eBPF Data Flow Agent, Nucleus, and Backend Platform

Portfolio Project Case Study

Kernel tracing • runtime interception • host-to-backend telemetry • Kubernetes deployment

Project type	Security and observability platform for outbound application data-flow monitoring
Core objective	Capture outbound requests and related context from Linux servers, then offload the events to a nucleus and backend for analysis, classification, and reporting
Core challenge	Encrypted HTTPS traffic hides the most useful request content, so I had to work across kernel, user-space, and runtime-specific interception layers
My role	End-to-end design and implementation across eBPF exploration, event transport, deployment model, runtime interception paths, and product architecture

Positioning note

This case study reflects the architecture, experiments, and deployment approach I developed across the project's evolution. A major strength of the work is that it combines low-level kernel engineering with real deployment practicality, runtime-specific interception, and product-level system design.

Contents

Why this project stands out

This was not just an eBPF tracing exercise. I treated it as a full product platform: a lightweight host-side agent, a nucleus layer for controlled relay and enrichment, and a central backend for storage, classification, ownership mapping, and dashboards. That combination of systems depth and product architecture is what makes the project portfolio-grade.

Project Summary

I worked on a host-based outbound data-flow monitoring platform designed to understand what applications running on Linux servers were sending to outside services. The central goal was to capture useful request metadata - and wherever possible, pre-encryption request content - then forward those events into a separate analysis plane instead of doing heavy analytics on the monitored host itself.

The architecture evolved into three layers. First, an agent installed close to the workload performed capture and emitted structured events. Second, a nucleus or node layer received data from one or more agents and handled deployment-specific buffering, relay, and context. Third, a backend platform stored the events, mapped them to customers and nodes, enriched the data, and surfaced it through APIs and dashboards.

A major reason the project became technically deep was that encrypted outbound traffic cannot be understood well from the network layer alone. That led me to explore multiple boundaries: kernel tracing, kernel-to-user-space event transport, library and runtime interception, SSL/TLS boundaries, and deployment models that could actually run broadly across Linux machines and Kubernetes nodes.

Problem Statement and Why Traditional Visibility Was Not Enough

The problem I was solving was broader than packet capture. If an application on a server makes an HTTPS request, a normal network view usually shows only encrypted traffic, destination

information, and limited socket-level metadata. For security, observability, and data-governance use cases, that is often not enough.

- I wanted to focus on outgoing requests only, especially external and third-party API calls made from the server.
- I wanted to avoid monitoring every OS service blindly; the preference was application-relevant visibility rather than generic noise.
- I wanted to understand whether a request looked legitimate, suspicious, or sensitive, and whether the payload carried things like tokens, PII, or payment data.
- I also needed a practical path for HTTPS traffic, because network capture alone cannot reliably show plain-text request bodies and headers before encryption.

That is why this project did not stay at the packet layer. I treated the problem as a layered visibility challenge and worked from kernel tracing up to runtime-level interception where needed.

My Role and Scope

I handled the project as a full-stack systems problem rather than a single tool task. My role covered architecture design, capture-path experiments, deployment strategy, event transport, and the surrounding product model.

Area	What I worked on
Capture layer	eBPF-based tracing exploration, event extraction, and outbound request visibility strategy
Event transport	How to move captured data from kernel to user space and then off the host safely
Runtime interception	Python, Node.js, Java, and .NET interception paths where kernel-only visibility was insufficient
Deployment	Hybrid containerized agent model, host path mapping, and Kubernetes DaemonSet rollout
Platform design	Agent -> nucleus -> backend architecture, registration model, storage direction, and analysis workflow

Architecture: Agent, Nucleus, and Backend

The platform architecture matured into a distributed design with clean separation of responsibilities. I intentionally kept the agent side lightweight so that monitored hosts did not become overloaded analytics nodes.

