Three Modes of Database-Kernel Integration via eBPF: Observability, Policy Injection, and Kernel-Resident State

Sheriff Adefolarin Adepoju * and Mildred Aiwanno-Ose Adepoju

Posted Date: 12 February 2026

doi: 10.20944/preprints202602.0959.v1

Keywords: eBPF; database systems; kernel observability; page-cache policy injection; XDP fast-path offload; kernel-resident transactional state

 doi:10.20944/preprints202602.0959.v1

Article

Three Modes of Database-Kernel Integration via eBPF: Observability, Policy Injection, and Kernel-Resident State

Sheriff Adefolarin Adepoju [1,*] and Mildred Aiwanno-Ose Adepoju [2]

[1] Department of Computer Science, College of Engineering, Prairie View A&M University, Texas, United States

[2] Department of Computer Information Systems, College of Engineering, Prairie View A&M University, Texas, United States

* Correspondence: 702 Santee Street, Hempstead, Texas, 77445, United States, sadepoju1@pvamu.edu

Abstract

eBPF is increasingly used around databases, but prior systems and papers often conflate goals and therefore blur the trade space: instrumentation versus control versus in-kernel state. This impedes principled comparisons and hides the kernel constraints that determine feasibility. We present a unified analysis organized into three modes of database–kernel integration via eBPF: observability (deriving DB-relevant signals from kernel and user-space hooks), policy injection (installing workload-specific cache and networking policies at kernel choke points), and kernel-resident state (providing safe transactional state semantics for eBPF programs beyond raw maps). For each mode, we characterized the hook placement, state model, and verifier/synchronization/portability constraints that shape feasibility, and then analyzed representative systems (programmable page-cache policies, XDP ingress offload, and ACID key-value with WAL export) against this framework. We show where microsecond overheads compound to core-scale costs when user-kernel crossings dominate tail latency and which integration mode fits the workload patterns. The outcome is a decision framework that guides when to measure, when to specialize the kernel policy, and when to introduce the kernel-resident state, enabling reproducible performance work without custom kernels.

Keywords: eBPF; database systems; kernel observability; page-cache policy injection; XDP fast-path offload; kernel-resident transactional state

1. Introduction

1.1. Context and Motivation

Modern database systems run on top of an operating system whose kernel controls the performance-critical substrate, namely, page cache behavior, block I/O submission and completion, TCP/IP processing, and scheduling. For many database workloads, these kernel subsystems are not background details; they are the mechanisms by which throughput, tail latency, and resource isolation are determined. A small change in caching or packet-processing behavior can dominate the impact of application-level optimizations because it occurs on paths that are executed at an extremely high frequency.

The eBPF has changed the practical boundary between database software and the kernel. It enables verified JIT-compiled programs to run at kernel and user-space hook points, with a shared state through maps and typed introspection through BTF/CO-RE. This makes it possible to (i) measure database-adjacent behavior with high fidelity, (ii) specialize kernel policies using workload knowledge, and (iii) maintain a limited kernel-resident state that supports fast-path decisions. These

 doi:10.20944/preprints202602.0959.v1

are qualitatively different objectives, but they are often discussed as if they were the same "eBPF for databases" story.

1.2. Problem Statement

The current discourse on eBPF in database contexts lacks a stable analytical framework. Various systems and papers regularly mix three distinct intents—observability, control via policy injection, and kernel-resident state—under a single label. Consequently, designs are compared on the wrong axes, feasibility constraints are underspecified, and the engineering trade-offs that actually decide success (verifier-bounded computation, synchronization limits, portability/dependency surface) are treated as implementation details rather than first-order design drivers.

1.3. Gap in Existing Work

Existing efforts typically contribute to a mechanism or system in one slice of the space: tracing and profiling pipelines for database operators, cache-control extensions that alter page-cache admission/eviction/prefetch, or network fast-path offloads that respond to simple RPCs at ingress. Each line of work is valuable, but the literature rarely provides the following:

- a shared taxonomy that separates *measurement* from*control* from *state semantics*,
- a consistent way to reason about where code executes (hook placement) and what state it can safely maintain,
- a feasibility analysis that treats verifier constraints, synchronization restrictions, and portability as primary design constraints rather than caveats, and
- a structured mapping from workload patterns (skew, scans, phase shifts, and strict tail latency budgets) to the appropriate integration mode.

Without this, the field accumulates point solutions and anecdotes rather than a principled understanding of when eBPF is the right integration tool, what class of kernel interaction is being attempted, and what must be reported to ensure reproducibility of results.

1.4. Approach and Contributions Overview

This paper provides an analytical framework for "database–kernel integration via eBPF" organized as three modes that share the same primitive (in-kernel, verified code at hooks) but differ in intent and feasibility boundaries: Observability: using eBPF to derive database-relevant signals from kernel and user-space execution without changing kernel policy. Policy injection: using eBPF to specialize kernel choke points (notably page cache and networking) with workload-specific access pattern hints and bounded decision logic. Kernel-resident state: providing structured state semantics for eBPF programs—up to transactional consistency—when fast-path decisions require coordinated updates beyond raw maps. For each mode, this study characterizes what "integration" concretely means in terms of hook placement, state model, and verifier/synchronization/portability constraints that shape feasibility. Representative systems, including programmable page-cache policies, XDP ingress offload, and ACID decision-state with WAL export, are analyzed against this framework. The goal is not to crown a single best approach but to supply a decision structure that makes trade-offs explicit and comparisons meaningful.

1.5. Paper Organization

Section 2 reviews related work and organizes it by integration mode rather than by individual systems. Section 3 introduces the three-mode framework and the feasibility constraints that govern each of the three modes. Section 4 applies the framework to representative designs and extracts the common design patterns and failure modes. Section 5 discusses the implications for database architects and kernel/observability engineers, including workload-to-mode fit and reporting requirements for reproducibility. Section 6 presents the limitations of this study. Finally, Section 7 concludes the paper and outlines future research directions.

 doi:10.20944/preprints202602.0959.v1

The main contributions of this paper are as follows:

- A three-mode taxonomy of database–kernel integration via eBPF—observability, policy injection, and kernel-resident state—that separates intent and prevents category errors in system comparison.
- Feasibility characterization for each mode is based on hook placement, state model, and verifier/synchronization/portability constraints that bound what can be built and what costs dominate.
- A structured analysis of representative systems (programmable page-cache policies, XDP ingress offload, and kernel-resident transactional state with WAL export) using a single framework enables a principled comparison across previously siloed lines of work.
- A decision framework that maps workload patterns and performance objectives to the appropriate integration mode, clarifying when to measure, when to specialize the kernel policy, and when the kernel-resident state is justified for correctness and tail-latency control.

2. Related Work

2.1. eBPF-Based Observability for Storage and Database-Adjacent Diagnosis

A large fraction of "eBPF for databases" work is fundamentally observability: extracting signals about I/O, scheduling, and networking behavior that databases experience but do not directly control. Cross-layer storage profiling exemplifies this approach. The zns-tools use eBPF to correlate events across the storage stack for NVMe ZNS devices, aiming to attribute performance effects across layers that are otherwise difficult to connect from user space alone [10]. This class of work is strong in terms of deployability because it reuses existing hook points rather than modifying the kernel policy.

The second strand focuses on the overhead and fidelity of observability itself, which is central when tracing is used in production-like database settings. Craun et al. showed that naïve tracing can impose costs on processes that are not being traced and proposed techniques to constrain the overhead to targeted processes [11]. Machado et al. demonstrated that eBPF libraries differ materially in performance, fidelity, and resource usage, meaning that "an eBPF-based measurement" is not a single design point; the tracing stack selection can alter results and threaten reproducibility if not controlled and reported [12]. For network-facing database components, Shahinfar et al. analyzed the performance costs in eBPF network applications, reinforcing that hook placement and per-packet work dominate the feasibility in tight fast paths [13].

Limitation: Observability work provides measurement and overhead-management primitives [10–13], but it does not justify kernel modification. Observability results are often implicitly used to motivate kernel policy changes without explicit feasibility analysis.

2.2. Policy Injection in Kernel Choke Points: Page Cache Specialization

Policy injection moves beyond measurement to specialize kernel decisions at choke points that dominate the database performance. The page cache is a canonical target because admission, eviction, and readahead behaviors directly shape the hit rate, writeback pressure, and interference under consolidation.

P2Cache proposes an application-directed page cache that exposes page cache events and allows applications to provide workload-specific cache control logic via eBPF [5]. The core argument is that data systems and databases often possess access pattern knowledge (e.g., scans vs. skewed hot sets, phase shifts) that the kernel cannot infer cheaply; therefore, programmable hints can reduce policy mismatches. The cache_ext advances this direction by enabling more general customization of page-cache behavior via eBPF, emphasizing in-kernel feasibility at high event rates, and supporting diverse eviction policies and data structures [7].

These systems differ in what they treat as "the interface contract"

P2Cache is framed around application-directed intent and explicit hints [5], whereas cache_ext is framed around enabling a wide class of kernel-resident cache policies [7].

 doi:10.20944/preprints202602.0959.v1

Both place logic on hot cache events, but cache_ext explicitly targets the generality of in-kernel policy composition and the associated runtime mechanisms required to make it practical [7].

Limitations specific to page-cache injection: Shared-resource externalities: improvements are workload- and colocation-dependent; specializing policy for one workload can harm others under mixed tenancy, complicating evaluation and "general benefit" claims [5,7]. Kernel coupling and maintenance: Deeper cache integration increases sensitivity to kernel interfaces and evolution, turning long-term portability into a feasibility constraint rather than an implementation detail [20].

2.3. Policy Injection and Fast Paths: Networking, Protocols, and RPC Handling

A parallel policy injection line targets the networking fast path, where distributed datastores are often dominated by tail latency and per-packet overhead. For high-QPS services, small per-packet costs can be translated into per-core CPU saturation and throughput ceilings.

BMC accelerates memcached by moving parts of request handling into the kernel and performing pre-stack processing, demonstrating that kernel-side handling can reduce the overhead for simple, high-volume request patterns [1]. The Electrode generalizes the target beyond a single service by accelerating distributed protocols with eBPF, showing that protocol machinery itself can be a viable in-kernel acceleration target [3]. DINT targets distributed transactions by offloading frequent-path transaction handling into XDP/eBPF, maintaining the hot-path state in kernel-accessible structures while deferring rare cases to user space [6]. eNetSTL addresses a different barrier, engineering complexity, by moving toward library-style support for building high-performance eBPF-based network functions in the kernel context [8].

Across these systems, the common architectural pattern is frequent-path acceleration with bounded work.

- From application-specific (memcached) [1] to protocol-level acceleration [3], to transaction-request handling [6].
- Frequent path handling is performed in the kernel, and complexity and exceptions are intentionally punted to the user space [6].
- eNetSTL emphasizes that adoption depends on reusable building blocks, not only feasibility proofs [8].

Limitations specific to fast network paths:

- Fast-path programs typically handle a constrained subset of operations; the quality of the frequent/rare split determines both performance and correctness risk [6].
- As fast paths require richer invariants (multi-object updates, transactional constraints), ad-hoc states in maps become fragile and motivate stronger state semantics (Section 2.5) [6].
- Cost attribution depends on hook placement and runtime behavior; without careful decomposition, "eBPF acceleration" results can be difficult to compare across systems [13].

2.4. Policy Injection for Storage: In-Kernel Storage Functions and Network-Attached Pushdown

The third policy-injection target is the **storage boundary**: placing programmable logic closer to storage submission/completion paths or pushing storage-adjacent functions toward network-attached storage endpoints. XRP presents in-kernel storage functions using eBPF, positioning eBPF as a mechanism for programmable logic associated with storage operations and the storage I/O path [2]. The BPF-oF explores the storage function pushdown over the network, reflecting the interest in moving computation toward storage endpoints in distributed environments where storage access costs dominate [4].

Framed in the same terms as Sections 2.2–2.3, this work is best understood as **policy injection at the storage layer** :

- Page cache injection changes *memory-side* caching decisions [5,7].
- Network fast-path injection changes the *ingress/transport-side* processing decisions [1,3,6,8].
-

 doi:10.20944/preprints202602.0959.v1

- Storage-boundary injection changes the *storage path* behavior (local in-kernel functions or network-attached pushdown) [2,4].
-

Limitations specific to storage-boundary injection:

Device and stack heterogeneity: feasibility depends on which layer the function attaches to (block, filesystem, network-attached endpoints) and what semantics are exposed; portability costs can be higher than those in purely observational tools [2,4,20].

Isolation and correctness scope: injecting storage-adjacent behavior can interact with ordering, durability expectations, and multi-tenant isolation; the acceptable scope is narrower than "general programmable logic" [2,4].

2.5 Kernel-resident state and transactional semantics for eBPF programs Kernel-resident state is often introduced informally as "store state in maps," but the key question is state semantics: what correctness guarantees exist under concurrency and how multi-step updates are coordinated. The BPF-DB makes this distinction explicit by proposing a kernel-embedded transactional DBMS for eBPF programs, offering ACID transactions and WAL export [9]. This elevates the kernel-resident state from ad-hoc map manipulation to structured, serializable state transitions suitable for non-trivial kernel-resident logic. DINT illustrates the contrasting strategy commonly used in fast paths: maintaining an adequate kernel-resident state for frequent-path decisions while punting complex cases to user space [6]. The comparison is instructive: DINT shows how far careful frequent/rare partitioning can go, whereas BPF-DB targets general-purpose consistency mechanisms inside the eBPF execution model.

