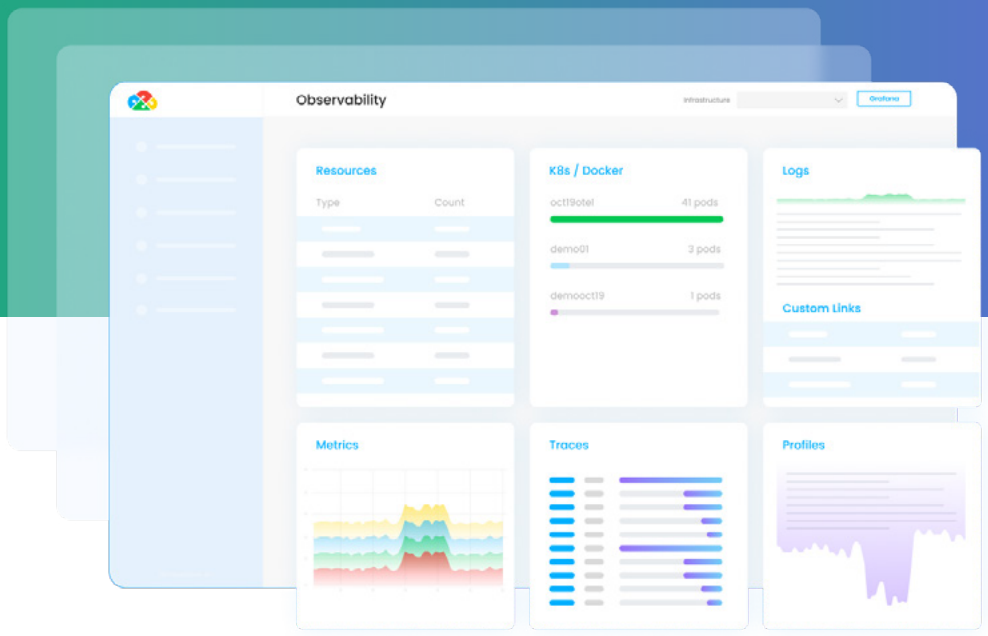
DuploCloud

WHITEPAPER

DuploCloud Observability



Contents

Introduction

Observability is a crucial aspect of any application or system. When done well, it becomes a mindset and culture requiring ongoing attention and dedication to provide valuable insights into the state of the digital footprint. This is especially important for understanding the context around errors and outages during critical moments when time is limited and focus is strained.

Observability can mean the difference between a happy retrospective and a painful outage. Recent advancements in open-source observability platforms have significantly increased the capability to tap into those valuable insights. Knowing the state of your application means knowing where to focus development efforts and being able to identify root causes in real time.

Definition

Succinctly put, in computer science theory, observability is "a measure of how well internal states of a system can be inferred from knowledge of its external outputs".[3] Practically speaking, observability is the proactive approach of designing and implementing visibility into an application to gain insights into its performance, issues, and failures, to know which key indicators matter most, and where to focus development efforts.

Observability is, by nature, a proactive approach to improving a system.[1]

Observability consists of collecting metrics, logs, traces, profiles, telemetry and other signals relevant to the team's KPIs, along with a strategy of who to alert for critical occurrences in the system.

Target Audience

Our audience for this whitepaper includes:

- **Business administrators**
- **Site Reliability Engineers**
- **DevOps Engineers**
- **MLOps Engineers**
- **Platform Engineers**
- **Cloud Engineers**
- **Infrastructure Engineers**
- **Software Developers**
- **Software Architects**
- **Product Owners/ Managers**

Observability, in reality, is about implementing and improving a culture of observability, which impacts both business and technical aspects of a software stack. It requires an ongoing effort to improve application and observability architecture continually, and it is most effective when there is buy-in from all stakeholders. It's a family affair.

Purpose

Observability is a rapidly advancing field with many brilliant open-source tools to choose from and integrate. Organizations at all levels struggle with different aspects of observability. It can be hard to justify spending time on it when the application code and infrastructure need attention and improvement.

Even more challenging is having the necessary experience to determine which observability tools to choose, especially when factors like development time and high budget requirements come into play.

This whitepaper offers a framework to create a team vision for adopting observability and a roadmap through the ambiguity often faced when implementing an observability model.