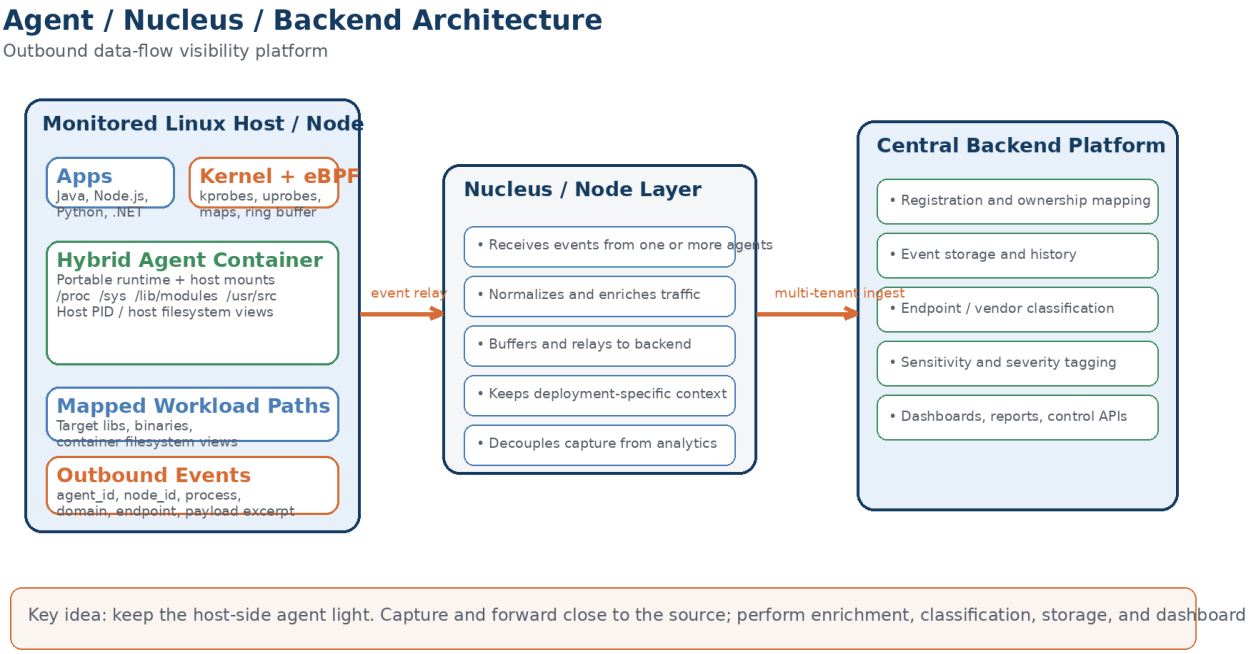

Figure 1 - High-level architecture of the eBPF data-flow monitoring platform

Architecture interpretation

The agent lived on the monitored machine and focused on capture, packaging, and forwarding. The nucleus sat between agents and the backend so that customer-specific deployments could have a local aggregation and control point. The backend handled the product-facing responsibilities: registration, storage, classification, severity mapping, and user-facing reporting.

Layer	Responsibility	Why it mattered
Agent	Observe outbound activity close to the source; capture minimal but useful context; forward structured events	Keeps the monitored server light and reduces local processing
Nucleus / node	Receive from one or more agents, normalize, buffer, relay, and maintain deployment-specific context	Decouples host capture from backend analytics and supports scale-out design
Central backend	Map agents and nodes to customers, store events, classify traffic, apply severity logic, and serve dashboards/APIs	Turns raw telemetry into an actual product and operations platform

Capture Strategy and Kernel-to-User-Space Data Path

A major engineering track in the project was deciding how the data should travel from the kernel and runtime boundaries into user space without creating unacceptable overhead. I wanted the event path to be efficient, predictable under burst, and structured enough for downstream classification.

- I explored eBPF event delivery patterns rather than assuming a single mechanism would fit every workload profile.
- Ring-buffer style transport became an important path because it is a strong fit for high-frequency event streaming in modern eBPF workflows.
- eBPF maps were also part of the design thinking for counters, filtering context, temporary correlation state, and controlled data sharing between kernel and user space.
- I treated kernel-to-user transfer and host-to-backend transfer as separate concerns: the first had to be efficient; the second had to be safe and scalable.

