A Practical Observability Primer

Author: Sumit More
Contributors: Sandeep Sharma, Anuj More

An Stakx publication
“A Practical Observability Primer” is our take on the philosophy of Observability. We hope that it evolves into a conscious, collective and continuous effort to understand and implement the concept of Observability in its truest sense, and serves as a good starting point for the adopters of Observability.
# Table of Contents

## Roots of Observability ......................................................... 1
- Emergence of SRE and DevOps ........................................... 1
- Observability, Context and Monitoring ............................... 2
- Monitoring vs Observability .............................................. 4

## The WHAT, The WHERE, And The WHY .......................... 5
- The WHAT ............................................................................. 5
- The WHERE ............................................................................ 7
- The WHY .............................................................................. 8

## Pillars of Observability ....................................................... 9
- Measurements ........................................................................ 10
- Inferences .............................................................................. 10
- Logging .................................................................................. 10
- Metrics .................................................................................... 12
- Traces ..................................................................................... 14

## Today’s tools ......................................................................... 17
- Elastic Stack ........................................................................... 17
- Prometheus: (Metrics Collection and Dashboarding) .............. 19
- Zipkin (Trace Collection and Dashboarding) ......................... 21
- Jaeger (Trace Collection and Dashboarding) .......................... 23
- OpenCensus (Metrics and Trace Collection) ........................... 25
- Grafana: (Analytics and Dashboarding) ................................. 27

## Churning The Magic ............................................................. 29

## Future of Observability ....................................................... 31

## References ............................................................................. 32
Chapter 1

Roots of Observability

Emergence of SRE and DevOps

Roots of Observability lie in the emergence of SRE (Site Reliability Engineering) and DevOps.

Software engineering has taught us that software needs to be Scalable, Available, Resilient, Manageable and Secure. Scalability and reliability form the cornerstones of these principles.

Scalability is the ability of the system to handle increased load. Applications can scale vertically (scaling up) i.e. means increasing the capacity of resource(s), or horizontally (scaling out) i.e. adding new instances of a resource(s).

Availability is a characteristic of a system, which aims to ensure an agreed level of operational performance, usually uptime, for a higher than normal period. Reliability is the probability of continuous correct operation. Having a system available and not reliable makes no sense. Similarly having a system reliable and not available makes no sense either. Hence, these terms are used congruously.

The early 2000s saw emergence of Site Reliability Engineering (SRE) with an intention to have dedicated teams whose sole target was to make software applications ultra-scalable and highly-reliable in an enterprise setting. After the introduction of SRE at Google, almost every tech giant saw the impending value of such teams, and ultimately incorporated such teams in their engineering organisations.

By late 2000s, the industry saw emergence of yet another vertical in form of DevOps. By this time most of the tech world was moving or had already moved towards Agile development. The need to 'reduce the time between committing a change to a system and the change being placed into normal production environment, while at the same time ensuring high quality' was of paramount importance.
The common thread between SRE and DevOps were effective monitoring systems. The monitoring system would address two questions: what’s broken, and why? The “what’s broken” indicated the symptom; the “why” indicated a (possibly intermediate) cause. The answers to these ‘Whats and Whys’ would give the development, deployment and management teams a bird’s eye view of the system, and valuable information which they could use in their corresponding decision making. Dashboarding of different metrics became a common practice in engineering teams. As the systems grew complex (and distributed), the need for monitoring changed from good-to-have to mandatory.

**Observability, Context and Monitoring**

By early 2010s the software industry had started adopting distributed architecture (microservices in particular) in their products. Major software products had seen the apparent positives of breaking their monoliths into distributed architecture. Google, Twitter and Amazon were few of the forerunners in this journey. At the same time, software infrastructure was moving towards cloud. AWS, Azure and the likes were providing cutting-edge solutions for software product/services companies to move their infrastructure in to the cloud. Cloud made way for the ideas like Containers, Orchestrators, Microservices and Serverless easy to explore & implement; and soon software underwent another transformation.

