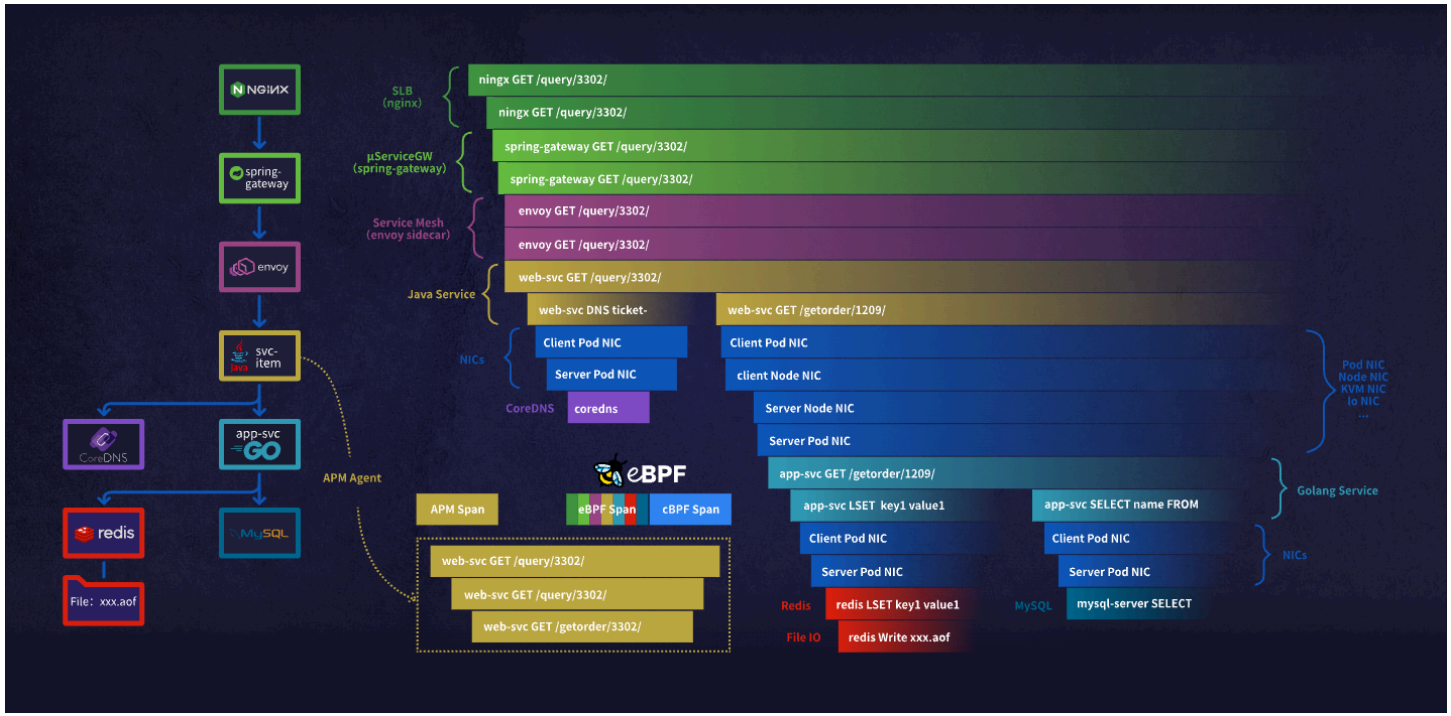
eBPF: The Key Technology to Observability

Jie JIN (金捷) 2023-11-17 Reads: 4969

Observability eBPF



eBPF: The Key Technology to Observability

Observability in control theory refers to the extent to which a system's internal state can be observed by its external outputs. In complex IT systems, having observability is necessary to achieve a predetermined level of stability and error rate. With the rapid expansion of microservices and the fast evolution of cloud-native infrastructure, building observability has become a necessary condition for ensuring business stability.

However, traditional application performance monitoring (APM) cannot achieve real observability. On one hand, instrumentation has modified the original program, making it logically impossible to achieve observability with original code. On the other hand, with the increasing number of cloud-native infrastructure components, it becomes harder to instrument basic services, resulting in more blind spots. In fact, instrumentation is almost impossible to deploy in core business systems of important industries such as finance and telecommunications. Due to its Zero Code advantages, eBPF avoids the drawbacks of APM instrumentation, and has become the key technology for achieving observability in the cloud-native era.

This article discusses the reasons why APM cannot achieve real observability, analyzes why eBPF is the key technology to observability, introduces three core functionalities of DeepFlow based on eBPF, and further explains how to inject business semantics into eBPF's observability data. Afterwards, this article shares nine real-world use cases of DeepFlow users and summarizes common questions that users had before adopting eBPF technology. Finally, this article further analyzes the significant implications of eBPF for new technology iterations.

0x0: Real Observability is Unachievable with APM

APM aims to achieve application observability through code instrumentation. By using instrumentation, applications can expose a wide range of observation signals, including metrics, tracing, logging, and function performance profiling. However, the behavior of instrumentation actually changes the internal state of the original program, which does not logically meet the requirement of observability, which is to "determine the internal state from external data". In core business systems of important industries such as finance and telecommunications, it is very difficult to deploy APM agents. In the era of cloud-native, this traditional approach faces even more serious challenges. Overall, APM's problems mainly

manifest in two aspects: the intrusiveness of agents makes deployment difficult, and observation blind spots make fault triage become impossible.

Firstly, the intrusiveness of probes makes it difficult to implement. The process of instrumentation requires modification of the source code of the application and re-releasing it. Even with bytecode enhancement techniques like Java Agent, it still requires modifying the application's startup parameters and re-releasing it. However, the modification of the application code is just the first hurdle. Typically, during the deployment process, many other issues may arise:

1. **Code conflicts**: Do you often encounter runtime conflicts between different agents when injecting multiple Java Agents for purposes such as distributed tracing, performance profiling, logging, or service mesh? Have you encountered compilation failures due to conflicting library versions when introducing an observability SDK? The more business teams there are, the more evident these compatibility issues become.
2. **Maintenance difficulties**: If you are responsible for maintaining the company's Java Agent or SDK, how frequently do you update them? At the same time, how many versions of probe programs are running in your company's production environment? How much time does it take to update them to the same version? How many different language probe programs do you need to maintain? When the enterprise's microservice framework or RPC framework cannot be unified, these maintenance issues become even more severe.
3. **Blurred boundaries**: All the instrumented code seamlessly integrates into the running logic of the business code, making it difficult to distinguish and control. It is hard to tell whether these code are causing performance degradation or runtime errors occur. Even if the probes have been extensively used in production and polished, suspicion still falls on them when problems arise.

In fact, this is also why **intrusive instrumentation schemes are rare in successful commercial products and more common in active open-source communities**. The activity of communities

such as OpenTelemetry and SkyWalking serves as evidence for this. In large enterprises with clear divisions, overcoming collaboration difficulties is an inevitable challenge for a technical solution to be successfully deployed. Especially in key industries such as finance, telecommunications, and power, where national welfare is at stake, the differentiation of responsibilities and conflicts of interest between departments often make intrusive solutions "impossible". Even in internet companies that embrace open collaboration, there are still issues such as developers' reluctance to instrument their code and blaming operation engineers for performance failures. After a long period of effort, people realized that intrusive solutions are only suitable in the case that each business development team voluntarily introducing, maintaining various agent and SDK versions, taking responsibility for performance risks and operational failures all by themselves. Of course, we have also seen noticed cases of large internet companies that have achieved success due to highly unified infrastructure. For example, Google openly admitted in their 2010 Dapper paper:

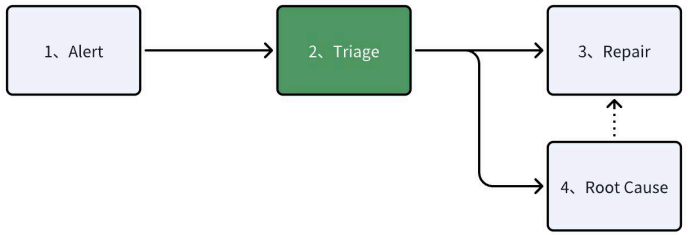
> True application-level transparency, possibly our most challenging design goal, was achieved by restricting Dapper's core tracing instrumentation to a small corpus of ubiquitous threading, control flow, and RPC library code.

In the 2022 external sharing of "分布式链路追踪在字节跳动的实践 (Distributed Trace in Practice at ByteDance)" ByteDance also stated:

> 得益于长期的统一基建工作，字节全公司范围内的所有微服务使用的底层技术方案统一度较高。绝大部分微服务都部署在公司统一的容器平台上，采用统一的公司微服务框架和网格方案，使用公司统一提供的存储组件及相应 SDK。高度的一致性对于基础架构团队建设公司级别的统一链路追踪系统提供了有利的基础。

> Thanks to long-term unified infrastructure work, there is a high level of consistency in the underlying technical solutions used by all microservices within ByteDance. The majority of microservices are deployed on the company's unified container platform, using the company's unified microservice framework and mesh scheme, and utilizing the company's provided storage components and corresponding SDKs. This high level of consistency provides a favorable foundation for the infrastructure team to build a company-level unified trace system.

