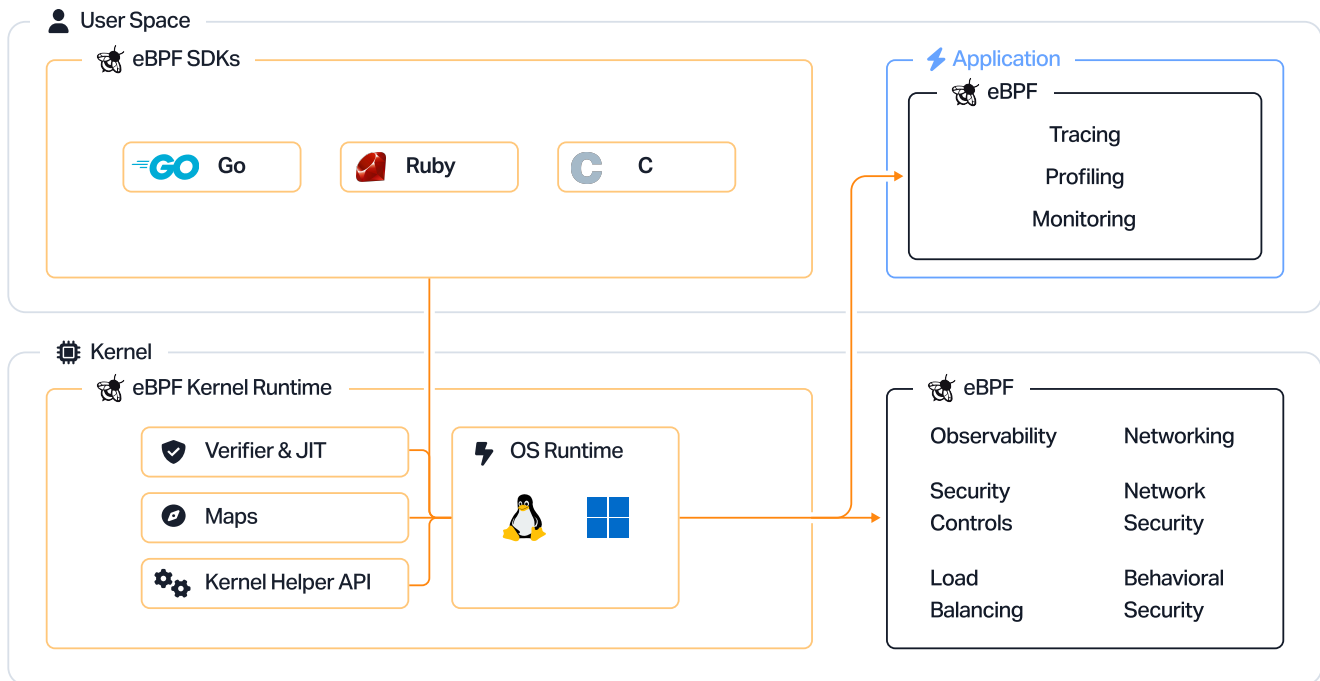


WHITE PAPER

Leveraging eBPF for Runtime Security: Upwind's Architecture for Kernel-Level Observability and Threat Detection

Cloud-native architectures introduce immense complexity. Monolithic applications with single language code bases no longer reflect reality. Ephemeral containers, microservices, dynamic networking, and frequent code changes are now the norm.