Distributed architecture meant more moving parts, which meant more communication between these moving parts. The software application communication (via gRPC, HTTPs, REST, GraphQL etc.) also increased in a distributed environment. With introduction of containers and serverless, the intra-system communication spiked up as well. More communication meant more messages, and more messages meant more events. Monitoring of the health and performance of the complex distributed architectures became important for quick root-cause analysis and debugging of the issues. The distributed teams felt the need of a system which could collect and monitor various events happening all over the system. Such events could be health logs of every node in the cluster, performance metrics of the hundreds of upstream/downstream services across the datacenters, logs
Building resilient and better fault-tolerant systems required understanding the context of the events that were being monitored. Observability was introduced to provide such a context-aware monitoring of the distributed infrastructure at hand. It is rather difficult to define Observability into a single concept. Some definitions consider Observability in terms of system failure, while some talk about Observability with reference to the testing pyramid. Distributed systems are inherently designed not be 100% available all the time. Hence, it makes sense to build Observability as a concept of collecting every possible snapshot of the system. These snapshots can then be used to develop intelligent analytics upon the data collected, which can further be used to provide alerts and probably self-healing triggers into the system. Final outcome of Observability stack are visualisations of these snapshots, analytics and alerts which could be useful to the engineering team.

We like to define it as: **An engineering philosophy wherein you observe the data flowing through the whole system via a set of tools and practices, and turn the collected data points and contexts into useful insights.**
Monitoring vs Observability

Monitoring inherently is tracking of alerts and the numbers of what’s going on in the system. Monitoring helps in answering “how is your system doing” by collecting and dashboarding the data.

Observability like visibility and availability is the quality of the service. It attempts to explore the system with an intention to deduce “what is your system doing” by exposing the data and exploiting context between the data to better understand the system. If something is observable, then you can monitor it (as well as do other things).

‘Observable’ monitoring systems inherently:

1. Collect and analyse high quality data in terms of correctness and completeness
2. Periodically update the metrics so as to avoid getting 'gamed'
3. Avoids wrong incentives/insights by analysing combination of metrics instead standalone analysis
Chapter 2

The WHAT, The WHERE, And The WHY

Observability framework comprises of the following building blocks:

1. Instrumentation (Edge Collection): Simply put, the responsibility of this block of the framework is to collect the logs, metrics and trace

2. Stack (Data Storage): Stack is where all the collected data (logs, metrics, and traces) is indexed and stored

3. Visualisation (Analysis): This block presents the collected data in a form that is useful for analysis. This is where the collected data is used to correlate and create dashboards which provide analytical insights of the system and (business) application

The visualisation layer queries its data from stack layer, which in turn stores the data provided by the instrumentation layer. The consumers of the visualisation layer range from the top-management (business), development teams, to SRE (Site Reliability Engineering) teams. Hence, it becomes imperative to have a clear understanding of WHAT data needs to be collected by the Instrumentation layer, WHERE should such data be collected and WHY is such data important.

As it turns out; these questions are not new. The SRE teams have been dealing with these questions for years now, and to our advantage, several SRE workbooks have addressed this in depth.

The WHAT

Building up from the experience of SRE teams we’ve come to a conclusion that the USE, and RED methods give a fair understanding as to what type of events need to be collected for an effective Observable system. Let us understand both of these methods in brief.
USE stands for: **Utilization | Saturation | Errors**

The USE method applies to infrastructure (network interfaces, storage disks, CPUs, memory etc.)

**Utilization**: the average time that the resource was busy servicing work

**Saturation**: the degree to which the resource has extra work which it can’t service, often queued

**Errors**: the count of error events

RED stands for: **Rate | Errors | Duration**

Since the USE method doesn’t really apply to services, the RED method addresses the monitoring of services.

**Rate**: The number of requests per second

**Errors**: The number of those requests that are failing

**Duration**: The amount of time those requests take

The Google SRE team has also defined ‘4 Golden Signals’ which prove to be insightful when collected.

Those **4 Golden Signals** are: **Latency | Traffic | Errors | Saturation**

**Latency**: The time it takes to service a request

**Traffic**: A measure of how much demand is being placed on your system, measured in a high-level system-specific metric

**Errors**: The rate of requests that fail, either explicitly (e.g., HTTP 500s), implicitly (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, “If you committed to one-second response times, any request over one second is an error”)

---

*mStakx*  
*A Practical Observability Primer*
**Saturation**: How “full” your service is. A measure of your system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O)

Sample events

1. Error Count (Eg: RPC Errors)
2. Request bytes
3. Number of request messages
4. Response bytes
5. Number of response messages
6. Round Trip latency
7. Server elapsed time
8. Uncompressed Request bytes
9. Uncompressed Response bytes
10. Page Load Time (PLT)