Limitations specific to kernel-resident state: Semantics vs. feasibility: stronger guarantees increase complexity under verifier and synchronization constraints, forcing careful design to keep updates bounded and safe [15,19]. Boundary with user space: WAL export acknowledges that durability and management remain user-space concerns; however, the split introduces an interface, overhead, and failure-mode complexity that differs from pure observability or pure policy injection [9].

2.6. Foundations: Verification, Security, and Portability as Feasibility Constraints

The feasibility of all three modes is dominated by cross-cutting constraints: what the verifier accepts, what synchronization is possible, and how stable the dependency surface is across kernels and deployments. Formal and mechanized analyses aim to increase confidence in the verifier reasoning (e.g., range analysis) [14,15]. Simultaneously, the HotOS work argues that kernel extension verification may not scale as a complete solution, reframing safety as broader than verifier correctness alone [16]. Security-oriented systems target safer extension mechanisms and memory-safety pathways for eBPF [18,19]. In parallel, portability and maintainability have become explicit concerns: dependency mismatches and unstable interfaces can make real-world eBPF extensions brittle across kernel versions, despite the tools intended to reduce fragility [20]. Finally, systems such as KFlex and VEP reflect ongoing efforts to broaden kernel extension practicality and programmability, which directly shifts the feasible design space over time [17,21].

As evident from the limitations identified across observability (Section 2.1), policy injection (Sections 2.2–2.4), and kernel-resident state (Section 2.5), several constraints recur as fundamental feasibility boundaries: verifier-bounded computation [15,19],synchronization restrictions [6,15], and dependency-surface instability [20]. This study treats these as first-order design drivers in Section 3.

2.7. Positioning and Novelty Relative to Prior Work

Existing work provides strong contributions in each mode—observability [10–13], policy injection [1,3,5–8], and kernel-resident state [6,9]—along with deep analyses of the feasibility constraints [14–21]. What is missing is a unified framework that:

 doi:10.20944/preprints202602.0959.v1

- separates observability, policy injection, and kernel-resident states as distinct intents with distinct feasibility boundaries.
- provides shared comparison axes (hook placement, state model, verifier/synchronization/portability constraints); and
- enables principled cross-system evaluation rather than siloed comparisons within a subcommunity.
-

This study fills this gap by analyzing representative systems across all three modes under one framework, making the trade space explicit, and tying feasibility constraints directly to database workload patterns and performance objectives.

3. Methodology: Framework Construction and Analysis Protocol

This section defines the analytical framework used in the remainder of the paper and the protocol for its application to representative systems. The methodology is deliberately "artifact-first": each subsection produces a reusable deliverable (tables, matrices, checklists, profile sheets, and a decision flowchart) that can be lifted into future database–kernel integration work.

3.1. Scope and Assumptions

Scope. The methodology targets Linux-based deployments, where eBPF programs are attached to kernel and/or user-space hook points to influence or observe database-relevant behavior. The analysis covers three integration objectives that frequently appear in database-adjacent eBPF systems: kernel observability [10–13], kernel policy specialization (cache/network/storage choke points) [1–8], and kernel-resident semantics-bearing states for eBPF programs [6,9].

Assumptions.

- A modern eBPF runtime with verifier-enforced safety properties (with known limitations) [14,15,19].
- Practical deployments must tolerate kernel upgrades and heterogeneity; interface instability and dependency mismatches are treated as feasibility constraints, not "engineering details" [20].
- The unit of analysis is a *system claim* : a paper's stated objective (e.g., lower tail latency), chosen hook placement, state model, constraint mitigation, and reported evaluation evidence.
-

Non-goals.

Proposing a new eBPF subsystem or new verifier design (covered by separate systems/security work [16–19,21]).

Full in-kernel DBMS designs are used as a replacement for user-space databases; the kernel-resident state mode is treated as an enabling substrate for bounded fast-path logic and correctness, not a migration of general DB functionality into the kernel [9].

3.2. Definitions and Terminology

We use the following terms throughout: Integration mode: one of three intents for using eBPF around databases: Observability, Policy Injection, or Kernel-Resident State (Section 3.3). Hook placement: the specific attachment points and execution contexts where the eBPF code runs (Section 3.4). Hook placement is a first-order determinant of event rate, allowable operations, and latency sensitivity [13]. Fast path: An execution path invoked at a high frequency (e.g., per packet, per cache event, per I/O completion), where the per-event overhead can saturate the CPU resources at a high QPS [6,7,13]. User/kernel crossing: transitions between kernel and user space induced by exporting data or deferring decisions, often dominating the tail latency when invoked on hot paths [6,7]. State model: Representation and semantics of the state used by eBPF programs (per-event, keyed, shared, or transactional) (Section 3.5). Dependency surface: the set of kernel interfaces, data structures, and assumptions on which an eBPF program depends. Instability is a primary source of breakage across

 doi:10.20944/preprints202602.0959.v1

kernels [20]. Feasibility constraints: verifier-bounded computation, synchronization restrictions, interface/portability constraints, and resource ceilings that bound what can be built and what it costs (Sections 3.6–3.7) [6,15,19,20].

3.3. Three-Mode Taxonomy

We formalized three integration modes. Each mode was specified using a fixed template to prevent category drift and enable principled comparison across systems that otherwise appear unrelated.

Mode template (applied to all three):

- **Objective** (what the system is trying to achieve)
- **Mechanism** (how eBPF is used)
- **Typical hook points** (where the code runs)
- **Allowed actions** (observe-only vs. modify policy vs. maintain semantics-bearing state)
- **Success criteria** (what "works" means)
- **Dominant risks** (what breaks feasibility in practice)

Table 1. Mode definitions and typical feasibility boundaries.

Mode	Objective	Typical hook points	Allowed actions	Dominant feasibility boundaries
Observability	Derive database-relevant signals about kernel-mediated behavior with minimal perturbation	Tracepoints/kprobes/fentry, uprobes/USDT, perf events, networking and block-layer tracepoints [10–13]	Read kernel/user-space state; emit events/metrics	Measurement overhead and perturbation [11], tracing-stack variance [12], hook availability and stability [20]
Policy Injection	Specialize kernel decisions at choke points using workload-specific intent	Page-cache events and memory/I/O policy points [5,7]; XDP/TC/cgroup networking hooks [1,3,6,8]; storage-path hooks/functions [2,4]	Modify admission/eviction/prefetch decisions; steer/handle packets; execute storage functions	Correctness under mixed workloads [5,7], hot-path CPU budget [6,7], portability/dependency surface [20]
Kernel-Resident State	Provide semantics-bearing state transitions for eBPF programs (up to transactional consistency)	Typically paired with fast-path hooks (e.g., XDP) or kernel events requiring coordinated updates [6,9]	Maintain shared state with defined semantics; optionally export logs/state	Verifier and synchronization constraints [15,19], state explosion, security boundary expansion [18,19]

This taxonomy is not an ontology of "all eBPF uses." This is a practical partition of *database–kernel integration intents* that appear repeatedly in systems from 2021 to 2025 [1–13].

3.4. Hook Placement Model

Hook placement determines (i) execution frequency, (ii) permissible operations, (iii) cost of accessing state, and (iv) whether logic sits on a latency-critical path [13]. We classify hooks by execution context and "event-rate regime" (qualitative) rather than claiming fixed rates.

Table 2. Hook placement taxonomy (qualitative).

Hook family	Typical context	Event-rate regime	Representative DB-adjacent use	Primary constraints
Tracing hooks (tracepoints, kprobes/fentry)	process context or kernel internal contexts	low → very high (workload-dependent)	Storage/network/kernel-path diagnosis [10–13]	Perturbation and overhead containment [11], fidelity variance across tooling [12], interface instability [20]
User-space instrumentation (uprobes/USDT)	user process	low → high	Correlating DB internals with kernel events [10,12]	Sampling bias, overhead, symbol stability
XDP	early ingress path	very high	Packet-level fast paths for KV/transactions [6]; network accel analyses [13]	Tight CPU budgets; limited semantics; state access costs [6,13]

 doi:10.20944/preprints202602.0959.v1

TC / cgroup networking hooks	networking stack	high	Protocol/request handling and traffic shaping [1,3,8]	Per-packet overhead; interaction with stack behavior
Page-cache / memory policy hooks	memory subsystem	high (cache events)	Admission/eviction/prefetch specialization [5,7]	Shared-resource interference; correctness and regressions [5,7]
Storage-path hooks / in-kernel storage functions	I/O path	medium → high	In-kernel storage functions and pushdown [2,4]	Heterogeneity; ordering/durability interactions [2,4]
Security hooks (LSM)	security decision points	medium	Guardrails and policy enforcement for extensions [18]	Privilege boundaries; policy correctness

This table is not exhaustive; it is a minimal hook taxonomy sufficient to place the 2021–2025 corpus into comparable execution contexts [1–13].

3.5. State Model and Semantics

State is the main differentiator between "fast tracing" and "integration." We model the state as a ladder of increasing semantic commitments and costs.

State ladder (from weakest to strongest): Stateless/counters: per-event metrics (rates, latencies). Keyed state: Maps keyed by the PID/connection/request ID. Shared coordinated updates: multi-map updates requiring concurrency discipline. Semantics-bearing state: explicit invariants (e.g., protocol-state machines). Transactional state: Atomic multi-step updates with isolation claims (serializability). The BPF-DB sits at the transactional end of the ladder by providing ACID transactions and a structured interface for log export [9]. DINT illustrates the "semantics-bearing but bounded" approach: it keeps enough state in the kernel for frequent-path transaction handling and defers the remainder to the user space [6]. cache_ext and P2Cache illustrate the keyed and shared states used to implement cache policy decisions at scale [5,7].

Table 3. State ladder and when it is required.

State level	Typical mode(s)	Example requirement	Typical risk
Stateless/counters	Observability	attribution of delays to kernel layers [10]	measurement perturbation [11]
Keyed state	Observability, Policy Injection	per-connection tracking; per-file hotness	state size and memory overhead
Coordinated updates	Policy Injection	eviction structures; fairness across flows [7]	contention; correctness under concurrency
Semantics-bearing	Policy Injection, Kernel-Resident State	protocol invariants in fast path [3,6]	verifier/sync limits; complexity
Transactional	Kernel-Resident State	atomic multi-key update with isolation [9]	feasibility under eBPF constraints; enlarged security boundary [19]

3.6 Feasibility Constraint Model We treat feasibility constraints as first-order design drivers. For each constraint, we recorded how it manifested, common failure modes, and mitigation strategies. This structure makes cross-paper comparisons explicit, even when papers emphasize different aspects.

Table 4. Constraint matrix with mitigation strategies (core constraints recur across modes).

Constraint	Manifestation	Typical failure mode	Mitigation strategies	Representative work
Verifier-bounded computation	bounded loops, restricted pointers, limited stack; verifier reasoning must accept program [14,15]	program rejected; forced simplification; unsafe workarounds	restructure into bounded phases; tail calls; push complex logic to user space; use verified toolchains where applicable [21]	[14,15,21]

 doi:10.20944/preprints202602.0959.v1

Synchronization restrictions	limited locking in hot contexts; contention dominates at high event rates [6]	correctness bugs; throughput collapse under contention	per-CPU state; lock-free designs; shard state; minimize shared updates; frequent/rare splitting [6]	[6]
Portability / dependency-surface instability	reliance on kernel internals; CO-RE/tooling does not eliminate all mismatches [20]	breakage across kernel versions; silent mismeasurement	minimize internal dependencies; prefer stable hooks; defensive probing; explicit version gating; document kernel/CONFIG requirements	[20]
Security boundary and privilege model	loading privileges; attack surface of in-kernel logic; extension governance [16,18,19]	escalation risk; abuse of hooks/state; unsafe extension deployment	privilege restriction; LSM-based guardrails; capability gating; minimize in-kernel semantics; auditing and policy controls [18]	[16,18,19]
Resource ceilings	map memory, per-CPU overhead, export bandwidth; runtime overhead [11–13]	dropped events; memory pressure; instability under load	bounded sampling; backpressure-aware export; limit state cardinality; report overhead explicitly [11,12]	[11–13]

This matrix was applied uniformly across the case studies (Section 3.11). This also explains why limitations recur across modes: the same fundamental boundaries constrain observability, policy injection, and kernel-resident states [6,15,19,20].

3.7. Cost Model

We decompose the cost into per-event and cross-boundary components. This allows comparisons across systems that operate at different event rates (e.g., tracepoints vs. XDP).

Per-event cost decomposition:

- Base overhead of reaching the hook and running the attached programs.
- Instruction path length, branching, and helper/kfunc usage.
- Map lookups/updates, contention, and cache locality.
- Ring-buffer/perf-event output, aggregation overhead, and user-space consumption [11,12].

Cross-boundary costs:

User/kernel crossings: exporting high-frequency events or deferring decisions to user space can dominate the tail latency and CPU consumption, especially on fast paths [6,7].