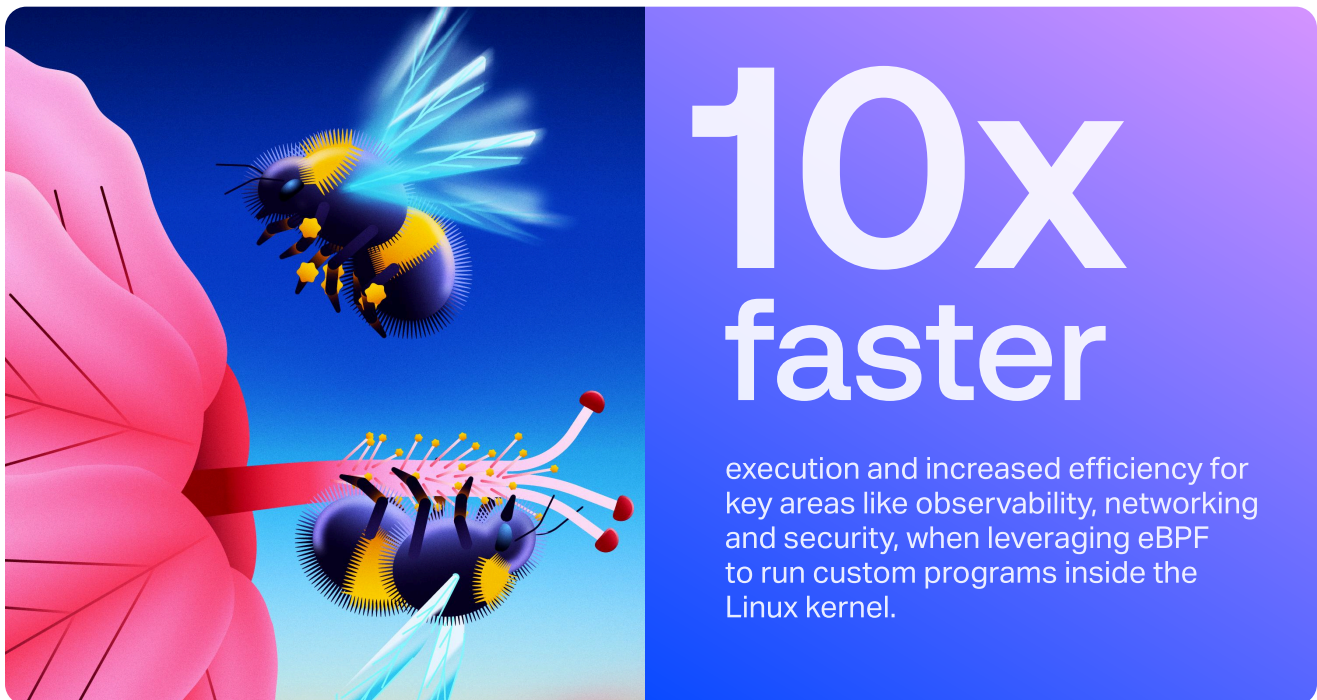


While these patterns bring many benefits and increase agility, they also widen the attack surface of user environments and obscure runtime behavior. Traditional security tooling struggles to keep up with modern software development, particularly when constrained by agent-based architectures or after-the-fact log analysis.

Upwind addresses these limitations by integrating eBPF at the foundation of its Cloud Native Security Platform (CNAPP), to offer deep, low-latency observability and threat response, in real time and directly from the Linux kernel.

What Is eBPF?

eBPF is a Linux kernel technology that safely runs user-defined programs at key kernel hook points. Operating in a sandbox, it enables deep instrumentation of kernel and user-space events without altering kernel code or using user-space agents, making it ideal for production environments demanding performance, safety, and maintainability.



10x faster

execution and increased efficiency for key areas like observability, networking and security, when leveraging eBPF to run custom programs inside the Linux kernel.

Because eBPF programs execute in kernel space, they provide several core benefits:

**Real-time execution:**

eBPF programs run instantly on kernel hook events (e.g., syscalls, tracepoints, kprobes, uprobes, XDP, TC), enabling fast response and visibility for profiling, monitoring, and security.

**Low overhead, high throughput:**

Statically verified and JIT-compiled for efficiency, eBPF minimizes CPU and memory use, enabling high-frequency tracing and metrics with minimal system impact.

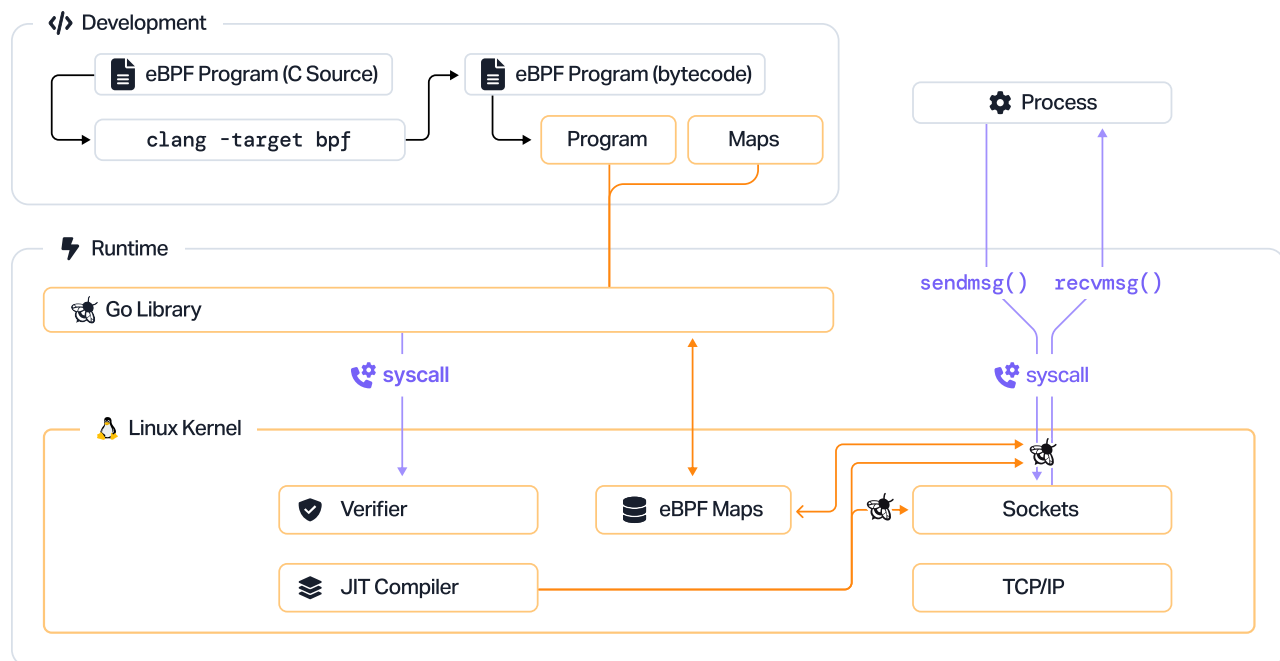
**Event-level granularity:**

eBPF attaches to low-level kernel events, capturing syscall params, stack traces, context switches, TCP retransmits, and more for detailed performance insight.

**Kernel-level visibility and privileged context:**

Running in kernel space gives eBPF access to kernel structures, memory, and execution paths unavailable to user space - vital for observability and security.

eBPF has evolved into a general-purpose runtime for kernel-level telemetry and control. Its flexibility and safety guarantees have led to its adoption in a wide range of domains—including performance profiling, networking, and security. It is increasingly the backbone of modern observability and security telemetry pipelines, offering deep, programmable insights into system behavior with production-grade performance characteristics.



