline the workflow of RDX.

2.1 Background: Runtime extensions

Runtime extensions, such as Wasm filters [44], eBPF programs [31, 65], and UDFs [23], are widely adopted in cloud data centers [56]. As shown in Figure 1(a), a runtime extension is a piece of executable logic that the control plane can load onto the data plane, without the need for draining traffic or restarting processes. Take service mesh as an example, which supports two distinct types of runtime extensions: (1) Wasm filters to customize application-level, request-aware policies (e.g., L7 routing), and (2) eBPF programs [15, 55] to support low-level network and security controls (e.g., mTLS termination). Regardless of extension types, existing frameworks typically involve two phases, namely control path injection and data path execution. In the injection phase, extensions written in high-level languages are validated (or verified) via static checks, then compiled to binary code using JIT, and eventually attached onto isolated sandboxes such as containers [14] or lightweight VMs [25, 36, 52, 59]. In the execution phase, the data path requests trigger the execution of runtime extension logic to implement the desired functionality.

Agent-based runtime extension frameworks are thus far the norm, and they rely on on node-local agents to manage extensions at scale. Concretely, each server runs a user-space daemon that retrieves configuration from a control plane (e.g., Kubernetes [9]) and handles deployment orchestration, access control, control-path injection, and extension state access. In containerized clouds, agents are often deployed per pod for isolating different workloads. For example, Istio users a per-pod agent [7] to load Wasm filters in service mesh. Likewise, eBPF also relies on user-level and kernel control agents [21] to manage data plane extension injection and removal.



(a) eBPF program injection overhead **(b) Update inconsistency duration** **(c) Application perf. degradation**

Figure 2: Motivating examples: (a) the relationship between program complexity and injection overhead; (b) the inconsistent time in eBPF and Wasm-based extensions, on four apps (1-4) with 4, 11, 17, and 33 microservices; (c) the impact of control path contention (updates injected per 10 second) on data path request completion.

2.2 Challenges: Runtime extension injection

Despite their popularity, modern agent-based runtime extension frameworks still face fundamental challenges.

Observation #1: *Microsecond-scale runtime extension injection is essential for various cloud-native applications; today, this is hard to achieve due to local CPU overhead.* Concretely, microsecond-scale injection is crucial for *fast auto-scaling* in serverless platforms [46, 60] and *per-query extensions* in data processing pipelines [19, 23], where the speed of extension injection has to match that of serverless function live migration and data processing queries at the microsecond scale. Figure 2a shows that extension injection time in existing frameworks is at the millisecond level even for small pieces of extension logic. Our profiling further attributes this delay to the compilation and validation overhead on local host CPU, which amounts to 90+% of overall loading time.

Observation #2: *Safety-critical runtime extension updates should avoid introducing inconsistencies; today, there is no principled approach to offering consistency guarantees.* We define update inconsistency time as the interval between the initiation and completion of runtime extension update. This interval exists due to both *intra-agent* and *inter-agent* dependencies: Within each local agent (e.g., service mesh sidecar or eBPF controller), pieces of extension logic can be attached to multiple locations (e.g., filters or hook points) as a dependency chain. Across local agents in a cluster, extension logic further forms a directed acyclic graph (DAG) structure according to inter-host dependencies. Figure 2b shows that the inconsistency could span hundreds of milliseconds even for applications with less than 20 microservices.

Update inconsistency has safety implications [4, 5]. If service B depends on A, but an extension update reaches B before it reaches A, then calls from A to B could fail during the update. In practice, developers have to rely on manual planning and testing to avoid such inconsistencies [3, 6, 8]. A more automated approach is to buffer all incoming requests before the update completes, but this is impractical given the time scale—if an application supports 10M req/sec [22], then a 100ms update interval would require buffering 1M requests, which is infeasible given resource and platform limits [2].

Observation #3: *The performance of control path injection and data path execution should not be affected by each other; today, they contend heavily due to shared resources.* Service mesh advocates separation of concerns by offloading operational complexities from application code to a dedicated infrastructure layer, so in principle its control path should be engineered to minimize interference with application-level requests. In reality, however, application requests and extension injection contend with each other over shared CPU/memory resources, especially when the request load is high.