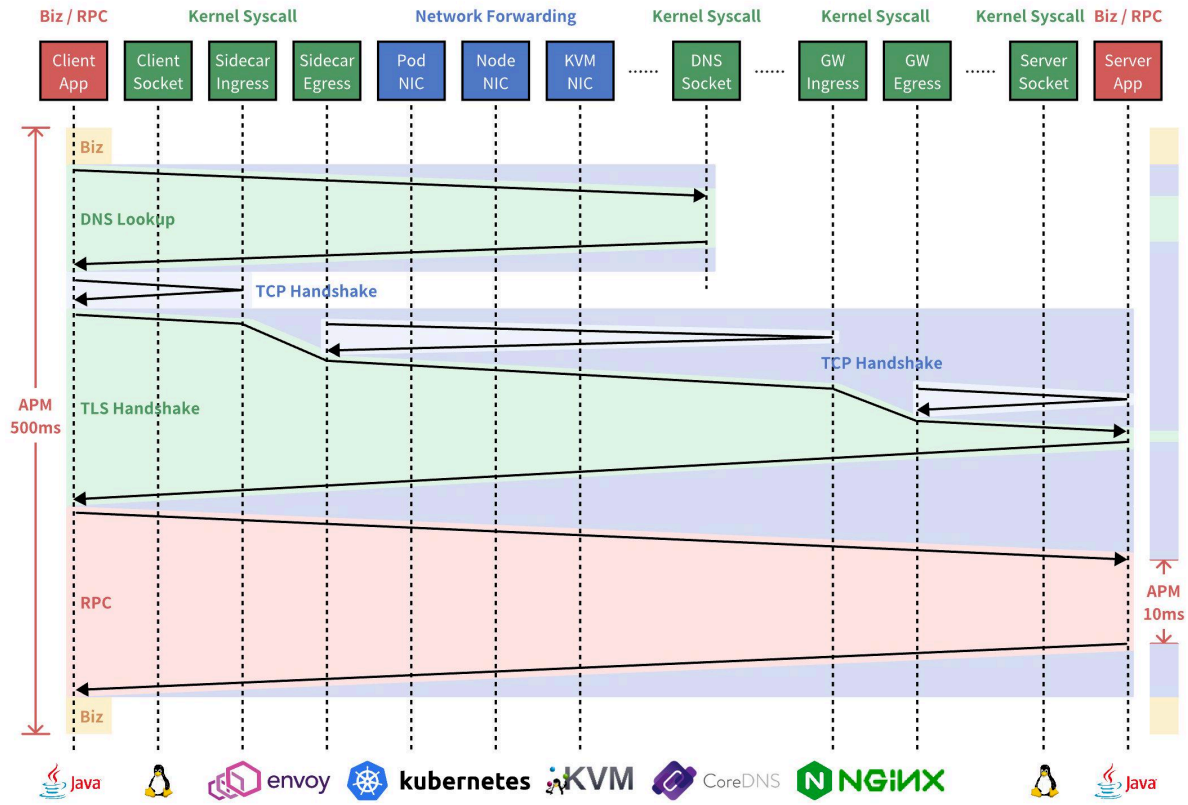
Secondly, obervation blind spots make fault triage impossible. Even if APM has been deployed within the organization, we still find it difficult to define troubleshooting boundaries, especially in cloud-native infrastructure. This is because developers and operations often use different terms for communication. For example, when a call latency is high, developers may suspect slow network, slow gateways, slow databases, or slow server-side response. However, due to the lack of full-stack observability, the responses from the network, gateways, databases usually indicate no dropped packets, low CPU usage, no slow log, and low server-side latency, which are a bunch of unrelated metrics. Consequently, the problem remains unresolved. Fault triage is the most critical aspect of the entire troubleshooting process, and its efficiency is of utmost importance.



The core role of triage in the process of fault handling

We would like to clarify two concepts: **Troubleshooting Boundary** and **Responsibility Boundary**. While the responsibility boundary of developers lies within the application itself, the

troubleshooting boundary extends to network transmission. For example, when a microservice experiences intermittent latency of up to 200ms when requesting RDS cloud services, if developers submit a ticket to the cloud service provider based on this, the response they are likely to receive is “RDS has not observed any slow logs, please self-examine”. We have encountered numerous cases like this with many clients, and the underlying causes vary, such as SLB in front of RDS or SNAT in K8s Node. **If fault triage cannot be done immediately, it will result in a prolonged back-and-forth battle between tenants (developers) and cloud service providers (infrastructure) lasting for days or even weeks.** From the perspective of troubleshooting boundary, if developers can provide info like “network latency between sending a request and receiving a response is as high as 200ms”, triage can be done quickly and push the cloud service provider to investigate. Once the right person is found, the subsequent problem resolution is usually very fast. We will also share several real-life cases in the following sections.



APM is difficult to achieve fault triage of cloud-native applications

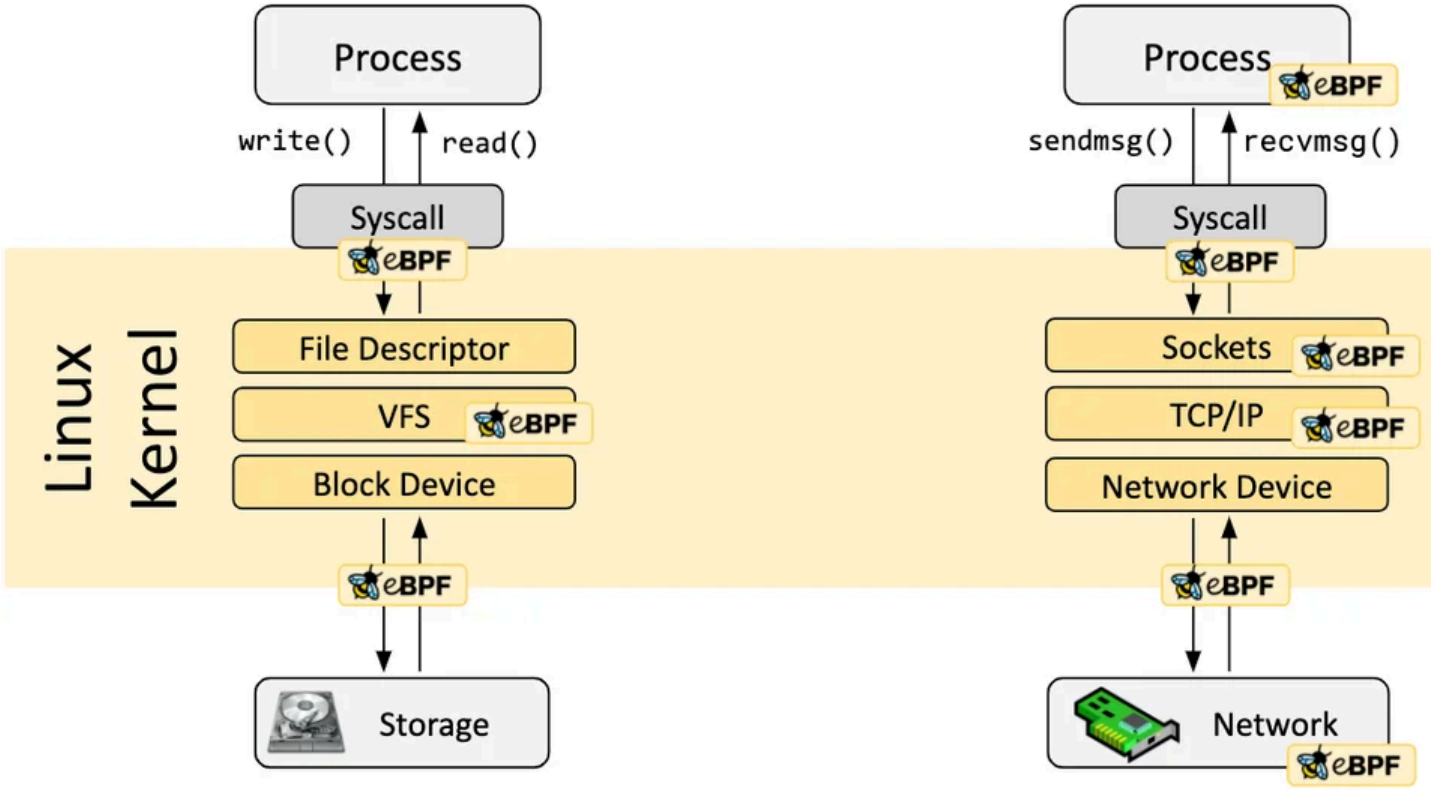
The above figure clearly explains why it is difficult to achieve fault triage of cloud-native applications using APM. Based on the request delay obtained from APM, people cannot well judge where the bottleneck comes from: business logic code, system calls, service mesh, K8s network, cloud network, DNS, TLS handshake, various gateways... If you are a microservice developer, you should not only focus on the business logic itself but also concern yourself with system calls and network transmissions. If you are a Serverless tenant, you may also need to pay attention to service mesh sidecars and their network transmissions. If you directly use virtual machines or build your own K8s cluster, container networking is a key issue that requires special attention, particularly with regards to core services such as CoreDNS and Ingress Gateway in K8s. If you are a private cloud computing service administrator, you should focus on

the network performance of KVM hosts. If you are part of the private cloud gateway, storage, or security team, you also need to pay attention to system calls and network transmission performance on service nodes. In fact, the more important aspect is that the data used for fault triage should be described in a similar manner: **How much time is consumed by each hop in the entire full-stack path of a single application call**. Through the aforementioned analysis, we have discovered that the observation data provided by developers through instrumentation may only account for 1/4 of the entire full-stack path. **Relying solely on APM to solve fault triage in the era of cloud-native is an illusion.**

0x1: Why eBPF is the Key to Obervability

This article assumes that you have a basic understanding of eBPF. It is a secure and efficient technology that extends kernel functionality by running programs in a sandbox. It is a revolutionary innovation compared to traditional methods of modifying kernel source code and writing kernel modules. You can visit ebpf.io for more information on eBPF. This article focuses on the revolutionary significance of eBPF for observability in cloud-native applications.

eBPF programs are event-driven, meaning that when the kernel or a user program encounters an eBPF hook, the corresponding eBPF program loaded on that hook point will be executed. Linux kernel provides a series of predefined common hook points, and you can also dynamically add custom hook points for the kernel and application programs using kprobe and uprobe. Thanks to the Just-in-Time (JIT) technology, the efficiency of eBPF code can match that of native kernel code and kernel modules. Thanks to the Verification mechanism, eBPF code will run safely without causing kernel crashes or entering infinite loops.



https://ebpf.io/what-is-ebpf/#hook-overview

Back to terms of observability, the sandbox mechanism is the core difference between eBPF and APM instrumentation. **The "sandbox" establishes a clear boundary between eBPF code and application code, allowing us to determine the internal state of the application by obtaining external data without modifying the application itself.** Now let's analyze why eBPF is an excellent solution to address the shortcomings of APM code instrumentation.

