happiest minds
AI FIRST. AGILE ALWAYS.

eBPF Unlocked

Redefining Kernel Observability, Security, and AI

In the vast, fluctuating landscape of today's infrastructure, where Kubernetes pods are ephemeral, and every microservice interacts with other microservices over an unpredictable web of APIs, traditional network tools are struggling to keep up.I

For many years, we have used tools such as Iptables, Tcpdump, and classical Linux routing to accomplish our goals. They served us well in the age of static servers, but now we send a single request through 100 microservices over 10 different servers. These tools introduce latency, complexity, and blind spots.

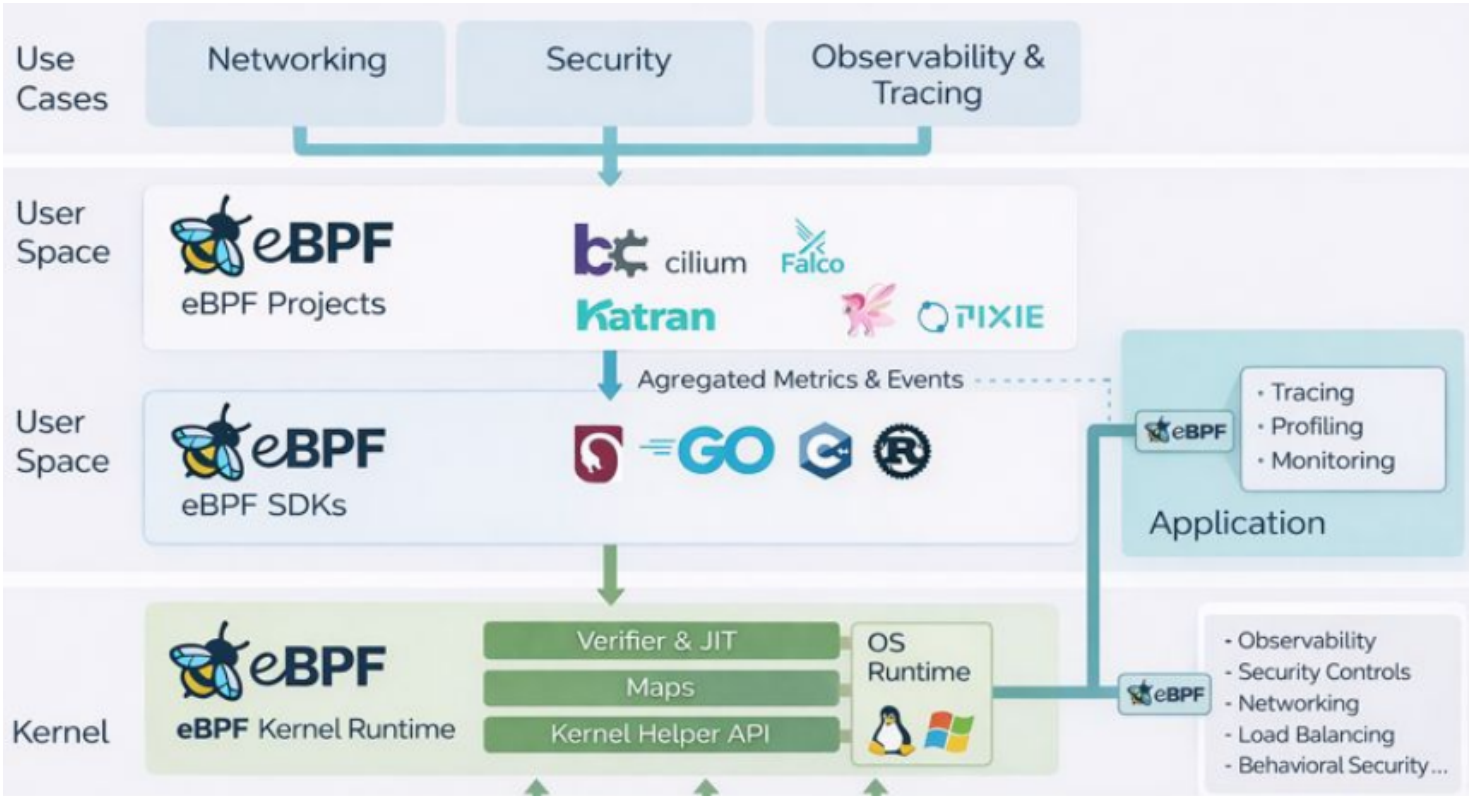
We needed a mechanism that enables the Linux kernel to be smarter, quicker, and safely programmable

Extended Berkeley Packet Filter (eBPF) represents, without question, the single largest evolution in Linux networking in the last 10 years.