As shown in Figure 2c, when the server CPUs are almost saturated with request load, application request completion rate could be halved during extension injection, leading to severe SLO degradation. The major overhead of injecting extensions in agent-based frameworks is from code validation, which uses CPU-intensive static analysis to check for correctness and safety. This overhead will be further exacerbated in cloud production due to high-density agent deployment. Apart from injection, periodic access to the injected extension's data structures for runtime state also introduces non-negligible CPU overhead (e.g., 25.3% performance degradation of Redis workload with eBPF extensions). We also note that network/RNIC contention between control and data paths is negligible, as network traffic used by extension injection is minimal compared to the data path.

Summary. Overall, the limitations of agent-based design include the mismatch between injection delay and workload demands, the lack of automation and global visibility for consistency guarantees, and the performance interference with application workloads due to resource contentions.

2.3 RDX: Remote Direct code eXection

RDX aims to arm RDMA with *remote direct code execution* capabilities, so that code logic can be loaded into remote hosts without involvement of their CPUs. As shown in Figure 1(b), this would enable an agentless architecture for runtime extension frameworks. Instead of relying on host agents to load extension logic into local data planes, we can use a centralized remote control plane to directly inject extension logic into hosts via RDMA. This approach holds the promise to (1) bring down extension (binary) injection delay to align with RDMA speed, thus covering microsecond-scale scenarios; (2) make it possible to buffer all requests during updates, thus ensuring consistency without extensive developer efforts; (3) minimize resource sharing between extension injection and application requests by eliminating the injection footprints locally, thus mitigating contention.

At first glance, realizing this agentless architecture is a daunting task. Essentially, the new paradigm requires the decoupling between control path extension injection and data path extension execution onto different nodes. The remote control plane has to take over the responsibility of local agents in terms of validating, compiling, and loading the extension logic, but not all control knobs available to local extension

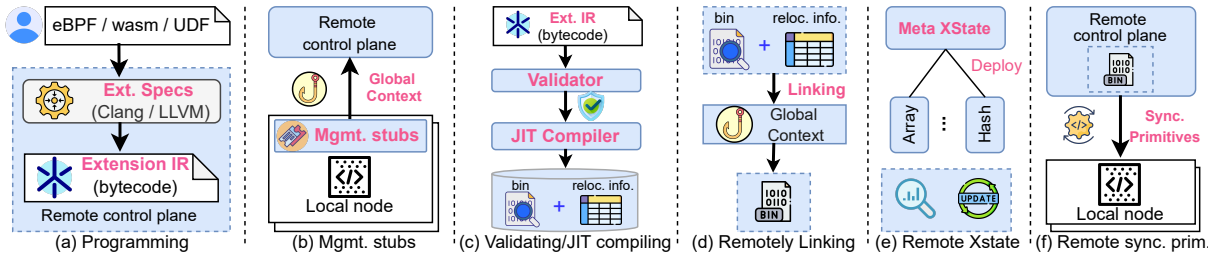


Figure 3: The technical roadmap of RDX and the key challenges; (a)~(f) correspond to the solution sketch in §3.1~§3.5.

injection are accessible to the remote. This is further complicated by our desire that this new architecture should stay *transparent* to developers, without affecting the programming model of existing extension frameworks (e.g., Wasm and eBPF specifications). To bridge these gaps, we propose an abstraction called *CodeFlow* to streamline remote runtime extension injection, abstracting away the low-level complexities of handling RDMA semantics and remote/local context gaps from developers.

Figure 3 shows the roadmap of RDX. First, when a new data plane sandbox boots, it initializes a setup module to expose control to the remote control plane. Subsequent steps are performed at runtime—the user informs the remote control plane the desired runtime extensions based on their goals. Next, the remote control plane validates, and compiles (using JIT or interpreters) the given runtime extensions into binaries (can be skipped if binary already exists), then performs binary rewriting to ensure that they are aligned with the context of target sandbox nodes (e.g., Global Offset Table). After that, the control plane sets up the needed shared data structures for eXtension State called `XState`, including their metadata and layouts, and eventually injects the extension binaries and `XState` into the local sandboxed data plane using remote transaction primitives. Throughout this process, the injection does not require application recompilation, traffic draining, or any other forms of downtime.