Scaling principle (used in analysis): If a path executes at E events/s and adds c seconds/event, then the CPU time consumed is $E \cdot c$ seconds/s. At high event rates, even microsecond-scale deltas saturate cores and reshape throughput ceilings, which is a central concern in fast-path designs [6,7], and a recurring theme in performance analyses of eBPF network applications [13].

Observability work provides cautionary evidence that overhead is not only a function of "the eBPF program," but also of the tracing stack and export path; library/runtime choices can change fidelity and resource usage [12], and the overhead must be constrained to avoid system-wide impact [11].

3.8. Correctness, Safety, and Failure Modes

The correctness is mode-dependent. We define correctness checklists per mode and explicitly include a **security boundary** dimension because "correct results" are insufficient if the integration expands the attack surface or violates operational constraints [16,18,19].

Table 5. Correctness checklist with security boundary.

Mode	Correctness criteria	Safety criteria	Security boundary
Observability	measurement fidelity; correct attribution; bounded perturbation; loss accounting [10–13]	overhead containment to avoid impacting unrelated processes [11]	who can attach probes; what data is exposed; leakage risk

 doi:10.20944/preprints202602.0959.v1

Policy Injection	preserves kernel invariants; improves objective without regressions under mixed workloads; stable under load [1,3,5–8]	bounded work on hot paths; rollback/disable mechanism; interference awareness [5,7]	privilege to load/attach; risk of policy abuse; isolation between tenants [19]
Kernel-Resident State	semantics claims (atomicity/isolation) are defined and evidenced; concurrency correctness; failure handling and export boundary [6,9]	bounded state growth; controlled contention; predictable degradation	expanded attack surface via richer in-kernel services; governance and hardening [18,19]

This checklist is applied as a failure mode screen before discussing performance wins. It prevents "fast but incorrect" comparisons (e.g., fast paths that silently drop semantics under concurrency) and maintains security feasibility [16,18,19].

3.9. Analysis Rubric and System Profile Sheet

To avoid narrative-only comparison, each system is analyzed using a fixed "profile sheet." This ensures uniform extraction across papers and makes gaps explicit (e.g., missing kernel-version constraints or missing overhead breakdowns).

Table 6. System profile sheet (template).

Field	Description
System	Paper/system name and year
Claimed objective	Throughput, tail latency, resource isolation, diagnosis fidelity, etc.
Mode	Observability / Policy Injection / Kernel-Resident State (Section 3.3)
Typical hook points	Specific attachment points (Section 3.4)
Event-rate regime	Qualitative (low/medium/high/very high)
State model	Position on state ladder (Section 3.5)
Semantics claims	Invariants, correctness guarantees, transactional claims [9]
Constraints encountered	Verifier, synchronization, portability, resources (Section 3.6)
Mitigations	Concrete strategies used (Table 4)
Cost drivers	Hook/program/map/export/crossing decomposition (Section 3.7)
Evaluation evidence	Workloads, baselines, metrics, variance reporting
Portability posture	Kernel/version assumptions; dependency surface notes [20]
Security boundary	Privileges required; potential abuse surface [18,19]
Reported limitations	Stated coverage gaps, failure modes

3.10. Workload-Pattern-to-Mode Mapping Procedure

We mapped workload objectives and patterns to integration modes using explicit decision rules. Mapping is not a "recommendation engine"; it is a reproducible procedure that exposes assumptions.

Workload signatures (inputs)

- **Access patterns:** hot-set skew, scans/streaming reads, phase shifts, write bursts (cache relevance) [5,7].
- **Service regime:** high-QPS request handling and strict tail-latency budgets (network relevance) [1,6].
- **Choke point:** cache, networking ingress/transport, or storage path (policy relevance) [2,4–7].
- **Semantic requirement:** whether correctness requires coordinated multi-step state transitions (state relevance) [6,9].
- **Operational constraints:** portability requirements, privilege models, and security governance [18–20].

 doi:10.20944/preprints202602.0959.v1

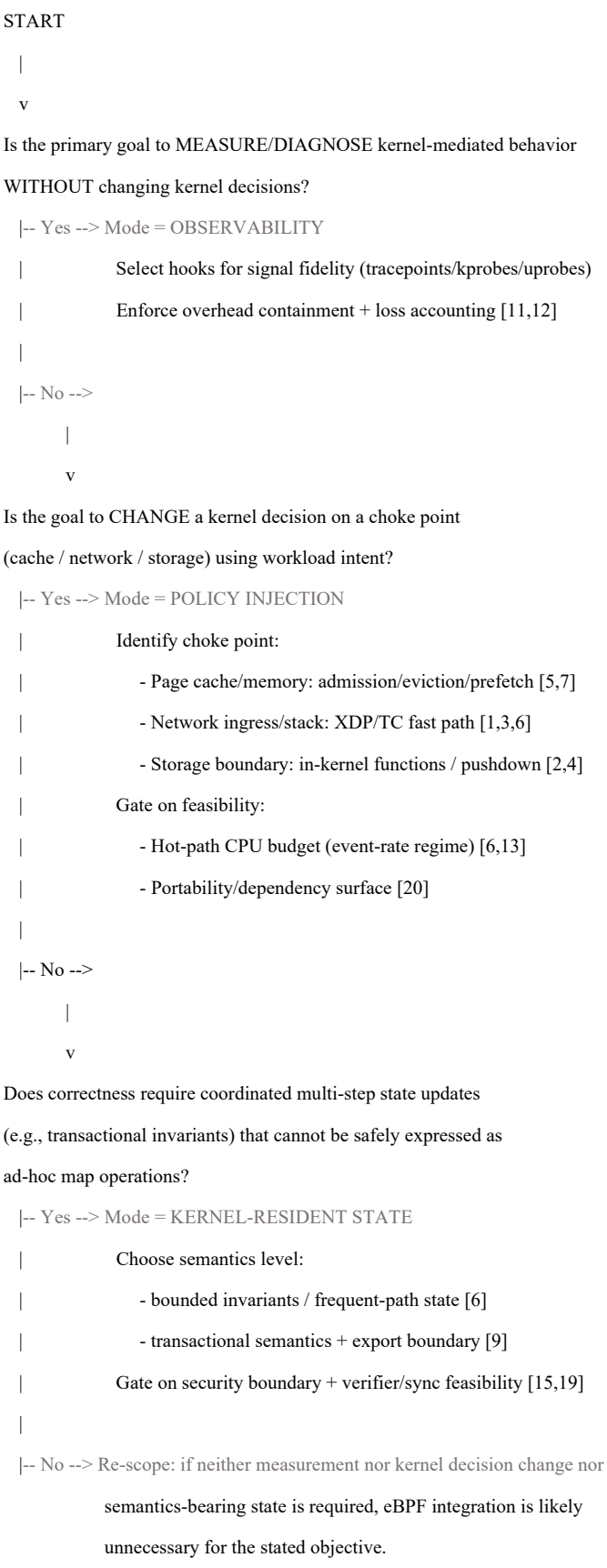


Figure 1. Simplified decision flowchart (concrete, mode-selecting).

This flowchart is intentionally conservative: it forces a paper to state which kernel decision is being changed (policy injection) or which semantics are required (kernel-resident state), instead of treating "eBPF" as a contribution.

 doi:10.20944/preprints202602.0959.v1

3.11. Case Study Selection and Extraction Protocol

Corpus. The primary corpus is the 2021–2025 set of systems and foundational works listed in Section 2, which spans:

- Observability and tracing overhead/fidelity: [10–13]
- Policy injection at cache/network/storage choke points: [1–8]
- Kernel-resident state semantics: [6,9]
- Feasibility foundations (verification, safety, portability): [14–21]
-

Inclusion criteria for representative systems (applied to [1–10]).

- Explicit use of eBPF to observe or influence a DB-adjacent kernel choke point.
- A stated performance or correctness objective and evaluation with workload evidence.
- Sufficient implementation details to identify hook placement and state model.

Extraction protocol. For each system, we populated the profile sheet (Table 6) and recorded the following:

- hook points, and event-rate regime
- state ladder position;
- Feasibility constraints encountered and mitigations used (Table 4)
- cost decomposition (Section 3.7).
- Correctness/safety/security boundary (Table 5);
- portability posture (dependency surface signals) [20].

This yields a comparable dataset of "system profiles" that supports cross-mode comparison without collapsing distinct intentions into a single axis.

3.12. Reproducibility and Reporting Requirements

Reproducibility is treated as an output artifact of the methodology rather than an afterthought. Two empirical hazards motivate explicit reporting requirements: (i) observability overhead can affect untraced processes if not carefully constrained [11] and (ii) tracing stacks and libraries can materially change fidelity and resource usage, impacting comparability across papers [12]. Portability concerns require explicit kernel/version reporting because dependency mismatches are pervasive in real-world extensions [20].

Artifact MR-1: Minimal Reproducibility Checklist (citable)

A paper claiming database–kernel integration via eBPF should report the following minimum information:

Kernel and platform

- kernel version and configuration flags relevant to eBPF and target subsystems
- CPU model, core count, NUMA topology; frequency scaling status
- NIC/storage device model; key offload features enabled/disabled (where relevant)

eBPF program and attachment

- hook points used (exact attach types and sites)
- program sizes (instruction count where available) and compilation toolchain
- map types, key/value sizes, cardinality bounds, and memory footprint
- export mechanism (ring buffer/perf events/etc.) and backpressure/drop handling

Runtime controls

- CPU pinning/affinity for kernel threads and user processes
- interrupt/NAPI affinity where networking fast paths are evaluated
- sampling rates, aggregation windows, and filtering rules (for observability)

Workloads and baselines

- workload generator and configuration (request mixes, dataset sizes, skew/scan behavior)

 doi:10.20944/preprints202602.0959.v1

- baselines with exact kernel/userspace configuration and tuning knobs
- warm-up procedure and run duration; how steady state is determined

Metrics and variability

- throughput and tail latency definitions (percentiles, time windows)
- overhead reporting (CPU%, memory overhead, dropped events, regression tests)
- variance reporting (repetitions, confidence intervals or equivalent)

Portability posture

- explicit statement of kernel-version assumptions and known incompatibilities [20]
- which interfaces are assumed stable vs. version-gated

MR-1 is used as a strict filter when interpreting results in later sections: claims that omit MR-1 items are treated as less comparable by construction, independent of the reported performance outcomes.

4. Framework Application and System Analysis

4.0. Section Goals and Reading Guide

Section 4 applies the framework defined in Section 3 to representative systems published between 2021 and 2025. The goal is not to re-explain the eBPF mechanisms. The goal is to normalize comparisons across papers that otherwise operate on different choke points, hook contexts and semantics.

The following artifacts were used in this section:

- **System profile sheets (Table 6 template):** condensed in-text, full versions assumed to be appended.
- **Constraint matrix with mitigation strategies (Table 4):** used to interpret feasibility choices rather than listing "limitations" ad hoc.
- **Cost decomposition (Section 3.7):** used to separate the hook cost, program cost, state-access cost, and export/crossing cost.
- **MR-1 reproducibility checklist (Section 3.12):** used to judge the interpretability of results, motivated by demonstrated tracing overhead and tooling variance [11,12].
- **Decision flowchart (Section 3.10):** validated using the case studies in Section 4.5.

Evidence versus inference rules:

A claim is treated as evidence-backed only when the paper reports a workload, baseline, and measurable outcome under the described configuration.

A claim is treated as an inference when it extrapolates beyond the measured workload regime (e.g., from microbenchmarks to production multi-tenant behavior).

4.1 Corpus and analysis protocol recap We analyze representative systems spanning three modes: Mode I (Observability): cross-layer storage profiling and trace/measurement feasibility under overhead and fidelity constraints [10–13]. Mode II (Policy Injection): cache, network, and storage-boundary specialization using eBPF at kernel choke points [1–8]. Mode III (Kernel-Resident State): semantics-bearing state inside the eBPF execution model, up to transactional semantics with an export boundary [6,9]. Extraction protocol. For each system, we populated the profile sheet fields (Table 6) and interpreted the feasibility using Table 4 (constraints × mitigation strategies). We explicitly track the hook-placement family and qualitative event-rate regime, state ladder position (Section 3.5), dominant cost driver under the unified decomposition (Section 3.7), correctness/safety/security boundary lens (Table 5), and MR-1 reporting completeness, with special attention to tracing stack choice and overhead containment motivated by [11,12].

 doi:10.20944/preprints202602.0959.v1

Table 7. Representative systems analyzed (by mode).

Mode	Systems (this paper's exemplars)
I: Observability	zns-tools [10]; Eliminating overhead on untraced processes [11]; eBPF library performance/fidelity variance [12]; eBPF network application performance demystification [13]
II: Policy injection	Cache: P2Cache [5], cache_ext [7]. Network: BMC [1], Electrode [3], DINT [6], eNetSTL [8]. Storage boundary: XRP [2], BPF-oF [4]
III: Kernel-resident state	BPF-DB [9] with DINT as contrast case for "bounded state + frequent/rare split" [6]

4.2. Mode I — Observability: What Can Be Measured, at What Cost, and with What Failure Modes

4.2.1. Mode I Overview

Mode I systems use eBPF to extract signals about kernel-mediated behavior relevant to database performance and correctness without changing kernel policy decisions. Typical targets include storage stack attribution, I/O path timelines, and network path cost decomposition [10,13].

