Département TIC
Filière Télécommunications
Orientation Réseaux et services

Travail de Bachelor

Analysis and deployment of an observability framework for micro-services-based applications

Non confidentiel

Étudiant : **Wènes Limem**
Travail proposé par : Graf Marcel

Enseignant responsable : **Graf Marcel**
Année académique : 2021-2022

Yverdon-les-Bains, le 22 avril 2022

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Le Chef du Département

Yverdon-les-Bains, le 22 avril 2022

Authentification

Le soussigné, Wènes Limem, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Yvedon-les-bains, le 22 avril 2022

Wènes Limem

Table of Contents

1 Introduction

For a long time, desktop clients ruled the IT world; monolithic design patterns were satisfying the needs of the existing number of clients and serving them with reliable performance. Thanks to the revolution in gadget development, clients are not just desktop clients (known as thick clients) anymore. But every connected device (Smartphones, tablets, IoT objects, also known as thin clients). Such diversity has pushed the number of clients to increase drastically.

The growing number of users will cause our applications to drop in performance. In addition, the infrastructure needed to run the applications has also changed, so that we do not need to buy and maintain the infrastructure anymore, but only to rent it from cloud-providers. Consequently, the evolution of technologies used in web applications development and deployment has left us no choice but to consider the traditional monolithic pattern as a non-fit for the development of modern web applications designed to run on the cloud.

To solve the problems and cover the insufficiencies caused by the monolithic design pattern, the micro-services design pattern has emerged for this purpose. This modern pattern is optimized to be coupled with cloud-services. However, this architecture, makes the application a black-box and adds a complexity layer, making us blind to what is happening in the application.

At first, engineers tried to uncover the black-box and decomplexify the application by using traditional monitoring and logging tools. In fact, these tools were developed to monitor monolithic applications on a certain type of infrastructure. With some tweaks made to them, they became usable in micro-services patterns and the various cloud infrastructures.

Decomplexifying and uncovering the application layers, will allow us to understand the magic happening inside the application, the communications between the micro-services and their states. On this account, the traditional tools, even tweaked, show insufficiencies in helping us uncover the application's black box. Thus, the need for modern monitoring and logging tools.

It is only natural that a question like "Are modern monitoring and logging tools enough to reach a general understanding of the workings of the application?". To answer this question, first, we need to agree on calling the general understanding of the application, observability. Finally, we need to define, what is observability? How can we characterize it? When is it required? What are its benefits for modern applications? And how does it apply to them?

2 Observability

2.1 What is Observability?

In IT and cloud computing, observability is the ability to measure a system's current state based on the data it generates, such as logs, metrics, and traces. It relies on telemetry derived from instrumentation that comes from endpoints and services in your multi-cloud computing environments. In these modern environments, every hardware, software, and cloud infrastructure component and every container, open-source tool, and microservice, generates records of every happening activity.

The goal of Observability is to understand what is happening across all these environments and among the technologies, so you can detect and resolve issues to keep your systems efficient and dependable.

Developers' teams inside an organization, usually implement observability using a combination of instrumentation methods and tools. They do also adopt an observability solution to help them detect and analyze the significance of events to their ops, development lifecycle, application security and end-user experience. [1]

2.2 Monitoring vs Observability

At first, we may think that observability is merely another name for sophisticated monitoring, but no. While the two terms share a lot in common, and can even complement each other, they are different concepts.

Figure 2—1 Monitoring vs Observability

One the one hand, *monitoring* allows us to watch and understand the state of our systems. It is based on gathering predefined metrics and logs. Thus, it can only show us what we have expected to occur. Furthermore, the questions we can answer with monitoring are: What is the state of my system? Is my system working?

On the other hand, observability allows us to actively debug our system. It is based on understanding how the system works. When it first emerged, the term indicated that if a system generates enough meaningful data that a human operator can understand its internal state based on external output, then such a system is observable. Therefore, systems provide insights, contexts and debugging data (like logs, metrics, and traces: they will be explained further) so that the system's administrator is able to answer the questions: Why did my system fail? What is my system doing?