Let's take a deep dive into what eBPF is and why it is transforming modern networking, and why we should all take notice

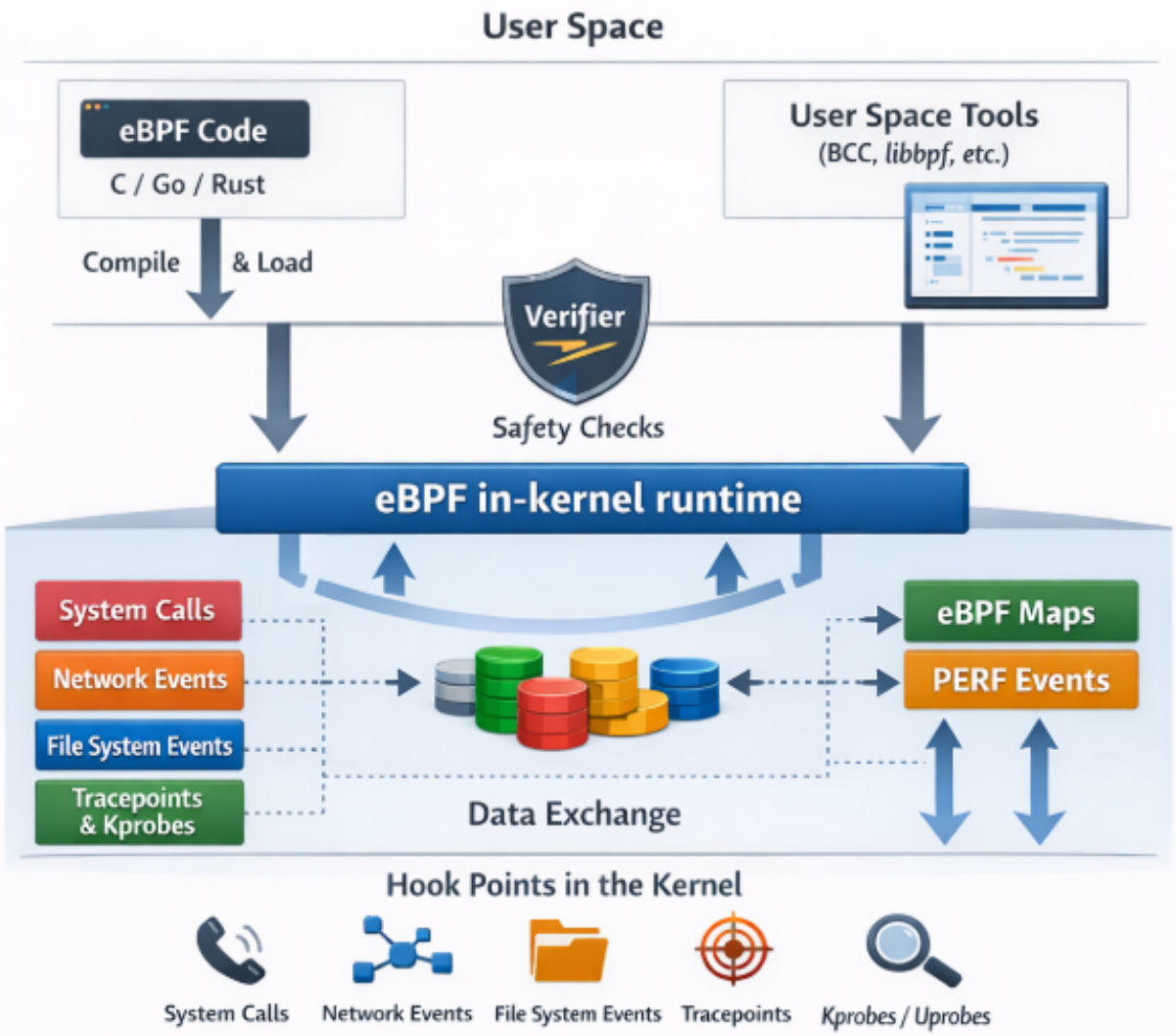
What is eBPF? Beyond Packet Filtering

Although the name suggests that ePBF was originally used only for filtering network packets, its capabilities have expanded significantly since then and now include many more things. Essentially, eBPF allows a user-defined program to run in the kernel/sandbox against various kernel hooks such as system calls, network events, tracepoints, and kprobes. As events are triggered, eBPF programs associated with those events automatically get executed. This capability enables eBPF to be used for live, real-time monitoring of kernel performance, filtering, and potentially changing the behaviour of the kernel.



The eBPF Ecosystem: A High-Level View

Imagine a finely tuned machine, and eBPF programs are the specialized sensors and actuators you can plug in to monitor and control it with incredible precision.



This figure depicts the whole eBPF ecosystem.

Developers of user space applications write their eBPF programs in C, Go, or Rust Languages, then compile them into bytecode to run within the eBPF runtime, which is integrated into the Linux kernel.

Besides that, they can set control points in the kernel at system calls, network events, file system events, tracepoints, and kprobes, which essentially give them ways to manipulate and monitor kernel operations for their applications.

The verifier checks every eBPF program to ensure it is safe to execute. Hence, eBPF Maps and PERF events jointly provide a channel for data transport between the kernel and user space applications and vice versa.

Below are the key components of eBPF:

eBPF Programs

These are bytecode programs that run in the kernel. Normally, the programs are written in a restricted C-like syntax so they can be compiled into eBPF bytecode through LLVM/Clang. Now, BPF Compiler Collection (BCC) and libbpf libraries are excellent examples of how this can be done effortlessly.