Mode I feasibility is defined by the following three constraints:

- **Perturbation:** tracing must not materially alter the workload being measured; otherwise, the measurement becomes a closed loop. Craun et al. showed that even per-process tracing strategies can impose non-trivial overheads on untraced processes, making "targeted tracing" an unsolved engineering problem under common approaches [11].
- **Fidelity under load:** event loss and export path limitations can silently corrupt conclusions. Machado et al. showed that tracing stacks and libraries vary widely in event loss and resource usage; therefore, "eBPF-based observability" is not a single comparable unit unless the toolchain is controlled and reported [12].
- **Interpretability:** Measurements must map onto actionable choke points. The zns-tools illustrate the value of cross-layer correlation to bridge abstraction gaps (file name/inode/block address) and expose semantic mismatches across layers [10].

Mode I does not claim "the kernel should be changed." Mode I claims "the bottleneck can be attributed." This boundary is important because Modes II and III introduce correctness, safety, and portability burdens that are not present in pure measurements.

4.2.2. System Profiles (Mode I)

Table 8. Mode I condensed profile summaries (normalized by the framework).

System	Primary intent	Typical hook points	State level	Dominant cost driver	Core limitation (framework view)
zns-tools [10]	Cross-layer storage profiling and timeline reconstruction	Kernel probes across VFS/MM/FS/block/NVMe + user-space probes; timestamped event traces	Keyed event traces; offline reconstruction	Export + post-processing (timeline building)	Heavy tracing must manage event volume and semantic translation across layers
Eliminating overhead on untraced processes [11]	Make per-process tracing viable without penalizing others	Tracepoints/kprobes; focuses on tracing attachment path and filtering placement	Minimal state (PID sets / gating)	Hook-trigger overhead for unrelated processes	Existing filtering strategies still tax untraced processes; requires deeper runtime changes to remove it
No Two Snowflakes Are Alike [12]	Quantify library-level performance/resource/fidelity trade-offs	I/O tracing hooks (storage syscall/I/O events) via different libraries	Keyed tracing + export	Export + user-space runtime/library overhead	Results vary by library; "measurement method"

 doi:10.20944/preprints202602.0959.v1

					becomes a confounder unless standardized
Demystifying performance of eBPF network applications [13]	Identify when eBPF offload helps/hurts; characterize cost drivers and interference	Primarily XDP and kernel↔user comm paths; microbenchmarks of maps, chaining, JIT	Keyed state via maps; controlled experiments	Program complexity + state-access + runtime/JIT effects	eBPF offloads can violate performance isolation; benefits are workload- and codegen-dependent

4.2.2.1. zns-Tools: Cross-Layer Storage Profiling for ZNS Stacks

The zns-tools is a representative Mode I system because it treats the storage stack as an opaque, layered pipeline and uses eBPF to reconstruct an end-to-end trace across multiple abstraction layers [10]. The tool explicitly decomposes tracing into subtools for the NVMe/block layer and device driver (zns-tools. nvme), and filesystem tracing (zns-tools. fs (including ZNS-aware filesystems) and application-level tracing (zns-tools. app, demonstrated RocksDB) [10]. It uses nanosecond-resolution timestamps and produces traces in a standard JSON format consumable by timeline tools such as Perfetto, separating collection from analysis/visualization [10].

Framework-relevant contributions

- The paper frames tracing not just as "collect events," but as "collect enough cross-layer identifiers to translate between abstractions" (file descriptor/inode/block/LBA/zone) [10]. This is an explicit instance of the "semantic gap" that Mode I exists to close.
- The zns-tools demonstrate that even for the same high-level workload, device utilization and behavior can vary "vastly," indicating that application-level outcomes can be misleading without cross-layer attribution [10].
- zns-tools does not attempt to change the page cache, filesystem, or device policy. It provides a structured route to localize where mismatches occur (e.g., cross-layer data placement/classification decisions), creating inputs for policy discussions rather than asserting policy changes.

Limitations (framework perspective):

The tool's value depends on maintaining fidelity under high event volumes and on correctly translating identifiers across layers; both are export- and tooling-sensitive, linking directly to the library/fidelity concerns raised by [12].

The analysis remains observational; it identifies semantic gaps but does not provide the feasibility analysis required to justify policy implementation.

4.2.2.2 Eliminating eBPF tracing overhead on untraced processes: observability cost containment Craun et al. isolated a structural observability problem: when an eBPF program is attached to a hook, every process triggering that hook pays at least some overhead, even if the tracer intends to target only a subset of processes [11]. They evaluated three per-process tracing strategies—post-eBPF filtering, in-eBPF filtering, and pre-eBPF filtering—and showed that all can impose "excessive overhead on untraced processes" under realistic tracing workloads, with costs scaling with the number of attached hookpoints/programs [11]. Their proposed solution modifies kernel virtual memory mappings to provide per-process kernel views such that untraced processes execute as if no eBPF programs are attached, targeting "zero-untraced-overhead" tracing [11]. Framework-relevant contributions: Cost model grounding: This study makes explicit that the main cost of observability is not only the program body; it is also the hook-trigger and dispatch overhead. This is important for databases because high-frequency syscalls and networking paths are shared across co-located services. Targeting as a first-order design axis: Mode I is often presented as "safe because it only observes," but this paper shows that safety does not imply negligible overhead, and that overhead can become a system-wide externality [11]. Bridge to later modes: The same overhead logic applies

 doi:10.20944/preprints202602.0959.v1

to policy injection and kernel-resident state: any hot-path eBPF presence introduces shared costs unless carefully bounded. Limitations (framework view): Achieving the stated goal requires kernel-side changes (per-process kernel views), which shift the deployability posture compared to the conventional "load-and-attach" eBPF tracing [11]. In practice, Mode I observability may not have access to such kernel modifications.

4.2.2.3. No Two Snowflakes Are Alike: The Tracing Stack Is Part of the Experimental Condition

Machado et al. formalized a problem that most systems papers treat implicitly: two papers can claim "eBPF tracing" but implement materially different measurement stacks with different performance, resource usage, and event loss [12]. They evaluated five widely used eBPF libraries (bpftrace, BCC, libbpf, ebpf-go, and Aya) using representative I/O tracing tools and assessed their performance impact, resource efficiency, and fidelity (lost events) [12]. They reported that no single library dominates across dimensions and that higher event capture capability can correlate with a higher performance penalty, whereas some configurations can suffer severe event loss [12].

Framework-relevant contributions:

- Without explicitly reporting the tracing stack, kernel version, export mechanism, and loss accounting, two Mode I "observability" results are not necessarily comparable [12].
- Event loss can convert causal reasoning into an artifact. For database diagnosis, missing events can invert the interpretations (e.g., attributing latency to one layer because events from another are dropped).
- This work justifies MR-1's insistence on reporting export mechanisms, drop behavior, and toolchain/runtime choices.

Limitations (framework view):

The results are specific to the evaluated libraries and workloads, but the methodological implication generalizes: any Mode I measurement must treat the tracing stack as a controlled variable, not as a convenience layer.

4.2.2.4. Demystifying Performance of eBPF Network Applications: Measurement of the Integration Medium

Shahinfar et al. contributed an analysis that is Mode I in intent (understanding when eBPF helps/hurts), even though they evaluated offload scenarios as part of the measurement program [13]. They questioned whether all networked applications benefit from eBPF, measured the performance under different workloads and test cases, and reported that many applications do not benefit; eBPF can also constrain deployment because it can lead to performance isolation violations [13]. Their results attribute limitations to runtime behavior (including when hooks execute relative to demultiplexing), state access patterns, and JIT/code generation quality for some program classes [13].

Framework-relevant contributions:

- The paper treats eBPF itself as the object of measurement maps, chaining mechanisms, kerneluser communication, and JIT behavior providing a diagnostic lens for feasibility claims made by Mode II systems [13].
- They empirically demonstrated cross-application interference when offloads were executed before demultiplexing, violating performance isolation in realistic NIC queue settings [13].
- The results clarify that "offload logic to eBPF" is not a monotonic performance story; it is workload- and mechanism-dependent.

Limitations (framework view):

The results are anchored in the evaluated hooks and scenarios; applying them requires matching the hook placement and workload regime. The contribution remains diagnostic and does not propose a general isolation mechanism.

 doi:10.20944/preprints202602.0959.v1

4.2.3. Cross-System Synthesis (Mode I)

Mode I systems cluster into two operational archetypes: where work is performed and what dominates the cost.

Archetype A: Emit-heavy tracing (high-fidelity export capacity required). zns-tools is primarily emit-heavy; it collects timestamped events across multiple layers and reconstructs an end-to-end timeline offline [10]. This archetype is powerful for semantic correlation but is export-bound and sensitive to event loss and tool choices. The snowflake study shows that the choice of tracing library and export approach can drastically change both overhead and fidelity, making emit-heavy systems particularly vulnerable to "measurement stack as confounder" [12].

Archetype B: Gate/aggregate-heavy tracing (high selectivity to constrain overhead costs).Craun et al. showed that even when the tracer intends to target one process, naïve attachment can tax every process that triggers the hook, motivating more aggressive gating strategies and even kernel-side mechanisms to eliminate untraced overhead [11]. This archetype treats targeting and overhead containment as the primary objectives, accepting that some observability flexibility may be traded for operational viability.

Cross-cutting findings under the framework:

Overhead containment is not "nice to have." Mode I can degrade unrelated workloads through the shared hook execution overhead, even before considering the export volume [11].

Fidelity should be reported as a first-class metric. Library/runtime differences can produce qualitatively different event losses, invalidating comparisons if not standardized [12].

The measurement medium can be a bottleneck. For network-facing systems, Shahinfar et al. demonstrated that runtime placement and mechanism choices (e.g., pre-demux execution) can cause isolation violations and performance regressions [13].

The Mode I value is the strongest when it closes semantic gaps. The zns-tools show that cross-layer identifier translation and timeline reconstruction turn raw traces into actionable hypotheses regarding kernel-mediated behavior [10].

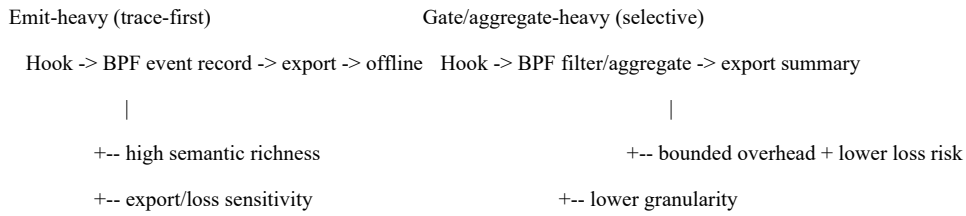


Figure 2. Observability pipeline archetypes (conceptual).

4.2.4. Observability Takeaways for DB Researchers

Mode I is sufficient when the objective is attribution, regression localization, or establishing a cost budget for subsequent integration purposes. This is insufficient when the objective requires changing kernel decisions, enforcing invariants, or guaranteeing semantics under concurrency.

Bridge Conditions checklist (Mode I → Mode II / Mode III) This checklist is the explicit bridge from "measured a problem" to "changed the kernel behavior." Passing it prevents category errors and forces feasibility reasoning. A. Bridge conditions for Mode I → Mode II (Policy Injection) All items required: Decision-point identification: the trace implicates a specific kernel choke point and decision (e.g., cache admission/eviction, pre-demux ingress handling, storage-path dispatch), not a diffuse correlation [10,13]. Causal plausibility: Evidence rules out obvious confounders (measurement perturbation, event loss, toolchain effects) via overhead accounting and fidelity reporting aligned with [11,12]. Hook feasibility: there exists a viable hook placement for control at the implicated decision point with an acceptable event rate regime and execution context constraints (bounded work, non-sleeping contexts where relevant) [13]. Policy-safety envelope: The intended intervention can be expressed as a bounded policy with explicit invariants and rollback/disable

 doi:10.20944/preprints202602.0959.v1

posture; otherwise, the proposal is not a policy injection, but an uncontrolled kernel modification. Isolation expectation: The proposal addresses shared-resource externalities (at minimum: states whether it can affect co-located workloads) motivated by observed isolation violations in eBPF-based fast paths [13]. Portability posture: the intervention states its dependency surface and upgrade assumptions; otherwise, results are non-transferable by construction (tie-in to instability evidence surfaced in Section 2.6 [20]). B. Bridge conditions for Mode I → Mode III (Kernel-Resident State) All items required: Semantic pressure: the bottlenecked decision requires coordinated multi-step state updates or invariants that cannot be safely maintained as ad-hoc map updates without a defined concurrency model (e.g., transactional invariants, protocol-state correctness) [13]. Crossing dominance: User/kernel crossings required to preserve semantics or fetch state are demonstrated to dominate tail latency or throughput at the target event rate, motivating kernel-resident state as a necessity rather than a preference [13]. Defined export boundary: persistence/complex management is explicitly kept out of the kernel-resident state (export boundary specified), preventing "kernel DBMS creep" and bounding the security surface (motivated by the state-semantics orientation of [9]). Security boundary statement: privileges, isolation assumptions, and abuse surface are explicitly stated before semantics claims are accepted, reflecting that a richer in-kernel state expands the security boundary [13,19]. Mode I outputs that do not satisfy these bridge conditions remain valid for diagnosis, but they are not a defensible justification for integration-mode escalation.