To illustrate this better, let us consider a scenario where our system design pattern is based on micro-services. If we encounter a service-disrupting incident, monitoring will not be able to show us where the incident occurred, it will just tell us that something is off. So, to fix this incident, we must find where the anomaly is occurring and why. However, if our system were designed to be observable, such an incident would be detected and debugging it would not take long.

2.3 Benefits of Observability

In enterprise environments, observability helps cross-functional teams understand and answer specific questions about what is happening in highly distributed systems. It enables us to understand what is slow or broken and what actions are needed to be taken to improve performance. With an observability solution in place, teams can receive alerts about issues and pro-actively resolve them before they affect users.

Due to the dynamic nature of cloud environments, most problems are neither predictable nor monitored. Observability addresses this common issue, enabling us to understand new types of problems continuously and automatically as they rise.

The value of observability does not stop at IT use cases. Once you begin collecting and analyzing observability data, you have an invaluable window into the business impact of our digital services. Such visibility allows us to optimize conversions, validate releases, and meet business goals.

Observability benefits IT teams, organizations, and end-users alike.

Some use cases that Observability eases:

- **Application performance monitoring,** full end-to-end observability enables organizations to get to the root of application performance issues much faster, including issues that arise from cloud-native and micro-services environments.

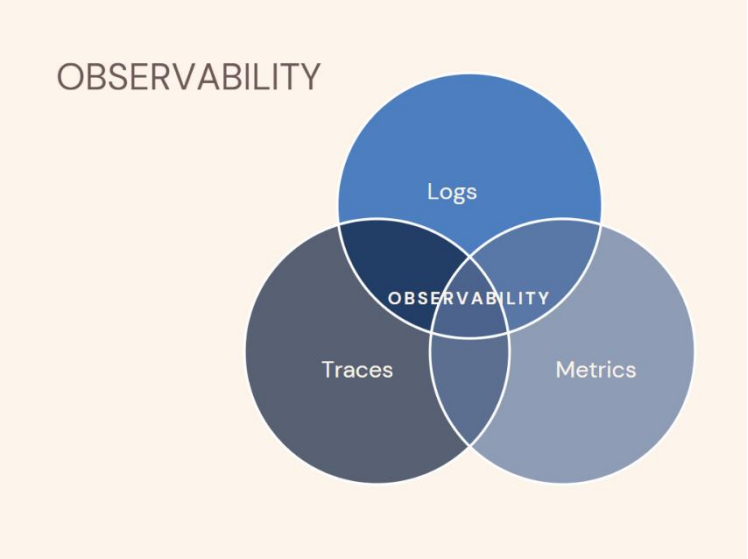
- **Infrastructure, Cloud and Kubernetes monitoring**, Infra and Ops teams can use the enhanced context an observability solution offers to improve application uptime and performance, cut down the time required to pinpoint and resolve issue, detect cloud latency issues, optimize cloud resource utilization, and impose administration of their Kubernetes environments and modern architectures.

- **End-use experience**, a good user experience can enhance a company's reputation and increase users' number, delivering an edge over the competition. By detecting and resolving issues well before the end-user notices and by making improvements before they are requested. There is also the possibility to optimize latency exactly as they see it, so everyone can quickly agree on where to make improvements.

- **DevSecOps**, observability is not just the result of implementing advanced tools, but a foundational property of an application and its supporting infrastructure. DevSecOps integrates three disciplines Development, Security and Operations. The three disciplines share their need for visibility so they can perform actions and reach desired goals. Observability offers a view to the actual actions the system takes, its performance and its issues, in near real-time. Architects and developers who create the software must design it to be observed. Then DevSecOps teams can use and interpret the observable data during the software delivery life cycle to build better, more secure, and more resilient applications. [1]

- **Business Intelligence,** observability can be extended to application level, more precisely to the business logic layer. In such a context, we can observe what is happening inside the application, the where and when of every occurring transaction. To interpret this, we can consider an example of an e-commerce web application. We would be interested to know the impact of a publicity campaign for the application. In that case, we need to observe the number of registered users. We can conclude the impact of the campaign based on the evolution of the observed variable.