Firstly, it solves the problem of difficult deployment with Zero Code. Since eBPF programs do not require modification of the application code, there are no runtime conflicts like Java Agent or compile-time conflicts like SDK, thereby **resolving code conflicts**. As eBPF programs can run without changing or restarting the application process, there is no need to re-release the

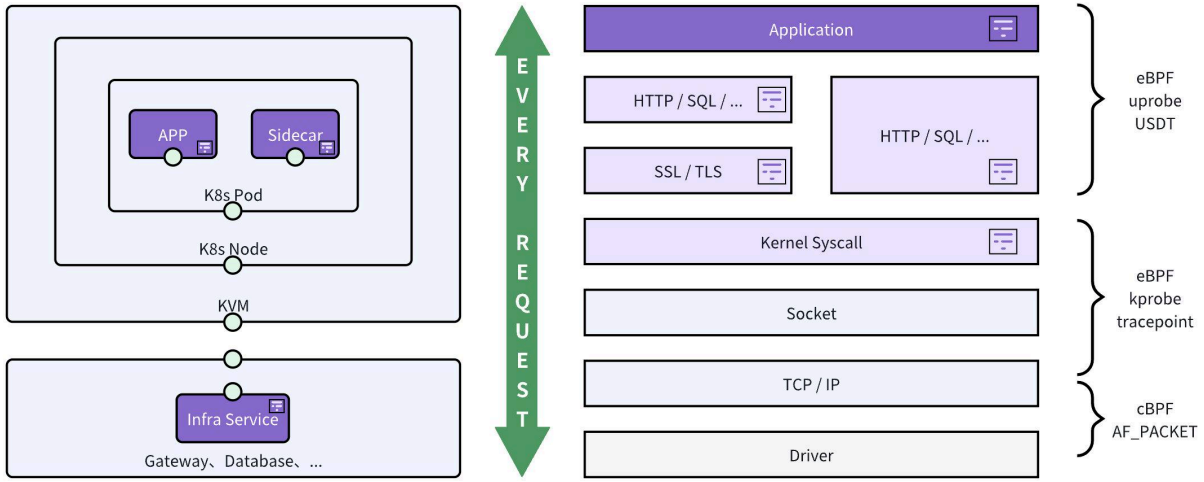
application, eliminating the pains of maintaining Java Agent and SDK versions, thereby **resolving maintenance difficulties**. Additionally, with the assurance of JIT technology and Verification mechanism, eBPF can run efficiently and securely, eliminating concerns about unexpected performance degradation or runtime errors, thus **resolving the issue of ambiguous boundaries**. Furthermore, on the management level, running a separate eBPF Agent process on each host allows for precise control of resources such as CPU consumption.

Secondly, it solves the problem of difficult fault triage with its full-stack capabilities. eBPF's capabilities cover every layer from the kernel to user programs, enabling us to trace a request's full-stack path starting from the application, passing through system calls, network transmission, gateway services, security services, and reaching the database service or peer microservice. **This provides ample neutral observability data to quickly identify faults.**

However, eBPF is not an easily mastered technology as it requires developers to have a certain understanding of kernel programming. Moreover, the raw data obtained lacks structured information. In the following sections, we will use our product DeepFlow as an example to explain how to overcome these obstacles and fully leverage the key role of eBPF in observability engineering.

0x2: Three Core Features of DeepFlow Based on eBPF

DeepFlow GitHub aims to provide a simple and deployable deep observability solution for complex cloud-native applications. With eBPF and Wasm technologies, DeepFlow enables Zero Code, full-stack data collection of metrics, tracing, call logs, and function profiling. It achieves Universal Tagging and efficient access to all data through Smart Tag techniques. By utilizing DeepFlow, cloud-native applications can automatically gain deep observability, relieving developers from the burdensome task of constant instrumentation. Additionally, it empowers DevOps/SRE teams with monitoring and diagnostic capabilities from code to infrastructure.

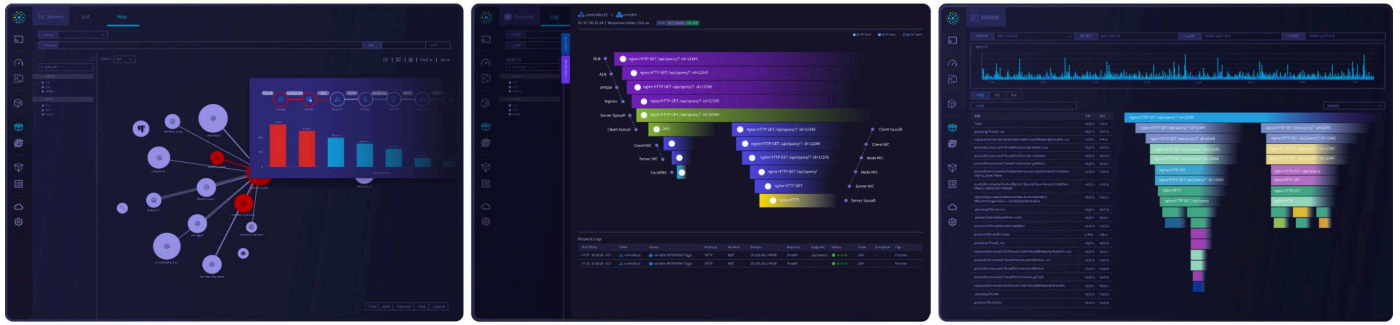


DeepFlow utilizes eBPF technology to achieve Zero Code observability for cloud-native applications

By utilizing eBPF and cBPF to collect data from application functions, system call functions, and network card traffic, DeepFlow first aggregates this data into TCP/UDP flow logs. Through application protocol recognition, DeepFlow further aggregates this data into application request logs, enabling the calculation of RED (Request/Error/Delay) performance metrics for the entire stack and facilitating distributed tracing through the correlation of call logs. Additionally, during the flow log aggregation process, DeepFlow calculates TCP throughput, latency, connection abnormalities, retransmissions, zero windows, and other network layer performance metrics. It also calculates IO throughput and latency metrics through hook file read/write operations, correlating all these metrics with each call log. Furthermore, DeepFlow supports obtaining OnCPU and OffCPU function flame graphs for each process through eBPF, as well as analyzing TCP packets to draw Network Profile temporal graphs. All of these capabilities ultimately manifest as three core features:

- Universal Map for Any Service
- Distributed Tracing for Any Request

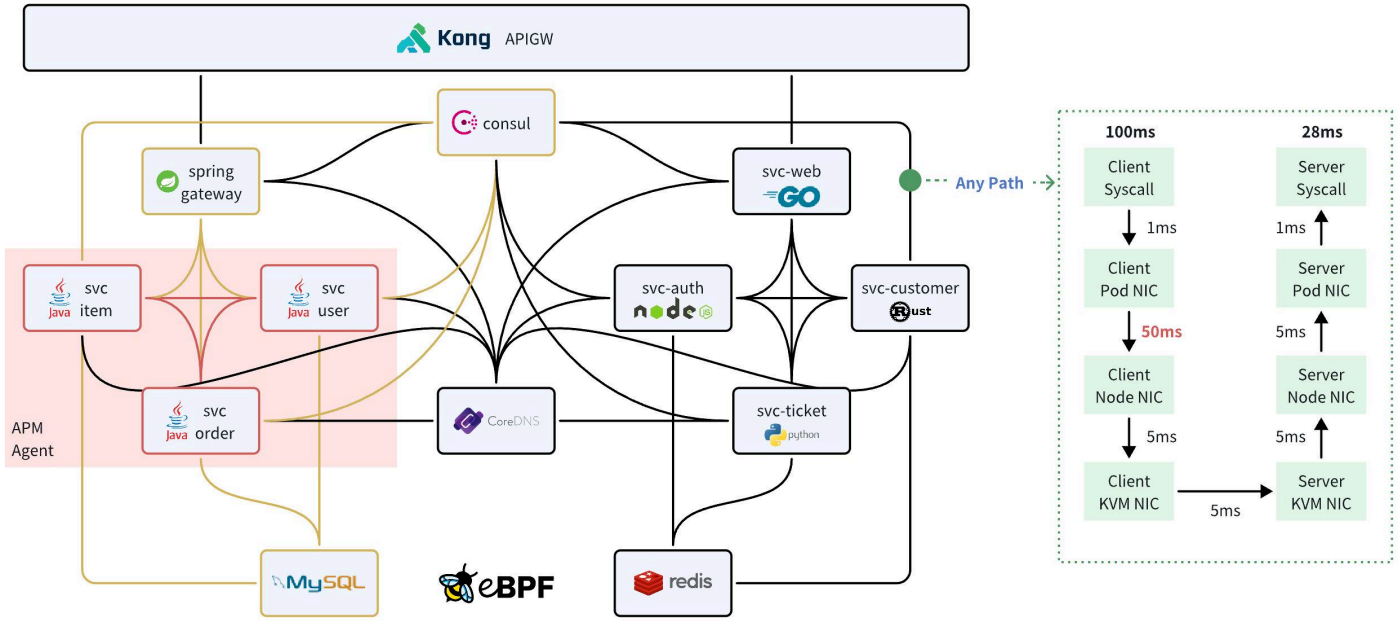
- Continuous Profiling for Any Function



Universal Map for Any Service | Distributed Tracing for Any Request | Continuous Profiling for Any Function