4.3. Mode II — Policy Injection: Cache, Network, and Storage Choke Points

Mode II systems use eBPF to change kernel decisions at a specific choke point using workload intent to specialize behavior where generic policies are suboptimal. The defining property is not "runs in kernel," but modifies an outcome on a high-leverage path: cache admission/eviction/prefetch, packet steering/response logic, or storage-path function execution [1–8]. Under this framework, a Mode II claim is only interpretable if it pins down four items: Decision point: the exact kernel decision being specialized (e.g., page-cache eviction, pre-stack request handling, storage function placement). Hook feasibility: hook placement and event rate regime consistent with bounded execution (tight per-event CPU budgets in networking; high-frequency cache events in page cache). State discipline: a state ladder position and contention plan that does not collapse under concurrency (per-CPU sharding, bounded shared updates, frequent/rare splitting). Invariant envelope: explicit safety envelope: what invariants are preserved, what regressions are possible under mixed workloads, and how rollback/disable is handled (Mode II changes shared kernel behavior and can induce negative externalities) [5,7,13]. The remainder of Section 4.3 applies the profile-sheet rubric and constraint/mitigation matrix to three Mode II subdomains: cache-layer injection, network-layer injection, and storage boundary injection.

4.3.2. Cache-Layer Policy Injection: Programmable Page-Cache Decisions

4.3.2.1. P2Cache

P2Cache targets a recurring database tension: applications often encode access pattern intent (hot sets, scans, phase shifts) that generic kernel caching policies do not directly observe. P2Cache's central move is to expose page-cache-relevant events and enable application-directed cache control logic using eBPF, shifting parts of the cache policy from "kernel-only generic" toward "kernel mechanisms + workload hints" [5].

Framework placement:

- Mode - Policy Injection (cache choke point).
- Typical hook points - page-cache/memory/I / O decision points (admission, eviction, prefetch/readahead).
- Keyed and coordinated state (tracking working sets, access phases or hint metadata).
- Shared-resource externalities (colocated workload regressions) and kernel coupling (interfaces deep in the memory subsystem).

 doi:10.20944/preprints202602.0959.v1

4.3.2.2. cache_ext

cache_ext addresses the same choke point but emphasizes policy generality by enabling a broad set of page-cache customizations using eBPF while remaining feasible at high event rates. The architectural argument is that pushing policy to user space is often infeasible because cache events occur too frequently; the policy must be executed in-kernel, with bounded per-event overhead, and with state structures designed for concurrency [7].

Framework placement:

- Mode: Policy Injection (cache choke point).
- Typical hook points: page cache events for admission, access, eviction, and removal decisions.
- Coordinated shared updates (eviction structures) with strong pressure to shard and bound the contention.
- Contention collapse and regressions under mixed tenancy, plus long-term stability issues when interfaces touch internal kernel cache machinery.

4.3.2.3. Cache-Layer Comparison (Normalized)

Table 9. Cache-layer policy injection comparison (framework-normalized).

Dimension	P2Cache [5]	cache_ext [7]
Integration intent	Application-directed cache decisions	Generalizable in-kernel cache policy customization
Hook regime	Page-cache decision points	Page-cache decision points (high-frequency event regime)
Control boundary	Explicit workload intent is part of the design contract	Emphasis on implementable policy space inside kernel
State pressure	Keyed hints + policy state	Coordinated eviction/admission structures; contention management
Dominant feasibility constraint	Mixed-workload externalities; interface coupling	Hot-path CPU budget + contention; interface coupling
Typical mitigation posture	Bound policy scope; rely on intent to reduce misprediction	Keep policy in-kernel to avoid crossings; structure state to reduce contention

The key distinction is the interface contract: P2Cache centers the application as a policy contributor, and cache_ext centers the kernel as a programmable policy host. Both face the same first-order feasibility boundary: any cache-layer injection must treat multi-tenant interference and event-rate scaling as primary, not incidental [5,7].

4.3.3 Network-layer policy injection: fast paths, protocol handling, and request offload Network-layer Mode II systems move logic into pre-stack or early-stack hooks to reduce per-request overhead and tail latency. The feasibility boundary is strict: per-packet overhead compounds into throughput ceilings and per-core saturation under high-QPS regimes [13].

4.3.3.1 BMC accelerates memcached by moving parts of request handling into the kernel and performing pre-stack processing. This demonstrates the performance appeal of in-kernel handling for a subset of simple, high-volume requests that are typical of key-value services [1]. Framework placement: Mode: Policy Injection (network choke point). Hook regime: Early packet-processing hooks (pre-stack). State ladder: keyed per-flow/request state with bounded logic. Dominant risk: restricted semantic coverage (fast path handles only some operations) and correctness envelope definition.

4.3.3.2 Electrode Electrode targets distributed protocol acceleration using eBPF, shifting the protocol machinery closer to the packet-processing path. It generalizes the “kernel fast path” beyond a single application by treating protocol steps as candidates for bounded in-kernel execution when

 doi:10.20944/preprints202602.0959.v1

they can be expressed safely and efficiently [3]. Framework placement: Mode: Policy Injection (network/protocol choke points). Hook regime: packet-processing hooks and bounded protocol-step handlers. State ladder: semantics-bearing state (protocol progress) with strong pressure to keep the state small and contention bounded. Dominant risk: Protocol state correctness and interference/isolation when multiple services share ingress resources.

4.3.3.3. DINT

DINT offloads frequent-path distributed transaction processing into XDP/eBPF, maintaining the hot-path state in kernel-accessible structures while punting complex or rare cases to user space. The system embodies a common Mode II pattern: frequent-path acceleration + rare-path fallback [6].

Framework placement:

- Mode: Policy Injection (network fast path) with explicit boundary conditions that begin to approach Mode III.
- Hook regime: XDP ingress (very high event rate regime).
- Semantics-bearing state (transaction request handling) managed under strict bounded execution constraints.
- Semantic drift between fast and slow paths; state pressure that pushes beyond ad-hoc map updates.
-

4.3.3.4. eNetSTL

eNetSTL focuses on systemization by making eBPF-based in-kernel network functions more practical through library-like support. Under this framework, a feasibility boundary often omitted by pure performance papers is addressed: engineering complexity and the cost of repeatedly rebuilding safe and efficient components for state management and packet handling [8].

Framework placement:

- **Mode:** Policy Injection enabler (network fast path).
- **Hook regime:** network function attachment points.
- **State ladder:** varies; aims to make higher-level constructs feasible without bespoke reinvention.
- **Dominant risks:** correctness-by-construction and predictable performance under constrained contexts.
-

4.3.3.5. Cross-Comparison: The Dominant Pattern Is Frequent/Rare Partitioning

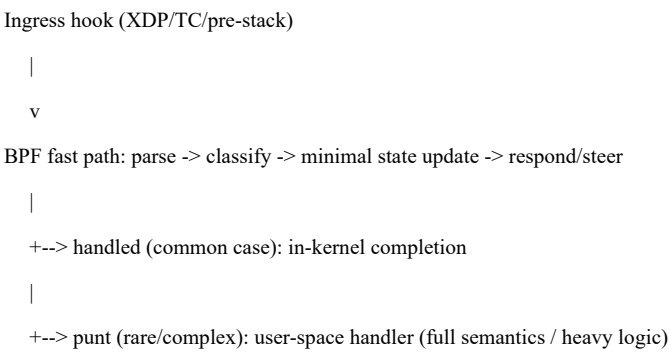


Figure 3.

This architecture is visible across application-specific acceleration and transactional request offloading [1,6]. The electrode extends the "what belongs in kernel" boundary toward the protocol machinery when the steps are bounded, and the state is disciplined [3]. The eNetSTL targets reuse and correctness scaffolding for fast path designs [8].

 doi:10.20944/preprints202602.0959.v1

State Pressure Indicators (Mode II → Mode III trigger signals)

The following indicators mark when network-layer policy injection stops being "bounded fast-path policy" and begins requiring semantics-bearing kernel-resident state (Mode III), because ad-hoc maps + frequent/rare splitting becomes fragile:

Multi-object atomicity: correctness requires updating multiple keys/records/metadata atomically per request (not just per flow counters).

Isolation requirement: Concurrent requests must observe a serializable (or otherwise specified) order; "best-effort" ordering is insufficient.

Conflict detection/retry loops: The fast path must detect conflicts and coordinate retries without unbounded loops (verifier pressure).

Cross-core coordination: correctness depends on coordination across CPU cores/queues beyond the simple per-CPU state, increasing contention risk.

Idempotency and replay handling: The system must handle duplicates, timeouts, and replays with strict semantics, requiring durable or structured state transitions.

Protocol state machines with invariants: the state must evolve through defined phases and reject invalid transitions; map updates become a de facto DBMS.

Export boundary pressure: correctness requires logging/exporting state transitions (for recovery or audit), pushing designs toward explicit WAL/export interfaces.

Security boundary expansion: Richer request-handling logic and state increase the kernel-side attack surface, making governance and hardening constraints primary [19].

DINT explicitly sits near this boundary: it uses kernel fast paths for frequent cases while maintaining a user-space fallback for complexity, illustrating how semantic pressure can be managed by partitioning until it cannot [6].

4.3.4. Storage-Boundary Policy Injection: In-Kernel Storage Functions and Network-Attached Pushdown

Storage-boundary Mode II systems target the I/O path directly, either by running storage functions in the kernel or by pushing computation toward storage endpoints in a distributed setting.

4.3.4.1. XRP

XRP proposes in-kernel storage functions using eBPF, framing eBPF as a mechanism for executing bounded logic associated with storage operations and I/O processing paths [2]. The intent is policy/function placement near storage events to reduce the overhead and enable customized behaviors.

Framework placement:

- **Mode:** Policy Injection (storage boundary).
- **Hook regime:** storage-path function attachment points (I/O submission/completion-adjacent).
- **State ladder:** keyed state and bounded semantics-bearing logic constrained by storage-path correctness expectations.
- **Dominant risks:** Semantic interactions with ordering/durability expectations and stack heterogeneity.
-

4.3.4.2 BPF-oF BPF-oF explores storage function pushdown over the network, extending the idea of "policy/function injection" to network-attached storage contexts, where data movement costs dominate and computation placement becomes part of the performance story [4]. Framework placement: Mode: Policy Injection (storage boundary + network-attached pushdown). Hook regime: network/storage boundary hooks, where pushdown is enforced. State ladder: varies; semantics-bearing states can be included when coordination is required. Dominant risks: Portability across heterogeneous endpoints and correct envelopes in distributed storage paths. 4.3.4.3 Storage-boundary comparison (normalized)

 doi:10.20944/preprints202602.0959.v1

Table 10. Storage-boundary injection summary (framework-normalized).

Dimension	XRP [2]	BPF-oF [4]
Integration intent	In-kernel storage functions	Storage function pushdown over network
Choke point	Local storage/I/O path	Distributed storage boundary
Primary value	Reduce overhead; enable bounded I/O-path functions	Reduce data movement; place computation near storage endpoints
Dominant constraints	Ordering/durability interactions; device/stack heterogeneity	Endpoint heterogeneity; distributed correctness envelope
Portability posture	Sensitive to kernel/storage stack interfaces	Sensitive to network/storage endpoint capabilities and interfa`ces

Storage-boundary injection is not a fourth mode of operation. This is Mode II applied to a different choke point, parallel to cache- and network-layer injection [2,4].

4.3.5. Mode II Cross-System Synthesis and Feasibility Gates

Synthesis: where Mode II wins and where it fails

Mode II tends to be justified when

- The decision point is on a hot path, and user/kernel crossings are the bottleneck, making in-kernel specialization the only plausible way to reduce tail latency or CPU overhead at the target QPS [6,7,13].
- The specialized logic is bounded (small instruction footprint, predictable state access) and can be expressed without escalating to Mode III semantics.
- The system explicitly manages externalities - cache policy changes consider mixed workload effects; network fast paths address performance isolation; and storage pushdown declares ordering/durability envelopes [2,5,7,13].

Mode II failures cluster around three patterns.

- The injected policy shifts the cost rather than removing it (e.g., state contention, export overhead, per-packet overhead) and saturates cores earlier than expected [13].
- The "policy" becomes an implicit state machine with correctness requirements that exceed ad-hoc maps and bounded execution, creating brittle designs (Mode II drifting into Mode III without acknowledging it) [6].
- Dependency-surface instability and insufficient reporting make it difficult to reproduce or deploy results across kernel versions and fleets [20].

Feasibility Gates (Mode II)

A Mode II claim should be treated as feasible and comparable only if it passes through the following gates:

The paper names the exact kernel decision being changed and why it is the correct choke point (cache/network/storage) [1–8].

Attachment type(s) are explicitly stated, with execution context constraints (e.g., XDP vs. TC vs. page-cache hooks) and qualitative event-rate regime [6,7,13].

