**Linux Foundation**

The [Linux Foundation](https://www.linuxfoundation.org/) is the world’s leading home for collaboration on open source software, hardware, standards, and data. Linux Foundation projects are critical to the world’s infrastructure, including Linux, Kubernetes, Node.js, ONAP, PyTorch, RISC-V, SPDX, OpenChain, and more. The Linux Foundation focuses on leveraging best practices and addressing the needs of contributors, users, and solution providers to create sustainable models for open collaboration. The Linux Foundation has registered trademarks and uses trademarks. For a list of trademarks of The Linux Foundation, please see its [trademark usage page](http://www.linuxfoundation.org/trademark-usage). Linux is a registered trademark of Linus Torvalds.

**Cloud Native Computing Foundation (CNCF)**

[Cloud Native Computing Foundation (CNCF)](https://www.cncf.io/) is an open source software foundation under the Linux Foundation umbrella dedicated to making cloud native computing universal and sustainable. Cloud native computing uses an open source software stack to deploy applications as microservices, packaging each part into its own container, and dynamically orchestrating those containers to optimize resource utilization. Cloud native technologies enable software developers to build great products faster.

CNCF serves as a vendor-neutral home for many of the fastest-growing projects on GitHub, including Kubernetes, Prometheus, and Envoy, fostering collaboration between the industry's top developers, end users and vendors.

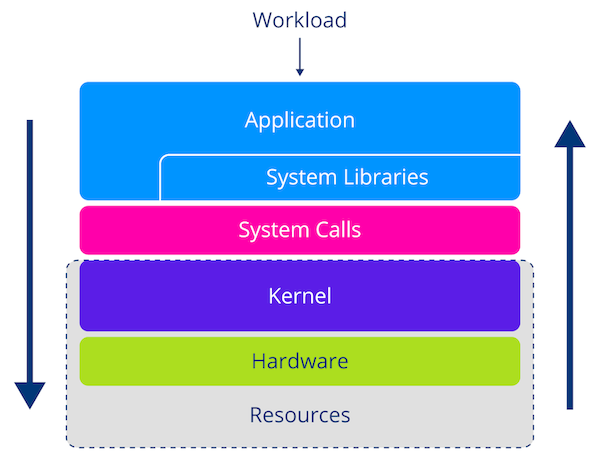
**Current State of Observability**

According to Wikipedia, "[observability](https://en.wikipedia.org/wiki/Observability) is a measure of how well the internal states of a system can be inferred from knowledge of its external outputs" (2024). In other words, observability refers to how easily you can understand what's happening inside a system, like an application or a service, by looking at the information it produces.

A distributed system is a network of independent computers, or nodes, working together to perform tasks as if they were a single system. These systems are widely used in applications like cloud computing, where different parts of an application run on different servers to share resources and balance the workload. Because of the complexity of distributed systems, it can be challenging to understand what's happening inside each component at any given time. This is where **observability** becomes crucial.

To make a distributed system observable, we must model its state in a way that lets us reason about its behavior. This model is composed of three factors:

1. First, there is the **workload**. These are the operations a system performs to fulfill its objectives. For instance, when a user sends a request, a distributed system often breaks it down into smaller tasks handled by different services. This is also often referred to as transactions.
2. Second, there are **software abstractions** that make up the structure of the distributed system. This includes elements such as load balancers, services, pods, containers and more.
3. Lastly, there are physical machines that provide computational **resources** (e.g. RAM, CPU, disk space, network) to carry out work.



**Workload and Resource Analysis**

Depending on our background, we often have a certain bias when investigating the performance of or troubleshooting problems in a distributed system. Application developers typically concentrate on workload-related aspects, whereas operations teams tend to look at physical resources.

To truly understand a system, we must combine insights from multiple angles and figure out how they relate to one another. However, before we can analyze something, we must first capture aspects of system behavior. This is commonly being done through a combination of *logs*, *metrics*, and *traces*.