Three core features of DeepFlow based on eBPF

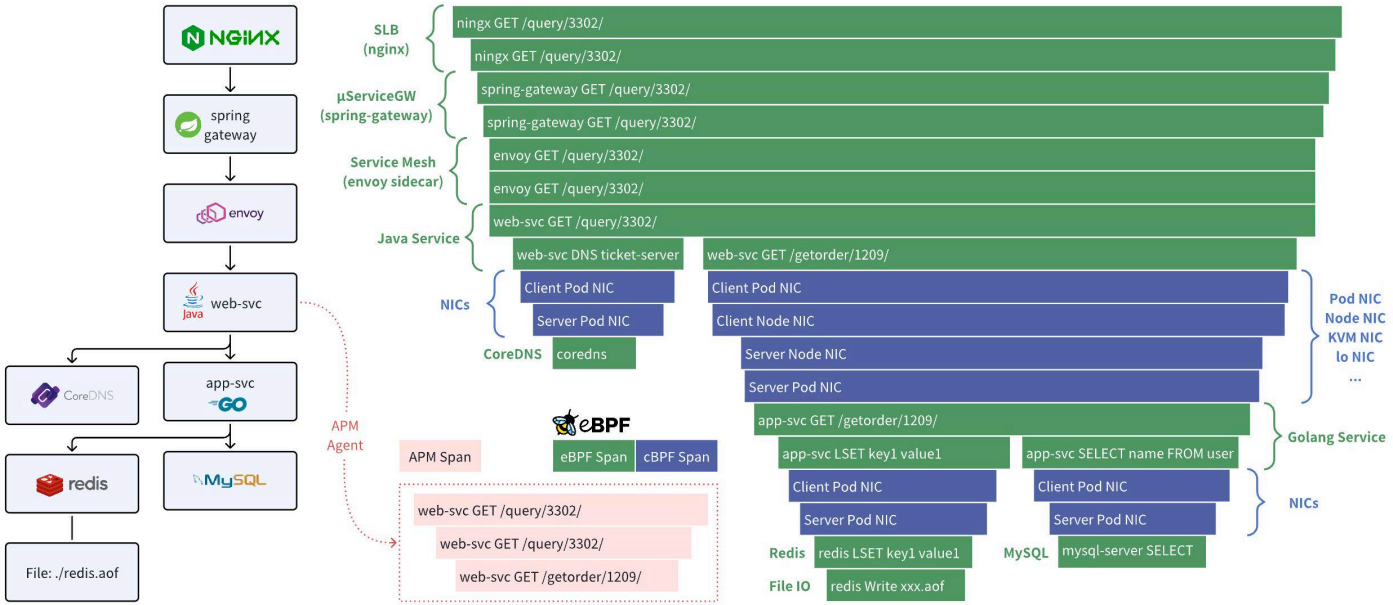
Core Feature 1: Panoramic View of Any Service. The Universal Map directly demonstrates the Zero Code advantages of eBPF, in contrast to the limited coverage ability of APM. All services can be displayed in the Universal Map. However, the call logs obtained by eBPF cannot be directly used for topology visualization. DeepFlow injects rich tags into all data, including cloud resource attributes, K8s resource attributes, and custom K8s labels. Through these labels, it is possible to quickly filter out the Universal Map of specific businesses and display them in different groups based on different labels, such as K8s Pod, K8s Deployment, K8s Service, custom labels, etc. **The Universal Map not only describes the call relationships between services but also showcases the full-stack performance indicators along the call paths.** For example, on the right side of the figure below, the inter-process delay changes when the two K8s services access each other. We can quickly identify whether the performance bottleneck lies in the business process, container network, K8s network, KVM network, or Underlay network. Sufficient neutral observation data is a necessary condition for rapid triage.



Comparison of DeepFlow Universal Map with the topology map obtained by the APM Agent

Core Feature 2: Distributed Tracing with Arbitrary Invocation. Zero Code distributed tracing (**AutoTracing**) is a significant innovation in DeepFlow. When collecting invocation logs through eBPF and cBPF, DeepFlow calculates information such as syscall_trace_id, thread_id, goroutine_id, cap_seq, tcp_seq based on the system call context. This allows for distributed tracing **without modifying application code or injecting TraceID and SpanID**. Currently, DeepFlow can achieve Zero Code distributed tracing for all cases except for cross-thread communication (through memory queues or channels) and asynchronous invocations. Additionally, DeepFlow supports parsing the unique Request ID injected by applications (such as the commonly used X-Request-ID in gateways) to resolve cross-thread and asynchronous issues. The diagram below compares the distributed tracing capabilities of DeepFlow and APM. APM can only trace instrumented services, typically achieved through Java Agent covering Java services. On the other hand, DeepFlow uses eBPF to trace all services, including SLBs such as Nginx, microservice gateways like Spring Cloud Gateway, service meshes like Envoy, and fundamental services like MySQL, Redis, and CoreDNS (including their file read/write latency).

Additionally, DeepFlow covers network transmission paths such as Pod NIC, Node NIC, KVM NIC, and physical switches. Furthermore, it provides seamless support for Java, Golang, and all other programming languages.

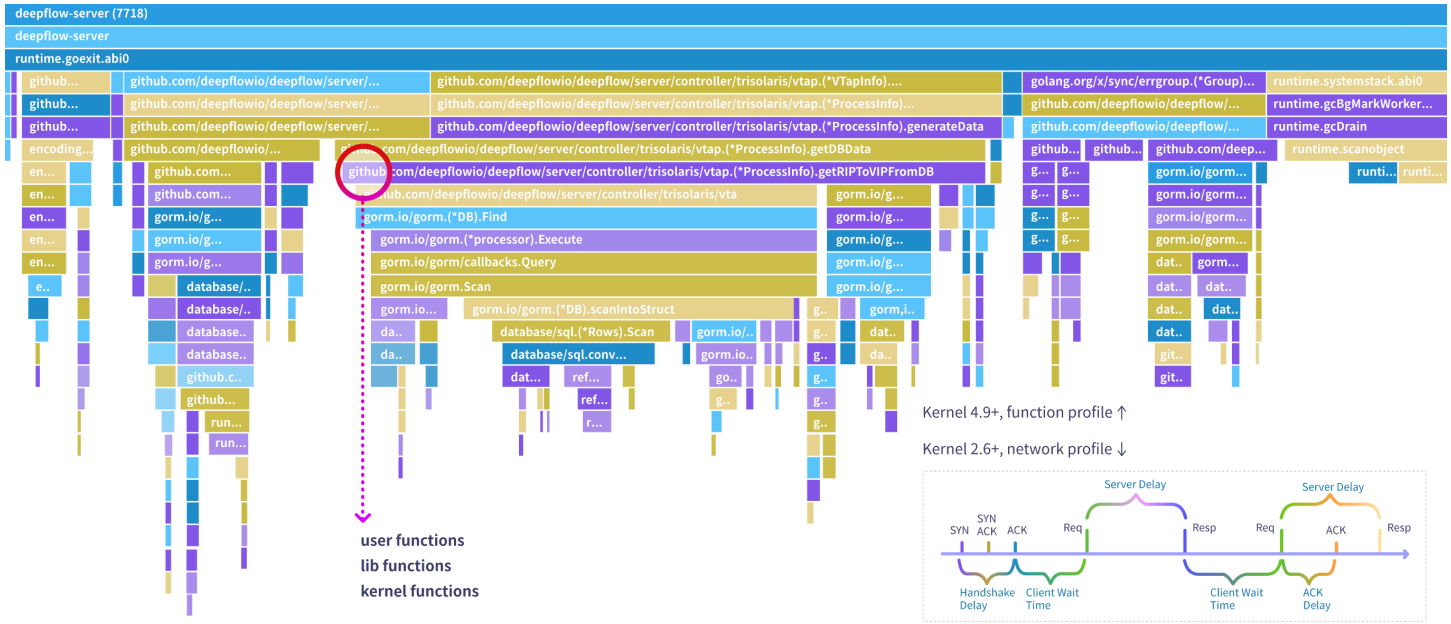


Comparison of DeepFlow distributed tracing with APM

Please note that the distributed tracing capabilities of eBPF and APM are not contradictory. APM can be used to trace the function call paths within an application process and excels at resolving cross-thread and asynchronous scenarios. On the other hand, eBPF has global coverage capabilities, allowing it to easily cover gateways, foundational services, network paths, and multi-language services. In DeepFlow, we support calling APM's Trace API to display the complete distributed tracing graph of APM + eBPF. Additionally, we provide the `Trace Completion API` to allow APM to call DeepFlow and obtain and correlate eBPF tracing data.