Signals

To start, we need to take a look at signals. What is a signal?

A signal is simply an output from the system or application.

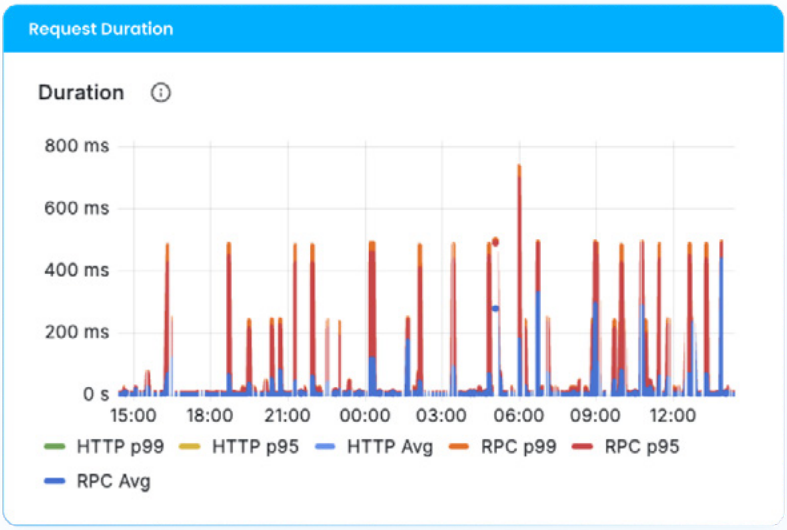
Metrics, traces, and logs are the three main signals, and while they are often referred to as the foundational signals or the three pillars of observability, they are not necessarily all required at once. Observability is as much an art as it is a cost-budget reality, and depending on the needs of your teams, they may not all be needed all the time. This is explored more in-depth by the CNCF's whitepaper[2] and is well worth the read.

The three primary signals are metrics, tracing, and logging.

Metrics	Numeric data or numeric representations of non-numeric data. Examples are the percentage of CPU usage or the total count of requests to a particular service. Here are a few examples of metrics that are useful for our team.

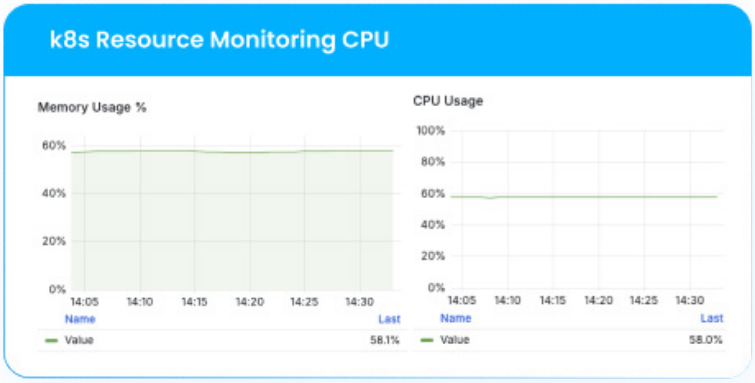
This shows statistical information about the duration of requests sent (via HTTP) to our website's server:

It's useful for showing the statistical outliers of requests that take longer than the rest, which signifies a delay in loading for the visitor. In this graphic, the HTTP average is only a few milliseconds, which is great, but some RPCs are between 200 ms and about 500 ms, which can add up against other delays throughout the system and cause issues for users, which we'll see better in traces.



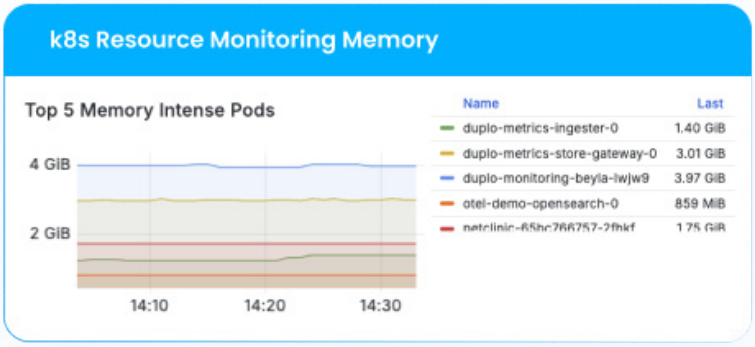
This example shows memory (RAM) and CPU usage.