**Logs**

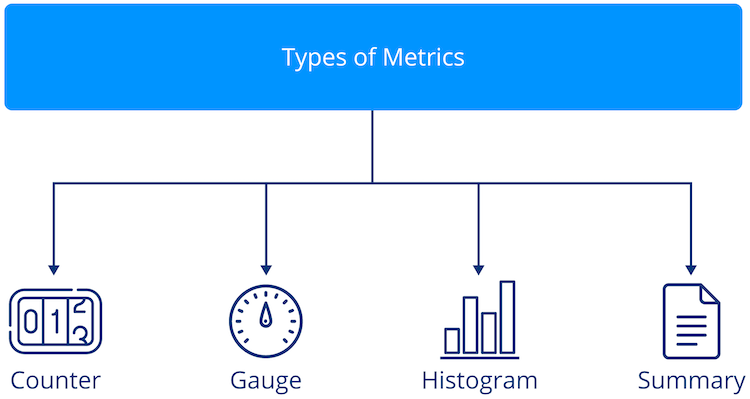


**Exemplary Log Files**

A**log**is an append-only data structure that records events occurring in a system. A log entry consists of a timestamp that denotes when something happened and a message to describe details about the event. However, coming up with a standardized log format is no easy task. One reason is that different types of software often convey different pieces of information. The logs of an HTTP web server are bound to look different from those of the kernel. But even for similar software, people often have different opinions on what good logs should look like.

Apart from content, log formats also vary with their consumers. Initially, text-based formats catered to human readability. However, as software systems became more complex, the volume of logs soon became unmanageable. To combat this, we started encoding events as key/value pairs to make them machine-readable, which is commonly known as structured logging. Moreover, the distribution and ephemeral nature of containerized applications meant that it was no longer feasible to log onto individual machines and sift through logs. As a result, people started to build logging agents and protocols to forward logs to dedicated services. These logging systems allowed for efficient storage as well as the ability to search and filter logs in a central location.