Approach	Why I tried it	Strength	Limitation / lesson
Perf / event buffer style delivery	Standard eBPF pattern for structured event streaming	Good baseline for emitting compact events	Needs careful tuning under load; not a free solution for large payloads
Ring buffer	Needed a lower-overhead event path for frequent events	Cleaner modern transport pattern for high-rate capture	Still requires disciplined payload sizing and user-space draining
Map-based state and counters	Needed shared state, filtering, and temporary context	Useful for correlation, limits, and selection logic	Best used for state - not as a substitute for disciplined event design
Immediate host-side offload	Wanted minimal user-space burden on the monitored machine	Supports a lean agent design	Requires good backpressure and reliability planning in the next layer

Runtime-Specific Interception Work

Kernel visibility alone was not enough for the most valuable part of the problem: understanding request bodies and headers before TLS encryption. Because of that, I explored runtime-aware interception strategies alongside the kernel-level work.

Stack	What I explored	Why it mattered
Python	Application-level visibility around request generation paths such as the requests stack	Improved alignment with actual application API calls instead of broad system noise
Node.js	Interception ideas around SSL wrappers and runtime-specific outbound request paths	Helped target pre-encryption boundaries in JavaScript-heavy services
Java	Deep work around SSL/TLS internals, ByteBuddy, ASM, and methods like encrypt / wrap in the SSL pipeline	Java required runtime instrumentation because network capture alone could not expose the request semantics
.NET	Exploration of SSL-related tracing paths and uprobes-based ideas	Extended the design philosophy beyond a single language runtime

Java interception became one of the most significant parts of the project.

For Java workloads, I explored multiple boundaries in the SSL/TLS pipeline because the request data of interest becomes encrypted very quickly. The work included investigating classes and paths such as SSLSocket, SSLEngine-related flows, Cipher and OutputRecord-style internals, and encryption boundaries exposed through methods such as encrypt and wrap.

To work at that layer, I explored bytecode instrumentation with ByteBuddy and ASM. This part of the project demonstrates an important engineering lesson: a host-based visibility platform cannot rely on one capture technique. It needs different strategies depending on whether the best observation point is the kernel, a shared library boundary, or a language runtime.

Design principle

I kept the focus on outbound requests initiated by applications, especially third-party API calls. That meant I was not trying to trace every response path or every internal service call blindly; the value came from targeted visibility into what the server was sending out.

Deployment, Portability, and Kubernetes Rollout

Another major part of the project was making the agent actually deployable across different Linux environments. An eBPF-based tool is not useful if it only works on one manually prepared machine, so I designed the agent around a hybrid containerized model.