This is useful for seeing the application's capacity to accept more workload. Those values are on the higher side, but there's still a lot of wiggle room for Kubernetes to scale out, and it likely will soon.

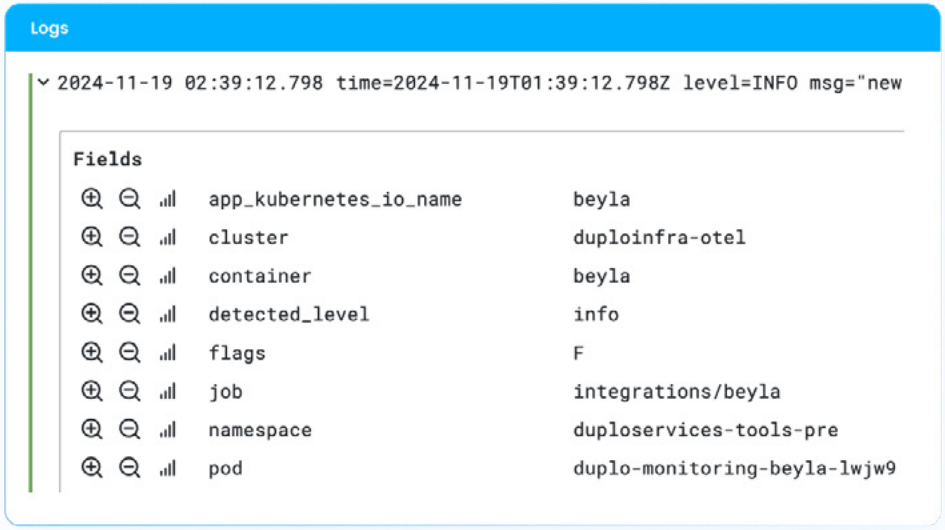


This one is related, showing the Pods deployed in Kubernetes that are hungriest for resources.

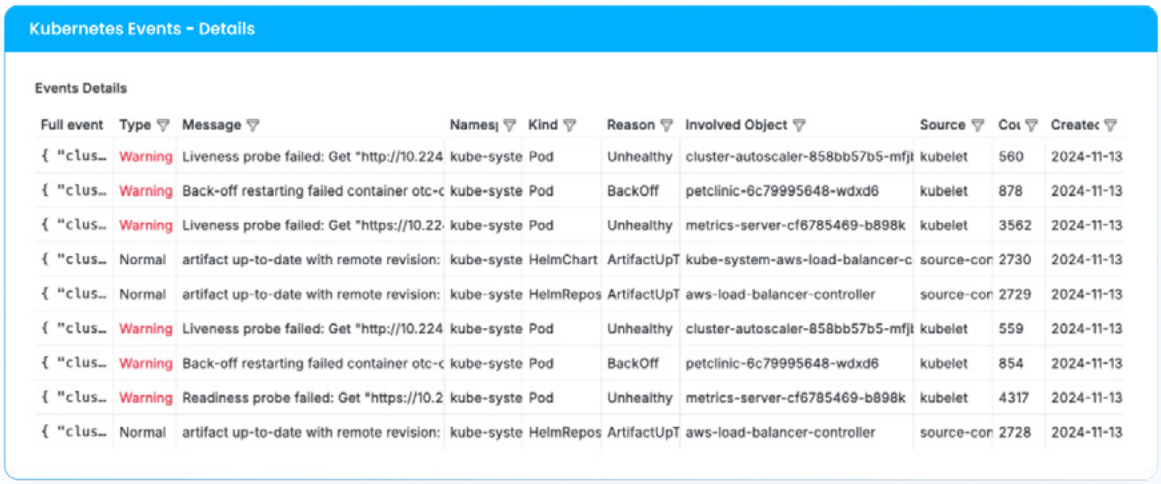
If these Pods are *supposed* to be gobbling up memory, then everything is fine, but if a pod that's supposed to be using just a few MiB of memory is topping the list, you'll know something is going on with it, and it'll need a set of eyes.