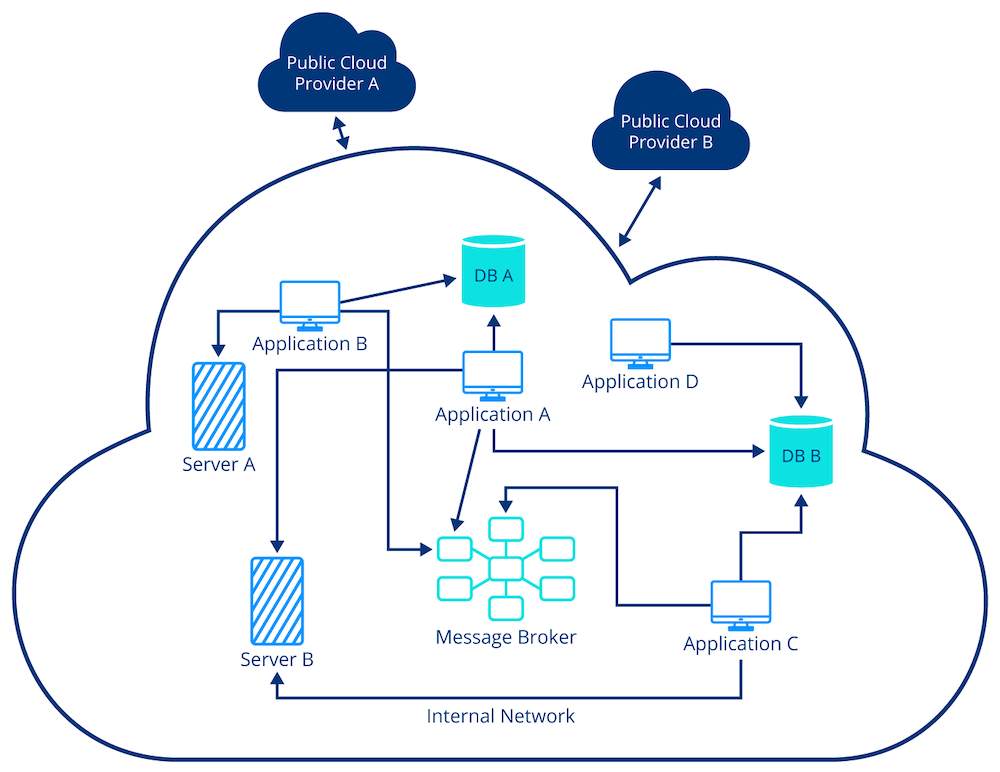
**Metrics**



**The Four Common Types of Metrics: Counters, Gauges, Histograms, and Summaries**

Logs shine at providing detailed information about individual events. However, sometimes we need a high-level view of the current state of a system. This is where metrics come in. A metric is a single numerical value derived by applying a statistical measure to a group of events. In other words, metrics represent an aggregate. This is useful because their compact representation allows us to graph how a system changes over time. In response, the industry developed instruments to extract metrics, formats and protocols to represent and transmit data, specialized time-series databases to store them, and frontends to make this data accessible to end-users.

**Traces**



**Exemplary Architecture of a Distributed System**

As distributed systems grew in scale, it became clear that traditional logging systems often fell short when trying to debug complex problems. The reason is that we often have to understand the chain of events in a system. On a single machine, stack traces allow us to track an exception back to a line of code. In a distributed environment, we don’t have this luxury. Instead, we perform extensive filtering to locate log events of interest. To understand the larger context, we must identify other related events, such as the specific requests or transactions that initiated the log entry and the sequence of services or microservices involved in processing that request across the system. This often results in a lot of manual labor (e.g. comparing timestamps) or requires extensive domain knowledge about the applications. Recognizing this problem, Google developed [Dapper](https://research.google.com/pubs/pub36356.html?spm=5176.100239.blogcont60165.11.OXME9Z), which popularized the concept of *distributed tracing*.

On a fundamental level, tracing is logging on steroids. The underlying idea is to add transactional context to logs. This makes it possible to infer causality and reconstruct the journey of requests in the system.

**Three Pillars of Observability**

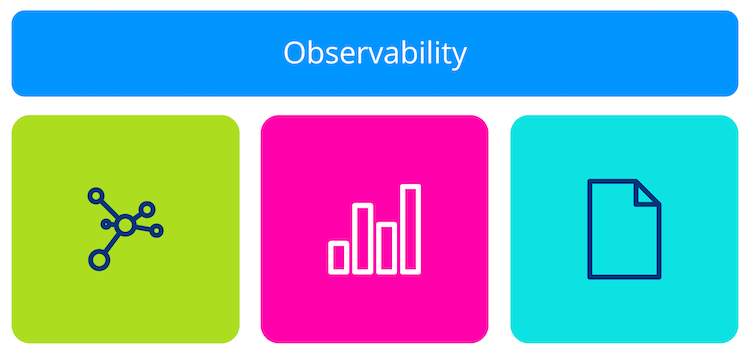
**Telemetry** is the process of automatically collecting and transmitting data from remote or distributed systems to monitor, measure, and track the performance or status of those systems. Telemetry data provides real-time insights into how different parts of an application are performing. Telemetry provides the data for observability tooling to help developers and system administrators observe, troubleshoot, and optimize the system without needing to manually check each component.

On the surface, logs, metrics, and traces share many similarities in their lifecycle and components. Everything starts with instrumentation that captures and emits data. The data has to have a specific structure defined by a format. Then, we need a mechanism to collect and forward a piece of [telemetry](https://en.wikipedia.org/wiki/Telemetry). Often, there is some kind of agent to further enrich, process, and batch data before ingesting it in a backend. This process typically involves a database to efficiently store, index, and search large volumes of data. Finally, there is a frontend analysis to make the data accessible to the end-user. However, in practice, we develop dedicated systems for each type of telemetry, and for good reason: Each telemetry signal poses its own unique technical challenge. These challenges are mainly due to the different natures of the data.