There is an explicit bound on per-event work, and the evaluation shows that it holds under target load (no hidden "slow path" frequency increase).

The state level is identified (keyed/coordinated/semantics-bearing), including cardinality bounds and a contention strategy (per-CPU sharding, partitioning, minimal shared updates) [6,7].

The paper states what correctness invariants are preserved and what semantics are intentionally excluded; for fast paths, this includes explicit frequent/rare split boundaries [1,6].

Mixed-workload regressions and isolation effects are measured or bounded; "wins on one workload" are not sufficient for shared kernel resources [5,7,13].

 doi:10.20944/preprints202602.0959.v1

Privileges required to load/attach, the attack surface added by the injected logic, and any guardrails are stated (especially for logic executed before demultiplexing or on shared ingress resources) [13,19].

Kernel-version assumptions and dependency-surface exposure are described; otherwise, deployability is undefined, and comparisons are unstable [20].

Kernel/hardware details, attachment points, map sizes/types, export/drop handling, workload configuration, and baseline configuration were reported (MR-1).

There is a clear operational control to disable the injection and revert to the baseline behavior without kernel rebuild (required for practical feasibility).

Passing these gates does not guarantee superiority; it guarantees interpretability and makes cross-system comparisons more principled.

4.4. Mode III - Kernel-Resident State: Semantics-Bearing State and Transactional Consistency

4.4.1. Mode III Overview

Mode III exists when policy injection alone is insufficient because the in-kernel logic requires structured, correctness-defining state transitions that exceed "bounded policy state." The defining move is not "more state," but state with semantics: explicit concurrency/correctness guarantees and a defined boundary to user space for persistence and management [9].

Under this framework, Mode III differs from Mode II along two axes:

- Mode II specializes in kernel decisions, and Mode III provides semantics-bearing state services to ensure that certain classes of kernel-resident logic are correct under concurrency.
- Mode II fails as a performance regression or interference, and Mode III fails as correctness violations, semantic drift, or unacceptable security boundary expansion.

This mode is inherently coupled to feasibility constraints, such as verifier-bounded computation, synchronization restrictions, and the security implications of richer in-kernel services [15,19].

4.4.2. System Profiles (Mode III)

4.4.2.1. BPF-DB

BPF-DB proposes a kernel-embedded transactional DBMS for eBPF programs, providing an ACID key-value interface and exporting a write-ahead log boundary to the user space [9]. Under the state ladder, it occupies the top rung: transactional semantics designed to support in-kernel programs that require coordinated and correct state evolution beyond ad-hoc map operations.

Framework placement:

- **Mode:** Kernel resident state.
- **Hook regime:** paired with arbitrary eBPF trigger points; the key contribution is a semantics-bearing state, not a specific hook.
- **State ladder:** Transactional.
- **Dominant cost drivers:** synchronization/coordination overhead, state footprint, and export boundary overhead (WAL export).
- **Dominant feasibility risks:** verifier/synchronization constraints and expanded security boundaries associated with in-kernel transactional services [15,19].

4.4.2.2. DINT as a Contrast Case

DINT is primarily a Mode II network fast-path system, but it is a useful contrast because it demonstrates the "near-Mode-III" approach: keeping enough state in the kernel to accelerate frequent-path transaction handling and punting complex cases to user space [6]. This illustrates how semantic pressure can be managed without introducing a general transactional substrateuntil the pressure indicators in Section 4.3.3 force escalation.

Framework placement:

 doi:10.20944/preprints202602.0959.v1

- **Mode:** Policy Injection with semantics-bearing state in the fast path.
- **State ladder:** semantics-bearing but not general transactional substrate.
- **Core trade-off:** avoids Mode-III complexity by limiting coverage and using fallback at the cost of semantic boundary management between fast and slow paths [6].
-

4.4.2.3. Normalized Comparison

Table 11. Mode III comparison: transactional substrate vs bounded fast-path state.

Dimension	BPF-DB [9]	DINT (contrast) [6]
Primary intent	Provide transactional semantics for eBPF programs	Accelerate frequent-path transaction handling
Semantics stance	Explicit ACID/transaction interface + export boundary	Frequent/rare split; semantics managed by partitioning
State ladder	Transactional	Semantics-bearing (bounded)
Where complexity lives	In-kernel state service + WAL/export integration	Boundary management between fast path and user-space fallback
Dominant feasibility risk	Sync/verifier feasibility + security boundary expansion [15,19]	Semantic drift; coverage gaps; contention at hot rates
When it is justified	Semantic pressure requires coordinated multi-step updates	Frequent-path dominates and can be safely bounded

The framework implication is direct: Mode III is not "better Mode II." It is a different commitment: it trades engineering and security complexity for stronger semantics and potentially simpler correctness arguments in the fast path.

4.4.3. Cross-System Synthesis (Mode III)

When Mode III is required

Mode III becomes defensible when at least one of the following holds:

- Correctness requires atomic multi-step updates that cannot be maintained by ad hoc maps without a defined concurrency model.
- Frequent/rare partitioning no longer isolates complexity because "rare" becomes common under contention, skew, or adversarial patterns, turning the fallback into the dominant path [6].
- Semantic pressure indicators accumulate (Section 4.3.3): isolation, conflict handling, replay safety, cross-core coordination, and explicit log/export requirements.
- The integration must expose a clean user-space boundary for durability or management, making an export model (e.g., WAL export) a part of the design contract [9].

Cost and feasibility drivers in Mode III

Mode III cost is dominated by the semantics-bearing state and tends to require coordinated updates; contention becomes a first-order bottleneck at high trigger rates [15,19]. Transactional or richer states increase memory pressure and cache effects, which can erase performance wins if not bounded. WAL/export mechanisms introduce cross-boundary costs that must be explicitly budgeted and measured, not assumed to be negligible [9].

Security boundary as a first-class constraint

Richer in-kernel state services expand the attack surface and privilege the consequences of bugs or abuse. Therefore, Mode III systems must treat the security boundary as part of feasibility, not as a separate "security paper concern," aligning with broader eBPF safety and memory-safety discussions [19].

Practical synthesis Mode II is the default for "change a kernel decision on a choke point." Mode III is justified only when the problem is not the decision itself but correctness of state evolution under concurrency in kernel-resident logic, and when bounded fast-path partitioning is insufficient [6,9].

 doi:10.20944/preprints202602.0959.v1

4.5. Cross-Mode Comparison and Decision Procedure Validation

4.5.1 Cross-mode matrix This subsection places the representative systems into a single normalized view: hook regime × event rate regime × state ladder × dominant feasibility boundary × mitigation posture. This makes cross-mode comparisons possible without collapsing distinct intents into a single "eBPF performance" axis.

Table 12. Foundational feasibility works [14] – [21] are not placed as "systems," but they anchor the constraint and mitigation columns (verification, safety, portability) and explain why the same feasibility boundaries recur across modes [15,19,20].

System	Mode	Choke point	Typical hook regime	Event-rate regime	State ladder level	Dominant feasibility boundary	Primary mitigation posture
BMC [1]	II	Network	Pre-stack / early packet path	Very high	Keyed / bounded semantics	Per-packet CPU budget; coverage limits	Fast-path subset; bounded parsing/handling
XRP [2]	II	Storage boundary	I/O-path storage-function hooks	Medium–High	Keyed / bounded semantics	Heterogeneity; ordering/durability envelope	Restrict function scope; place logic where semantics are exposed
Electrode [3]	II	Network/protocol	Packet-processing hooks	High	Semantics-bearing state	Correctness under bounded execution; isolation	Protocol-step bounding; state minimization
BPF-oF [4]	II	Storage boundary (distributed)	Network-attached pushdown boundary	Medium–High	Keyed / semantics-bearing (varies)	Endpoint heterogeneity; distributed envelope	Pushdown scoping; capability-aware deployment
P2Cache [5]	II	Cache	Page-cache policy points	High	Keyed / coordinated	Mixed-workload externalities; kernel coupling	Explicit intent/hints; bounded policy scope
DINT [6]	II (boundary case)	Network (transactions)	XDP ingress	Very high	Semantics-bearing state	Semantics vs bounded execution; crossings	Frequent/rare split; bounded in-kernel work
cache_ext [7]	II	Cache	Page-cache policy points	High	Coordinated shared updates	Contention + hot-path CPU; kernel coupling	In-kernel policy to avoid crossings; shard/bound structures
eNetSTL [8]	II (enabler)	Network	Network-function hooks	High	Varies	Engineering complexity under constraints	Library-style components to reduce bespoke errors/costs
BPF-DB [9]	III	Kernel-resident state service	Paired with triggers (varies)	Varies (often high)	Transactional	Verifier/sync feasibility; security boundary	Structured semantics + explicit export boundary (WAL)
zns-tools [10]	I	Observability (storage)	Cross-layer probes + app probes	Medium–High	Keyed traces	Export/fidelity; semantic translation	Cross-layer correlation; offline reconstruction
Untraced overhead elimination [11]	I	Observability feasibility	Trace hooks + targeting mechanisms	High	Minimal gating state	System-wide overhead externalities	Target isolation; eliminate untraced overhead paths
Library variance study [12]	I	Observability comparability	Multiple tracing stacks	High	Keyed traces	Fidelity variance; toolchain confounding	Treat library/runtime as experimental variable

 doi:10.20944/preprints202602.0959.v1

Network eBPF perf analysis [13]	I	Observability (medium analysis)	XDP/comm-path microbenchmarks	Very high	Keyed state	Isolation violations; runtime-dependent costs	Cost decomposition; isolation-sensitive placement analysis

4.5.2. Validate the Decision Flowchart Against Case Studies

This subsection validates that the decision procedure (Section 3.10) consistently classifies real systems and explicitly flags boundary cases rather than implicitly.

Mode I leaf (observability)

Systems that primarily measure/diagnose without changing kernel decisions fall cleanly into Mode I.

- Cross-layer storage attribution and timeline reconstruction [10].
- Overhead containment is a prerequisite for valid measurement (untraced-process overhead externality) [11].
- Measurement stack variance is a confounder that requires explicit control and reporting [12].
- Characterization of eBPF network application costs and isolation effects as feasibility evidence rather than an acceleration claim [13].

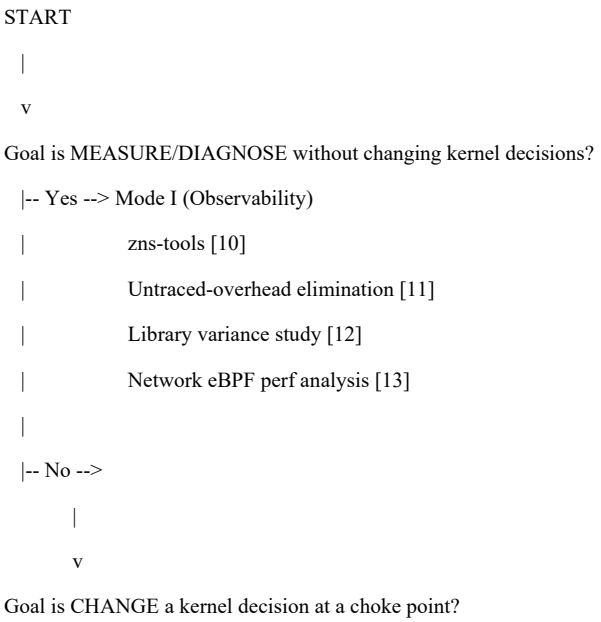
Mode II leaf (Policy Injection)

Systems that change a kernel decision at a specific choke point fall into Mode II.

- Cache policy specialization (admission/eviction/prefetch) [5,7].
- Network fast-path specialization (application-specific handling, protocol steps, and transactional frequent-path handling) [1,3,6,8].
- Storage-boundary function injection/pushdown as a storage-layer policy injection [2,4].
-

Mode III leaf (kernel-resident state). Systems whose primary contribution is **semantics-bearing state services** (beyond ad-hoc maps), with explicit concurrency and export boundaries, fall into Mode III [9].

Boundary case resolution (DINT) is defined asDINT is classified as Mode II because its primary objective is ingress **fast-path specialization,** and it relies on a frequent/rare split with a user-space fallback rather than providing a general semantics-bearing state substrate [6]. The framework still treats DINT as a *boundary case*: its design triggers multiple state-pressure indicators (Section 4.3.3), which is precisely why Mode III exists as a separate category [6,9].



 doi:10.20944/preprints202602.0959.v1

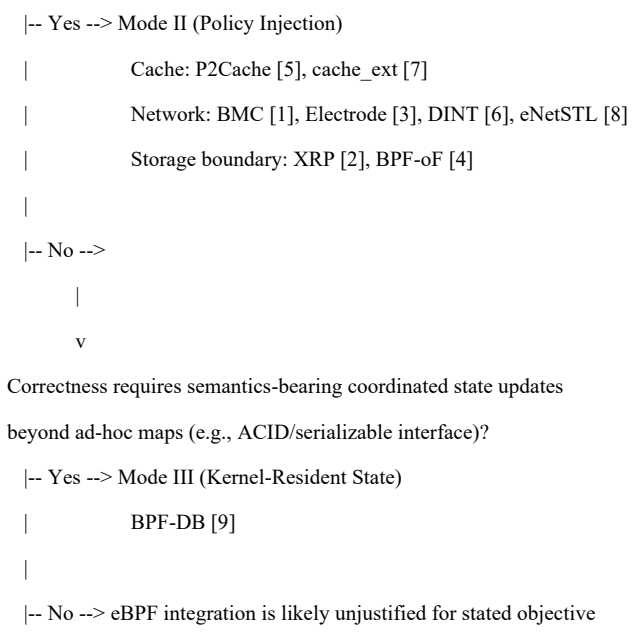


Figure 4. Decision flowchart annotated with representative systems (concrete validation).