The WHERE

Good places to add instrumentation (collect data) in the system are at the points of ingress & egress, inside system (application logic) etc. For instance:

1. Logging requests and responses specific webpage-hits or API endpoint (If you’re instrumenting an existing application then make a priority-driven list of specific pages or endpoints and instrument them in order of importance.
2. Measure and log all calls to external services and APIs. For e.g: Calls (or queries) to database, cache, search service etc.
3. Measure and log job-scheduling and the corresponding execution. E.g. cron jobs
4. Measure significant business and functional events, such as users being created or transactions like payments and sales

5. Measure methods and functions that read and write from databases and caches

The WHY

Credit: The New York Times

‘Observing the software system’ is a good idea as it enables the engineering teams to:

1. Identify and diagnose faults, failures, and crashes
2. Measure and analyse the operational performance of the system
3. Measure and analyse the business performance and success of the system or its component(s)
4. Discover different insights by combining the results of different metrics
5. Identify gaps in business logic

Thus, avoiding a serious business and operational risk.
Chapter 3

Pillars of Observability

The concept of Observability is built on collecting every possible snapshot of the application. These snapshots can then be used to develop intelligent analytics upon the data collected, which can further be used to provide alerts and probably self-healing triggers into the system. Final outcome of Observability stack are visualizations of these snapshots, analytics and alerts which could be useful to the engineering team. Observability is a process which is more than just tooling. It is a culture which when adopted makes the system context aware.

Making systems observable inherently is based upon collecting factual measurements, and concluding insightful inferences based on those measurements.
Measurements

Being cognizant of all the events happening in the system and collecting them serves as a rich dataset towards achieving observability. Such measurements include:

1. **Logging**: Logging comprises of recording discrete events in the system. These events can be structured (JSON based application/system logs) or unstructured (text strings)

2. **Metrics**: Metrics are aggregatable events like counters (Eg: HTTP requests), gauges (HTTP queue depth), histogram etc. which can help identify trends

3. **Tracing**: Recording events with causal ordering across services and distributed systems as well; hence, enabling them to identify cause across borders

Inferences

Inferencing comprise of extracting information out of the data collected, and correlating multiple sources of data to give a better understanding of the system.

Logging

Logs provide strategic insight by capturing the snapshot of the system along with the context between the multiple subsystems of an application. Logs are generally instrumented as per their usability. Depending on the storage rules, they’re processed, aggregated and eventually stored in a centralized data store from where they can be indexed, queried and analysed/processed further.

Sources

There are a number of sources from which these logs can originate such as:

1. Application logic code (Business logic)

2. Middleware and Network communication (Request Brokers, JDBC, Switches etc.)
3. Communication over the network (HTTP requests/responses)

4. Communication with underlying Database(s)

5. Communication channels (Message brokers)

6. Messages with task-queues, caches (Celery, Redis, etc.)

7. Interaction with load balancers

8. Communication with security and authentication modules (Firewalls)

Types of Logs

1. Plaintext Logs: This is generally a timestamped free-form text.

2. Structured Logs: This type of log has a well defined structure. For instance: Logs conforming to the JSON format.

3. Binary Logs: Logs in the Protobuf format, MySQL binlogs used for replication and point-in-time recovery, systemd journal logs, the pflog format used by the BSD firewall etc.

Benefits of Logs

1. Logs are omniscient about every single request. Hence, logs can be queried with simplest to most complex of tools to gather insights into the working of the system.

2. With intelligent logging one can literally play Dr. Strange and recreate the system state of past or future.

3. The analysis performed using logs can further be logged and stored for future comprehensive analysis.

4. Structured logging helps capture just about anything that one might perceive to be of interest. Such type of log data can support high dimensionality, the sort that is great for things like:
   a. Exploratory analysis of outliers
   b. Auditing
   c. Analytics like measuring revenue, billing, user engagement
d. Real time fraud analysis

e. DDoS detection

Limitations of Logs
1. Log generation and storage overhead tends to increase exponentially with n number of application components, nodes and communication channels

2. The cost of logs increases in lockstep with user traffic or any other system activity that could result in a sharp uptick in Observability data.