The design of data models, interchange formats, and transmission protocols, highly depends on whether you are dealing with unstructured or semi-structured textual information, compact numerical values inside a time series, or graph-like structures depicting causality between events. Even for a single signal, there is no consensus on these topics.

Furthermore, how we work with and derive insights from telemetry varies dramatically. A system might need to perform full-text searches, inspect single events, analyze historical trends, visualize request flow, diagnose performance bottlenecks, and more. These requirements manifest themselves in the design and optimization of storage, access patterns, query capabilities, and more.

When addressing these technical challenges, [vertical integration](https://en.wikipedia.org/wiki/Vertical_integration) emerges as a pragmatic solution. In practice, observability vendors narrow the scope of the problem to a single signal and provide instrumentation to generate and tools to analyze telemetry, as a single, fully integrated solution.



**The Three Pillars of Observability: Metrics, Traces and Logs**

Having dedicated systems for logs, metrics, and traces is why we commonly refer to them as the three pillars of observability. The notion of pillars provides a great mental framework because it emphasizes the following:

* There are different categories of telemetry
* Each pillar has its own unique strengths and stands on its own
* Pillars are complementary and must be combined to form a stable foundation for achieving observability

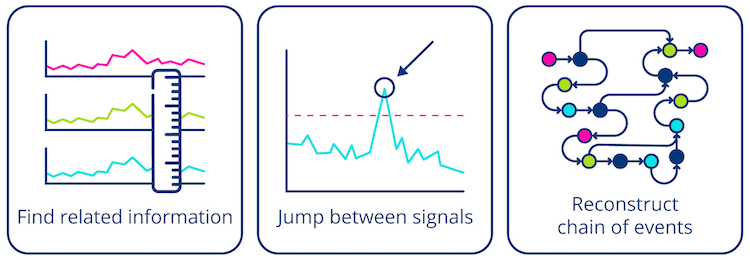
**Problems with Our Approach to Observability**

With loads of open source and commercial observability solutions on the market, you might (rightly) ask yourself:

* If there are plenty of mature solutions for generating, collecting, storing, and analyzing logs, metrics and traces, why should I care?
* What’s wrong with the current state of observability?
* Oh, great … is this yet another attempt at standardization?

These are valid questions. To answer them, we must identify (some) downsides that result from building and working with pillar-based observability systems.

**Siloed Telemetry is Difficult to Work With**



**The Need for Correlated Telemetry**  
Image inspired by [[Young, 2021]](https://www.oreilly.com/library/view/the-future-of/9781098118433/)

First, there are deficits in the quality of telemetry data. To illustrate this, let’s imagine that we want to investigate the root cause of a problem. The first indicator of a problem is usually an alert or an anomaly in a metrics dashboard. To confirm the incident is worth investigating, we have to form an initial hypothesis. The only information we currently have is that something happened at a particular point in time. Therefore, the first step is to use the metrics system to look for other metrics showing temporally correlated, abnormal behavior.

After making an educated guess about the problem, we want to drill down and investigate the root cause of the problem. To gain additional information, we typically switch to the logging system. Here, we write queries and perform extensive filtering to find log events related to suspicious metrics.

After discovering log events of interest, we often want to know about the larger context in which the operation took place. Unfortunately, traditional logging systems lack the mechanisms to reconstruct the chain of events in that particular transaction. Traditional logging systems often fail to capture the full context of an operation, making it difficult to correlate events across different services or components. They frequently lack the ability to preserve critical metadata, such as trace IDs or span IDs, which are essential for linking related events together. This limitation results in fragmented views of the system’s behavior, where the story of a single operation is spread across multiple logs without a clear narrative. Furthermore, the absence of standardized query languages or interfaces adds to the difficulty of searching and analyzing logs effectively, as operators must rely on custom scripts or manual filtering to uncover patterns and anomalies.