Core Feature 3: Continuous Profiling of Any Function. By capturing snapshots of the application's function call stack, DeepFlow is able to create CPU profiles of any process, helping

developers quickly identify function performance bottlenecks. **The function call stack not only includes business functions but also shows the time spent on dynamic linked libraries and kernel system call functions.** Furthermore, DeepFlow generates a unique identifier when collecting the function call stack, which can be used to associate with call logs, achieving the synergy between distributed tracing and function performance profiling. Specifically, DeepFlow uses cBPF to analyze individual packets in the network, allowing for the creation of Network Profiles for each TCP flow, analyzing connection establishment latency, system (ACK) latency, service response latency, and client waiting latency. **Network Profile can be used to infer the code range of performance bottlenecks in the application**, We will also share relevant case studies in later sections.



CPU Profile and Network Profile in DeepFlow

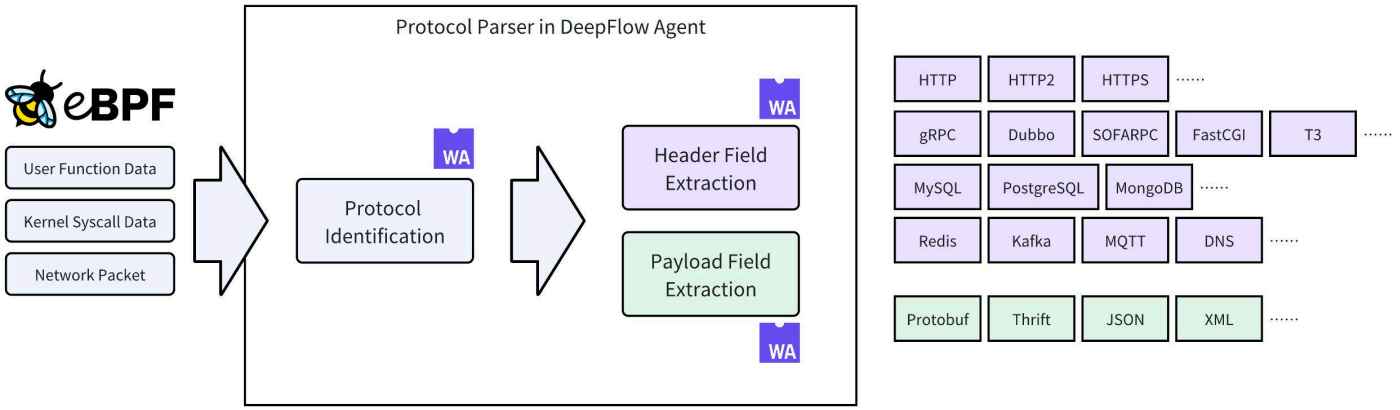
This article cannot fully explain the principles behind these exciting features. DeepFlow is also an open-source project, and you can read our GitHub code and documentation to learn more. You can also read our paper published at the top conference in the field of network communication,

ACM SIGCOMM 2023, for more information. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code

0x3: Inject Business Semantics into eBPF Observation Data

Another demand for using the APM Agent is to inject business semantics into the data, such as **user information, transaction information, and the name of the business module associated with a call**. It is difficult to extract business semantics from the raw byte stream collected by eBPF using general methods. In DeepFlow, we have implemented two plugin mechanisms to address this limitation: injecting call-level business semantics through the Wasm Plugin and injecting service-level business semantics through the API.

Firstly, injecting call-level business semantics through the Wasm Plugin: The DeepFlow Agent has built-in parsing capabilities for common application protocols, and we continuously add support for more protocols, as indicated by the blue parts in the diagram below. However, we have found that the actual business environment can be more complex. For example, developers may insist on returning an HTTP 200 status code while placing error information in a custom JSON structure. Additionally, a significant portion of RPC payloads may use serialization methods that rely on schemas, such as Protobuf and Thrift. Furthermore, there may be issues with eBPF AutoTracing being disrupted due to cross-thread calls in the processing flow. To address these challenges, DeepFlow offers a Wasm Plugin mechanism that allows developers to enhance the ProtocolParser in the pipeline.



Inject call-level business semantics with DeepFlow Wasm plugin

In fact, we have also observed the existence of "natural" distributed tracking markers in industries such as finance, telecommunications, and gaming. For example, there are global transaction IDs in financial transactions, call IDs in telecommunications core networks, and business request IDs in gaming. These IDs are carried in all calls, but their specific locations are determined by the business itself. With the flexibility provided by the Wasm Plugin, developers can easily write plugins to extract these information as TraceIDs.

Secondly, injecting service-level business semantics through APIs: By default, DeepFlow SmartEncoding mechanism automatically injects cloud resources, container K8s resources, and K8s custom label tags into all observation signals. However, these tags only reflect application-level semantics. To help users inject business semantics from systems such as CMDB into observation data, DeepFlow provides a set of APIs for business tag injection.

0x4: Real Use Cases of DeepFlow

In this chapter, we will share nine types of practical cases of DeepFlow users with you. These cases are difficult and unexpected problems. We'll see that when there is only APM data, it can take several days or even weeks to find a direction, but with the power of eBPF, the

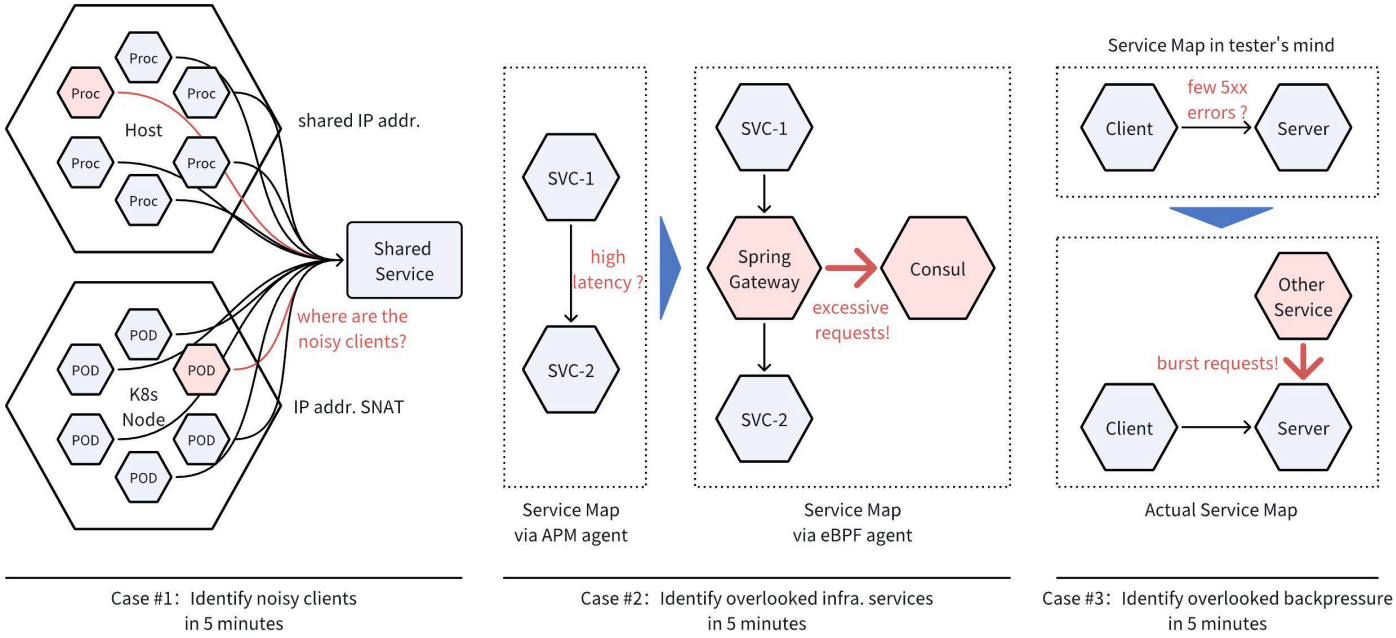
troubleshooting can often be completed within 5 minutes. **Before introducing them, I want to clarify that this does not mean that eBPF is only good at solving difficult problems.** We now know that eBPF can collect metrics, request logs, profiles, and other observation signals with Zero Code. DeepFlow has also implemented a Universal Map (including performance indicators, call logs, etc.), distributed tracing, and continuous performance profiling based on these signals.

Case Type 1: Fast triage of the problematic service:

- Case 1: **Locating the top clients accessing shared services in 5 minutes.** Infrastructure such as MySQL, Redis, and Consul are often shared by many microservices. When their load is high, it is usually difficult to determine which clients are causing the issue. This is because container pods accessing these shared services usually perform SNAT, so the server sees the IP of the container node. In non-container environments, there are also many processes sharing the host's IP. It is inefficient to analyze IP addresses from the server's call logs, and we cannot expect all clients to inject APM Agents. With DeepFlow, one of our large banking clients was able to quickly locate the microservices with the highest frequency of requests to the RDS cluster from nearly 100,000 pods in 5 minutes. Similarly, one of our smart car clients was able to quickly locate the microservices with the highest frequency of requests to Consul from tens of thousands of pods in 5 minutes.
- Case 2: **Identificating overlooked shared services in 5 minutes**. A major bank client of DeepFlow encountered poor performance of their "Distributed Core Transaction System" after migrating from a physical environment to a private cloud during online testing. After two weeks of investigation and injecting numerous APM Agents, the problem can only be identified in the link between the service named `cr****rs` and the access authorization transaction service `au****in`. However, these two services did not have any performance issues before migrating to the cloud. The development team initially suspected the private cloud infrastructure, but had no supporting data. When they were at a loss, they found the

DeepFlow team. After deploying the eBPF Agent, all access relationships and performance indicators between microservices became visible. It was immediately discovered that when `cr****rs` accessed the authorization transaction service `au****in`, it also passed through Spring Cloud Gateway, which was making requests to the service registry center Consul at an extremely high rate. At this point, the problem became clear - it was due to improper cache configuration in the gateway, which caused the service registry center to become a bottleneck.