2.4 Observability pillars



Based on what we have seen so far, we can characterize a successful observable system based on the measurements collected from these three: Logs, Metrics and Distributed traces are the key pillars to achieving success. Although, we can gather such data from back-end components, we cannot have a full picture of how our system is behaving based on the raw form of these data.

The three pillars mentioned previously can be defined as follows:

- **Logs**, structured or unstructured text records of events that occurred at a specific time. [1]

- **Metrics**, numeric values that represent counts or measurements that are often calculated or aggregated over a temporal period. Metrics can originate from a variety of sources, including infrastructures, hosts, services, application-level metrics, cloud platforms, and external resources.

- **Distributed tracing**, activity of a transaction or request as it flows through applications and shows how services connect, may include code-level-details. Distributed tracing uses spans and alerts to notify admin systems of what is going on.

2.4.1 Logs

As defined above, logs are records of events that took place at a given time.

Logging is a mechanism to collect logs from various input sources. Usually, logs are in raw format [2]. To gain real insights, we must parse these logs and query-type them. Often, logs are sent to an output tool which organizes and stores them in a central database. Ergo the user can search, analyze, these logs. To push the matter further, we can even create alerts based on logs.

This process defines what to log, how it should be logged, and how logs are shipped to an external system for aggregating and centralized storage.

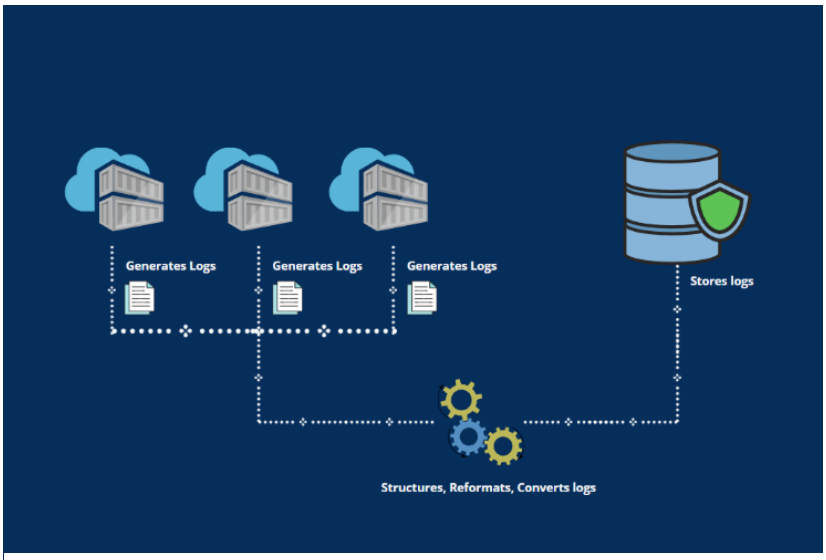


Figure 2—3 Logging process in a distributed system

2.4.2 Metrics

Metrics are the measurements of resource usage that are observed and collected through a system. These can be low-level summary usage (CPU, Memory, Storage, Network, I/O throughputs, etc.), or high-level types of data (Requests served, number of sold articles in a certain shop, etc.)

Usually, the most common metrics are the ones provided by the operating system. These metrics cover the usage of the underlying physical resources. Data related to these components is made available by the OS and might be forwarded to a metrics collection point.