If we switch our perspectives from someone building an observability solution to someone using it, an inherent disconnect is revealed. The real world isn’t made up of logging, metrics, or tracing problems. Instead, we have to move back and forth between different types of telemetry to build up a mental model and reason about the behavior of a system. Since observability tools are silos of disconnected data, figuring out how pieces of information relate to one another causes a significant cognitive load for the operator.

**Lack of Instrumentation Standard Leads to Low Quality Data**

Another factor that makes root-cause analysis hard is that telemetry data often suffers from a lack of consistency. This leads to difficulties in correlating events across different services or components, as there is no standardized way to identify related events, such as through trace IDs or span IDs. Additionally, there is no straightforward method to integrate multiple solution-specific logging libraries into a coherent system, resulting in fragmented and disjointed views of the system’s behavior.

**No Built-in Instrumentation in Open Source Software**

Let’s look at this from the perspective of open source software developers. Today, most applications are built on top of open source libraries, frameworks, and standalone components. With a majority of work being performed outside the business logic of the application developer, it is crucial to collect telemetry from open source components. The people with the most knowledge of what is important when operating a piece of software are the developers and maintainers themselves. However, there is currently no good way to communicate through native instrumentation.

One option would be to pick the instrumentation of an observability solution. However, this would add additional dependencies to the project and force users to integrate it into their systems. While running multiple logging and metrics systems is impractical but technically possible, tracing is outright impossible as it requires everyone to agree on a standard for trace context propagation to work.

A common strategy for solving problems in computer science is to add a layer of indirection. Instead of embedding vendor-specific instrumentation, open source developers often provide observability hooks. This allows users to write adapters that connect the open source component to their observability system. While this approach provides greater flexibility, it also has its fair share of problems. For example, whenever there is a new version of software, users have to notice and update their adapters. Moreover, the indirection also increases the overhead, as we have to convert between different telemetry formats.

**Combining Telemetry Generation with Results in Vendor Lock-in**

Let’s put on the hat of an end user. After committing to a solution, the application contains many solution-specific library calls throughout its codebase. To switch to another observability tool down the line, we would have to rip out and replace all existing instrumentation and migrate our analysis tooling. This upfront cost of re-instrumentation makes migration difficult, which is a form of vendor lock-in.

**Struggling Observability Vendors / High Barrier for Entry**

The last part of the equation is the observability vendors themselves. At first glance, vendors appear to be the only ones profiting from the current situation. In the past, high-quality instrumentation was a great way to differentiate yourself from the competition. Moreover, since developing integrations for loads of pre-existing software is expensive, the observability market has a relatively high barrier to entry.

With customers shying away from expensive re-instrumentation, established vendors have faced less competition and pressure to innovate. However, they are also experiencing major pain points. The rate at which software is being developed has increased exponentially over the last decade. Today’s heterogeneous software landscape makes it impossible to maintain instrumentation for every library, framework, and component. As soon as a vendor starts struggling with supplying instrumentation, customers will start refusing to adopt their product. As a result, solutions compete over who can build the best n-to-n format converter instead of investing these resources into creating great analysis tools. Another downside is that converting data that was generated by foreign sources often leads to a degradation in the quality of telemetry. Once data is no longer well-defined, it becomes harder to analyze.

**What is OpenTelemetry (in a nutshell)?**

OpenTelemetry (OTel) is an open source project designed to provide standardized tools and APIs for generating, collecting, and exporting telemetry data such as traces, metrics, and logs. It aims to give developers deep visibility into applications, helping to monitor, troubleshoot, and optimize software systems.

The main goals of OpenTelemtry are:

* **Unified telemetry**: Combines tracing, logging, and metrics into a single framework enabling correlation of all data and establishing an open standard for telemetry data.
* **Vendor-neutrality**: Integration with different backends for processing the data.
* **Cross-platform**: Supports various languages (Java, Python, Go, etc.) and platforms, making it versatile for different development environments.