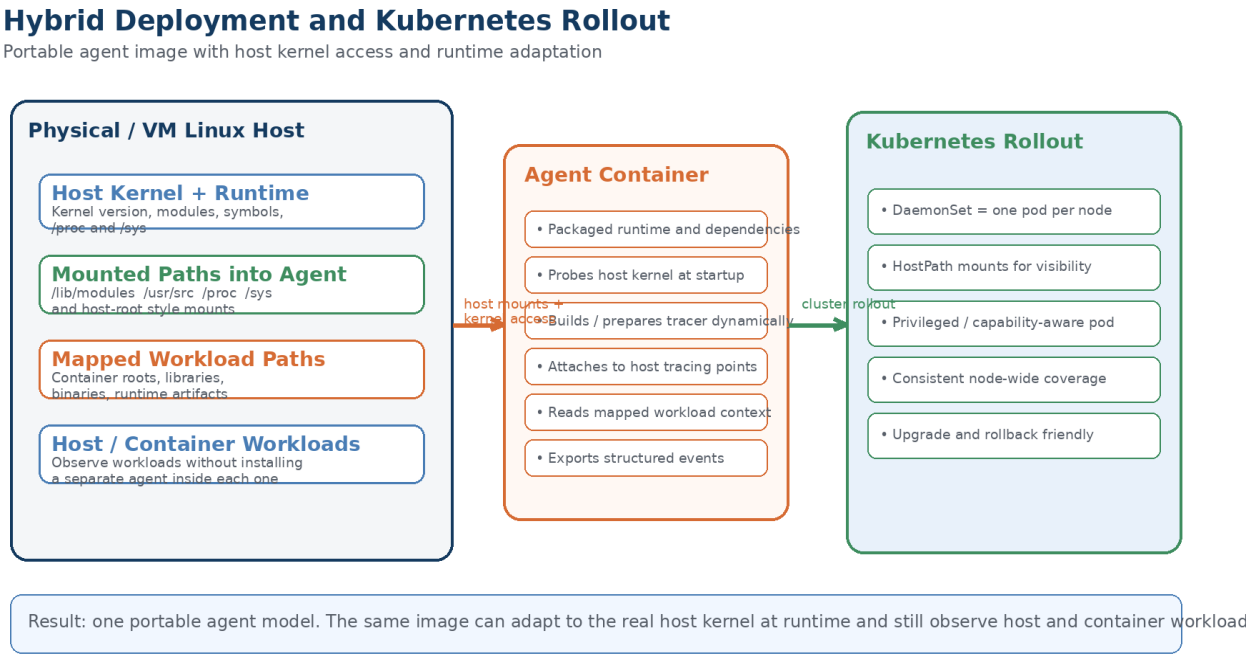

Figure 2 - Hybrid deployment model with host access, dynamic kernel adaptation, and Kubernetes rollout

How the deployment model worked

- The agent runtime and dependencies were packaged in a container for portability and repeatable rollout.
- The container was deliberately given controlled access to the host kernel and host filesystem through mounts such as /proc, /sys, /lib/modules, /usr/src, and host-root style mappings.
- Because containers share the host kernel, the agent could still work against the real node kernel while keeping its user-space toolchain isolated inside the image.
- At startup, the agent could inspect the host kernel and prepare the tracing side dynamically for that environment instead of assuming one static build would work everywhere.

Dynamic kernel-aware compilation / preparation

A key practical challenge was kernel variation. I wanted the agent to be usable across many Linux machines instead of hard-coding it to one distro or one carefully matched kernel. That led to a design where the containerized agent would detect the actual host kernel, use the mounted host kernel resources, and dynamically build or prepare the kernel-aware tracing component against that environment.

This was one of the strongest operational ideas in the project because it converted a fragile host-specific installation problem into a more portable pattern: ship the agent once as a container, then adapt it at runtime to the real machine.

Mapping other container paths into the agent

For containerized workloads, the agent did not need to run inside every target application container. Instead, I explored a host-integrated observability model where the agent container could access mapped workload paths from outside. That allowed the agent to inspect libraries, binaries, and runtime artifacts used by target containers while still operating from one separate observability container.

This was especially relevant for runtime and SSL interception work, where the agent often needed awareness of the target application's library paths and mounted filesystems. Being able to map those paths into the agent container made the design much more practical than trying to inject a separate instrumented agent into every workload.

Kubernetes DaemonSet rollout

The natural Kubernetes deployment model for this kind of node-level observability component was a DaemonSet. I designed the agent so that one pod could run per node with host path mounts and the necessary privileges or capabilities, giving node-wide visibility without manual placement.

DaemonSet element	Purpose	Portfolio significance
One pod per node	Keep the agent close to the host kernel and node-local workloads	Shows a production-ready rollout strategy rather than a lab-only design
HostPath mounts	Expose kernel resources and target workload paths to the agent	Makes the hybrid container model workable across clusters
Privileged / capability-aware pod	Allow tracing and host-level observation operations	Highlights the real operational constraints of observability tooling
Central relay to nucleus/backend	Move events off node quickly and consistently	Fits the larger product architecture and scaling model

Data Model, Classification, and Control Plane

The platform was designed to do more than collect raw trace events. The event model had to be structured enough that the backend could group, classify, and present the captured traffic meaningfully.

- Representative event fields included agent identity, node identity, process/application context, destination domain, endpoint, payload excerpt or masked payload, and classification metadata.
- The downstream goal was to classify endpoints, vendors, and payload sensitivity rather than store undifferentiated blobs of traffic.
- Severity logic mattered as well; for example, different categories such as credit-card data, PII, tokens, or email-like identifiers could be ranked differently.
- The control plane also needed agent registration, node registration, mapping to customers or companies, and a clean ownership model for the backend product.

In the broader platform design, the backend side was planned as the product control layer: storing events, associating them with users and nodes, surfacing them through APIs, and eventually driving endpoint/vendor views and security-focused reports.