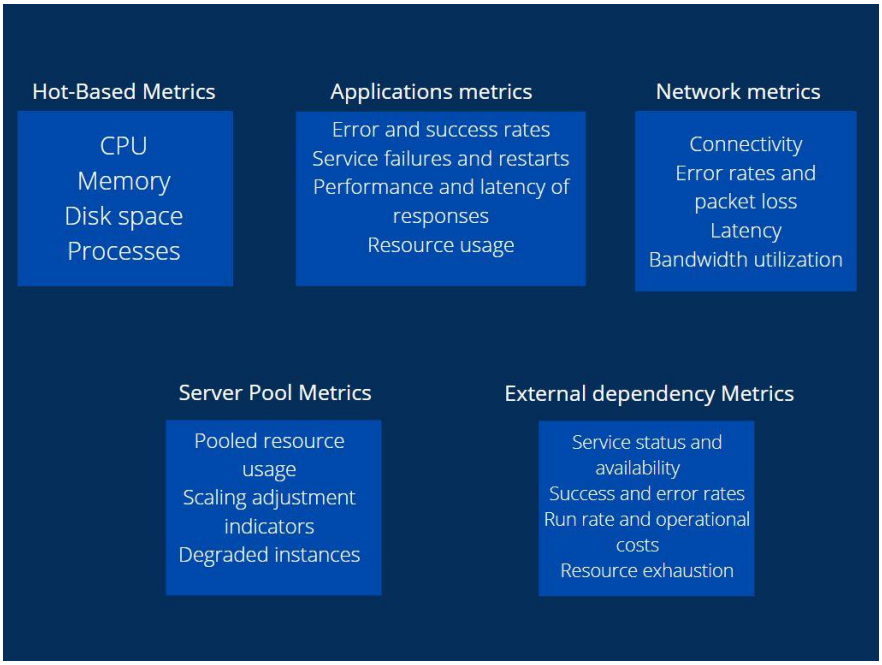


Figure 2—5 – Metrics classification

Moreover, any application can provide metrics; it is called instrumentation. In fact, it is becoming necessary and a good coding practice to instrument the application. This allows monitoring tools to gather metrics if the search mechanism, if pull-based, or push metrics in the other case to metrics endpoints. Which explains the existence of an enormous number of libraries that help developers instrument their applications.

Even though instrumenting one's code is a tiny action compared to an application development process, it has a beneficial effect. For starters, the insight provided by the metrics allows the exterior observer to the system to understand, not only its health, but also its behavior.

2.4.3 Spans & Distributed tracing

A trace is a mark, object, or other indication of the existence or passing of something. Projecting the linguistic definition of the term in our environment, we should be able to observe each request made between applications services from user's interaction till database action, with every detail (runtime, service name, annotations, tags).

External monitoring only tells us the overall response time, and in best cases, the number of invocations. However, in our case, each service manages a request by executing one or more operations.

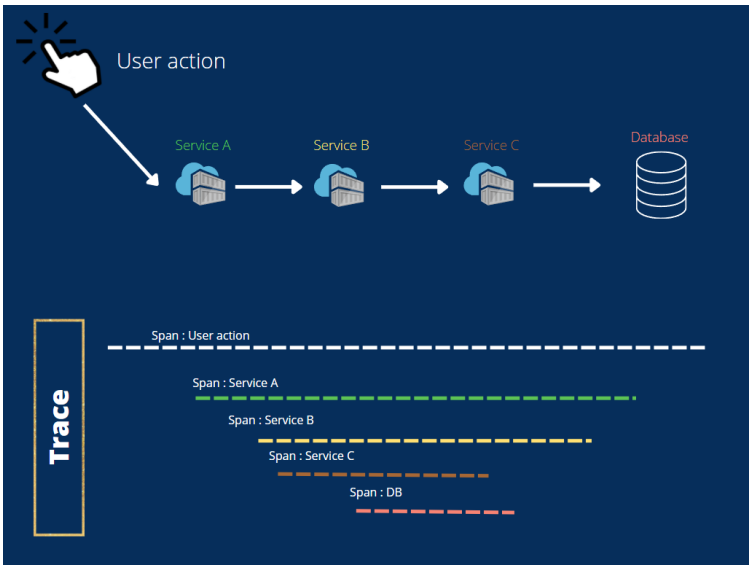


Figure 2—6 - Tracing

Hence the need for observing each service at a close range, since each service can execute database queries, generate logs, and publishes messages.

Tracing is a very important piece of Observability. It traces the calls between various micro-services. Priorities defined for different failures of service and the one with the highest priority caught and alerted in near real-time. A trace is composed of one to multiple spans.