Logs Text entries from applications or systems consisting of data developers thought relevant and showing a step-by-step record of what happened leading up to a result. There are several levels of log verbosity ranging from `debug,` useful during development, to `critical,` which is helpful for production applications. Log verbosity is typically turned down as applications are deployed live to reduce noise, cost, and compute time.



This dashboard shows Kubernetes events and the first section of this dashboard shows the logs resulting from each event.



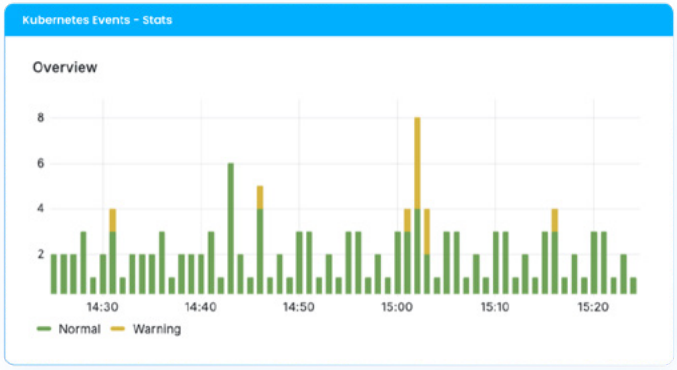
Full event	Type	Message	Namesj	Kind	Reason	Involved Object	Source	Cot	Createc
{ "clus…	Warning	Liveness probe failed: Get "http://10.224	kube-syste	Pod	Unhealthy	cluster-autoscaler-858bb57b5-mfjt	kubelet	560	2024-11-13
{ "clus…	Warning	Back-off restarting failed container otc-c	kube-syste	Pod	BackOff	petclinic-6c79995648-wdxd6	kubelet	878	2024-11-13
{ "clus…	Warning	Liveness probe failed: Get "https://10.22	kube-syste	Pod	Unhealthy	metrics-server-cf6785469-b898k	kubelet	3562	2024-11-13
{ "clus…	Normal	artifact up-to-date with remote revision:	kube-syste	HelmChart	ArtifactUpT	kube-system-aws-load-balancer-c	source-con	2730	2024-11-13
{ "clus…	Normal	artifact up-to-date with remote revision:	kube-syste	HelmRepos	ArtifactUpT	aws-load-balancer-controller	source-con	2729	2024-11-13
{ "clus…	Warning	Liveness probe failed: Get "http://10.224	kube-syste	Pod	Unhealthy	cluster-autoscaler-858bb57b5-mfjt	kubelet	559	2024-11-13
{ "clus…	Warning	Back-off restarting failed container otc-c	kube-syste	Pod	BackOff	petclinic-6c79995648-wdxd6	kubelet	854	2024-11-13
{ "clus…	Warning	Readiness probe failed: Get "https://10.2	kube-syste	Pod	Unhealthy	metrics-server-cf6785469-b898k	kubelet	4317	2024-11-13
{ "clus…	Normal	artifact up-to-date with remote revision:	kube-syste	HelmRepos	ArtifactUpT	aws-load-balancer-controller	source-con	2728	2024-11-13

An unhealthy Pod is something the Kubernetes scheduler will banish and replace. In contrast, one stating BackOff might need more immediate attention, although both are looking into what in the application is causing the underlying issue in the first place. ArticaftUpToDate is a friendly reporting-for-duty type notification that may or may not be useful.