Hook Points

Specific locations within the kernel where eBPF programs can attach and execute. Here is a list of some examples:

- **System Calls :** Think of this as the point at which user applications interact with the kernel.
- **Network Events :** One can look at and manipulate network packets at different layers, for example, ingress/egress, XDP
- **Tracepoints :** Statically defined instrumentation points within the kernel for debugging and tracing
- **kprobes/uprobes :** Dynamically attaching to any kernel or user-space function for deep introspection.
- **File System Events :** Monitoring file access and manipulation

eBPF in-kernel runtime

The entire premise of eBPF is that it runs in the kernel. The in-kernel runtime is the “host” in which all eBPF programs run. It is designed with security and performance as top priorities.

eBPF Maps

These are efficient key-value stores shared between eBPF programs in the kernel and user-space applications. They enable data exchange, aggregation, and configuration.

Verifier

An important security component. Before an eBPF program is loaded, the verifier performs a static analysis to ensure it's safe to run within the kernel. It checks for infinite loops, out-of-bounds memory access, and other potential vulnerabilities, guaranteeing that eBPF programs cannot crash or compromise the kernel.

User Space Tools

Applications interact with eBPF programs and maps from user space. Frameworks like BCC, libbpf, and specialized eBPF tools provide the necessary APIs and utilities.

Why is eBPF such a breakthrough?

The power of eBPF stems from several key advantages:

Reliable by Design

The in-kernel verifier ensures that eBPF programs are safe and cannot crash the kernel. This is a significant improvement over traditional kernel modules, which, if buggy, could lead to system instability.

Performance

eBPF programs execute directly within the kernel, avoiding costly context switches between user and kernel space. They are highly optimized and can perform complex logic at line speed, especially for networking tasks (e.g., XDP - Express Data Path).

Programmability and Flexibility

Developers can write custom logic to suit their exact needs, offering unprecedented flexibility in observability, security, and networking

Dynamic Updates

eBPF programs can be loaded, updated, and unloaded dynamically without requiring a kernel recompilation or system reboot. This dramatically simplifies development and deployment cycles.

Rich Context

eBPF programs have access to a wealth of kernel context, allowing for highly granular and insightful analysis.

How eBPF Works: A Deeper Dive

Let's trace the journey of an eBPF program:

1. Code Generation (User Space)

A developer writes an eBPF program, typically in a restricted C dialect. This code is compiled into eBPF bytecode by a compiler like LLVM/Clang.

2. Loading (User Space to Kernel)

The compiled eBPF bytecode is then loaded into the kernel via the bpf() system call.

3. Verification (Kernel)

Before the program is actually attached, the in-kernel verifier rigorously analyzes the bytecode. This static analysis checks for:

- **Termination**: Guarantees the program will always exit (no infinite loops).
- **Memory Safety**: Ensures no out-of-bounds memory access.
- **Privilege Escalation:** Prevents malicious behaviour.
- **Resource Limits:** Enforces limits on instruction count and stack usage

4. Attachment (Kernel)

Once verified, the eBPF program is attached to a specific hook point in the kernel. This could be a kprobe on a particular function, a tracepoint, or a network interface for XDP.

5. Execution (Kernel):

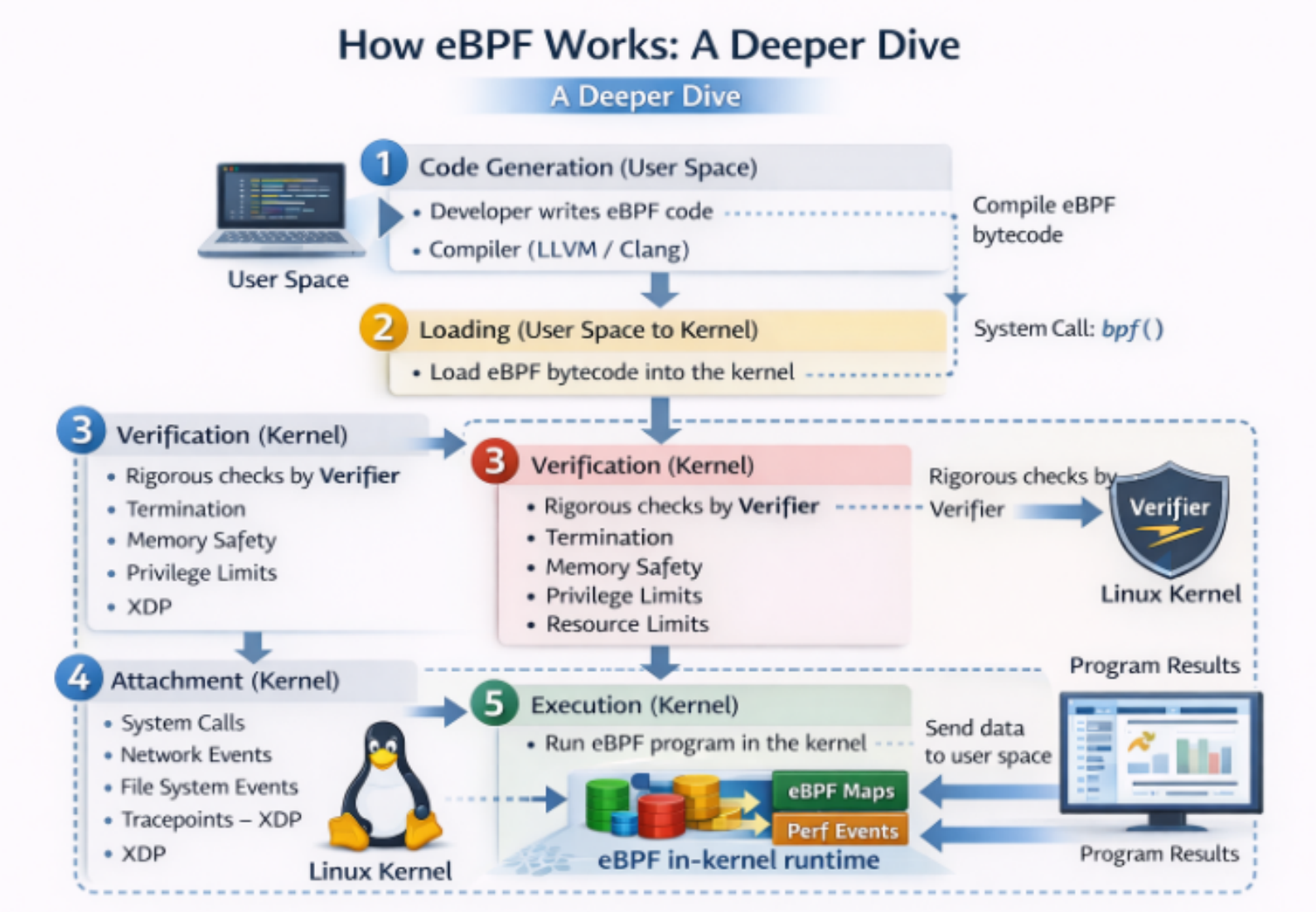
When the event associated with the hook point occurs, the eBPF program is executed by the eBPF runtime. The program can read kernel data, write to eBPF maps, and even modify kernel behavior (e.g., dropping packets, returning custom values).