The initial request is called a parent span (or root span) it stores the end-to-end response time of the request. To dive deeper into the subject, let us consider an example.

As shown in figure 5, we can consider a user's action as a root span. The request is initiated by the front-end service (if there is only one); the action will trigger other actions that will need interaction with other services of the application; for example, when a user request requires data from database, a series of services will be crossed till the database. Now each service will contribute a span as soon as it responds to a request from the parent span. It is called a child span.

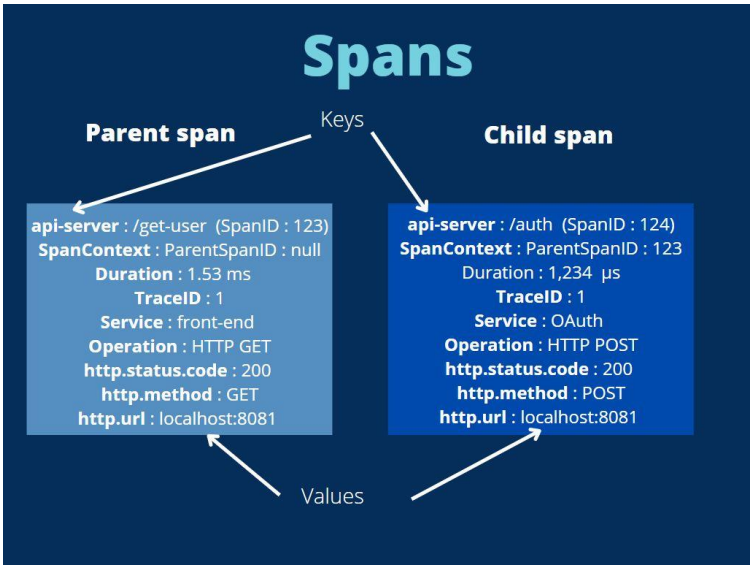


Figure 2—7 - Illustration of Spans and their tags

Linguistically speaking, a span the full extent of something from end to end. In our case, Spans are the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Each component of the distributed system contributes a span: a named, timestamped operation representing a step in the workflow.

Spans are identified by tags in addition to the name. Tags allow us to query, filter and comprehend trace data. A span context identifies the requests root span. It can provide request, service name, and duration metrics. Information like this can help us understand which service is taking longer than it should, and from there we can suggest different solutions depending on the found inconsistency.

3 Micro-Services architecture

3.1 Monolithic… A relic of the past

General Motors CEO, Mr. J. Welch said, “Change before you have to.” For a long time, desktop clients ruled the IT world; monolithic design patterns were satisfying the needs of the existing number of clients and serving them with reliable performance. Thanks to the revolution in gadget development, clients are not just desktop clients (known as thick clients) anymore. But every connected device (Smartphones, tablets, IoT objects, also known as thin clients). Such diversity has pushed the number of clients to increase drastically. Consequently, our applications must be scaled so it can serve client requests with reliable performance. Monolithic designs are not well suited for scaling, which scaling in monolithic designs requires multiple instances of the application to run. As a result, we would need to tweak the application because of future potential inconsistencies.

Besides, adding features has become a necessity for most applications. Unfortunately, the monolithic design pattern does not fit properly into the feature addition process. Since the application runs as a whole entity; from one side, we need to stop the application, add the feature, run the tests, reproduce the application, and redeploy it; this will cause a failure in availability. From another side, adding a feature is not as easy as it is said; in worst-case scenario, we might run into a variety of inconsistencies that obliges us to re-architect the whole application; otherwise, we would run into stability or consistency issues.