3 System Sketch

Next, we propose a set of building blocks to realize RDX.

3.1 Interacting with remote mgmt. stubs

RDX should expose the local sandbox's runtime context to the remote control plane. If extension code were just a standalone piece of raw binary data, this task would not be difficult—remote code injection would simply be an RDMA memory write, with its payload containing the extension binary. However, the challenge is that extensions come with complex metadata structures (e.g., hook points) and global dependencies (e.g., global offset tables (GOT)). For instance, an eBPF program's metadata `struct bpf_program` contains no less than 30 variables far beyond a code pointer, while a Wasm filter carries more than 20 metadata variables. Thus, extension injection not only requires supplying the code itself but also "filling in" its associated variables with correct values; additionally, this and other data structures are continuously created and destroyed at runtime. Directly "handcrafting" all these variables from afar is exceedingly difficult, as we need to infer variable values remotely without context; even a slight error could cause sandbox failures.

We propose to plant a minimal set of *management stubs* when each sandbox boots. There are three management stubs in total—`ctx_init` which creates the sandbox context (e.g., extension metadata) for runtime extensions by preloading empty extensions at locations of interest; `ctx_register` which registers the memory address for control knobs (e.g., select metadata variables and scratchpad pages) with RDMA, and `ctx_teardown` which provides assistance in detaching extensions based on a reference counting mechanism. All three stubs are contained in a loadable module, which is installed on each sandbox as a one-time setup operation. Each node spawns a CodeFlow (**rdx_create_codeflow**()) via remote management stubs to handle subsequent extension injection, removal, and access.

3.2 Remotely validating & JIT-compiling code

Validating and compiling extensions are essential for ensuring correctness and efficiency. Today, both steps are performed locally: the validator checks the extension IR (e.g., bytecode) against static safety rules such as memory boundary and extension termination, and the JIT compiler then generates the architecture-specific machine code (e.g., x86 or ARM). These steps must remain context-aware, as they rely on local architectural specifications and runtime environments (e.g., global variables and functions) in the local host. For example, an eBPF program may refer to its metadata descriptor and helper functions; a Wasm filter relies on host-function calls in Proxy-Wasm. Thus, RDX's remote control plane must identify and align with (1) the target architectures and (2) the required local context.

RDX relies on existing validators and JIT compilers (on the remote control plane) whenever possible to reuse their functionalities. To start with, RDX uses cross-architecture JIT compilers [16, 17] to generate binary for each architecture (e.g., X86 and ARM), removing the architecture dependency of the target nodes. Within each generated binary, RDX (1) locates entity references (e.g., function calls) using bytecode-level descriptors, (2) inserts placeholders for future relocation from remote to local contexts, and (3) records entity references information in a symbol table. The instrumented binaries, along with the symbol table, are packaged and stored

Operation	Description
rdx_create_codeflow(node, ext_spec) ⇒ handle	Create a CodeFlow handle bounded with a remote node
rdx_validate_code(handle, ext_prog) ⇒ result	Remotely validate the ext_prog using CodeFlow
rdx_JIT_compile_code(handle, ext_prog) ⇒ result	Remotely JIT compile the ext_prog into binary code using CodeFlow.
rdx_link_code(handle, ext_prog) ⇒ result	Link the binary of ext_prog to remote context via rewriting
rdx_deploy_prog(handle, ext_prog) ⇒ result	Deploy the binary of ext_prog onto remote node bounded with handle
rdx_deploy_xstate(handle, XState) ⇒ result	Deploy the data structure of XState onto remote node
rdx_tx(handle, inter_obj, qword_swap)	Transactionally update remote qword_swap obj with new inter_obj
rdx_cc_event(handle, event_hook, mem_addr)	Flush remote mem_addr by injecting cache-coherent binary to event_hook
rdx_mutual_excl(handle, hook_ctx)	Update remote pointers with mutual exclusion locks in hook_ctx
rdx_broadcast(codeflow_group, ext_progs, n) ⇒ result	Transactionally "broadcast" *n* ext_progs to *n* nodes codeflow_group

Table 1: The proposed set of CodeFlow API operations.

in the control plane for future reuse. Working together, the **rdx_validate_code()** and **rdx_JIT_compile_code()** APIs enable RDX to validate and compile each extension *once* and deploy them *anywhere*, on demand.