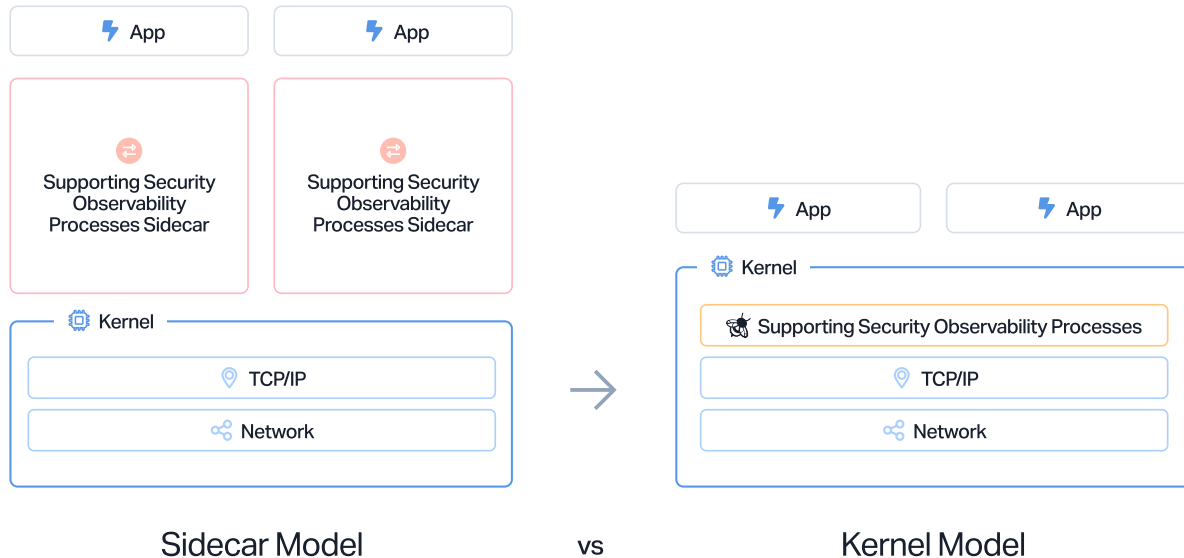
Upwind's Architecture: Embedding Security into the Kernel

In-Kernel Event Monitoring

Upwind deploys eBPF sensors to monitor kernel-level events including system calls, process lifecycle, file operations, and network activity. These probes run asynchronously in the kernel and emit telemetry data to the Upwind Platform via ring buffers or perf events.

Security-Driven Contextualization

The power of eBPF-based telemetry lies not just in capturing kernel events, but in enriching and correlating them with the dynamic state of cloud-native environments. Upwind enhances this by converting low-level kernel signals into high-fidelity security and observability data through layered contextualization, bridging raw system activity and meaningful insights.



Upwind's eBPF instrumentation records detailed kernel events - like syscalls, process lifecycles, file and network I/O, and container transitions - and correlates them in real time with high-level context across the cloud-native stack, including:



Kubernetes control plane metadata:

Each event is tagged with pod- and service-level context - such as service names, ports, labels, annotations, ingress configs, namespaces, and deployment versions - enabling kernel events to be tied to microservices and their ownership domains for actionable telemetry and fine-grained policy enforcement.



Cloud provider identity and infrastructure context:

Events are enriched with cloud metadata like EC2 instance IDs, VPCs, ENIs, IAM roles, security groups, and region/zone info, anchoring telemetry in infrastructure identity and trust boundaries—crucial for multi-account and multi-region setups.

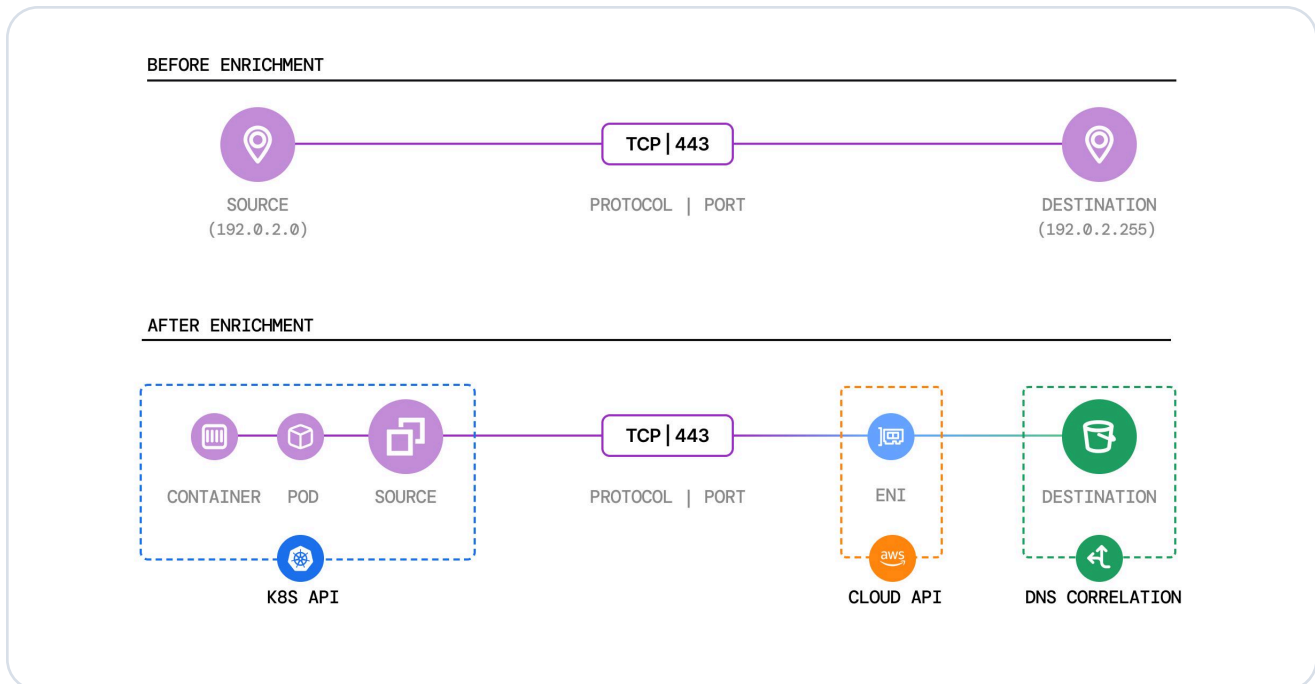


Runtime execution context:




Upwind correlates runtime details like image digests, process trees, entrypoint commands, and PID namespaces, providing insight into process lineage and drift, including unexpected binaries, lateral movement, or post-exploitation activity.