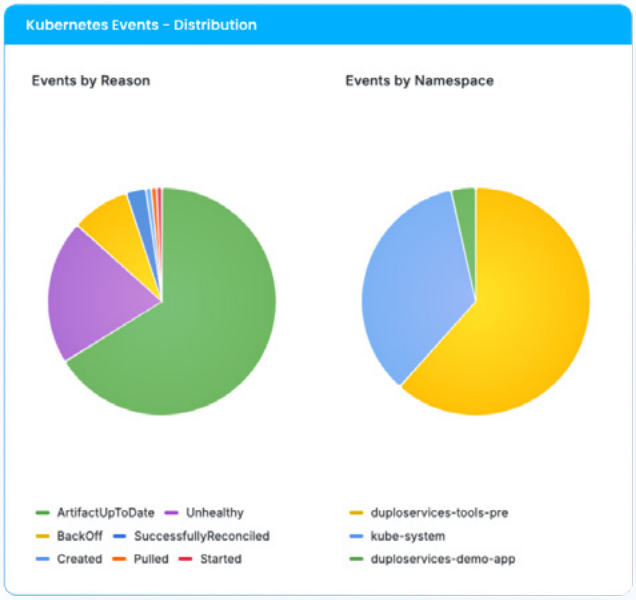
Logs are most commonly sent around in JSON format, and while they may look a little scary, it's just a format that humans and machines can read. See this example of a pod currently in BackOff:


```
{
 "cluster": "duploinfra-otel",
 "count": "901",
 "eventRV": "17086192",
 "instance": "loki.source.kubernetes_events.cluster_events",
 "job": "integrations/kubernetes/eventhandler",
 "kind": "Pod",
 "msg": "Back-off restarting failed container otc-container ir
 "name": "petclinic-6c79995648-wdxd6",
 "namespace": "kube-system",
 "objectAPIversion": "v1",
 "objectRV": "17024478",
 "reason": "BackOff",
 "reportingcontroller": "kubelet",
 "reportinginstance": "ip-10-224-15-136.us-west-2.compute.inte
 "service_name": "integrations/kubernetes/eventhandler",
 "sourcecomponent": "kubelet",
 "sourcehost": "ip-10-224-15-136.us-west-2.compute.internal",
 "type": "Warning"
}
```

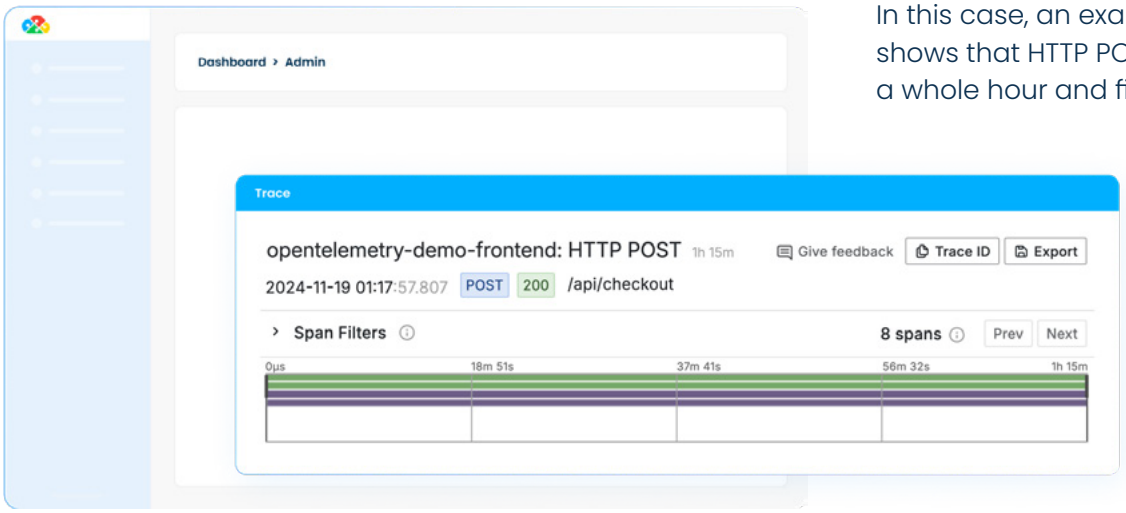
This dashboard also has some metrics describing logs, such as this graph indicating log event levels (in this case, warning) are the highest/most urgent level.



A pie chart breaking down events by reason and by namespace:



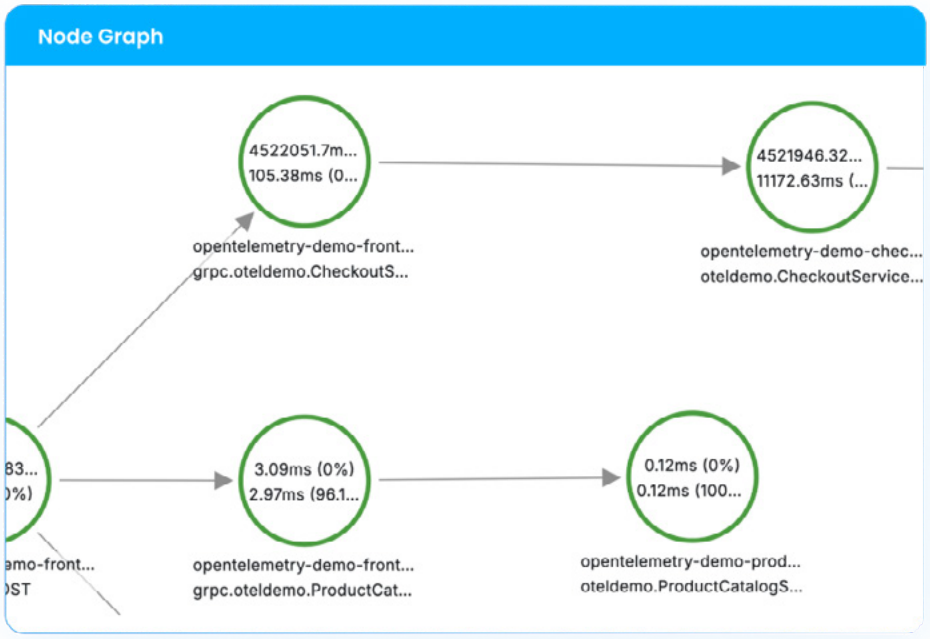
Traces show what happens downstream in a system due to an event. Traces are useful for knowing how long each segment or call takes to complete and getting a full view of any related errors or log entries.



In this case, an example of a trace shows that HTTP POST requests take a whole hour and fifteen minutes.

The traces dashboard also includes a handy node graph to see how events are related.

As a bonus, there is a fourth main signal, `profiles,` that is becoming more and more relevant to cloud-native workloads.



Profiles	Show the *why* behind the traces, linking the system calls the code is making relative to each trace. This makes it amazingly easy to drill into issues.

To recap, metrics show you how the system is behaving as a whole; the logs show the actions the system is taking; the traces show how long each of those actions takes; profiles link those lengths of time through the code down to what the system is doing underneath it all as a result of the calls the code makes.

Modern observability is a wonder.

Prometheus vs OpenTelemetry

It's worth pointing out that Prometheus is an end-to-end monitoring tool that collects metrics and serves as a backend to transfer, store, and query them.

OpenTelemetry is simply a signals and telemetry collector and typically uses Prometheus as its backend for distribution and storage.

OpenTelemetry supports traces and profiles and offers a more standardized approach to metrics, such as delta representation, which measures the change in value over time, with more information about the application and its underlying systems.

eBPF

eBPF is a promising new technology that provides a way to create helper functions that interact with the Linux kernel and probe it for information without modifying the actual kernel, crossing namespace boundaries, or installing kernel modules. Helper functions work in concert with the kernel, taking advantage of existing kernel functionality and tooling, and as a result, are very lightweight and safe as they always terminate and use an in-kernel verifier to ensure crashing the kernel isn't possible. Likewise, functions are limited in length and cannot loop, ensuring they won't use precious system resources.

eBPF functions are always event-driven and excellent for gathering tracing information and metrics.[4]

A list of eBPF providers:

- **Pixie**

  An open-source observability tool for Kubernetes applications. Pixie uses eBPF to automatically capture telemetry data without the need for manual instrumentation.

- **Cilium**

  An open source, cloud-native solution for providing, securing, and observing network connectivity between workloads, fueled by the revolutionary Kernel technology eBPF.

- **Beyla**

  An OSS project that enables automatic instrumentation for HTTP/gRPC applications written in Go, C/C++, Rust, Python, Ruby, Java (including GraalVM Native), NodeJS, .NET, and more. It's based on eBPF, which allows you to attach your programs to different points of the Linux kernel.

- **OpenTelemetry eBPF Collector**

  The OpenTelemetry eBPF project develops components that collect and analyze telemetry from the operating system, cloud, and container orchestrators. Its initial focus is on collecting network data to enable users to gain insight into their distributed applications.

DuploCloud has evaluated these and various eBPF tools and leverages Beyla and OpenTelemetry's eBPF capabilities for its advanced tracing, metrics, and other insights into applications and distributed systems to offer a mature observability stack.

eBPF implemented well is likewise impactful for security and compliance due to the wealth of information it provides. Liz Rice, an eBPF specialist and member of the governing board at CNCF, states, "The more contextual information that's available to the investigator, the more likely they will be able to find out the root cause of the event and determine whether it was an attack, which components were affected, how and when the attack took place and who was responsible."[6]

eBPF functions are run through a verifier at the kernel level, preventing loops, crashes, and out-of-bounds access so that no information from the kernel leaks into user space.