3.3 Linking code to local context

Next, RDX should link the instrumented binaries to target runtime contexts. If one extension is fully inline, this is easy for RDX as it just needs to remotely write the binary to the target node without linking. However, extensions often rely on some global utilities (e.g., global variables and functions) of the target sandbox runtime. Without careful relocation, injecting a binary would cause runtime failures. So the challenge is how to align the generated binary with the context of the local runtime (i.e., node or sandbox).

This can be achieved by *binary rewriting* techniques. During the linking stage (**rdx_link_code()**), the entity placeholders within the JIT-compiled binary are replaced with the addresses of their local counterparts in target node's runtime. Since JIT-compiled extension binaries rely on node-specific global context, such as the global offset table (GOT), we expose this global context—containing addresses of all global variables and functions—to the RDX remote control plane when creating the CodeFlow via **rdx_create_codeflow()**. By combining this global context information with relocation metadata generated during **rdx_JIT_compile_code()**, CodeFlow can remotely link the extension binary accurately to the target node's runtime, followed by **rdx_deploy_prog()** call to remotely deploy the well-linked binary over RDMA.

3.4 Managing remote XState data structures

While runtime extensions are event-driven, they may keep state in various data structures called XState, such as eBPF maps and shared queues in Wasm filters. Thus, an important task of RDX is remote XState management, including allocation, destruction, lookup and update. The key challenge is the dynamic lifecycle of XState—due to the wide variety of XState types and sizes that users request for at runtime. A strawman solution is thus to register many XState instances for each possible type with a maximal allowed size. However, this could cause non-trivial memory waste.

Our key insight is to create one level of indirection, instantiating a "Meta" XState at setup time in the ctx_register management stub, which allocates a scratchpad of reserved pages upon boot. The top layer "Meta" XState, which is simply an "array", indexes other bottom-layer XState instances that are created at runtime. When injecting an extension with a XState, the RDX remote control plane (1) allocates a memory chunk from the scratchpad according to XState size, (2) injects a header (including XState's type and size) before the XState. (3) and injects an entry (i.e., the memory address of the header) in the "Meta" XState. The design, which is implemented in **rdx_deploy_xstate()**, enables RDX to create or destroy XStates with *any* sizes at runtime. RDX also offers compatible lookup and update interfaces to access XState data elements from RDX remote control plane (via RDMA) and local extensions.

3.5 Remote Sync. primitives

Due to the asynchronous nature between CPU and RNICs, injecting code remotely to the local data plane introduces synchronization challenges. Concretely, one-sided RDMA operations lead to three critical issues: (1) Non-atomic RDMA writes may cause objects (e.g., injected Wasm filters) to be partially read by CPUs; (2) the lack of cache coherence between the RNIC and CPU could delay CPUs from noticing injected objects in a timely manner; and (3) software support for mutual exclusion operations between the RNIC and CPUs.

To address these challenges, RDX proposes a set of remote synchronization primitives, as summarized in Table 1. *Remote transaction* (**rdx_tx()**) ensures atomicity by fully loading all required objects into the local runtime before they become visible for execution. *Remote cache coherence* (**rdx_cc_event()**) allows the remote control plane to flush cachelines in the local data plane, immediately exposing newly injected objects to CPUs. Lastly, *remote mutual exclusion* (**rdx_mutual_excl()**) provides sandbox-level locks, enabling safe concurrent interactions between CPUs and RDMA operations. These primitives are also used for XState injection.

4 Using RDX: Case Studies

We now demonstrate how RDX extends today's runtime extension frameworks and enables new use cases.

Agentless architecture for runtime extensions. We explore how RDX realizes an *agentless* runtime extension framework. For service mesh Wasm filters, we extend Istio's Extension

Discovery Service (XDS) with a new *filter manager*, where each Envoy sidecar hosts a CodeFlow handle. The local data plane exposes contexts via management stubs at boot time and only handles data plane execution at runtime. On the control plane, a *filter registry* stores Wasm binaries and relocation records validated and JIT-compiled through RDX APIs. A *filter dispatcher* remotely links and deploys requested filters, while a *filter inspector* introspects runtime states via `XState` APIs. For eBPF programs, we integrate RDX with Cilium in a similar fashion, though with additional sandboxing challenges due to kernel dependencies.