This layered enrichment enables Upwind to map a low-level kernel signal to a high-level entity, such as a specific Kubernetes microservice running a particular container image in a given namespace with an associated IAM role and ingress exposure.

As a result, what would otherwise be opaque kernel noise becomes a semantically rich event tied to a specific workload, version, and ownership group.



This enables:

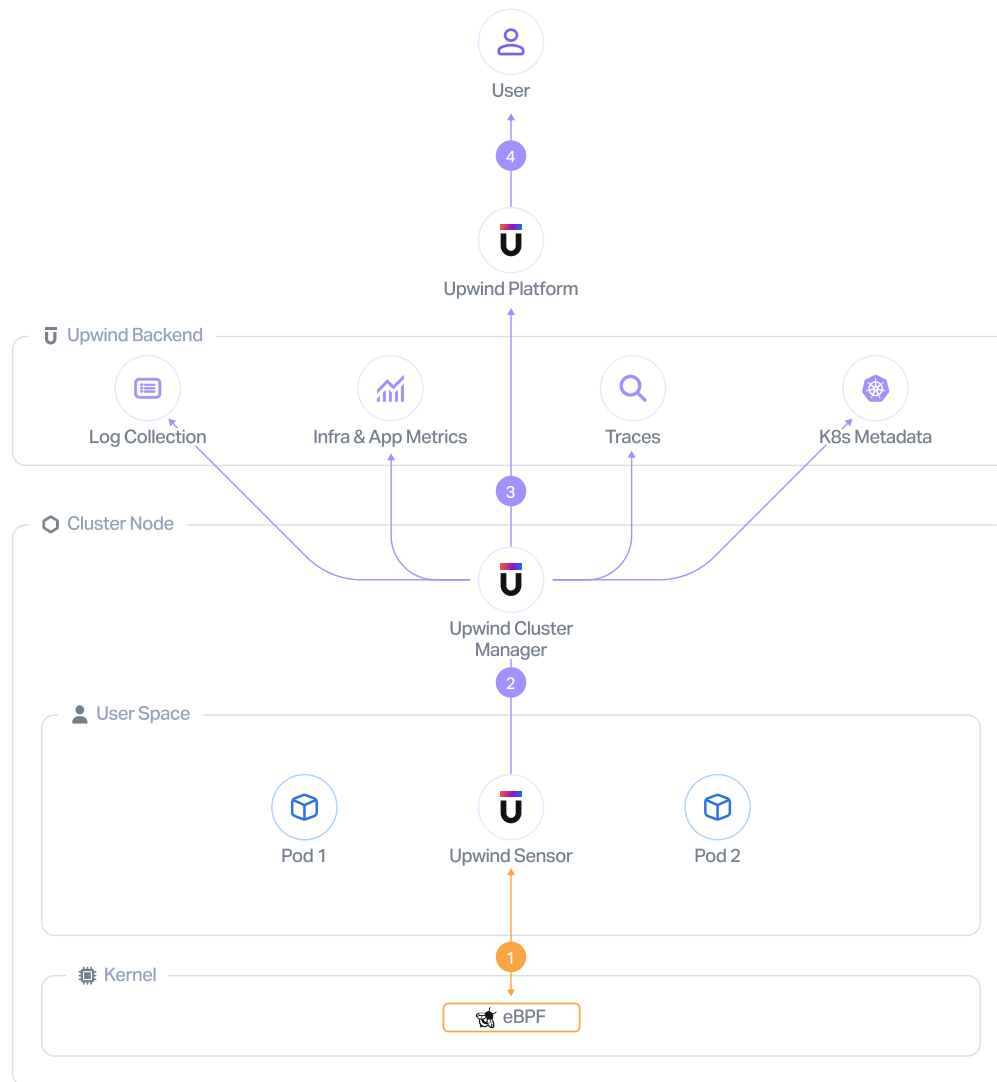
-  **Precise, context-aware alerting:**
Security rules use workload identity and behavior norms instead of generic syscall patterns, reducing false positives and improving signal quality.
-  **Forensic investigation with full stack traceability:**
Anomalous activity can be traced to a container image, build version, or deployment—even after the container is gone—supporting incident response in ephemeral environments.
-  **Fine-grained policy enforcement:**
Policies use high-level constructs instead of static IPs or ports. eBPF enables real-time enforcement tied to containerized process identities.

This approach makes kernel-level telemetry a first-class signal in cloud-native security. Unlike traditional CNAPP tools that depend on user-space agents or scraped metadata, Upwind delivers runtime visibility with in-kernel precision - closing the gap in observability and enforcement for ephemeral, distributed systems.

Data Flow and API Visibility

Upwind uses eBPF to observe runtime data flows and API interactions without relying on OpenAPI schemas, inline proxies, or traffic mirroring.