Note: Stream analytics (for data in motion) and compression (data at rest) can dramatically reduce amount of storage needed

Metrics

Metrics are a set of numbers that give information about a particular process or activity. They are measures of properties in pieces of software or hardware. To make a metric useful we keep track of its state, generally recording data points or observations over time. An observation is a value, a timestamp, and sometimes a series of properties that describe the observation, such as a source or tags. The combination of these data point observations is called a time series.

Metrics can be visualized in different types of graphs such as gauges, counters, and timers.

Type of Metrics
1. **Gauges**: Gauges are numbers that are expected to change over time. A gauge is essentially a snapshot of a specific measurement. The classic metrics of CPU, memory, and disk usage are usually articulated as gauges. For business metrics, a gauge might be the number of customers present on a site.

2. **Counters**: Counters are numbers that increase over time and never decrease. Good examples of application and infrastructure counters are system uptime, the number of bytes sent and received by a device, or the number of logins. Examples of business counters might be the number of sales in a month or cost of sales for a time period.
useful thing about counters is that they let you calculate rates of change. A lot of useful
information can be understood by understanding the rate of change between two
values. For example, the number of logins is marginally interesting, but create a rate
from it and you can see the number of logins per second, which should help identify
periods of site popularity.

3. **Timers**: Timers track how long something took. They are commonly used for application
monitoring—for example, you might embed a timer at the start of a specific method and
stop it at the end of the method. Each invocation of the method would result in the
measurement of the time the method took to execute.

**Benefits of Metrics**
1. Since metrics are just numbers measured over intervals of time, they can be compressed,
   stored, processed and retrieved very efficiently
2. Metrics are optimized for storage and enable longer retention of data, which can in turn
   be used to build dashboards to reflect historical trends
3. Metrics allow for effective and valid aggregations (daily or weekly frequency)
4. Metrics transfer and storage has a constant overhead
5. The cost of metrics doesn’t increase in lockstep with user traffic or any other system
   activity
6. Metrics, once collected, are more malleable to mathematical and statistical
   transformations such as sampling, aggregation, summarization and correlation, which
   make it better suited for monitoring and profiling purposes
7. Metrics are also better suited to trigger alerts, since running queries against an in-
   memory time series database is far more efficient than running a query against a
   distributed system storage, and then aggregating the results before deciding if an alert
   needs to be triggered

**Limitations of Metrics**
1. One of the biggest drawback of historical time series databases has been the
   representation of metrics which didn’t lend itself very well toward exploratory analysis or
filtering. The hierarchical metric model and the lack of tags or labels in systems like Graphite especially hurt in this regard. Modern monitoring systems like Prometheus represent every time series using a metric name as well as additional key-value pairs called labels. This allows for a high degree of dimensionality in the data model. A metric is identified using both the metric name and the labels.

2. Metrics in Prometheus are immutable; changing the name of the metric or adding or removing a label will result in a new time series. The actual data stored in the time-series is called a sample and it consists of two components—a float64 value and a millisecond precision timestamp.

**Traces**

A trace is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system. Traces are a representation of logs; the data structure of traces looks almost like that of an event log. A single trace can provide visibility into both the path traversed by a request as well as the structure of a request. The path of a request allows software engineers and SREs to understand the different services involved in the path of a request, and the structure of a request helps one understand the junctures and effects of asynchrony in the execution of a request.

**Components of a Trace**

1. **Span**: Span is a set of annotations that correspond to a particular RPC. A span represents a logical unit of work that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships.

2. **Trace**: A trace is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans. At the highest level, a trace tells the story of a transaction or workflow as it propagates through a (potentially distributed) system. Traces are built by collecting all spans that share a traceId. The spans are then arranged in a tree based on span-Id and parent-Id thus providing an overview of the path a request takes through the system.
Although discussions about tracing tend to pivot around its utility in a microservices environment, it's fair to suggest that any sufficiently complex application that interacts with or rather, contends for resources such as the network, disk, or a mutex in a non-trivial manner can benefit from the advantages tracing provides.

The basic idea behind tracing is straightforward - identify specific points (function calls or RPC boundaries or segments of concurrency such as threads, continuations, or queues) in an application, proxy, framework, library, runtime, middleware, and anything else in the path of a request that represents the following:

- Forks in execution flow (OS thread or a green thread)
- A hop or a fan out across network or process boundaries

Traces are used to identify the amount of work done at each layer while preserving causality by using happens-before semantics.