6. Data Exchange (Kernel to User Space)

eBPF programs can communicate results back to user-space applications in several ways:

- **eBPF Maps:** Programs can write data to shared maps, which user-space applications can then read.
- **Perf Events:** For high-volume, asynchronous data, eBPF programs can generate perf events that user-space tools can consume.

Simplified eBPF Program Flow



The diagram illustrates how an eBPF program's simple process runs in user space before executing. First, code is created and compiled before being integrated into the Linux kernel for production use. The Linux kernel is central to determining whether an eBPF program can execute, as it performs a verification step through its verifier. After verification, the eBPF program is bound to specific kernel hook points, such as file system, system calls, or network events. When triggered, eBPF programs run inside the eBPF virtual machine, access kernel data, and communicate with user-space applications via eBPF Maps for structured data and through Perf Events for asynchronous data. The entire lifecycle of an eBPF program, from creation to execution, provides developers with powerful, flexible mechanisms for instrumenting and controlling the Linux kernel core.

Industry Adoptions and Current Products Using eBPF

eBPF is rapidly being adopted across various industries, from cloud providers to security vendors, due to its unparalleled capabilities. Here are some key areas and products leveraging eBPF:

1. Observability and Smart Monitoring

eBPF provides unparalleled visibility into system behaviour without the overhead of traditional tracing tools. It can collect metrics, trace function calls, and analyze network traffic with minimal impact on performance.

Cilium

Cilium is a cloud native networking, security, and observability solution for Kubernetes. Cilium leverages eBPF to create high performance policy enforcement, load balanced applications and give detailed insight into network protocol and application protocol usage. Hubble, part of Cilium, expands on the observability of Cilium.

Pixie

An open-source observability platform for Kubernetes applications. Pixie uses eBPF to automatically collect full-stack telemetry data (CPU, memory, network, I/O, application requests) without any code changes or manual instrumentation.

Falco

This is a cloud-native runtime security project primarily uses kernel modules, it is increasingly integrating eBPF for richer event sources and improved performance in detecting suspicious activity.

Datadog

One of the popular monitoring and analytics platform. Datadog utilizes eBPF for enhanced network performance monitoring, system call tracing, and deep process-level visibility, allowing for more comprehensive and efficient infrastructure and application monitoring.

Grafana Labs (Grafana Phlare):

For continuous profiling, tools like Pyroscope are integrating eBPF to collect CPU and memory profiles with very low overhead, helping identify performance bottlenecks.

2. Networking and Advanced Load Balancing

Cilium

Beyond security, Cilium uses eBPF for advanced load balancing (replacing kube-proxy), efficient IP address management, and deep network visibility within Kubernetes clusters.

Cloudflare

Uses XDP extensively for DDoS mitigation and load balancing, leveraging eBPF's performance to handle massive traffic volumes.

XDP (Express Data Path)

A core Linux kernel technology powered by eBPF. XDP allows eBPF programs to process network packets directly from the network driver, before they even reach the network stack. This enables extremely high-performance packet filtering, DDoS mitigation, and custom packet processing at near wire speed.

Meta (Facebook)

Employs eBPF in their data centers for load balancing, network security, and advanced traffic engineering.

3. Security

eBPF's ability to process packets in the kernel at line speed makes it ideal for high-performance networking solutions.

Tracee

An open-source runtime security and forensics tool by Aqua S ecurity. Tracee uses eBPF to trace system calls and other kernel events to detect malicious behavior, policy violations, and collect forensic data.

Cilium (once more)

Provides robust network policy enforcement at the kernel level, ensuring that only authorized traffic flows between multiple pods and services.

Sysdig

A container and cloud security platform that utilizes eBPF for deep visibility into system calls and other kernel events to detect and prevent threats in real-time.