Traditional tools depend on code instrumentation, API contracts, or sidecars - methods that don't scale in dynamic microservice environments. Upwind avoids these by operating at the kernel socket layer with kprobes on syscalls like `sendmsg` and `recvmsg`, capturing real-time network activity across encrypted and plaintext traffic with minimal overhead.



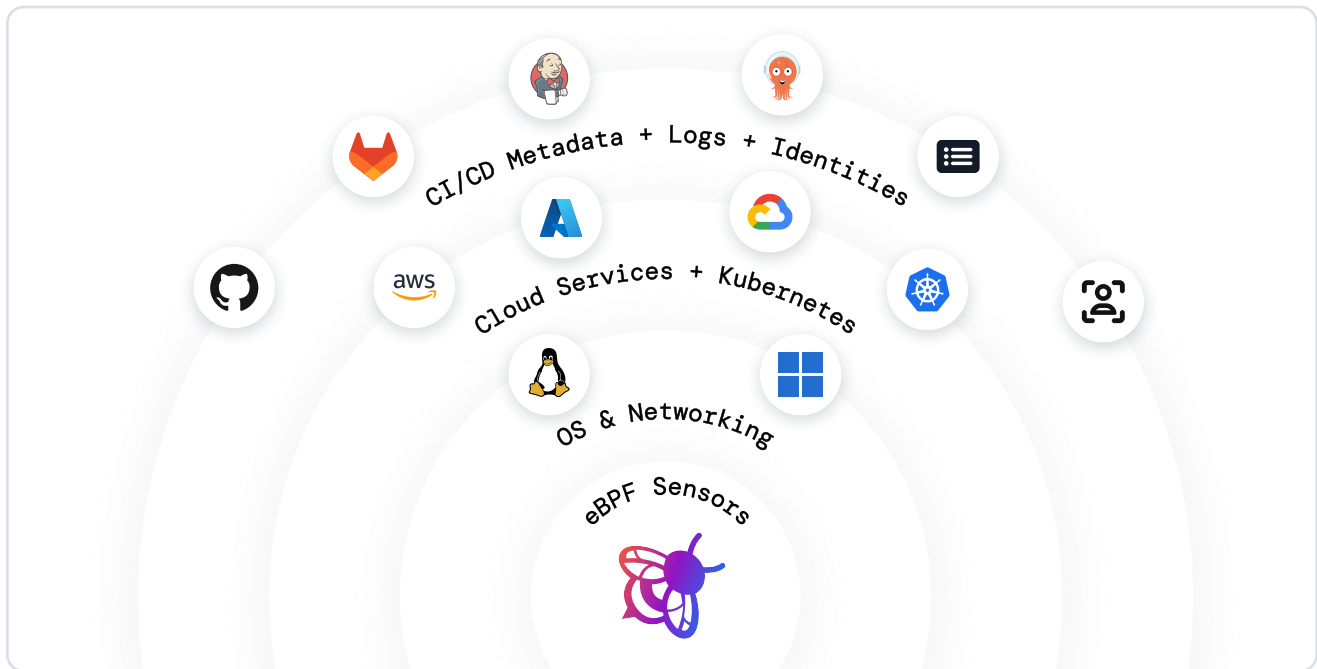
Step 1:
Real time telemetry data via Maps, Buffers, and Events sent to Upwind Sensor from eBPF

Step 2:
Data is sent from the Upwind Sensor to the Upwind Backend via the Upwind Cluster Manager

Step 3:
Data is contextualized with data from other sources in the Upwind Backend then sent to the Upwind Platform

Step 4:
You consume the data from the Upwind Platform

This low-level vantage point allows Upwind to automatically extract and analyze:



HTTP and gRPC traffic semantics:

Upwind inspects socket buffers to reconstruct API calls - paths, methods, headers, response codes - for HTTP/1.x and gRPC, including TLS traffic when accessible at the syscall level or in-memory.



Sensitive data payloads:

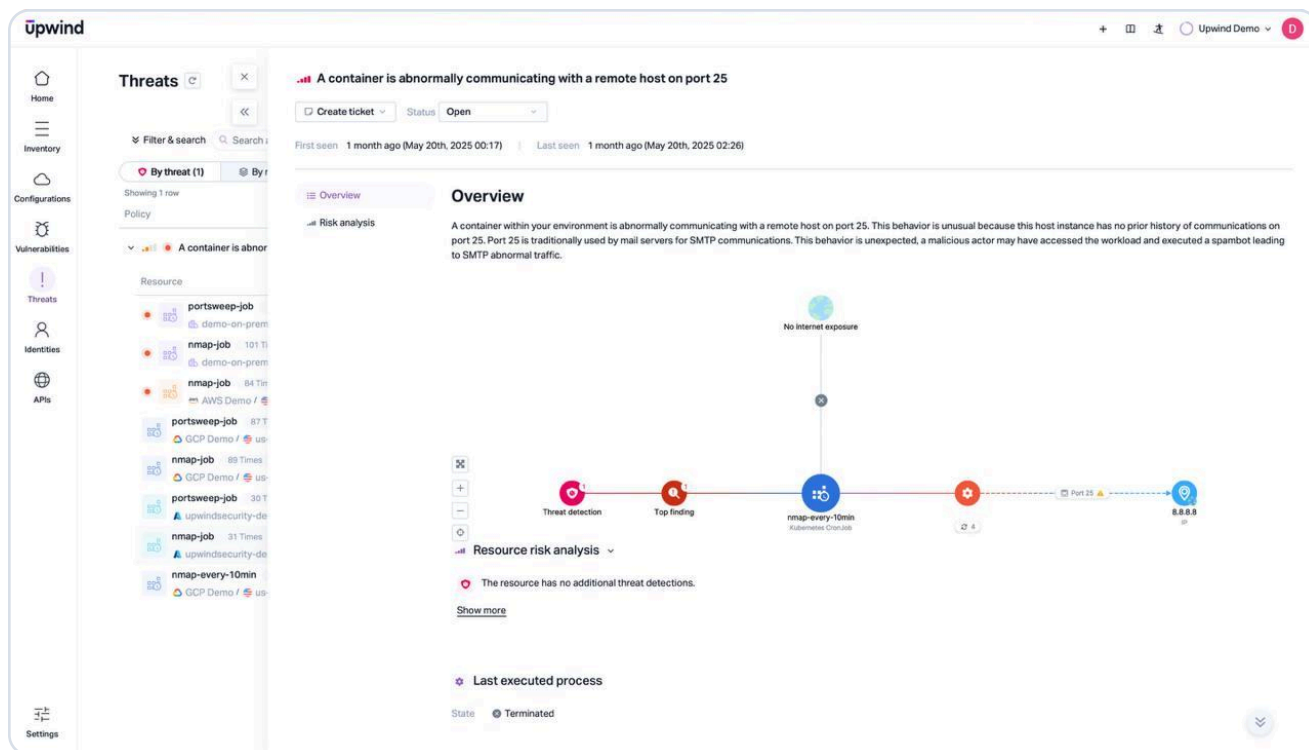
eBPF probes scan payloads in real time for sensitive data signatures (PCI, PII, PHI) without duplicating packets or using user-space engines, enabling high-throughput, in-kernel detection.