The validation outcome is that the taxonomy partitions the corpus without introducing a fourth category: storage pushdown remains policy injection at a different choke point, and transactional state remains a separate mode driven by semantic pressure rather than by hook placement [2,4,9].

4.5.3. MR-1 Compliance Scan with Compliance Score

MR-1 (Section 3.12) was used as the interpretability gate. Two empirical observations motivate treating MR-1 as non-optional: (i) tracing overhead can bleed onto untraced processes if targeting is not handled correctly [11] and (ii) toolchain/library choices can alter fidelity and resource usage, turning the measurement stack into a confounder if not controlled and reported [12]. Portability risk further requires explicit kernel/version assumptions because dependency mismatches are pervasive in real-world eBPF extensions [20].

Compliance score definition

We define a compliance score (CSCSCS) on a 0–10 scale computed from MR1 disclosure as follows:

- **(2 pts)** Kernel/platform disclosure
- **(2 pts)** Attachment disclosure
- **(2 pts)** State/export disclosure
- **(1 pt)** Runtime controls
- **(2 pts)** Workloads/baselines disclosure
- **(1 pt)** Metrics/variance disclosure

Table 13. Observed distribution and interpretation.

title	year	kernel_platform	attach	state_export	controls	workloads_baselines	metrics_variance	compliance_score
BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing	2021	2	0	1	1	1	0	5
XRP: In-Kernel Storage Functions with eBPF	2022	1	1	0	1	2	0	5
Electrode: Accelerating Distributed Protocols with eBPF	2023	2	1	2	1	0	0	6
BPF-oF: Storage Function Pushdown Over the Network	2023	1	1	1	1	1	0	5
P2Cache: Application-Directed Page Cache Optimization	2023	2	1	0	1	1	0	5

 doi:10.20944/preprints202602.0959.v1

DINT: Fast In-Kernel Distributed Transactions with eBPF	2024	2	2	1	1	2	0	8
cache_ext: Customizing the Page Cache with eBPF	2025	1	0	0	0	0	0	1
eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions	2025	2	2	1	1	1	0	7
BPF-DB: A Kernel-Embedded Transactional DBMS for eBPF Applications	2025	2	0	1	1	1	0	5
zns-tools: eBPF-Powered Cross-Layer Storage Profiling for ZNS SSDs	2024	1	1	0	1	1	0	4
Eliminating eBPF Tracing Overhead on Untraced Processes	2024	1	0	0	0	1	1	3
No Two Snowflakes Are Alike: Studying eBPF Librariesâ€™ Performance, Fidelity and Resource Usage	2025	2	1	1	1	1	0	6
Demystifying Performance of eBPF Network Applications	2025	2	2	1	1	1	0	7

Using the interpretability bands defined earlier:

- **CS≥8CS \ge 8CS≥8: 1/13 papers** (DINT [6])
- **5≤CS<85 \le CS < 85≤CS<8: 9/13 papers**
- **CS<5CS < 5CS<5: 3/13** papers (cache_ext [7], zns-tools [10], Untraced-overhead elimination [11])

Two disclosure gaps dominate this corpus under strict scoring.

- **Metrics/variance disclosure:** Only the of **1/13** papers scored the variance criterion (Table 13). This severely limits cross-paper quantitative synthesis, especially for tail-latency assertions.
- **Attachment specificity and state/export details:** Several systems score on attach or state/export, which blocks precise cost attribution and complicates the replication of “fast path” vs “control path” effects (Table 13).

The following interpretation rules were used throughout the study:

- CS≥8CS \ge 8CS≥8: results were comparable with minimal ambiguity.
- 5≤CS<85 \le CS < 85≤CS<8: results are usable but require qualification; missing details likely affect cost/fidelity/portability.
- CS<5CS < 5CS<5: results are discussed qualitatively and not used for cross-system quantitative conclusions.

4.6. Consolidated Takeaways as Numbered Decision Rules

The integration mode is explicitly stated. Every claim must declare whether it is an observability (Mode I), policy injection (Mode II), or kernel-resident state (Mode III). Mode ambiguity is treated as a methodological defect. [10–13]

Hook placement was considered the primary cost determinant. A “fast” eBPF program attached to the wrong hook can be infeasible; a “complex” program attached to a low-rate hook can be acceptable. The Hook and event-rate regimes precede the algorithm discussion. [13]

Use Mode I when the objective is attribution, not control. Mode I is sufficient for localizing bottlenecks and cross-layer semantic mismatches only when the measurement perturbation and fidelity are controlled and reported. [10–12]

Do not escalate from Mode I to Mode II/III without the bridge conditions. Mode I traces that do not isolate a control-feasible kernel decision point are not a defensible basis for policy injection or a semantics-bearing state. [10–13]

Mode II requires a named kernel decision and a bounded invariant envelope. “Improves performance” is not a Mode II statement. The paper must specify the decision being changed, the invariants preserved, and the rollback/disable position. [1–8]

For cache layer injection, externalities are measured as first-order outcomes. Page cache specialization must quantify mixed-workload interference and regressions; single-workload wins are not general claims on a shared kernel resource. [5,7]

 doi:10.20944/preprints202602.0959.v1

For network-layer injection, a strict per-packet CPU budget and isolation analysis are enforced. Ingress-path execution can saturate cores and violate performance isolation, depending on the placement. Explicitly report fast-path coverage and slow-path frequency. [1,3,6,13]

Use state pressure indicators as a hard stop for Mode II complexity growth. When multi-object atomicity, isolation requirements, replay safety, cross-core coordination, or log/export needs become correctness drivers, stop expanding the ad-hoc map logic. The partition boundary was redesigned or escalated to Mode III. [6,9]

Mode III must precisely define semantics and bound the export boundary. Transactional or semantics-bearing states in the kernel require explicit guarantees, a concurrency model, and a defined user-space boundary (e.g., log/WAL export), not implicit persistence assumptions. [9]

Treat verifier and synchronization constraints as design constraints rather than implementation details. Bounded computation and restricted synchronization shape feasible architectures, restructuring into bounded phases and shard states to avoid contention collapse. [15,19]

Treat the dependency-surface stability as a deployment constraint. Kernel versions and dependency surfaces must be declared, and portability must be argued and tested rather than assumed. [20]

Gate comparability on MR-1 with a compliance score and constrained quantitative synthesis accordingly**.** In this corpus (Table 13), only **1/13** systems reached **CS ≥ 8**, **9/13** fell in **5 ≤ CS < 8**, and **3/13** fell below **CS < 5**. The dominant disclosure gap is metrics/variance (satisfied by **1/13**), followed by incomplete state/export disclosure (full credit in **1/13** and zero credit in **5/13**) and missing attach specificity (zero credit in **4/13**). Cross-paper quantitative claims—especially tail-latency comparisons—must therefore be restricted to (i) within-paper baselines or (ii) the high-compliance subset; low-compliance systems remain usable for qualitative design insights but not for numerical cross-system rankings. This strict gating is justified because overhead bleed-through and tooling variance are known confounders, and dependency instability renders underspecified setups non-portable. [11,12,20]

Stop condition: Avoid eBPF integration when it does not change the feasibility frontier. If the objective can be met with Mode I measurement alone or with user-space changes without hot-hook execution under strict budgets and governance constraints, kernel integration is not a contribution. [13,19]

5. Discussion

5.1. Why it Worked

Across the 2021–2025 corpus, eBPF is used for three different intents that are routinely conflated: measurement (Mode I), control via policy injection (Mode II), and semantics-bearing kernel-resident state (Mode III). Partitioning by intent forces each system to be judged against the correct evidence standard: fidelity/perturbation for observability [11,12]; bounded invariants and externalities for policy injection [5,7]; and explicit semantics plus export boundaries for kernel-resident state [9]. This prevents category errors, such as using Mode I traces to justify kernel policy changes without a feasibility analysis [10–13]. Once the systems are normalized by the hook and event-rate regimes, most performance and failure outcomes become structurally predictable. XDP and early ingress hooks live on a per-packet critical path and expose isolation risks depending on their placement relative to demultiplexing [13]. Page-cache decision points are high-frequency and shared, making contention and mixed workload externalities first-order constraints [5,7]. Tracing hooks can impose system-wide overhead even when “targeted,” because of dispatch and filtering placement [11]. The framework works because it places hook placement ahead of algorithmic details. Mode-II systems succeed when user/kernel crossings or generic kernel policies dominate the hot path. cache_ext argues that user-space feedback loops are infeasible at cache-event rates and therefore keeps the policy in-kernel with bounded work and a disciplined state [7]. DINT maintains frequent-path transaction handling in XDP and punts rare cases to user space, explicitly managing the boundary

 doi:10.20944/preprints202602.0959.v1

[6]. The benefit is not that "kernel execution is inherently faster," but "avoid repeated crossings and generic policy mismatch where the kernel owns the decision" [6,7,13].

Ad-hoc map usage works until the correctness demands coordinated multi-step updates under concurrency. BPF-DB makes semantics explicit (transactional interface with WAL export), shifting correctness from "implicit map behavior" to "defined state transitions + explicit boundary to user space" [9]. This is a different commitment than the Mode II fast-path policy injection and carries different feasibility and security burdens [15,19].

5.2. What the Results Imply

Implication 1: Table 13 shows a severe disclosure gap under strict MR-1 scoring: only **1/13** systems reports variance in a way that satisfies the MR-1 metrics/variance criterion, and only **1/13** achieves **CS ≥ 8**. Most systems land in the "usable but qualified" band (**9/13** with **5 ≤ CS < 8**), while **3/13** fall below **CS < 5**. This does not invalidate the systems, but it blocks reliable numeric cross-paper comparisons, especially for tail latency claims. The constraint is consistent with known confounders: tracing can perturb non-target workloads [11], tracing stacks differ materially in terms of fidelity and overhead [12], and dependency-surface instability can silently change behavior across kernels [20].

Implication 2: The most common omissions are (i) variance reporting, (ii) attach specificity (0 score for **4/13**), and (iii) state/export disclosure (0 score for **5/13**). These omissions map directly onto the framework's feasibility drivers: without attach specificity, hook placement, and event-rate regime cannot be reproduced or compared; without state/export details, contention and export-path costs cannot be budgeted; without variance, tail-latency conclusions are fragile.

Implication 3: Mode II work is divided cleanly by choke point: page cache [5,7], networking ingress/stack [1,3,6,8], and storage boundary [2,4]. Each choke point has distinct invariants and externalities. Treating them as a single category encourages overgeneralized conclusions. Mode III is orthogonal: it is driven by semantics pressure rather than by the location of the hook [6,9].

Implication 4: Many high-performance systems succeed by handling a bounded frequent path in-kernel and punting rare cases to user space [1,6]. The breaking point appears when correctness requires atomic multi-object updates, isolation, replay safety or explicit logging/export semantics. At that point, continuing to expand ad hoc map logic increases correctness risk faster than it increases performance [6,9].

5.3. Where it Outperforms Existing Methods

Mode I: observability vs user-space-only diagnosis

Mode I outperforms user-space-only profiling when the bottleneck is mediated by kernel subsystems and attribution requires cross-layer correlation. The zns-tools demonstrate practical cross-layer reconstruction across the storage stack, enabling explanations that are not recoverable from user space metrics alone [10]. The outperformance boundary is operational: if tracing overhead leaks onto untraced processes, it becomes unsuitable for shared environments [11]; if the tracing stack changes fidelity and overhead, the results are not comparable and may be misleading [12].

Mode II: policy injection vs user-space workarounds and generic kernel policy

Mode II outperforms user-space workarounds when the kernel owns the decision point, and the event rate renders user/kernel feedback loops infeasible. Cache policy injection can outperform generic caching under workload-specific access patterns when admission/eviction/prefetch policy mismatches dominate the outcomes [5,7]. Network fast-path injection can outperform user-space handling for common-case requests by avoiding repeated traversal overhead and boundary crossings, provided that the per-packet work remains bounded and the isolation effects are managed [1,6,13]. Storage-boundary injection can outperform higher-layer approaches when computation placement and data movement dominate the cost; however, feasibility is narrower because of heterogeneity and ordering/durability envelopes [2,4].

Mode III: kernel-resident semantics vs ad-hoc maps + expanding fast paths

 doi:10.20944/preprints202602.0959.v1

Mode III outperforms ad hoc map-centric designs when correctness requires explicit semantics under concurrency. The BPF-DB provides a transactional interface and WAL export boundary designed to support correctness claims that do not scale with ad-hoc map discipline [9]. This outperformance is conditional; it trades higher feasibility and security burden for stronger semantics and simpler correctness arguments [15,19].