Benefits of Tracing

1. Identify the amount of work done at each component/layer/service while preserving causality
2. Ability to track a request as it travels through each of the services
3. Ability to collect metrics of any interest for a specific span

Limitations of Tracing
1. Most of the distributed tracing tools still do not support quite a few programming languages
2. Library instrumentation still lacks support for a few big frameworks
3. Tracing tools themselves are distributed in nature. Hence, compatibility needs to be checked at the time of orchestration
Today’s tools

Today a few technology companies like honeycomb.io, humio.io, lightstep.com etc. that provide out-of-the-box paid frameworks and products in the Observability area.

However, there are quite a few mature open-source (OSS) tools and frameworks available in order to realise Observability via Instrumentation (Logging, Metrics and Tracing collection), Stack (Data storage) and Visualisation (Analysis). These include the following:

Elastic Stack

Formerly known as “ELK” Stack, “ELK” stood for three open source projects: Elasticsearch, Logstash, and Kibana.

Credit: Elastic
Components

1. **Logstash**: Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a “stash” like Elasticsearch. In context of Observability it serves as a Logging instrumentation component.

2. **Elasticsearch**: Elasticsearch is a search and analytics engine. In context of Observability it serves as a centralised data storage component to which varied data can be ingested, and from which data can be exported to any UI tooling framework for analytical purposes.

3. **Kibana**: Kibana lets users visualize data with charts and graphs in Elasticsearch. In context of Observability, Kibana serves as a Dashboarding/Analytics component.

4. **Beats**: In 2015, Elastic introduced a family of lightweight, single-purpose data shippers into the ELK Stack equation called Beats. The community Elastic framework continues to grow stronger as the need for Observability finds firm roots in current engineering scenario.

Features

1. Elastic provides enterprise-level proactive cluster alerting which automatically notifies changes in the cluster state, application state and host of other metrics

2. Elastic provides multi-stack support & analysis to record, track, and compare the health and performance from a single place.

3. Elastic provides ability to go beyond rule-based alerting by combining alerting with unsupervised machine learning

4. Elastic provides ability to generate, schedule & email reports of any Kibana visualization or dashboard based on specified conditions and is architected to scale and travel well

5. Elastic provides ability to export raw documents, saved searches, and metrics for seamless integration into existing monitoring frameworks
Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

Components
The Prometheus ecosystem consists of multiple components, many of which are optional:
1. The main Prometheus server which scrapes and stores time series data

2. Client libraries for instrumenting application code

3. A push gateway for supporting short-lived jobs

4. Special-purpose exporters for services like HAProxy, StatsD, Graphite, etc.

5. An alertmanager to handle alerts

6. Various support tools (E.g. node_exporter)

Most Prometheus components are written in Go, making them easy to build and deploy as static binaries.

Features

1. Prometheus provides a multi-dimensional data model with time series data identified by metric name and key/value pairs which fits both machine-centric monitoring as well as monitoring of highly dynamic microservices architectures

2. Prometheus has a flexible query language to leverage this dimensionality monitoring of highly dynamic microservices architectures

3. Prometheus is designed for reliability, to be the system you go to during an outage to allow you to quickly diagnose problems

4. In Prometheus time series collection happens via a pull model over HTTP

5. Pushing time series is supported via an intermediary gateway

6. Targets are discovered via service discovery or static configuration

7. Multiple modes of graphing and dashboarding support (Eg: Grafana and other API consumers)

8. No reliance on distributed storage; single server nodes are autonomous

9. Each Prometheus server is standalone, not depending on network storage or other remote services, thus making it reliable even when parts of the infrastructure are broken
Zipkin (Trace Collection and Dashboarding)

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures, and manages both the collection and lookup of this data. Applications are instrumented to report timing data to Zipkin.

Credit: Zipkin Architecture
Components

1. **Zipkin Collector**: Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

2. **Storage**: Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema. In addition to Cassandra, there is a support for ElasticSearch and MySQL. A few other back-ends are also available as third party extensions.

3. **Zipkin Query Service**: Once the data is stored and indexed, we need a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

4. **Web UI**: A GUI that presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI.

For example, when an operation is being traced and it needs to make an outgoing http request, a few headers are added to propagate IDs. Headers are not used to send details such as the operation name.

Credit: Zipkin Sample Trace
The component in an instrumented app that sends data to Zipkin is called a Reporter. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI.

Features

1. **Web UI**: The Zipkin UI presents an easy-to-understand web UI which provides a dependency diagram showing how many traced requests went through each application.