Lateral data movement across services or tenants:

By correlating socket communications with metadata (pod, namespace, cloud account), Upwind maps inter-service flows and flags unauthorized cross-tenant or cross-workload traffic.

Upwind's eBPF model enables dynamic API mapping in production - even for undocumented or versionless services - by observing real traffic at the syscall level, unlike tools reliant on OpenAPI specs or SDKs.



The benefits of this approach are both operational and strategic:



Real-time threat detection:

Upwind detects suspicious API behavior - like unexpected POSTs, data exfiltration, or access from compromised containers - based on live behavior, not static rules or offline analysis.



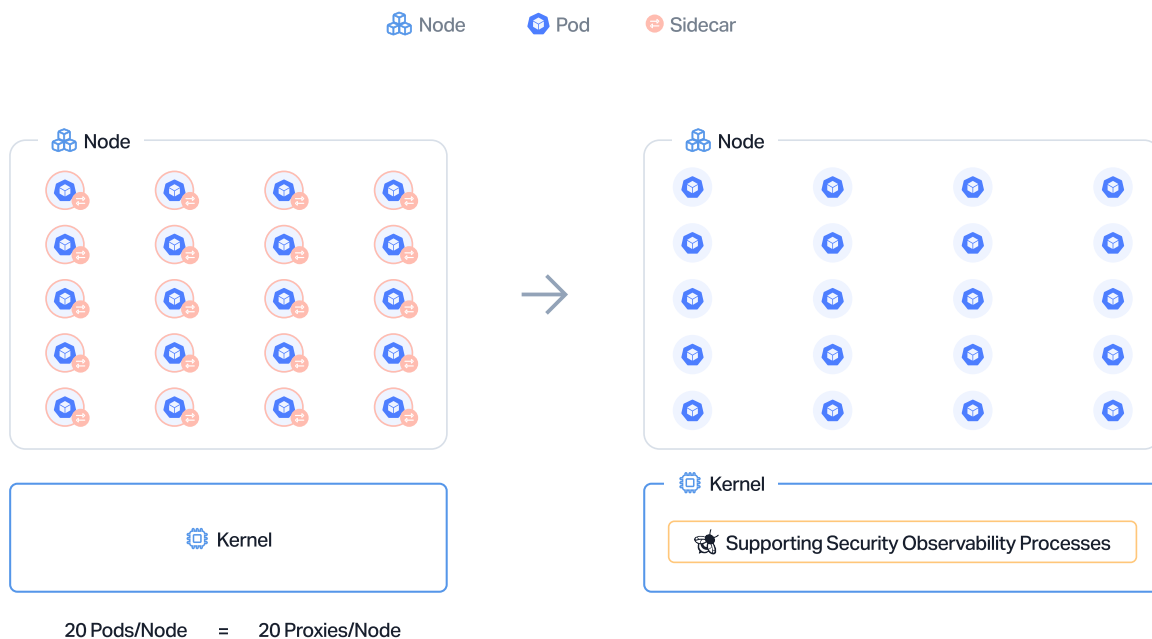
Compliance and data governance:

It continuously monitors sensitive data flows without scans or annotations, supporting GDPR, HIPAA, and PCI-DSS by tracking access and movement for audit and breach response.

Upwind leverages eBPF as a live, distributed runtime sensor, giving security and platform teams deep visibility into service behavior, data flows, and API usage across cloud-native infrastructure.

Scalable by Design

Rather than deploying sidecars and increasing overhead with scale, Upwind's architecture uses controlled, minimal-privilege execution paths to load and manage eBPF programs directly into the kernel. These programs are attached dynamically to relevant kernel or user-space hook points, like tracepoints, kprobes, uprobes, network sockets via system APIs and netlink interfaces. This design yields several critical advantages:



Elimination of traffic duplication and sidecar resource overhead:

Traditional systems duplicate traffic using sidecars or packet capture tools, adding CPU, memory overhead, and latency. Upwind uses eBPF to attach directly to L3/L4 kernel paths, enabling zero-copy, in-kernel visibility without user space involvement.