4. Performance Engineering and Debugging

eBPF tools provide unprecedented visibility into system performance bottlenecks, allowing engineers to pinpoint issues quickly and efficiently.

BCC (BPF Compiler Collection) Tools

A collection of powerful eBPF-based tools for tracing, monitoring, and analyzing various kernel subsystems. Examples include execsnoop (trace new processes), biolatency (disk I/O latency), opensnoop (file opens), and many more. These tools are invaluable for performance tuning and troubleshooting.

bpftrace

A high-level tracing language for Linux, built on top of eBPF. It provides a dtrace-like syntax for easy and powerful kernel and user-space tracing, making it accessible to a wider range of users.

Netflix

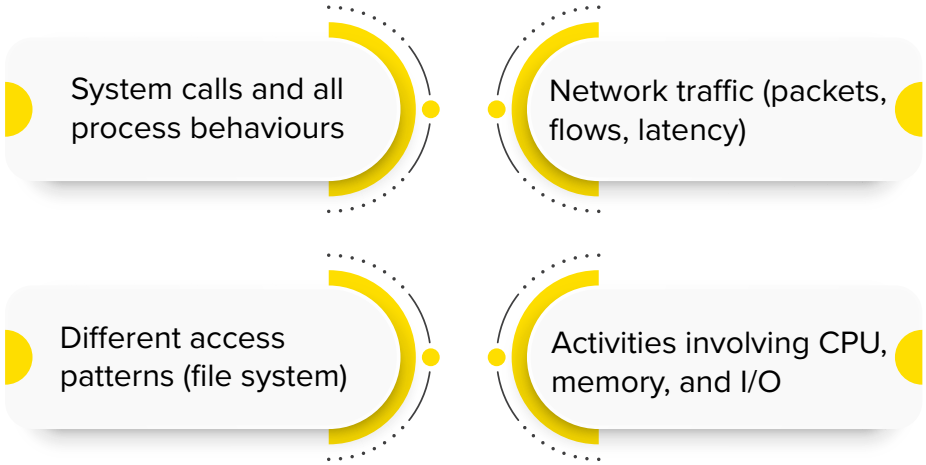
Pioneer in using eBPF for performance analysis and debugging across their massive infrastructure. Many of the foundational BCC tools and bpftrace scripts originated from their work.

The Role of eBPF in Enabling AI-Driven Systems

eBPF plays a foundational role in enabling AI-driven observability, security, and automation by providing real-time, low-overhead access to rich kernel-level signals. It acts as the data acquisition and enforcement layer that AI systems rely on to learn, infer, and act.

1. High-Fidelity Data Collection for AI Models

AI systems require large volumes of accurate, contextual data. eBPF enables this by safely observing kernel events such as:



Because eBPF runs inside the kernel, it captures ground-truth telemetry that is far more reliable than user-space polling or logs—ideal for training and feeding ML models.

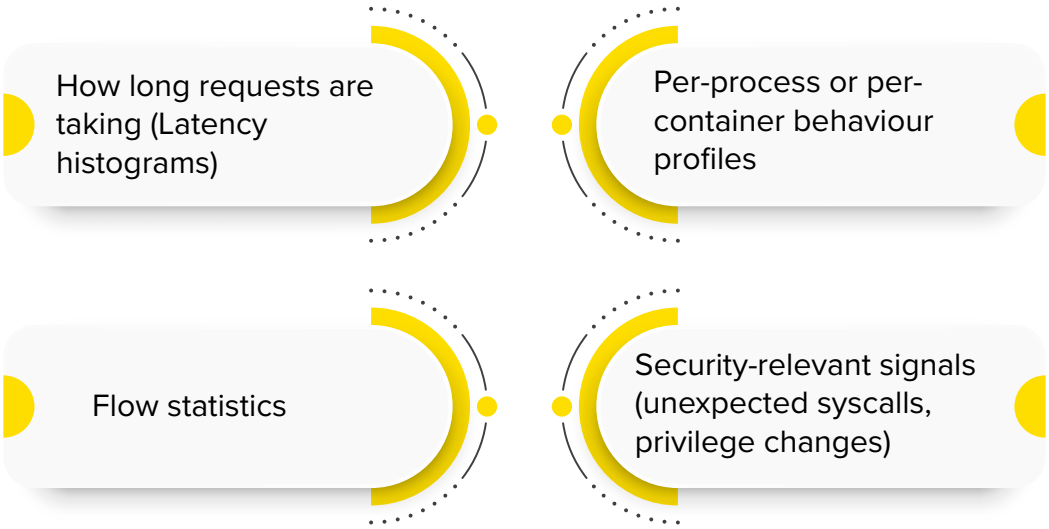
2. Low-Overhead, Real-Time Signal Generation

Traditional monitoring tools introduce significant overhead, which can distort data and impact performance. eBPF programs are:

This makes eBPF suitable for real-time AI inference, where latency and performance are critical, such as anomaly detection or adaptive traffic control.