**Why is OpenTelemetry Promising?**

At the time of writing (May 2024), OpenTelemetry is the [second fastest-growing project within the CNCF](https://www.cncf.io/reports/cncf-annual-report-2023/#projects). OpenTelemetry receives so much attention because it promises to be a fundamental shift in the way we produce telemetry. It’s important to remember that observability is a fairly young discipline. In the past, the rate of innovation and conflicts of interest prevented us from defining widely adopted standards for telemetry. However, the timing and momentum of OpenTelemetry appear to have a realistic chance of pushing for standardization of common aspects of telemetry.

To learn more about the promising aspects of OpenTelemetry, select *"Expand"* in each box.

**Benefits of OpenTelemetry**

* Instrument once, use everywhere

A key promise of OpenTelemetry is that you instrument code once and never again, giving you the ability to use that instrumentation everywhere. OpenTelemetry recognizes that, should its efforts be successful, it will be a core dependency for many software projects. Therefore, it follows strict processes to provide [long-term stability guarantees](https://opentelemetry.io/docs/specs/otel/versioning-and-stability/). Once a signal is declared stable, the promise is that clients will never experience a breaking API change.

* Separate telemetry generation from analysis

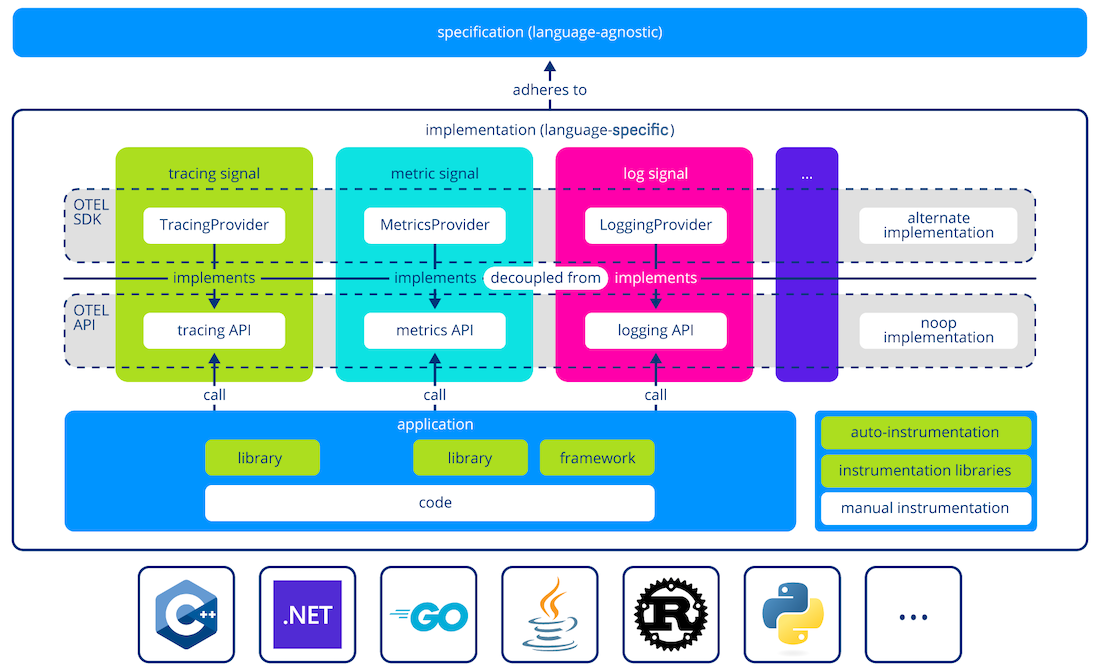
Another core idea of OpenTelemetry is to *separate the mechanisms that produce telemetry from the systems that analyze it*. Open and vendor-agnostic instrumentation marks a fundamental *change in the observability business*. Instead of pouring resources into building proprietary instrumentation and keeping it up to date, vendors must differentiate themselves through feature-rich analysis platforms with great usability. OpenTelemetry *fosters competition*, because users are no longer stuck with the observability solution they chose during development. After switching to OpenTelemetry, you can move platforms without having to re-instrument your entire system.