Horizontal scalability across highly dynamic containerized environments:

Sidecars and daemonsets struggle to scale in Kubernetes due to scheduling and lifecycle constraints. Upwind deploys eBPF centrally, injecting programs into kernel contexts as needed for consistent visibility and enforcement across ephemeral containers - without per-container agents.



Minimal privileged footprint with scoped access and reduced attack surface:

Upwind uses narrowly scoped privileges (e.g., `CAP_BPF` and `CAP_SYS_ADMIN`) and avoids long-lived privileged processes, kernel patches, or out-of-tree modules. All logic runs in verified, bounded kernel contexts, minimizing the attack surface while maintaining operational safety.

Performance Efficiency and Safety Guarantees

Upwind's runtime engine is designed for production-grade performance, resilience, and observability at scale. It leverages advanced eBPF kernel primitives to execute safely and efficiently in high-density, high-throughput environments such as Kubernetes clusters with thousands of pods and multi-tenant workloads.

Advantages	eBPF	Kernel Extensions
Safety	✓ eBPF programs run in a sandbox, preventing kernel / OS crashes and instability due to errors.	✗ Kernel extensions can crash the kernel if poorly written or unoptimized.
Performance	✓ eBPF programs are JIT-compiled, providing competitive performance.	✗ Kernel extensions can be fast if well-optimized but risk instability.
Flexibility	✓ eBPF is flexible and versatile, suitable for various use cases, including networking and security.	✗ Kernel extensions are limited to specific kernel interfaces and require deep kernel knowledge.
Dynamic Loading	✓ eBPF allows dynamic loading and unloading of programs without kernel restarts.	✗ Kernel extensions typically require a kernel restart to update or remove.
Maintenance and Debugging	✓ eBPF is easier to maintain and debug due to user-mode development and modern tooling.	✗ Kernel extensions involve complex kernel-level development and debugging.
Security	✓ eBPF provides more fine-grained control over program behavior, enhancing security.	✗ Kernel extensions can potentially introduce security vulnerabilities.
Containers	✓ eBPF provides the ability to extract all containers' network traffic from the host level at low overhead.	✗ Potential inaccurate correlation of network traffic of running containers with actual sources and destinations.

Key to this performance and safety is Upwind's use of core eBPF features, including:



Per-CPU ring buffers:

Upwind uses **BPF_RINGBUF** structures allocated per CPU for lock-free, low-latency event transport from kernel to user space. This avoids global locks and ensures real-time visibility under high syscall or network load.



Tail calls for modular execution chains:

Upwind chains small, specialized eBPF programs at runtime using tail calls. This modular model supports complex, distributed logic across hook points while staying within eBPF limits for verifier approval and efficiency.

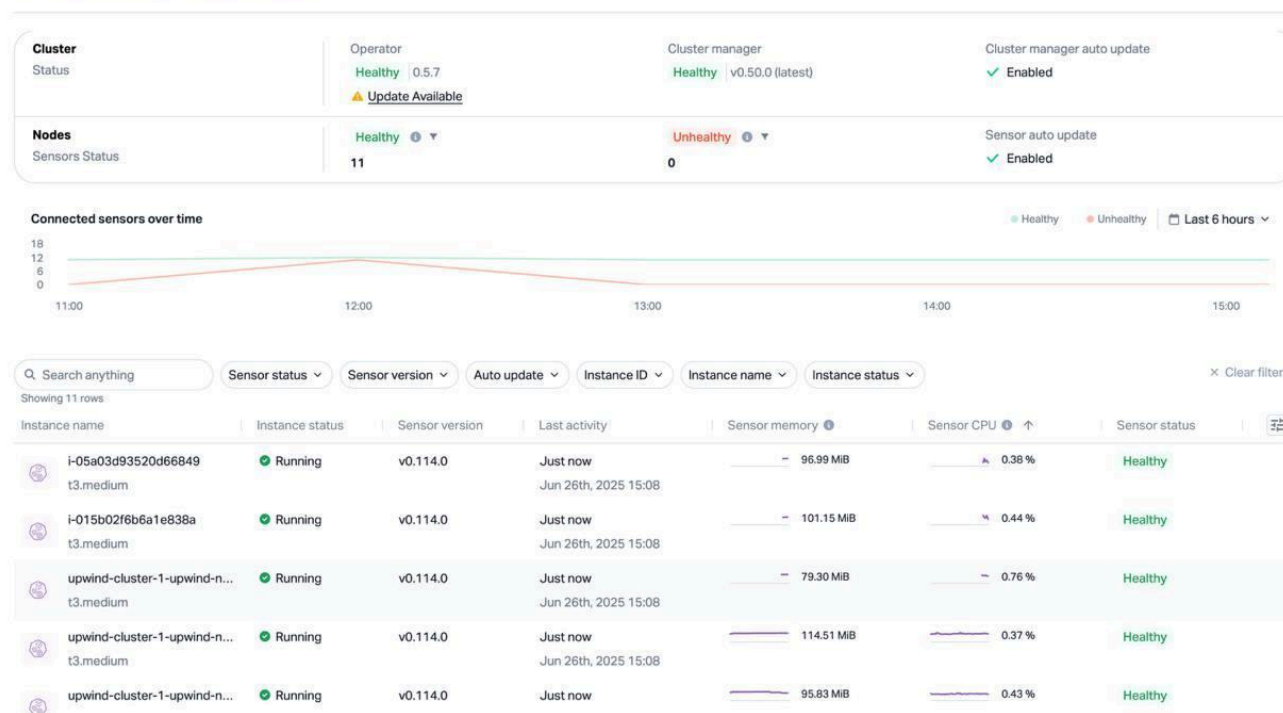


BPF maps for efficient state management:

Upwind uses **BPF_MAP** to cache runtime state like process lineage, connection tracking, and access patterns. These fast, in-kernel stores enable efficient correlation without persistent agents or external storage.