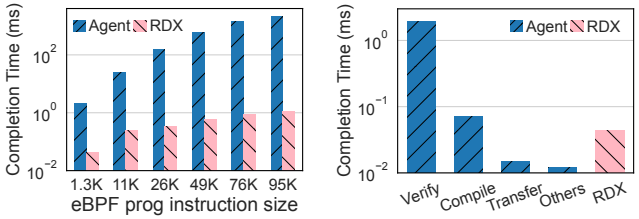
Fast and consistent extension updates. Modern deployments often need to broadcast extension updates across many nodes without introducing inconsistencies [11, 12, 18]. RDX introduces a *Collective CodeFlow* API (**rdx_broadcast()**) for microsecond-scale, atomic updates that simplify cluster-wide rollouts. Inspired by RDMA-based distributed transactions [61, 63], RDX treats a group update as a transaction whose write set spans all target hooks, deploying code in parallel with remote synchronization. To ensure consistency, RDX supports Big Bubble Update (BBU) [32], which buffers incoming requests during updates so that no request observes mixed logic. Once deployed, buffered requests are released in dependency order. This automated, transactional workflow eliminates inconsistencies and relieves operators from manually reasoning about update correctness.

Rollback and hot-patching for buggy extensions. Runtime extension failures have increasingly caused production outages, forcing emergency rollbacks or node restarts that requires draining request traffic for seconds or more [10, 24, 43]. Recovery can be delayed or stalled under CPU contention, leading to deadlocks and prolonged maintenance downtime. We rearchitect the current extension failure handling service with a remote rollback module that reverts faulty extensions to stable states in microseconds using CodeFlow *link+deploy* APIs or Collective CodeFlow operations. Hot patching can then be rapidly deployed online via the CodeFlow injection pipeline, improving overall service reliability.

Extension live migration for microsecond auto-scaling. There is a growing trend towards microsecond-scale elasticity in cloud-native deployments [46, 60, 62]. Existing studies leverage a "warm pod pool" and transfer intermediate states over RDMA to achieve scale-out in microseconds. However, scaling out a microservice pod means migrating both the application container and its sidecar (e.g., Envoy Proxy) to the new replica pod; reloading Wasm filters in the sidecar adds overhead by up to seconds, becoming the bottleneck. By contrast, RDX provides a promising optimization opportunity to cut the filter reloading cost to microsecond-scale with its CodeFlow APIs, enabling seamless live migration of extensions for auto-scaling.

5 Discussion

We now discuss the security implications of RDX in terms of confidentiality, integrity, and availability. For confidentiality, the control plane serves as a remote gatekeeper with role-based privilege model that prevent unauthorized access to extension entities (e.g., helpers, `XState`). For integrity, RDX can adopt JIT hardening [33] to insert runtime guards against tampering and speculative attacks, and perform signature-based remote runtime checks or remote memory introspection [49] over code, states, and hooks. For availability, RDX control plane can enforce strict runtime limits (e.g., instruction count), maintain extension state machines, and support atomic preemption with versioning to safely recover faulty extensions. Together, we envision RDX to naturally incorporate security techniques to protect remote code execution.



(a) Prog. load overhead. **(b) Time breakdown (1.3K).**

Figure 4: eBPF program load overhead. Agent baseline (shown on the left) incurs verify, JIT compile, and other forms of overhead. The breakdown is shown on the right.

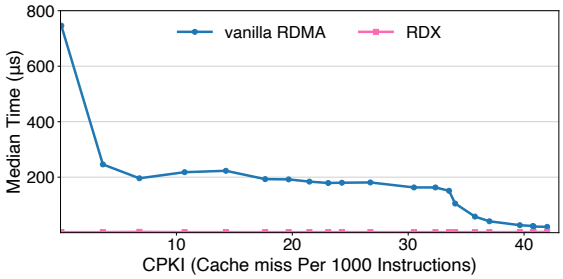


Figure 5: Remote synchronization primitives significantly reduce the incoherence time of remote extension injection.