3. Feature Extraction at the Source

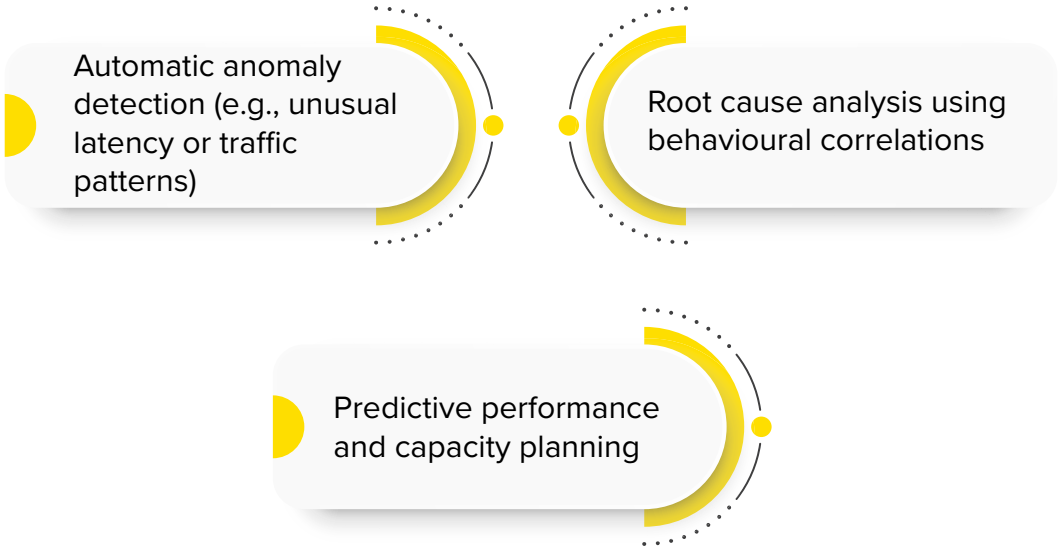
Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat



This reduces data volume and produces AI-ready features, improving model efficiency and accuracy.

4. Enabling AI-Driven Observability

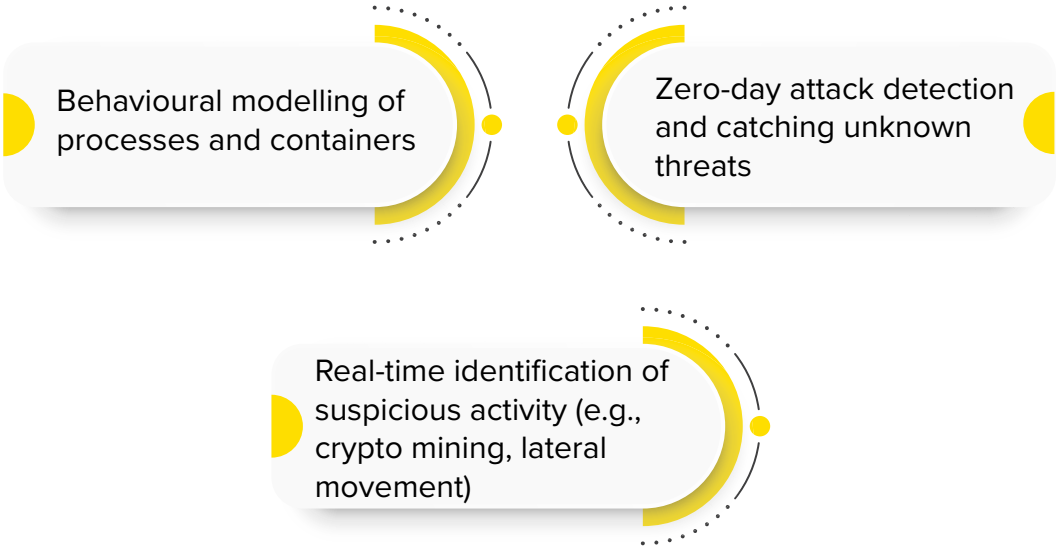
In observability platforms, eBPF feeds AI models with continuous telemetry that enables:



AI systems can learn “normal” system behaviour and flag deviations—without requiring manual instrumentation.

5. AI-Powered Security and Detection of Threats

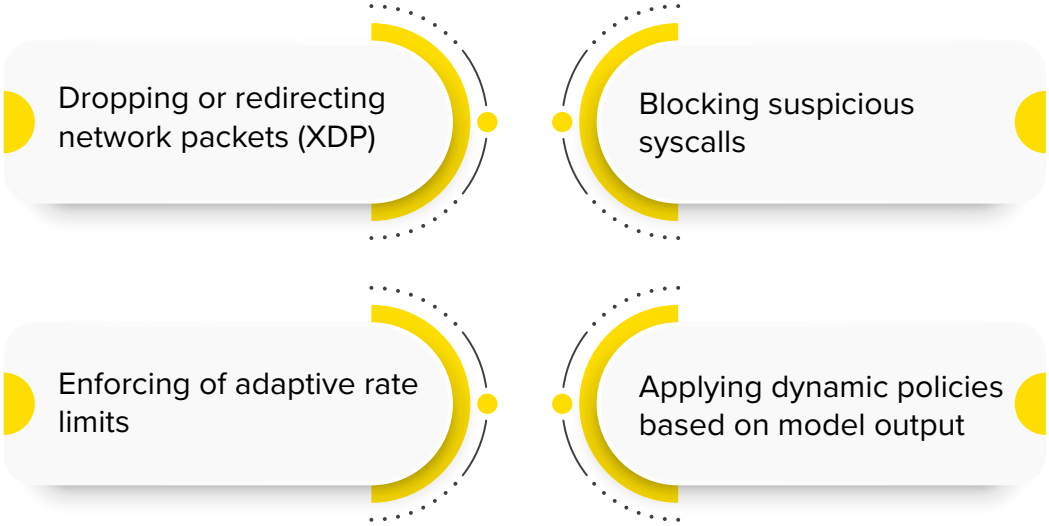
eBPF is especially powerful for security-focused AI use cases:



When combined with ML models, eBPF enables behavior-based security, not just signature-based detection.