- Case 3: **Identificating overlooked background pressure in 5 minutes**. During software development, the stress testing environment is usually shared by multiple people, and even multiple teams, such as development and testing, may use the same set of stress testing environments. In a test conducted by a smart car client of DeepFlow, the testing personnel found that there were always a small number of HTTP 5XX errors occurring, which directly invalidated the test results. When the testing personnel were at a loss, they immediately opened the DeepFlow Universal Map and discovered that other services were accessing the tested service at a non-negligible rate.



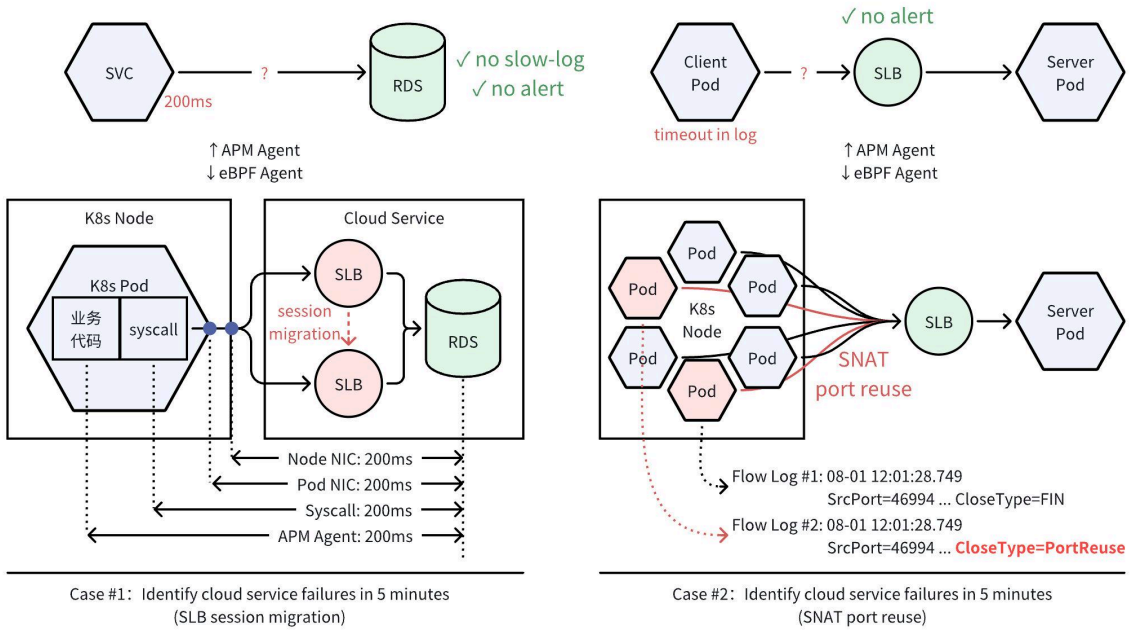
Fast triage of the problematic service

Case Type 2: Fast triage of managed service failures:

- Case 1: **Triage failure of managed cloud service in 5 minutes** - SLB Session Migration. In the past, troubleshooting failures in managed services has been difficult due to the inability to instrument the services. For a DeepFlow smart car customer, there was a high latency issue occurring every 10 minutes in the charging service. Using the APM Agent, the issue was only traced to the RDS access by the charging core service. However, the public cloud service provider closed the ticket after careful examination of slow logs and RDS performance metrics because no anomalies were found. This problem persisted for a week without resolution. Through DeepFlow's full-stack metric data, it was clear that during the failure, the RDS access latency observed from system calls, pod NIC, and node NIC all exceeded 200ms, accompanied by a significant increase in the "Server RST" count in the network metrics. These data pushed the public cloud service provider to re-investigate the issue and eventually discovered that the SLB cluster before RDS triggered session migration under high

concurrency, causing this problem. It can be seen that full-stack observability is crucial for cross-team troubleshooting.

- Case 2: **Triage failure of managed cloud service in 5 minutes** - K8s SNAT Conflict. In this case, SLB also appeared, but the root cause was different. For a DeepFlow smart car customer, the vehicle control service occasionally timed out when accessing the account service, occurring 7 times per pod every day. The public cloud service provider also did not see any abnormal SLB metrics, and this ticket remained unresolved for a month. After reviewing the DeepFlow Universal Map, the troubleshooting was once again quickly completed. It was observed that during the occurrence of the failure, the "Connection Exception" count in the network metrics increased significantly. Further examination of the associated flow logs revealed that the cause of TCP connection failure at that time was "SNAT port conflict". It can be seen that even for timeout-type failures "without call logs", the use of full-stack performance metrics can quickly localize the cause of the failure.

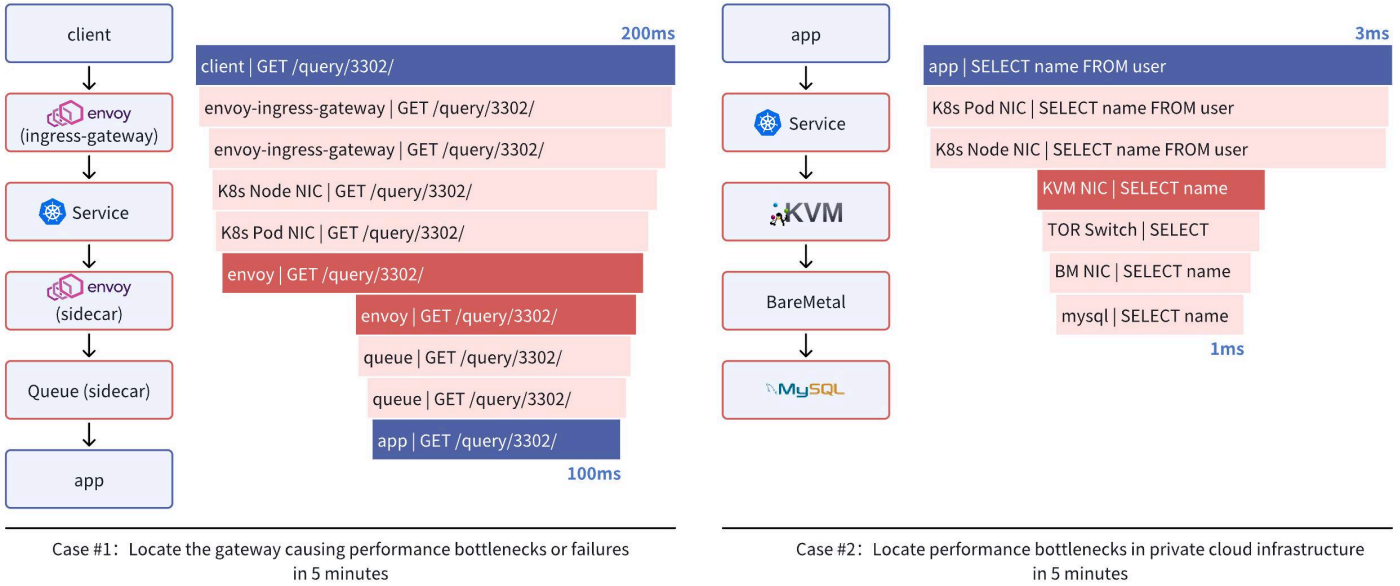


Fast triage of managed service failures

Case Type 3: Fast pinpointing problems with various gateways and cloud infrastructure:

- Case 1: **Pinpointing gateway causing performance bottlenecks or failures in 5 minutes**. In order to centrally implement functionalities such as load balancing, security auditing, microservice splitting, rate limiting, and circuit breaking, various gateways are usually deployed in cloud-native infrastructure. A game client of DeepFlow uses KNative as Serverless infrastructure. In this environment, any client needs to pass through four types of gateways: Envoy Ingress Gateway, K8s Service, Envoy Sidecar, and Queue Sidecar when accessing microservices. In the past, clients mainly used log file and tcpdump packet capturing to troubleshoot issues when the call latency on the client side was much higher than that on the server side or when HTTP 5XX call failures occurred. However, using DeepFlow, it is possible to locate performance bottlenecks or failure points on the gateway path within 5 minutes. For example, one time a slow request problem caused by unreasonable configuration of the Envoy Sidecar was quickly discovered.
- Case 2: **Pinpointing performance bottlenecks in private cloud infrastructure in 5 minutes**. A large bank client of DeepFlow conducted a significant amount of performance testing before launching a private cloud for their “Distributed Core Transaction System”. During the testing, it was found that when accessing the MySQL service on bare metal servers in the K8s cluster, the latency on the client side (3ms) was significantly higher than the latency observed by the DBA team (1ms). This indicated that the infrastructure accounted for 67% of the total time spent during the entire process, but it was unclear which specific part was causing it. Through DeepFlow, it was found that the main latency consumption during the entire access process occurred on the KVM host machine. After providing this data to the private cloud provider, a quick investigation revealed that the host machine in this environment was using ARM CPU and SRIOV network card, and VXLAN Offloading was enabled. In a complex environment, some unreasonable configurations led to high latency in traffic forwarding. By modifying the configuration, DeepFlow observed an 80% decrease in

latency at the KVM, effectively ensuring the smooth launch of the entire distributed core transaction system.