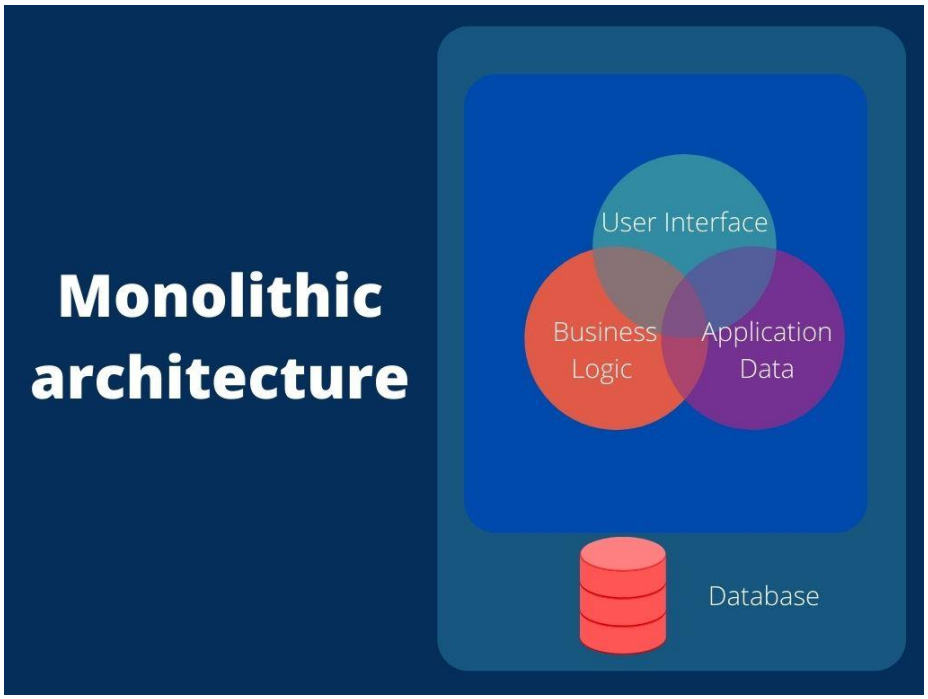


Figure 3—1 Illustration of monolithic architecture

Furthermore, cloud-computing services unlock fresh solutions. These novel solutions allow us to run our application on a rentable infrastructure (a brochure of variety infrastructure customization is provided by all cloud providers). Deploying a tremendous codebase, adopting it to modern technologies and keeping it up to date with market changes can be a challenging task.

To conclude, monolithic design patterns are quite useful in a lot of other use-cases but have shown insufficient to keep up with evolving technology.

3.2 Micro-services design architecture

As stated by Mr. Robert C. Martin's definition of the Single Responsibility Principle "Gather together those things that change for the same reason and separate those things that change for different reasons ", micro-services design is modularizing an application to small modules, so that each module focuses on a certain service of an application. In fact, these modules will be coupled loosely together to respond to a certain request. From a developer point of view, we can view our application as a collection of orchestrated small applications working together, hence the name micro-services. In contrast, from a client's point of view, the application is still viewed as a single entity. Figure 3-2 represents the design of a famous web application named sock-shop, developed, and maintained by *Weave-Works*[1].

Such an approach gave us numerous advantages over the monolithic pattern.

First, scalability is not an issue anymore. In fact, services run independently of each other, if one of them has reached its load limit, we simply scale it up by adding another instance of it.

Secondly, since the application is not a single point of failure anymore, pushing updates to a certain service of the application would not cause a downtime for the whole application, but the service in question. Such an advantage will allow us to increase our updates from 2-3 updates per week to 2-3 updates per day.