* Make software observable by default

With OpenTelemetry, open source developers are able to add *native instrumentation to their projects without introducing vendor-specific code* that burdens their users. The idea is to make observability a first-class citizen during development. By having software ship with built-in instrumentation, we no longer need elaborate mechanisms to capture and integrate it after the fact.

* Improve how we use telemetry

Last (and definitely not least), OpenTelemetry tries to change how we think about and use telemetry. Instead of having three separate silos for logs, metrics, and traces, OpenTelemetry follows a paradigm of linking telemetry signals together. With context creating touch points between signals, the overall value and usability of telemetry increase drastically. For instance, imagine the ability to jump from conspicuous statistics in a dashboard straight to the related logs. Correlated telemetry data helps to reduce the cognitive load on humans operating complex systems. Being able to take advantage of linked data will mark a new generation of observability tools.  
  
**Vendor-Agnostic, Language-Specific Instrumentation**



**Generate and Emit Telemetry via the OTel API and SDK Packages**

To generate and emit telemetry from applications, we use language-specific implementations, which adhere to OpenTelemetry’s specification. OpenTelemetry supports a wide-range of popular [programming languages](https://opentelemetry.io/docs/instrumentation/#status-and-releases) at varying levels of maturity. The implementation of a signal consists of two parts:

* API
  + defines the interfaces and constants outlined in the specification
  + used by application and library developers for vendor-agnostic instrumentation
  + refers to a no-op implementation by default
* SDK
  + provider implements the OpenTelemetry API
  + contains the actual logic to generate, process and emit telemetry
  + OpenTelemetry ships with official providers that serve as the reference implementation (commonly referred to as the SDK)
  + it is possible to write your own

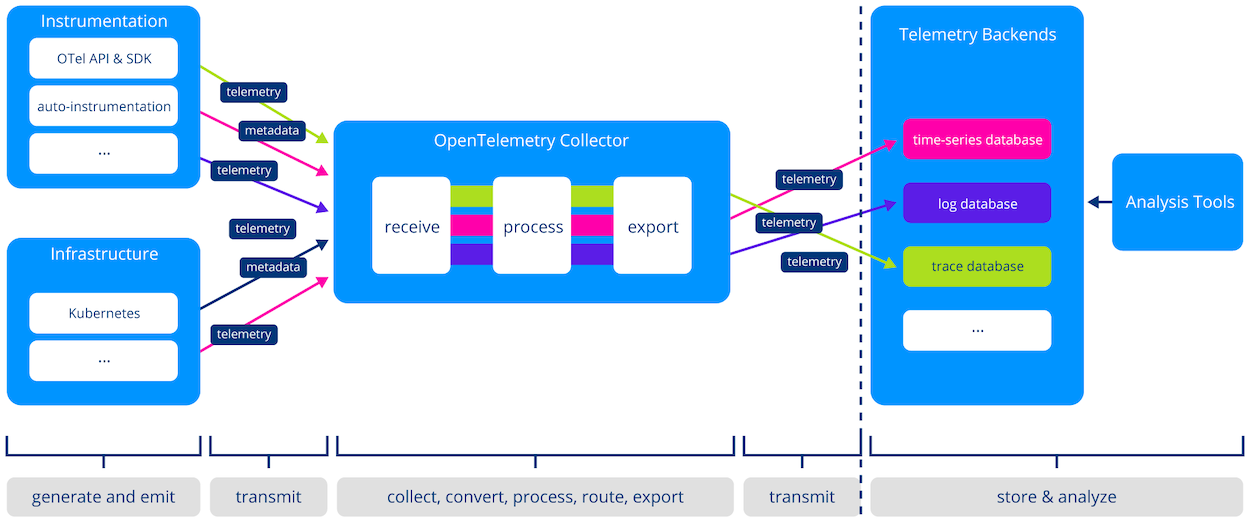
Generally speaking, we use the OpenTelemetry API to add instrumentation to our source code. In practice, this can be achieved in various ways, such as:

* zero-code or automatic instrumentation (if available and to avoid code changes)
* instrumentation libraries that provide simplified OpenTelemetry integration (which may or may not require code changes)
* manual or code-based instrumentation (for fine-grained control, deeply embedded in the code)

For now, let’s skip further details and focus on why OpenTelemetry decided to separate the API from the SDK. On startup, the application registers a provider for every type of signal. After that, calls to the API are forwarded to the respective provider. If we don’t explicitly register one, OpenTelemetry will use a fallback provider that translates API calls into no-ops.