2. **Trace Filtering**: Zipkin provides filter or sort all traces based on the application, length of trace, annotation, or timestamp for troubleshooting latency problems or errors

3. **Production Safe**: Instrumentation is written to be safe in production and have little overhead

---

**Jaeger (Trace Collection and Dashboarding)**

Jaeger is a distributed tracing system released as open source by Uber Technologies. It is used for monitoring and troubleshooting microservices-based distributed systems, including:

1. Distributed context propagation

2. Distributed transaction monitoring

3. Root cause analysis

4. Service dependency analysis

5. Performance/latency optimization
Components

1. **Jaeger Agent**: The Jaeger agent is a network daemon that listens for spans sent over UDP, which it batches and sends to the collector. It is designed to be deployed to all hosts as an infrastructure component. The agent abstracts the routing and discovery of the collectors away from the client.

2. **Jaeger Collector**: The Jaeger collector receives traces from Jaeger agents and runs them through a processing pipeline. Currently our pipeline validates traces, indexes them, performs any transformations, and finally stores them.

3. **Jaeger Query**: Jaeger Query is a service that retrieves traces from storage and hosts a UI to display them.

Features

1. **High Scalability**: Jaeger backend is designed to have no single points of failure and to scale with the business needs.

Credit: [Jaeger Architecture](#)
2. **Native support for OpenTracing**: Jaeger backend, Web UI, and instrumentation libraries have been designed from ground up to support the OpenTracing standard.

3. **Multiple storage backends**: Jaeger supports two popular open source NoSQL databases as trace storage backends: Cassandra 3.4+ and Elasticsearch 5.x/6.x. There are ongoing community experiments using other databases, such as ScyllaDB, InfluxDB, Amazon DynamoDB. Jaeger also ships with a simple in-memory storage for testing setups.

4. **Modern Web UI**: Jaeger Web UI is implemented in Javascript using popular open source frameworks like React.

5. **Cloud Native Deployment**: Jaeger backend is distributed as a collection of Docker images. The binaries support various configuration methods, including command line options, environment variables, and configuration files in multiple formats (yaml, toml, etc.) Deployment to Kubernetes clusters is assisted by Kubernetes templates and a Helm chart.

6. **Metrics Support**: All Jaeger backend components expose Prometheus metrics by default (other metrics backends are also supported). Logs are written to standard out using the structured logging library zap.

7. **Backwards compatibility with Zipkin**: Jaeger provides backwards compatibility with Zipkin by accepting spans in Zipkin formats (Thrift or JSON v1/v2) over HTTP. Switching from Zipkin backend is just a matter of routing the traffic from Zipkin libraries to the Jaeger backend.

---

**OpenCensus (Metrics and Trace Collection)**

OpenCensus is a metric and trace collection tool. It is available as a vendor-agnostic set of libraries which can be used to collect traces and metrics from any application. Instrumenting the application code with OpenCensus helps gain the ability to understand exactly how a request travels between the services, and gather any useful metrics about the entire
architecture. OpenCensus is used to visualize request lifecycle, perform root-cause analysis, and optimize service latency by gaining key insights into the latency and performance of every (micro)service and data storage that is being managed.

Components

1. **Context**: Some of the features for distributed tracing and tagging need a way to propagate a specific context (trace, tags) in-process (possibly between threads) and between function calls. Context component makes sure that such contexts, and their corresponding sub-contexts are propagated.

2. **Trace API**: Trace component is designed to support distributed tracing apart from data collection and export.

3. **Tags API**: The Tag API allows for creating, modifying and querying objects representing a tag (key-value pair) which propagate through the context subsystem via RPC, HTTP, etc.

4. **Stats API**: The Stats API component is designed to record measurements, dynamically break them down by application-defined tags, and aggregate those measurements in user-defined ways. It is designed to offer multiple types of aggregation (e.g. distributions) and be efficient (all measurement processing is done as a background activity); aggregating data enables reducing the overhead of uploading data, while also allowing applications direct access to stats.

Features

1. **Low latency**: OpenCensus is simple to integrate and use, it adds very low latency to your applications and it is already integrated into both gRPC and HTTP transports.
2. **Vendor Agnosticity**: OpenCensus is vendor-agnostic and can upload data to any backend with various exporter implementations. Even though, OpenCensus provides support for many backends, users can also implement their own exporters for proprietary and unofficially supported backends.