6 Preliminary Validation

We prototyped RDX with eBPF as it offers a mature and representative proving ground for remote runtime extensions: (1) eBPF already includes a verifier, JIT compiler, and LLVM toolchain—allowing us to focus on remote injection, not basic plumbing. (2) eBPF supports service-mesh use cases (e.g., L3 policy in Cilium), mirroring one role played by Wasm filters. (3) eBPF underpins a broader range of use cases—networking, storage, telemetry, and security—giving RDX a wider payoff than other candidates. Given their shared injection mechanics, our success with eBPF suggests other extensions (e.g., Wasm) are feasible, which we leave as future work.

Testbed setup. The RDX remote control plane operates on a dedicated server within the same rack as the servers running local data planes. All servers run Ubuntu 20.04, each equipped with a 24-core Intel Xeon E5-2643 CPU (3.40GHz), 128GB DRAM, and a Mellanox CX-4 RDMA NIC. The *Agent* solution uses per-node eBPF agents as the baseline.

End-to-end feasibility. We repeatedly deploy the synthetic *Socket Filter* eBPF programs from the official Linux eBPF stress test [1], where the program instruction size ranges from 1.3K to 95K. Every program is repeatedly deployed 100K times to accurately measure the average completion time, with automated checks ensuring functional correctness. As Fig. 4a shows, RDX remotely injects all programs successfully, and consistently reduces injection time by orders of magnitude (47× ∼ 1982×). RDX mainly benefits from eliminating the program verification and JIT compilation overheads in injection path relative to *Agent* baseline (Fig. 4b).

RDX's benefits. RDX greatly reduces CPU contentions with workloads by eliminating per-node agent "tax." For example, *agentless* eBPF over RDX improves Redis throughput by up to 25.3% over the agent baseline. Likewise, injecting Wasm filters via RDX could improve microservice performance by up to 65%, based on CPU interference observed in §2. Finally, RDX enhances reliability through hardware-level injection—avoiding *lockout* effects and ensuring microsecond-level policy rollout consistency, even under full CPU load, injecting each policy in microseconds.

Effectiveness of remote sync. primitives. The remote sync. primitives in RDX effectively improve the injection synchronization efficiency between RNIC and host CPU. For instance, in Fig. 5, RDX consistently achieves orders of magnitude lower incoherence time (∼2μs across all CPKI levels) compared to the vanilla RDMA without remote synchronization primitives (up to ∼746μs under low cache stress).

7 Summary & Future Work

We present RDX, a vision that extends RDMA from memory access to code execution, using runtime extensions as a motivating use case. RDX enables an agentless architecture that decouples control path injection from data path execution, achieving microsecond-scale updates, strong consistency, and minimal contention. Our proposed CodeFlow abstraction exposes remote stubs, validates and JIT-compiles on the control plane, links binaries with local context, manages remote states, and performs hardware-level synchronization. Early case studies indicate its feasibility and performance benefits, which could lead to further RDMA innovation.

Looking forward, there is a set of open directions. (1) A declarative language for cluster-wide extension orchestration. (2) QoS-aware isolation to mitigate cross-extension interference. (3) Seamless integration with Wasm/UDF frameworks via formalized semantics. (4) New use cases, including living patching [34] and fault injection for reliability testing. (5) Data-driven control loops for datacenter resource management. (6) Replace RDMA with cache-coherent PCIe interconnects [40] to cut synchronization overhead. (7) Rigorous, at-scale validation and optimization for CodeFlow.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. We are especially grateful to Brighten Godfrey for shepherding our paper. This work is partially supported by a VMware Early Career Faculty Grant, a Cisco grant, and NSF grants CNS-1942219, CNS-2106751, CNS-2107147, CNS-2214272, and SaTC-2330065.

References

[1] BPF selftest notes. https://github.com/torvalds/linux/tree/master/tools/testing/selftests/bpf/.

[2] Envoy circuit breaker. https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/cluster/v3/circuit_breaker.proto/.

[3] Envoy filter dependency. https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/common/dependency/v3/dependency.proto/.