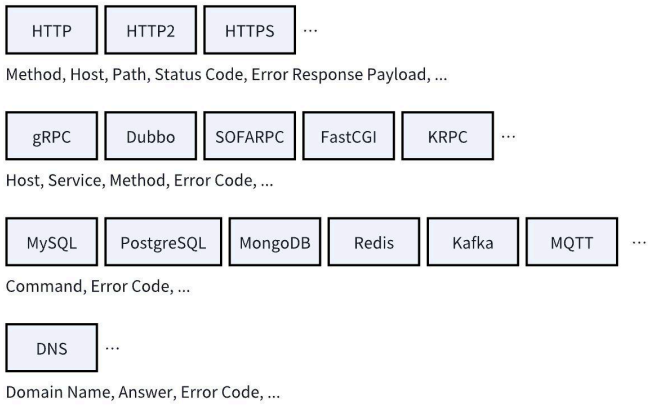
Fast pinpointing problems with various gateways and cloud infrastructure

Case Type 4: Fast locating code issues:

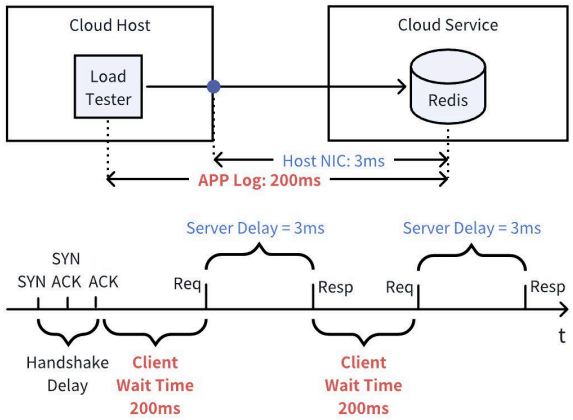
- Case 1: **Identifying legacy code issues by analyzing call logs**. Here, legacy code refers to code that has been inherited from developers who have left the company, or the code has been handed over to different developers multiple times, or it is a service provided by an external vendor without source code access. Even if customers want to enhance the observability of such services through instrumentation, it is difficult. Many of our clients have been able to identify issues with such services on the first day of deploying DeepFlow. For example, a gaming client discovered errors in a Charge API of a game which did not impact the players but resulted in continuous financial losses for the company. Another example is a cloud service provider's development team discovering that a service is writing to a non-existent database table, and the team responsible for the service has already been changed

multiple times. Although it did not cause any business failures, it led to incorrect operational data.

- Case 2: **Identifying code performance bottlenecks using Network Profile**. In the runtime environment of Linux kernel 4.9 and above, using eBPF Profile to locate code performance bottlenecks is a convenient capability. DeepFlow Network Profile can achieve some effects in more common kernel environments. For example, one of our gaming clients discovered that the latency shown by the load testing program for the Redis hosting service reached as high as 200ms. After checking the DeepFlow performance metrics, it showed that the observed latency on the host network interface was less than 3ms. The load testers are not the developers of the load testing program, and the server kernel on which the load testing program runs does not have eBPF capability. In order to understand the reason, the load testers looked at the Network Profile generated by cBPF data and immediately discovered that the Client Wait Time was as high as 200ms. This meant that the load testing program spent too much time between two invocations. When this information was relayed to the development team of the load testing program, they were pleasantly surprised and immediately optimized it, achieving significant improvements.



Case #1: Discover logical problems in ancestral code through Request Log



Case #2: Infer application performance bottlenecks through Network Profiling

Fast locating code issues

All the cases introduced in this chapter are real cases from DeepFlow customers' actual work, hoping to give you a more realistic understanding of the importance of eBPF technology for observability.

0x5: FAQs before applying eBPF

Question 1: To what extent can the eBPF Agent replace the APM Agent? If we only consider distributed tracing purposes, even with cross-thread and asynchronous calls, it is possible to track them by utilizing the unique ID field in the request headers of typical businesses such as finance, telecommunications, and gaming, with the support of the Wasm Plugin. The Wasm Plugin can also be used for extracting business semantics. Therefore, the eBPF Agent can completely replace the APM Agent. For the requirement of tracing the calling paths between internal functions of an application, it generally focuses on microservice frameworks, RPC frameworks, and ORM frameworks. As these types of functions are relatively standard, we believe that in the future, eBPF-based dynamic hook driven by Wasm Plugin can be implemented to obtain Span data within the program.

Question 2: Does the eBPF Agent have high requirements for the kernel? More than half of the capabilities in the DeepFlow Agent can be achieved with cBPF based on kernel 2.6+. When the kernel reaches 4.9+, it supports function performance profiling, and when the kernel reaches 4.14+, it supports eBPF AutoTracing and SSL/TLS encrypted data collection. In addition, with the support of the Wasm Plugin, AutoTracing does not strongly rely on a 4.14+ kernel. It can be implemented on any 2.6+ kernel by extracting the existing unique ID field in the request.

Question 3: Does collecting full-stack data consume a large amount of storage space? A layer 4 gateway does not change the content of a call, and a layer 7 gateway generally only modifies the protocol header of a call. Therefore, the call logs collected from network traffic can be very simple, only containing a few associated information and timestamp information, without the

need to retain detailed request and response fields. This way, the Span collected on the network forwarding path will only add a small storage burden.

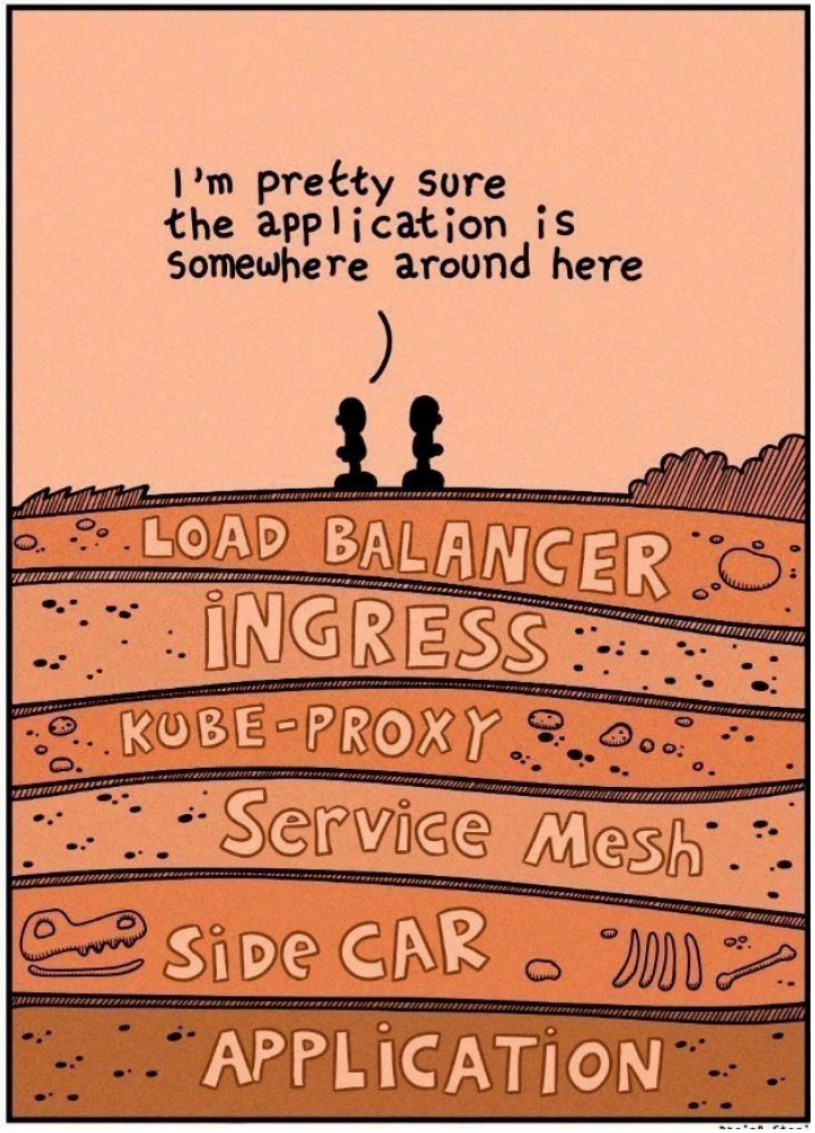
Question 4: Can eBPF be used for implementing RUM? eBPF is not a technology used in browsers, so it is not suitable for web-side. eBPF is a host data collection technology, so it is not suitable for capturing data from all apps on personal mobile devices. However, for endpoint systems fully controlled by enterprises, eBPF has a wide range of applications, such as IoT endpoints based on Linux or Android operating systems, and in-vehicle entertainment systems in smart cars.

0x6: The Significance of eBPF for the Iteration of New Technologies

In the past, APM Agents were unable to achieve observability for infrastructure, leading users to prioritize stability and infrequent changes in infrastructure, which inevitably stifles innovation. Therefore, implementing observability based on eBPF is of great significance for the iterative development of new technologies. Innovation in various industries is addressing pain points in business, and people will accelerate the adoption of innovation after seeing the benefits. Zero Code observability is a strong guarantee for the speed of innovation.