3. **Simplified tracing**: Distributed traces track the progression of a single user request as it is handled by the internal services until the user request is responded.

4. **Context Propagation**: Context propagation is the mechanism by which information (of your choosing) is sent between your services. It is usually performed by sending data in headers and trailers on HTTP and gRPC transports.

---

**Grafana: (Analytics and Dashboarding)**

Grafana is an open source, feature rich metrics dashboard and graph editor that allows you to query, visualize, alert on and understand your metrics no matter where they are stored. It gives engineering teams ability to create, explore, and share dashboards and foster a data driven culture.

**Features**

1. **Visualize**: Fast and flexible client side graphs with a multitude of options. Panel plugins for many different way to visualize metrics and logs.

2. **Alerting**: Visually define alert rules for your most important metrics. Grafana will continuously evaluate them and can send notifications.

3. **Notifications**: When an alert changes state it sends out notifications. Receive email notifications or get them from Slack, PagerDuty, VictorOps, OpsGenie, or via webhook.

4. **Dynamic Dashboards**: Create dynamic & reusable dashboards with template variables that appear as dropdowns at the top of the dashboard.
5. **Mixed Data Sources**: Mix different data sources in the same graph. You can specify a data source on a per-query basis. This works even for custom datasources.

6. **Annotations**: Annotate graphs with rich events from different data sources. Hover over events shows you the full event metadata and tags.

7. **Ad-hoc Filters**: Ad-hoc filters allow you to create new key/value filters on the fly, which are automatically applied to all queries that use that data source.
Chapter 5

Churning The Magic

With a fair understanding of Observability, and introduction to some of the tools/ frameworks to achieve observable systems; we thought it would be a good idea to create a boilerplate to demonstrate the whole system in action. This boilerplate is a single click deployment built as a set of Docker images.

Components

The key components of this boilerplate are the docker images of the following:

1. **Application Layer**: A simple web application built in Python Django with Postgres as database, and nginx running the web server

2. **Instrumentation Layer**: A set of Logstash, Prometheus and Zipkin libs for Python serving as conduits to collect data (logs, metrics and traces)

3. **Stack (Data Storage) Layer**: Elasticsearch, Zipkin and Prometheus storage

4. **Visualization (Analysis) Layer**: Kibana, Prometheus UI, Grafana, and Zipkin UI
You can get started with this boilerplate with a single click, play around with it, and use it in your development/production environments to turn your applications into Observable systems. The boilerplate code is available on Github. Pull-requests are welcome to make it better and more useful for a varied production environments.

Features

1. Advantages of OSS (Open Source Software): The boilerplate inherently enjoys the advantages of OSS since all the tools used in creation of this boilerplate. These include:
   a. Cost effectiveness
   b. Being able to be flexible in terms of changing tools
   c. Ability to take the community versions and get started quickly in order to attain faster time to market
   d. Visibility into the internal working of the tools, and scope of customizing them to specific needs

2. Cloud Agnostic: With a little bit of production automation, the system is designed to fit seamlessly on to any cloud platform (AWS, Azure, Google, DigitalOcean, etc.)

3. Containerised Solution: The system is containerised (using Docker) which enables exploiting the advantages of cloud-native development
In our opinion context-awareness is at the core of an effective Observability framework.

Data-science in general and machine-learning in particular has the capability to handle large volume of data and the potential to unravel unpredictable behaviours; and hence could perform a strategic role in Observability. Machine Learning could essentially be at the forefront of processing the Observability data so that the “What?” question can be answered. The problem is the lack of training data to train the models, which at the moment only web scale companies have.

There is an increasing consensus among the technology leaders that machine learning could be a good way forward to fully comprehend extremely complex IT stacks that are getting more distributed with containers, serverless functions, edge computing and IoT.

At a minimum level, machine learning can make the Observability data more useful for humans. At present, the use of machine learning for Observability is at very early stages and it is going to take a long time before it becomes the norm.
References

10. Logs and metrics: https://medium.com/@copyconstruct/logs-and-metrics-6d34d3026e38
11. OpenCensus: https://opencensus.io/
15. Prometheus Monitoring and Alerting: https://prometheus.io/
16. Elastic Stack: https://www.elastic.co/
17. OpenCensus: https://github.com/census-instrumentation/opencensus-specs