6. Closed-Loop AI Control and Automation

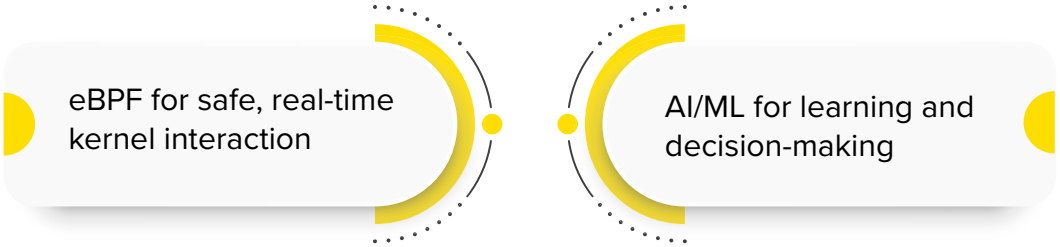
Beyond observation, eBPF allows AI systems to take action:



This enables closed-loop AI systems where models observe, decide, and enforce actions in real time—directly at the kernel level.

7. Ideal Foundation for Autonomous Systems

By combining:



Linux systems can evolve toward self-observing, self-protecting, and self-optimizing platforms.

Proposed Technical Approach: AI-Driven Self-Healing Linux Runtime

The proposed architecture is designed as an autonomous, self-healing runtime that utilizes eBPF for deep kernel sensing and real-time enforcement. Instead of relying on external agents, this approach embeds intelligence directly into the kernel path.

1. Kernel-Level Sensing & Feature Extraction

The system is designed to use eBPF programs to monitor syscalls, network flows, CPU scheduling, and file system activity. Rather than simply exporting raw data, the eBPF layer will:

- Perform lightweight behavioural fingerprinting in-kernel.
- Stream aggregated, high-signal data to user space to reduce overhead.

2. Intelligent Decision Loop

A high-performance user-space agent (developed in Go or Rust) will handle the heavy lifting:

Model Inference

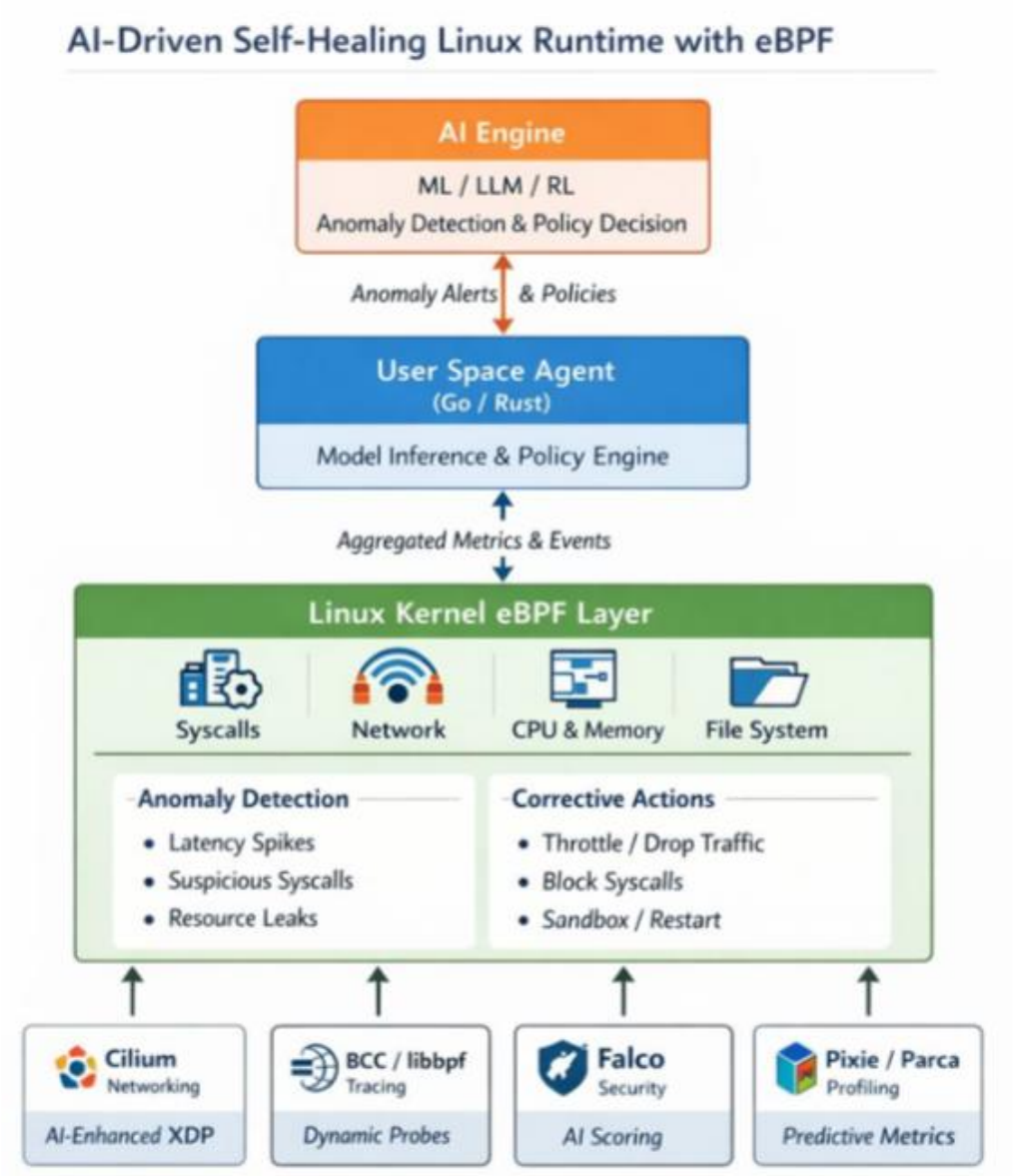
Learning "normal" system behaviour to identify anomalies early.