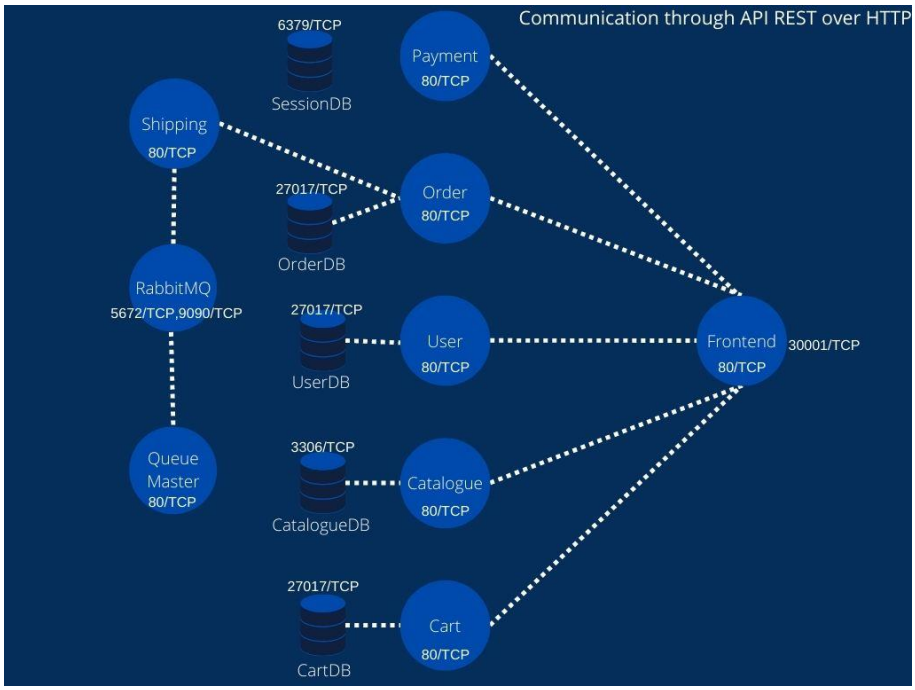


Figure 3—2 Microservices design of sock-shop web application

Lastly, modularizing the application will also affect the codebase of each service. With a lighter codebase and environment, it will be possible to deploy each of our services on a lightweight virtual machine called containers. In cloud environments, destroying & deploying an instance of a container will not exceed the seconds.

In conclusion, micro-services architecture, beneficial as it looks, is not a silver-bullet pattern. In fact, it increases application complexity and requires developer teams to have certain skills that they do not necessarily have.

[1] Weave-works: One of the first members of the Cloud Native Computing Foundation, founded in 2014, specialized in infrastructure management.

3.3 Containers

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. [3]

Concisely, containers are lightweight, standalone, executable packages of software. In addition, these dependencies include libraries, binaries & required configurations. In depth, a container runs on a host OS like a virtual machine with two key differences: First, it shares the OS kernel, so it boots faster using less memory. Second, it is hosted on a container runtime, offering the possibility to run multiple containers (several MB in size for each container) on a single server.

The name was not chosen loosely; in fact, shipping containers standardized globally, cargo transportation globally. Applying this to our field, it is safe to say that containers are also a standardization of application packaging over the cloud-environment despite the nature[2]& the provider[3] of the cloud.

It is only natural that questions like, what is the link between micro-services patterns and container technology? How does the one fit the other? comes to one's mind.

To answer these questions, let us agree on the fact that containers are a technology for packaging applications and micro-services are a design pattern. With that said, it is not always fit to choose the micro-service pattern when one is charged with the task of architecting an application. But if we are bound to opt for it, to fully benefit from the technology and the design pattern, we should consider the following: deploying an entire application to a single virtual machine represents a risk of a single point of failure independently of the design pattern chosen. However, modularizing the application to services that will later be deployed over containers is more beneficial on many levels, such us: scalability, availability, specific targeted improvements.

To conclude, both *the single responsibility* principle and *the single application per container* principle align to fit the couple micro-services design deployed on cloud.

3.4 Orchestration

S. Newman explained, in his book *Building microservices*, orchestration as follows: [4] "[…] With orchestration, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra […]". Orchestration is one of many architectural patterns for distributed systems that play a crucial role in architectural readability and workload management. For that reason, it can fit software development environment that is focused on long-term scalability. In an orchestration-driven architecture, software transactions are translated into workflows, managed, and identified by the orchestrator. In addition to these two tasks, the orchestrator manages payload construction, transformation, and protocol interoperability.