As eBPF functions are only run upon events, the attack surface is limited, and functions run inside a sandbox. This event-driven architecture opens up worlds of possibilities to ensure certain actions by the kernel are further limited and observed. Any actions of a specific type can include eBPF functions to verify or impose limitations, such as network traffic to isolate certain components from others based on network packet details. Cilium is a potent tool in this regard.

eBPF is relatively new in the observability world, and DuploCloud is proud to be an early adopter of such technology. To illustrate the capabilities and creative applications of eBPF, [Pixie]() currently has a dynamic logging functionality[5] in Alpha that uses eBPF to generate structured logs of a running application without redeploying application code, something that is truly groundbreaking. Dynamic logging allows for the capturing of function arguments, function return values, and latency with little overhead and without stopping the execution to redeploy.

Observability Maturity Model

With so many capable tools to choose from on the market, knowing which to focus on and when is an increasingly complex decision. This is where the observability maturity model comes into play. Created by AWS, it provides a framework to orient an organization to a mature, comprehensive observability stack.

It does this by breaking down observability into four stages, starting with the most important aspects and building from there to end up with potent insights into a digital footprint.

The observability maturity model enables startups to trailblaze a complete roadmap of current observability capabilities and future needs. An organization can develop a way to increase the resiliency and reliability of their applications. An observability maturity assessment is well worth the time and is a crucial building block of a robust digital footprint.

Observability done well reduces cognitive load, increases awareness, prevents downtime, and minimizes response time, along with providing detailed information for root cause analysis.

Foundational Monitoring	Collecting Telemetry and Data – Siloed data with no cohesive strategy to monitor the various systems owned by the organization, typically divided by team. Identify critical workloads, define metrics, logs, and KPIs, and aim to observe them together. Implement telemetry as a core part of the application design to understand its health and state.
Intermediate Monitoring	Telemetry Analysis and Insights – Signals from metrics, logs, and traces are established with visualizations and alerts in dashboards, giving teams the insight needed to prioritize issues. This puts teams in the driver's seat instead of being reactive. However, outages and issues often overwhelm teams, requiring excess cognitive effort and time to debug and solve. The remedy is to identify KPIs, create policies such as disaster recovery, and review them regularly along with application architecture. This helps reduce MTTR (Mean Time to Resolution).
Advanced Observability	Correlation and Anomaly Detection – Context is immediately apparent through a maturing set of metrics, traces, and logs. Traces include relevant information from services external to the application, such as managed services from the cloud provider. Observability KPIs such as MTTR and SLOs (Service Level Objectives) are low. This enables organizations to increase their applications' SLAs (Service Level Availability) while increasing the complexity of system architectures. Anomaly detectors are used to alert based on outliers that don't match usual patterns. At this level, AI can correlate signals, perform RCAs, and suggest resolutions using custom machine learning models based on past collected data, a process now described as AIOps.
Proactive Observability	Automatic and Proactive Root Cause Identification – Observability data is used in real-time with GenAI to provide relevant insights into issues, and options to resolve, all displayed in dynamic dashboards with information relevant to the matter at hand, saving precious time and reducing costs associated with querying and visualizing data that is irrelevant.

Conclusion

DuploCloud can help you perform an observability maturity assessment to identify areas of focus and create a plan to overcome deficiencies in your system visibility, identify issues more quickly, and pour a stronger foundation to improve your applications, teams, and critical business offerings. An observability strategy is key to your application's growth, and DuploCloud has a team of observability gurus ready to draw a roadmap and begin trailblazing.

Sources

1. AWS Observability Maturity Model
2. CNCF TAG Observability Whitepaper
3. Kalman R. E., On the General Theory of Control Systems, Proc. 1st Int. Cong. of IFAC, Moscow 1960 1481, Butterworth, London 1961, as quoted in the CNCF TAG Observability Whitepaper.[2]
4. Cilium, eBPF Architecture
5. Pixie, Dynamic Logging
6. Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security