5.4. Practical Implications

For systems authors and database researchers

- Mode selection is a correct decision, not a presentation choice. Treat Mode I evidence as diagnosis unless bridge conditions are met; treat Mode II as kernel behavior change requiring invariant envelopes and externality evaluation; treat Mode III as semantics work requiring explicit guarantees and an export boundary [5–7,9–13].
- MR-1 is not optional if the results are meant to be generalized. Table 13 shows that missing variance reporting and underspecified attachment/state/export details are the norm, which blocks rigorous cross-paper quantitative conclusions, even when headline metrics look compelling. This is predictable given the known confounders in tracing overhead and tooling variance [11,12] and known instability risks [20].
- Externalities are the primary outcomes of shared choke points. Cache policy work must measure mixed workload regressions and interference; network fast-path work must report isolation effects and slow-path frequency; storage pushdown must bound ordering and deployment assumptions [2,4,5,7,13].

For platform operators

- Treat eBPF programs as privileged production codes, whose risk increases sharply from Mode I to Mode III. The security boundary expands with statefulness and semantics-bearing logic; governance and least privilege are important [19].
- Prefer interventions that are bounded, disabled, and version-scoped. Dependency surface instability makes implicit portability assumptions unsafe; operational feasibility requires an explicit kernel/version posture [20].
- Demand disclosure artifacts are required. A system that cannot disclose attach sites, state/export budgets, and variance is not operationally characterizable under the load.

For kernel and eBPF ecosystem designers

Stability and isolation dominate the adoption process. Interface instability and dependency mismatches limit durable deployments [20]. Isolation failures and shared overhead effects limit safe multi-tenant use in both observability and fast-path execution [11,13]. Mechanisms that reduce dependency surfaces and enforce isolation expand the feasible design space more than incremental micro-optimizations.

6. Limitations

This study provides a framework and structured application to representative eBPF–database systems (2021–2025). The limitations below describe what this study does not claim to solve and what bounds the applicability of its conclusions.

6.1. What the Framework Does Not Handle

The analysis assumes the Linux eBPF semantics, hook families, and verifier-enforced constraints. The results do not directly transfer to other kernels or extension systems without re-deriving the hook/constraint model. Feasibility framing depends on eBPF's verifier and runtime properties, as studied in prior work [15,19]. The three modes deliberately partition the *database–kernel integration intents*: observability, policy injection, and kernel-resident state. They do not attempt to classify unrelated eBPF domains (e.g., generic traffic filtering, purely security-only enforcement) except

 doi:10.20944/preprints202602.0959.v1

where these affect the feasibility and security boundary [19]. This study normalizes comparisons using hook placement, state ladder, constraint/mitigation matrix, and a unified cost model. It does **not** attempt to compute a single cross-paper leaderboard of throughput/latency improvements because workloads, hardware, kernels, and tracing stacks differ in ways that can dominate the outcomes if treated as comparable units [12]. This study is analytical and relies on paper-reported evaluation evidence. While MR-1 provides a minimal disclosure standard, this study does not execute or reproduce each system to verify the claims. The feasible design space is shaped by the verifier capabilities, bounded computation requirements, and synchronization restrictions. As these mechanisms evolve (e.g., changes in verifier reasoning or toolchains), the precise boundary between "feasible" and "infeasible" can shift, although the framework's insistence on treating these as first-order constraints remains stable [15,19,21]. Deep integration into page cache, networking paths, or storage interfaces increases the reliance on kernel internals. The framework flags this as a feasibility constraint, but does not remove it; portability and upgrade resilience remain open operational problems [20]. The framework requires externality evaluation for Mode II systems (mixed-workload regressions, isolation violations); however, it does not provide a unified quantitative interference model. Isolation properties remain hook- and placement-dependent, as shown in the network eBPF performance analyses [13].

6.3. Data and Deployment Limitations (Now Grounded in Table 13)

Table 13 shows that only **1/13** systems achieves **CS ≥ 8**,**9/13** fall in **5 ≤ CS < 8**, and **3/13** fall below **CS < 5**. This does not negate the contributions of the underlying systems but constrains how far the paper can push quantitative cross-paper conclusions. The framework can normalize the axes but cannot invent missing experimental details. Under the strict MR-1 criterion used here, **12/13** systems do not provide enough metrics/variance detail to score the variance point (Table 13). This sharply limits the ability to compare tail-latency improvements numerically across studies, even when each study reports internal comparisons. Given the known confounders in tracing overhead and measurement stack variance [11,12], the absence of explicit variance reporting forces a conservative interpretation. Several systems omit attach specificity (0 score for **4/13**) and state/export details (0 score for **5/13**) in a way that prevents reconstructing hook placement, event-rate regime, or export-path bottlenecks from the paper alone (Table 13). This is important because hook placement and export/state discipline are the dominant feasibility variables for both observability and fast-path policy injection [11,13]. The strict protocol only considers artifact/repo details if the paper explicitly references them. When details exist only in external artifacts without explicit linkage in the paper text, the framework treats them as missing by construction, which may understate practical reproducibility for some systems but preserves auditability and reviewer-grade strictness.

7. Conclusion and Future Work

7.1. Conclusion

This study introduces a unified framework for three modes of database–kernel integration via eBPF—observability, policy injection, and kernel-resident state—and applies it to representative systems from 2021 to 2025 across cache, networking, and storage choke points [1–13]. The framework normalizes comparison along the first-order drivers of feasibility and performance: hook placement and event-rate regime, state model and semantic commitments, and verifier/synchronization/portability constraints, supported by a common cost decomposition and a uniform system-profile rubric [6,7,13,15,19,20]. This separation prevents category errors (e.g., treating observability evidence as sufficient to justify kernel policy change) and makes explicit when bounded fast paths suffice versus when a semantics-bearing kernel-resident state is required [6,9–13]. This study also contributed to the development of a minimal reproducibility standard (MR-1) and a strict compliance scoring method. Table 13 shows that cross-paper quantitative synthesis is currently constrained by systematic disclosure gaps, especially variance reporting and attachment/state/export

 doi:10.20944/preprints202602.0959.v1

specificity, reinforcing the need for standardized reporting to make future results comparable and deployable [11,12,20].

7.2. Future Work

An open, machine-checkable schema and tooling that automatically records kernel version/config, attach points, map budgets, export/drop accounting, workload parameters, and variability statistics should be developed. This directly targets the dominant MR-1 gaps observed in Table 13 and reduces the confounding effect from the tracing-stack variance and overhead bleed-through [11,12].

Create benchmark suites for page-cache and ingress-path interventions that report not only the target workload gains but also regressions on the co-located workloads. This is necessary because externalities are first-order feasibility constraints for cache policy injection and pre-demux fast paths [5,7,13].

Investigate stable, narrow interfaces for page cache, networking, and storage-boundary injection to reduce dependency-surface fragility across kernel versions. This directly addresses the brittleness of the deployment documented for real-world eBPF extensions [20].

Develop design patterns and tools that detect "state pressure" early (multi-object atomicity, isolation needs, replay safety, log/export requirements) and provide safe migration paths from frequent/rare fast paths to explicit semantics-bearing substrates. Transactional/export-boundary designs provide a reference point for this transition [6,9].

As systems move toward Mode III, the security boundaries expand. Future work should establish governance models—privilege restriction, auditing, policy enforcement, and safe extension lifecycles—aligned with memory safety and kernel extension safety challenges in the eBPF ecosystem [19].

Acknowledgments: The authors would like to thank God for His grace and guidance throughout this research. We also thank Mildred Adepoju for her valuable technical assistance, analytical support, and insightful feedback. We acknowledge Prairie View A&M University for providing library access and resources that facilitated this work.

References

1. Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing," in *Proc. 18th USENIX Symp. Networked Systems Design and Implementation (NSDI '21)*, Apr. 2021, pp. 487–501.
2. Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, and A. Cidon, "XRP: In-Kernel Storage Functions with eBPF," in *Proc. 16th USENIX Symp. Operating Systems Design and Implementation (OSDI '22)*, Jul. 2022, pp. 375–393.
3. Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating Distributed Protocols with eBPF," in *Proc. 20th USENIX Symp. Networked Systems Design and Implementation (NSDI '23)*, Apr. 2023, pp. 1391–1407.
4. I. Zarkadas, T. Zussman, J. Carin, S. Jiang, Y. Zhong, J. Pfefferle, H. Franke, J. Yang, K. Kaffes, R. Stutsman, and A. Cidon, "BPF-oF: Storage Function Pushdown Over the Network," *arXiv* preprint arXiv:2312.06808, Dec. 2023, doi: 10.48550/arXiv.2312.06808.
5. D. Lee, I. Choi, C. Lee, S. Lee, and J. Kim, "P2Cache: Application-Directed Page Cache Optimization," in *Proc. ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23)*, Jul. 2023, doi: 10.1145/3599691.3603408.
6. Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu, "DINT: Fast In-Kernel Distributed Transactions with eBPF," in *Proc. 21st USENIX Symp. Networked Systems Design and Implementation (NSDI '24)*, Apr. 2024, pp. 401–417.

7. T. Zussman, I. Zarkadas, J. Carin, A. Cheng, H. Franke, J. Pfefferle, and A. Cidon, "cache_ext: Customizing the Page Cache with eBPF," in *Proc. ACM SIGOPS Symp. Operating Systems Principles (SOSP '25)*, Oct. 2025, doi: 10.1145/3731569.3764820.
8. B. Yang, D. Shen, J. Zhang, H. Yang, L. Zhao, B. Wang, G. Liu, and K. Chen, "eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions," in *Proc. European Conf. on Computer Systems (EuroSys '25)*, Mar.–Apr. 2025, doi: 10.1145/3689031.3696094.
9. M. Butrovich, S. Arch, W. S. Lim, W. Zhang, J. M. Patel, and A. Pavlo, "BPF-DB: A Kernel-Embedded Transactional Database Management System for eBPF Applications," *Proc. ACM Manag. Data*, vol. 3, no. 3 (SIGMOD), Art. 135, Jun. 2025, doi: 10.1145/3725272.
10. N. Tehrany, K. Doekemeijer, and A. Trivedi, "zns-tools: eBPF-Powered Cross-Layer Storage Profiling for ZNS SSDs," in *Proc. Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '24)*, Apr. 2024, doi: 10.1145/3642963.3652205.
11. M. Craun, K. Hussain, U. Gautam, Z. Ji, T. Rao, and D. Williams, "Eliminating eBPF Tracing Overhead on Untraced Processes," in *Proc. Workshop on eBPF and Kernel Extensions (eBPF '24)*, Aug. 2024, doi: 10.1145/3672197.3673431.
12. C. Machado, B. Gião, S. Amaro, M. Matos, J. Paulo, and T. Esteves, "No Two Snowflakes Are Alike: Studying eBPF Libraries in the Real World," in *Proc. Workshop on eBPF and Kernel Extensions (eBPF '25)*, Sep. 2025, doi: 10.1145/3748355.3748364.
13. F. Shahinfar, S. Miano, A. Panda, and G. Antichi, "Demystifying Performance of eBPF Network Applications," *Proc. ACM Netw.*, vol. 3, CoNEXT3, Art. 16, Sep. 2025, doi: 10.1145/3749216.
14. S. Bhat and H. Shacham, "Formal Verification of the Linux Kernel eBPF Verifier Range Analysis," Tech. Rep., The University of Texas at Austin, May 2022.
15. H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the Verifier: eBPF Range Analysis Verification," in *Proc. Int. Conf. Computer Aided Verification (CAV 2023)*, LNCS 13966, 2023, pp. 226–251, doi: 10.1007/978-3-031-37709-9_12.
16. J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, "Kernel Extension Verification is Untenable," in *Proc. Workshop on Hot Topics in Operating Systems (HotOS '23)*, Jun. 2023, doi: 10.1145/3593856.3595892.
17. K. K. Dwivedi, R. Iyer, and S. Kashyap, "Fast, Flexible, and Practical Kernel Extensions," in *Proc. ACM Symp. Operating Systems Principles (SOSP '24)*, Nov. 2024, doi: 10.1145/3694715.3695950.
18. K. Huang, J. Sampson, M. Payer, G. Tan, Z. Qian, and T. Jaeger, "SoK: Challenges and Paths Toward Memory Safety for eBPF," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2025, pp. 848–866, doi: 10.1109/SP61157.2025.00134.
19. H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "MOAT: Towards Safe BPF Kernel Extension," in *Proc. 33rd USENIX Security Symp. (USENIX Security '24)*, Aug. 2024, pp. 1153–1170.
20. S. (Wanxiang) Zhong, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Revealing the Unstable Foundations of eBPF-Based Kernel Extensions," in *Proc. European Conf. on Computer Systems (EuroSys '25)*, Mar.–Apr. 2025, doi: 10.1145/3689031.3717497.
21. X. Wu, Y. Feng, T. Huang, X. Lu, S. Lin, L. Xie, S. Zhao, and Q. Cao, "VEP: A Two-stage Verification Toolchain for Full eBPF Programmability," in *Proc. 22nd USENIX Symp. Networked Systems Design and Implementation (NSDI '25)*, Apr. 2025, pp. 277–299.