[4] Envoy resource updates. https://www.envoyproxy.io/docs/envoy/latest/api-docs/xds_protocol/.

[5] Envoy update issue. https://github.com/envoyproxy/envoy/issues/13009/.

[6] Gcp canary update. https://cloud.google.com/service-mesh/docs/revisions-overview.

[7] Istio pilot agent. https://istio.io/latest/docs/reference/commands/pilot-agent/.

[8] Istio traffic management. https://istio.io/latest/docs/ops/best-practices/traffic-management.

[9] Kubernetes. https://kubernetes.io/.

[10] Istio sidecar consuming high cpu istio 1.30-1.3.3, 2019. https://github.com/istio/istio/issues/18229.

[11] Istio 1.5 proxy sidecar crash on wasm filters that made a httpcall to an endpoint without response body, 2020. https://github.com/istio/istio/issues/23890.

[12] Proxy startup stalled by warming eds clusters, 2022. https://github.com/istio/istio/issues/38709.

[13] Exploring webassembly (wasm): Enhancing application performance, 2024. https://medium.com/@harsh.manvar111/exploring-webassembly-wasm-enhancing-application-performance-3233018b85e3.

[14] Kata containers, 2024. https://github.com/kata-containers/kata-containers.

[15] ebpf-based networking, observability, security. 2025. https://cilium.io/.

[16] Userspace/gpu ebpf vm with llvm jit/aot compiler, 2025. https://github.com/eunomia-bpf/llvmbpf.

[17] Wasmtime: Platform support, 2025. https://docs.wasmtime.dev/stability-platform-support.html.

[18] Airbnb. Improving istio propagation delay: A case study in service mesh performance optimization, 2023. https://medium.com/airbnb-engineering/improving-istio-propagation-delay-d4da9b5b9f90.

[19] Alibaba. User-defined functions in polardb, 2023. https://www.alibabacloud.com/help/en/polardb/polardb-for-xscale/udfs.

[20] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, et al. Empowering azure storage with {RDMA}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[21] T. A. Benson, P. Kannan, P. Gupta, B. Madhavan, K. S. Arora, J. Meng, M. Lau, A. Dhamija, R. Krishnamurthy, S. Sundaresan, et al. Netedit: An orchestration platform for ebpf network functions at scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 721–734, 2024.

[22] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka. On-demand container loading in {AWS} lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, 2023.

[23] G. Cloud. User-defined functions in google bigquery, 2024. https://cloud.google.com/bigquery/docs/user-defined-functions.

[24] I. community. Proxy readiness probe failurethreshold is too high, 2021. https://github.com/istio/istio/issues/29727.

[25] I. community. Userspace ebpf vm, 2025. https://github.com/iovisor/ubpf.

[26] Databricks. 100x faster bridge between apache spark and r with user-defined functions on databricks, 2018. https://www.databricks.com/blog/2018/08/15/100x-faster-bridge-between-spark-and-r-with-user-defined-functions-on-databricks.html.

[27] Databricks. Introducing sql user-defined functions, 2021. https://www.databricks.com/blog/2021/10/20/introducing-sql-user-defined-functions.html.

[28] Databricks. User-defined functions (udfs) in unity catalog, 2025. https://docs.databricks.com/aws/en/udf/unity-catalog.

[29] J. Dejaeghere, B. Gbadamosi, T. Pulls, and F. Rochet. Comparing security in ebpf and webassembly. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 35–41, 2023.

[30] DrDroid. Envoy filter chain mismatch, 2025. https://drdroid.io/stack-diagnosis/envoy-filter-chain-mismatch.

[31] eBPF.io authors. Dynamically program the kernel for efficient networking, observability, tracing, and security, 2025. https://ebpf.io/.

[32] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 635–649, 2022.

[33] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2405–2419, 2017.

[34] M. Fruth and S. Scherzinger. The case for dbms live patching. *Proceedings of the VLDB Endowment*, 17(13):4557–4570, 2024.

[35] A. Fuerst, A. Rehman, and P. Sharma. Ilúvatar: A fast control plane for serverless computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 267–280, 2023.

[36] Google. What is v8?, 2025. https://v8.dev/.