The primary reason for separating the API from the SDK is that it makes it easier to embed native instrumentation into open source library code. OpenTelemetry’s API is designed to be lightweight and safe to depend on. The signal’s implementation provided by the SDK is significantly more complex and likely contains dependencies on other software. Forcing these dependencies on users could lead to conflicts with their particular software stack. Registering a provider during the initial setup allows users to resolve dependency conflicts by choosing a different implementation. Furthermore, it allows us to ship software with built-in observability without forcing the runtime cost of instrumentation onto users that don’t need it.

**Telemetry Processor (Standalone Component)**



**Processing and Forwarding Telemetry to Backends**

So far, we have seen that OpenTelemetry provides tooling for vendor-agnostic instrumentation to application and library developers. This alone marks a significant milestone, but OpenTelemetry’s framework goes much further. After generating and emitting telemetry, operators are responsible for managing and ingesting it into the respective backends. This includes tasks such as:

* gathering data from various sources
* parsing and converting it for downstream processing
* enrichment with additional metadata
* filtering out irrelevant data to reduce noise and storage requirements
* normalization and applying transformations
* buffering for resilience and performance
* routing to steer subsets of telemetry to different destinations
* forwarding to backends

To build and configure such telemetry pipelines, operations teams often deploy additional infrastructure. A popular example is the [fluentbit](https://fluentbit.io/" \t "_blank) telemetry agents. Similarly, OpenTelemetry provides a standalone component with these capabilities: the [OpenTelemetry Collector](https://opentelemetry.io/docs/collector/" \t "_blank).

**Wire Protocol**

Completing the standardization, generation, and management package, OpenTelemetry also defines how to transport telemetry between producers, agents, and backends.

The [OpenTelemetry Protocol (OTLP)](https://opentelemetry.io/docs/specs/otel/protocol/" \t "_blank) is an open source and vendor-neutral wire format that defines:

* how data is encoded in memory
* a protocol to transport that data across the network

As a result, OTLP is used throughout the observability stack. Emitting telemetry in OLTP means that instrumented applications and third-party services are compatible with countless observability solutions. The Collector supports receiving telemetry from and exporting to a various formats (e.g., Prometheus Metrics, Zipkin traces, etc.). However, OTLP is generally preferred because the Collector uses it internally to represent and process telemetry. Thereby, we avoid the cost of converting between formats and increase consistency. This is because the native format closely aligns with the ideas proposed by the framework (having attributes follow semantic conventions, cross-signal correlation, etc.).

Similarly, most observability backends support OTLP right out of the box. Given the rapid adoption of OpenTelemetry, integrating with OTLP automatically gives you access to a broad audience of potential users. Moreover, an open and vendor-neutral telemetry protocol means less work for developers of observability tools. Before, you had to develop countless adapters to be able to ingest data arriving in various proprietary formats. In other words, OTLP is a significant push for interoperability between tools and services in the observability ecosystem.

OTLP offers three transport mechanisms for transmitting telemetry data: HTTP/1.1, HTTP/2, and gRPC. When using OTLP, the choice of transport mechanism depends on application requirements, considering factors such as performance, reliability, and security. OTLP data is often encoded using the Protocol Buffers (Protobuf) binary format, which is compact and efficient for network transmission and supports schema evolution, allowing for future changes to the data model without breaking compatibility. Data can also be encoded in the JSON file format, which allows for a human-readable format with the disadvantage of higher network traffic and larger file sizes.