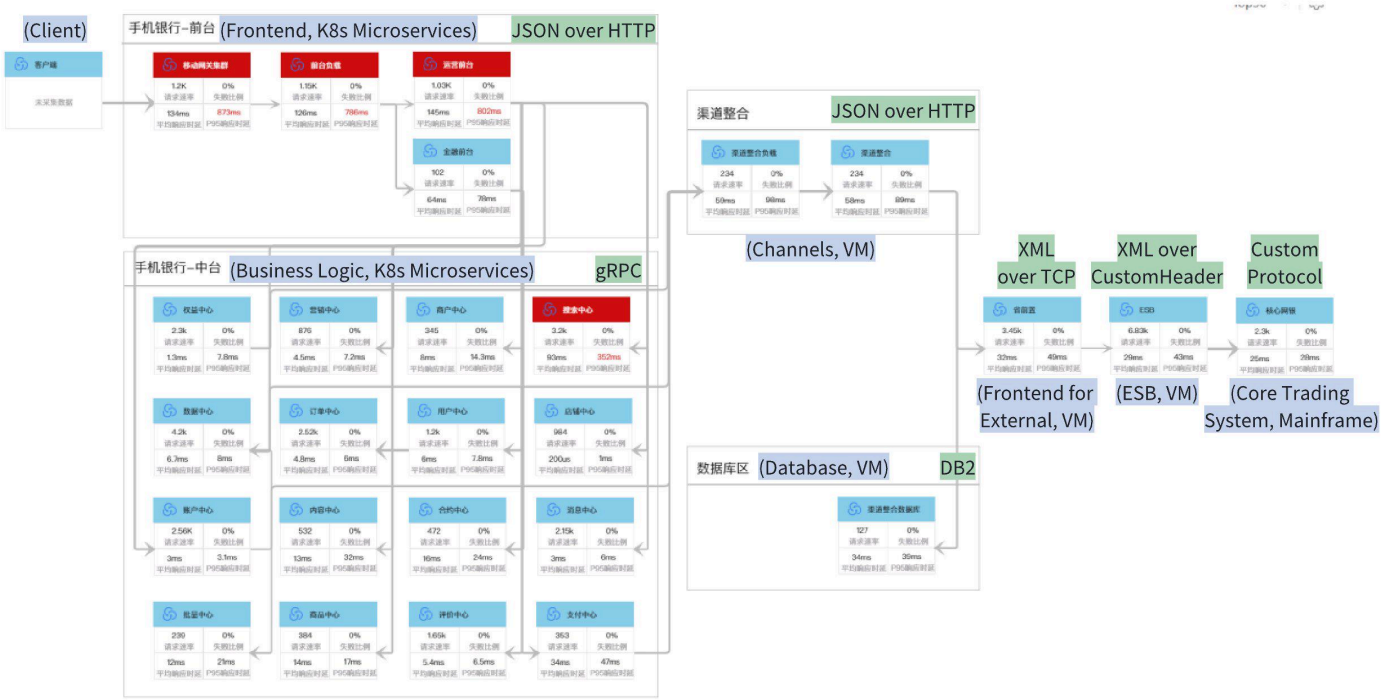
Continuous innovation of cloud-native infrastructure: Take the gateway as an example. The number of gateways for microservice access in a cloud-native environment may surprise you. The cartoon below vividly illustrates this situation. These gateways are solving practical problems encountered in business operations. Load balancers avoid single points of failure. API gateways ensure the security of exposed APIs. Microservice gateways allow the front-end of a business system to conveniently access any microservice in the backend. Service Mesh provides rate limiting, circuit breaking, and routing capabilities, reducing repetitive work in business development. Although different gateways may have overlapping capabilities, this is an inevitable intermediate state in the process of technological development. Furthermore,

different gateways are often managed by different teams, and the management personnel usually do not have the ability for secondary development. If Zero Code observability of gateways cannot be achieved, it will have disastrous consequences for troubleshooting.



Various gateways microservices access to, from theburningmonk@twitter

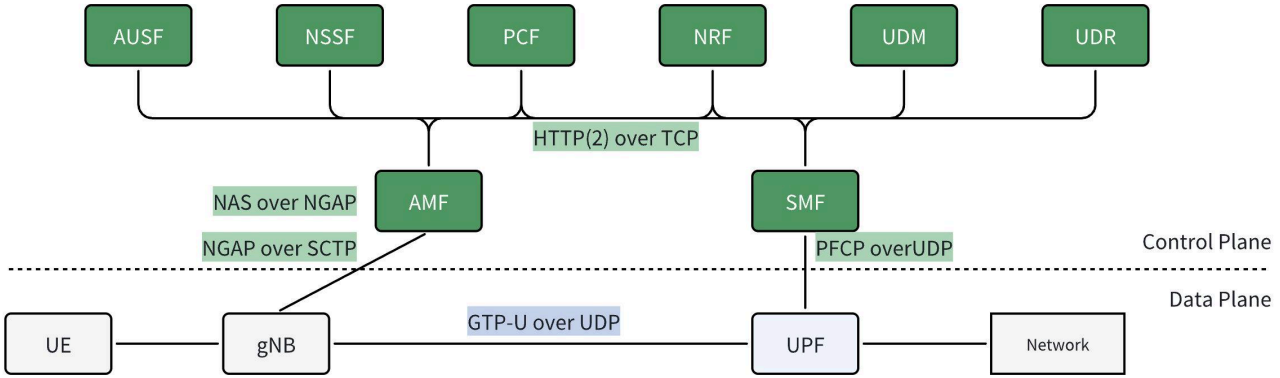
Distributed transformation of financial core trading systems: In the past, the core trading systems in the financial industry were supported by specialized hardware, making it difficult to scale and iterate at an expensive cost. DeepFlow's banking, securities, and insurance customers have embarked on the distributed transformation of their core trading systems in the past two years. These systems are crucial to the country's economy and people's livelihoods, and the Zero Code observability is a prerequisite for ensuring the smooth launch of such systems.



Service topology of a mobile bank service

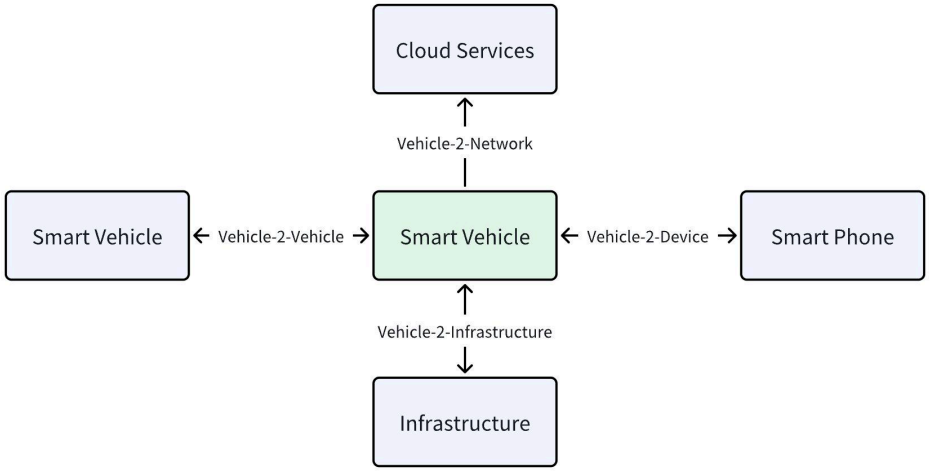
Telecommunication core network service-oriented architecture transformation: Similar to the financial industry, the telecommunication core network used to be supported by dedicated hardware. However, starting from the 5G core network, the 3GPP has clearly proposed the Service-Based Architecture (SBA) specification, where core network functions have been decoupled into a series of microservices running in a K8s container environment. Similarly, Zero

Code observability is also a prerequisite for ensuring the smooth deployment of telecommunication core business systems.



5G core network elements and their communication protocols, each control plane element adopts SBA architecture

Development of network connected smart vehicle: The intelligent automotive network consists of a central cloud, edge cloud (factory/park), and terminal (in-vehicle system). In order to provide users with continuously updated software experiences, all services within the entire intelligent automotive network adopt a microservices architecture and cloud-native deployment. An infrastructure with observability is also a prerequisite for the continuous iteration of this extensive network.



Network connected smart vehicle

The Importance to AIOps Development: In the past, before the implementation of AIOps solutions, observation data (usually metrics and logs) needed to be centralized and cleansed. This was a lengthy process that often took several months to complete. eBPF has the potential to fundamentally change this situation. Due to the data collected by eBPF covering all services, having highly consistent label information and data formats, it will greatly reduce the hurdles to implementing AIOps solutions.

0x7: Conclusion

APM Agents, due to their intrusive nature, are difficult to deploy in core business systems in industries such as finance, telecommunications and power, and are hard to instrument in cloud-native infrastructure. eBPF's Zero Code advantages effectively address these pain points, making it a key technology for achieving observability in the cloud-native era. DeepFlow, based on eBPF, has already served various industries with its Universal Map, Distributed Tracing, and Continuous Profiling capabilities, enabling rapid implementation of Zero Code observability in places such as the distributed core trading systems in the finance industry, the 5G core network

in the telecommunications industry, the distributed power trading systems in the energy industry, intelligent connected vehicles, and cloud-native gaming services. This ensures the continuous innovation of the new generation of businesses and infrastructure.

Related Posts

DeepFlow 基于 eBPF 的高度自动化可观测性实践

基于 eBPF 的云原生可观测性深度实践

中国移动磐基PaaS平台基于eBPF的应用可观测性建设实践

微拍堂基于 DeepFlow 建设零侵扰的可观测平台

开启高度自动化的可观测性新时代