[37] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[38] B. Huang, L. Jin, Z. Lu, M. Yan, J. Wu, P. C. Hung, and Q. Tang. Rdma-driven mongodb: An approach of rdma enhanced nosql paradigm for large-scale data processing. *Information Sciences*, 502:376–393, 2019.

[39] B. Huang, L. Jin, Z. Lu, X. Zhou, J. Wu, Q. Tang, and P. C. Hung. Bor: Toward high-performance permissioned blockchain in rdma-enabled network. *IEEE Transactions on Services Computing*, 13(2):301–313, 2019.

[40] Y. Huang, Y. Huang, M. Yan, J. Hu, C. Liang, Y. Xu, W. Zou, Y. Zhang, R. Zhang, C. Huang, et al. An ultra-low latency and compatible pcie interconnect for rack-scale communication. In *CONEXT*, 2022.

[41] Y. Huang, Y. Qiu, Y. Xiao, A. Bhatnagar, S. Ratnasamy, and A. Chen. Exposing rdma nic resources for software-defined scheduling. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*, pages 1–8, 2025.

[42] Y. Huang, Y. Qiu, Z. Yang, Y. Dai, D. Wu, F. Lai, J. Xing, and A. Chen. Towards fully disaggregated recommendation model serving. In *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 38–45, 2025.

[43] Istio. context deadline exceeded (client.timeout exceeded while awaiting headers), 2022. https://github.com/istio/istio/issues/40758.

[44] Istio. Why choose istio?, 2025. https://istio.io/latest/docs/overview/why-choose-istio/.

[45] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.

[46] Z. Jia and E. Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pages 152–166, 2021.

[47] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[48] J. Langlet, R. Ben Basat, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi. Direct telemetry access. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 832–849, 2023.

[49] H. Liu, J. Xing, Y. Huang, D. Zhuo, S. Devadas, and A. Chen. Remote direct memory introspection. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6043–6060, 2023.

[50] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM symposium on cloud computing*, pages 412–426, 2021.

[51] Microsoft. User-defined functions (udfs) in unity catalog, 2023. https://docs.azure.cn/en-us/cosmos-db/nosql/query/udfs.

[52] Q. Monnet. Rust virtual machine and jit compiler for ebpf programs, 2025. https://github.com/qmonnet/rbpf.

[53] Nvidia. Doca argus service guide, 2025. https://docs.nvidia.com/doca/sdk/doca+argus+service+guide/index.html.

[54] M. Orenbach, R. Ailabouni, N. Masalha, T. Nguyen, A. Saleh, F. Block, F. Alder, O. Arkin, and A. Atamli. Blueguard: Accelerated host and guest introspection using dpus. 2025.

[55] Pixie. Open source kubernetes observability for developers, 2018. https://px.dev/.

[56] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment*, 11(4):432–444, 2017.

[57] Y. Sun, H. Chen, Z. Fu, W. Lv, Z. Liu, and H. Liu. Wasmguard: Enhancing web security through robust raw-binary detection of webassembly malware. In *Proceedings of the ACM on Web Conference 2025*, pages 1942–1950, 2025.

[58] Z. Wang, T. Ma, L. Kong, Z. Wen, J. Li, Z. Song, Y. Lu, G. Chen, and W. Cao. Zero overhead monitoring for cloud-native infrastructure using {RDMA}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 639–654, 2022.

[59] WAVM. Webassembly virtual machine, 2025. https://github.com/WAVM/WAVM.

[60] X. Wei, F. Lu, R. Chen, and H. Chen. {KRCORE}: A microsecond-scale {RDMA} control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, 2022.

[61] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586, 2018.

[62] J. Zhang, H. Tian, X. Huang, W. Li, K. Xu, D. Shen, Y. Wang, and K. Chen. Swift: Rethinking rdma control plane for elastic computing. *arXiv preprint arXiv:2501.19051*, 2025.

[63] M. Zhang, Y. Hua, P. Zuo, and L. Liu. {FORD}: Fast one-sided {RDMA-based} distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, 2022.

[64] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. *VLDB*, 2020.

[65] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn. bpftime: userspace ebpf runtime for uprobe, syscall and kernel-user interactions, 2023.

[66] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[67] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, et al. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 142–157, 2023.