Data concern	Why it was needed	Example of what it enabled
Identity	Associate each event with the right monitored deployment	agent_id / node_id / customer mapping
Traffic semantics	Know what the request was trying to do	domain, endpoint, host, request type, payload excerpt
Security relevance	Prioritize what deserves attention first	PII, token, card-data, or suspicious external call classifications
Product analytics	Support a usable UI rather than raw trace dumps	endpoint grouping, vendor rollups, trend views, hit counts

Approaches Tried and Why the Design Evolved

One of the strongest parts of the project is that I did not pretend one technique would solve everything. The design evolved because each layer taught something different about cost, portability, and visibility.

Approach	Why I tried it	What it solved	Why the design moved forward
Kernel/network-oriented capture	Needed broad outbound visibility close to the source	Connection-level and process-adjacent observability	Not enough to expose high-value plain-text request content under TLS
Ring buffer and kernel-user transfer tuning	Needed efficient event movement into user space	Made the event path more realistic for high-frequency capture	Still required disciplined event sizing and host-side offload design

Application/runtime interception	Needed better semantic visibility than the network layer could provide	Brought the observation point closer to the request itself	Different runtimes demanded different methods and trade-offs
Hybrid container deployment	Needed portability across Linux machines	Made rollout repeatable and practical	Required careful host mounts, privileges, and kernel adaptation logic
Agent -> nucleus -> backend architecture	Needed a real product platform instead of a single host script	Separated capture, relay, and analytics cleanly	Created a scalable operational model for multi-node, multi-customer deployments

Key Engineering Decisions and Practical Challenges

The project involved many practical constraints beyond just making a trace happen once. The design had to balance capture depth, overhead, deployment realism, and the messy diversity of Linux application stacks.

- I focused on outbound traffic because value came from understanding what the server was sending externally, not from indiscriminately tracing everything.
- I kept the host-side agent as light as possible and pushed heavier work downstream.
- I treated portability as a first-class concern and designed the agent to adapt to real host kernels instead of assuming a uniform environment.
- I used a hybrid model for containerized workloads: keep the agent in its own observability container, but map enough host and workload context into it to make tracing practical.
- I accepted that a realistic platform needed more than eBPF alone; runtime instrumentation and language-specific interception were part of the solution space.

Challenge	What made it hard	How my design responded
HTTPS encryption boundary	Network views lose the most useful request details	Explored runtime-aware interception and pre-encryption boundaries
Cross-kernel portability	Linux environments differ in kernel versions and available resources	Used a hybrid container model with host mounts and runtime adaptation
Containerized workloads	Target libraries and filesystems may live in other containers	Mapped workload paths into the observability container instead of instrumenting every app container directly
Host overhead	Tracing can become too expensive if the host also does analytics	Separated capture from processing through nucleus/backend relay

Technical Stack and Portfolio Value

Layer	Technologies / directions involved
Kernel tracing	eBPF, state maps, ring-buffer-oriented event transport, Linux observability tooling
Runtime interception	Python request-level ideas, Node.js SSL wrapper exploration, Java ByteBuddy/ASM instrumentation, .NET interception direction
Deployment	Docker-style hybrid agent container, host path mapping, host kernel access, Kubernetes DaemonSet rollout
Relay / platform architecture	Agent -> nucleus -> backend design, event offload, customer/node registration, central analytics
Security analytics	Endpoint classification, vendor grouping, sensitivity detection, severity tagging, external API inventory

Why this is one of my strongest portfolio projects

This project combines kernel engineering, runtime instrumentation, container and Kubernetes deployment design, and product architecture in one coherent system. It shows that I can move from low-level capture problems all the way to rollout strategy and platform thinking.

Just as importantly, it shows engineering realism. I did not stop at the first tracing idea. I worked through multiple approaches, identified where each one failed or became insufficient, and pushed the design toward something that could actually operate on real Linux hosts and clusters.

Portfolio summary

I designed and worked on an outbound data-flow monitoring platform built around an eBPF-driven agent, a nucleus relay layer, and a central backend. The project covered kernel-level capture, kernel-to-user-space event transport, runtime-specific interception, hybrid container deployment, and Kubernetes rollout. Its strongest portfolio value is the combination of systems depth, deployment practicality, and product-level architecture.