Policy Logic

Determining the best corrective action based on real-time scoring.

Feedback Loop

Pushing decisions back to the kernel via eBPF maps for immediate execution.



3. Real-Time In-Kernel Enforcement

When a threat or failure is identified, the system will trigger automated responses directly in the kernel:

Throttling

Restricting resources for misbehaving processes.

XDP Filtering

Dropping or rate-limiting malicious traffic at the network driver level.

Syscall Blocking

Instantly preventing high-risk system calls.

Dynamic Sandboxing

Isolating workloads without requiring a container restart.

4. Component Ecosystem

The architecture is designed to integrate and extend proven open-source technologies:

- **Cilium** for XDP and networking, extended with AI-driven policy injection
- **BCC / libbpf** for syscall and scheduler tracing, extended with behavioural fingerprints
- **Falco** for security signals, extended with model-based scoring instead of static rules
- **Pixie / Parca** for profiling, extended with predictive baselining

The Path Forward for eBPF

eBPF is not just a passing fad; it's a foundational technology that is continually evolving. Its integration into the Linux kernel is deep and growing, with new features and hook points being added regularly. We can expect to see even wider adoption in areas like:

Serverless Computing: Providing efficient and secure sandboxing for serverless functions.

Service Mesh: Enhancing performance and observability of service mesh implementations.

Next-generation Firewalls and IDS/IPS: Leveraging eBPF for even more granular and high-performance packet inspection and policy enforcement.

Edge Computing: Enabling lightweight and efficient monitoring and security on resource-constrained devices

eBPF has truly empowered developers and system administrators with a "superpower" to understand, secure, and optimize their Linux systems that were previously beyond imagination. Its impact on the future of cloud-native infrastructure and beyond is undeniable.

Conclusion

eBPF represents a fundamental shift in how we interact with the operating system kernel. What began as a packet filtering mechanism has evolved into a powerful, secure, and extensible platform for observability, security, networking, and AI-driven systems.

By unlocking kernel-level insight and control—without sacrificing safety—eBPF is redefining how modern infrastructure is built, monitored, and protected.

About the Author

Debjyoti Mukherjee
Senior Technical Architect, PDES

Debjyoti Mukherjee is a Senior Technical Architect with over 18 years of experience in the design and development of high-performance networking products. A hands-on leader, Debjyoti specializes in Datacom networking, with deep expertise spanning L2/L3 protocols, advanced datapath architectures, SDN, and NFV.

Throughout his career, he has taken the lead on designing various features across diverse customer product lines. His technical contributions are particularly noted in the development of both hardware and software-based pipelines, leveraging technologies such as SmartNICs, FPGAs, and DPDK to optimize network performance. His extensive industry perspective is shaped by his work with renowned companies such as Cisco, AMD, Netscout and others

happiest minds
AI FIRST. AGILE ALWAYS.

www.happiestminds.com

About Happiest Minds Technologies

Happiest Minds Technologies Limited (BSE, NSE: HAPPSTMNDS) is an AI First, customer-centric digital engineering company committed to delivering 'Happiest People . Happiest Customers'. With an integrated approach that spans from chip to cloud, Happiest Minds delivers secure and scalable solutions across product engineering, cybersecurity, analytics , and automation platforms. Happiest Minds brings purpose and precision to every engagement, helping enterprises solve complex business challenges and fast-track their digital evolution across industry sectors such as Banking, Financial Services & Insurance (BFSI), EdTech, Healthcare & Life Sciences, Hi-Tech and Media & Entertainment, Industrial, Manufacturing, Energy & Utilities, and Retail, CPG & Logistics.

Happiest Minds has been honored by both the Golden Peacock Awards and the Institute of Company Secretaries of India (ICSI) for its exemplary Corporate Governance practices. Guided by its mission of 'Happiest People . Happiest Customers' and consistently recognized as a great place to work, Happiest Minds is headquartered in Bengaluru, India, with a global presence across the Americas, UK, Europe, Australia, the Middle East, Africa, and Asia.

For more information, please write to us at business@happiestminds.com