These optimizations keep Upwind's telemetry engine lightweight and non-intrusive, even during bursty traffic, rapid pod churn, or resource-heavy activity from noisy neighbors.

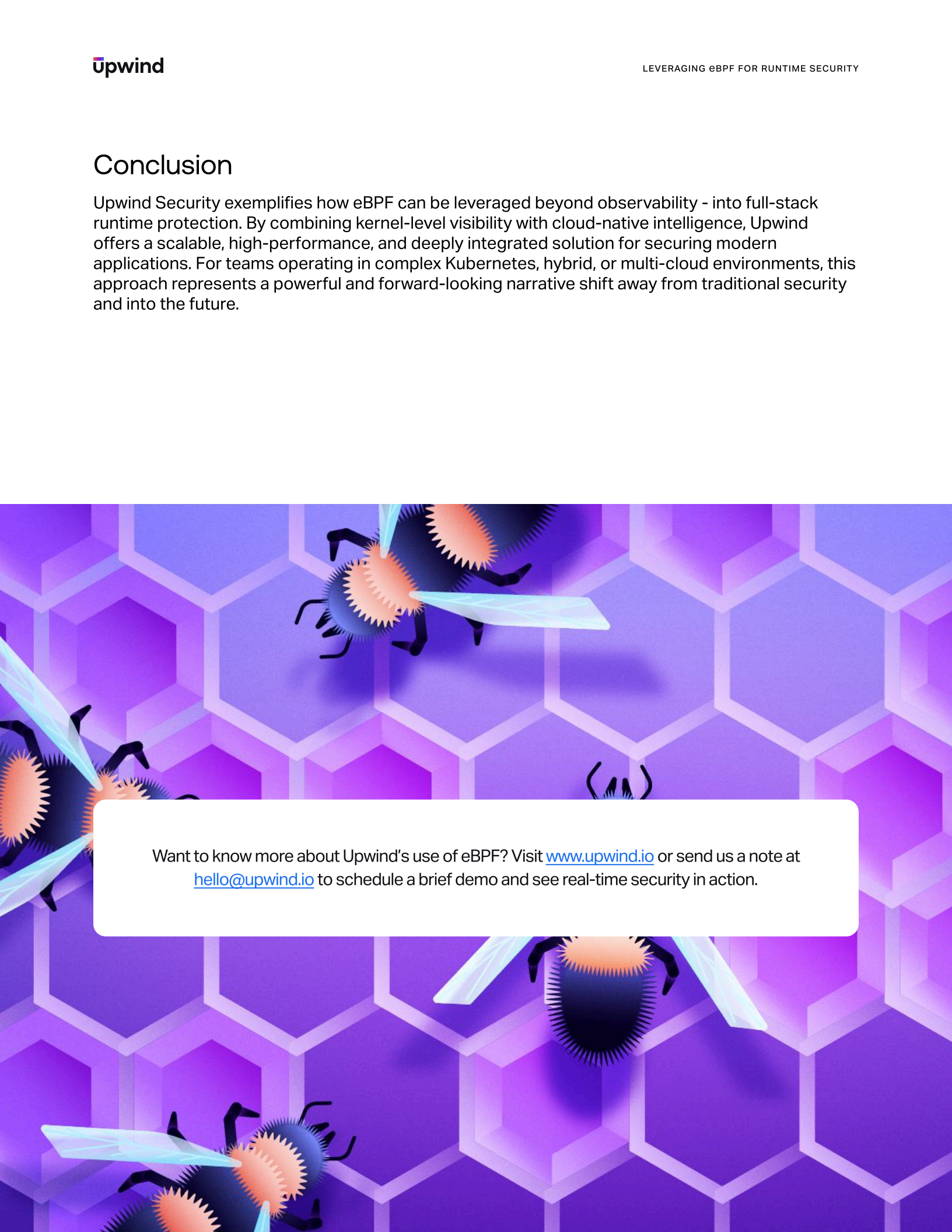
All eBPF programs are **safely loaded into the kernel only after passing the static verifier**, which enforces a strict set of constraints: bounded loops, safe memory access, known instruction paths, and valid stack frame usage. This verifier acts as a kernel-level static analysis engine, rejecting any program that could lead to undefined behavior or kernel crashes, ensuring system stability and reliability.



The result is a runtime instrumentation layer that is both performance-efficient and security-hardened, capable of operating continuously in production without degrading latency, exhausting resources, or increasing the operational burden. This design allows Upwind to deliver precise, real-time telemetry and enforcement capabilities in environments where reliability, safety, and scale are non-negotiable.

Conclusion

Upwind Security exemplifies how eBPF can be leveraged beyond observability - into full-stack runtime protection. By combining kernel-level visibility with cloud-native intelligence, Upwind offers a scalable, high-performance, and deeply integrated solution for securing modern applications. For teams operating in complex Kubernetes, hybrid, or multi-cloud environments, this approach represents a powerful and forward-looking narrative shift away from traditional security and into the future.

The background of the bottom half of the page is a vibrant purple with a repeating pattern of hexagons. Overlaid on this pattern are several stylized bees. The bees have black bodies with orange, spiky segments and translucent, light blue wings. They are depicted in various orientations, some appearing to fly towards the viewer and others away from it.

Want to know more about Upwind's use of eBPF? Visit www.upwind.io or send us a note at hello@upwind.io to schedule a brief demo and see real-time security in action.