Furthermore, the orchestration pattern aims to centralize workflow management. Developers can query these workflows to identify individual service responsibilities without knowing the rest of the overall application architecture. In addition, this pattern offers more flexibility around deployments and changes, as it allows the different services of an application to start communicating synchronously and switch to asynchronous communication after that. An orchestrator manages transforming payload inter-services. Since all payloads are managed centrally, not only are changes easy to track, locate and manage, but also data can be converted to any needed format.

To achieve operational transparency, the orchestrator must be able to check and report system health of each service. Such information is much needed for system administrators. Let us consider an application with three services, in a scaled-up environment; each service will have ten replicas. Unless we did setup observability principles for our distributed systems, it can take forever to find out the faulty replica of a service deployment.

---

[2] Cloud nature: Public, Private, or Hybrid

[3] Cloud provider : AWS, GC, Azure, etc.

3.4.1 Kubernetes

"Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications." [5]

To look closer to Kubernetes use case, we need to consider a developer creating a multi-container application. He also needs to plan out how the parts fit and work together, what should happen in case of failure, what should happen in case of high load, etc. The following files represents an example of a multi-container application configuration files.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: carts
  labels:
    name: carts
  namespace: e-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: carts
  template:
    metadata:
      labels:
        name: carts
    spec:
      containers:
        - name: carts
          image: w3n3s/carts:2.0.1
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 200Mi
          ports:
            - containerPort: 4003
          volumeMounts:
            - mountPath: /tmp
              name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
```

Figure 3—5 Carts micro-service pod configuration file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users
  labels:
    name: users
  namespace: e-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: users
  template:
    metadata:
      labels:
        name: users
    spec:
      containers:
        - name: users
          image: w3n3s/users:2.0.0
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 200Mi
          ports:
            - containerPort: 4000
          volumeMounts:
            - mountPath: /tmp
              name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
```

Figure 3—4 Users micro-service pod configuration file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: products
  labels:
    name: products
  namespace: e-shop
spec:
  replicas: 1
  selector:
    matchLabels:
      name: products
  template:
    metadata:
      labels:
        name: products
    spec:
      containers:
        - name: products
          image: w3n3s/products:2.0.0
          resources:
            limits:
              cpu: 300m
              memory: 500Mi
            requests:
              cpu: 100m
              memory: 200Mi
          ports:
            - containerPort: 4002
          volumeMounts:
            - mountPath: /tmp
              name: tmp-volume
      volumes:
        - name: tmp-volume
          emptyDir:
            medium: Memory
      nodeSelector:
```

Figure 3—3 Products micro-service pod configuration file

To deploy the application we need pods, the smallest unit building unit of Kubernetes. The above files contain the configuration of such said components. These containerized applications are then stored (via YAML[4] config files) in a container registry. Next step will be to apply them to Kubernetes.

Kubernetes will then analyze the configuration, aligning the requirements of each pod. Next, it will locate the appropriate requested resources for running the container. As a last step of mounting the application, it will then pull the images from the registry hub, starts the container, and helps them connect to one another and to system resources.

The orchestrator work does not end here; in the contrary, this is but the end of the first task. Kubernetes will, afterwards, monitor everything, and when events diverge from a certain desired state, it will try to fix it and adapt the change. For example, if a container crashes, K8s will restart it; if the Kubernetes node

[4] YAML: stands for Yet Another Markup Language, however it is not a markup language, but a data serialization language that is often used for writing configuration files.

is out of resources, it will look for another node to run the containers, if application load spikes. Kubernetes will locate the service in question and make it scale up.

3.4.2 Service registry and microservices addressability in Kubernetes

As we mentioned previously, to deploy an application we need a set of pods. The next step will be to configure services to expose these pods and make communication between them possible.

Each microservice has a unique identifier (name) that is used to resolve its location (IP address). If we must keep a list of where to find each service things can catch fire quickly. To avoid this, we have a look-alike DNS mechanism that translates service names to addresses. For that, services must have a unique name independent of the infrastructure it is deployed on, so that its location is discovered. Such an approach implies that there is an interaction between the way a microservice is deployed and its discovery.

Along the same vein, when a container fails the orchestrator will automatically launch a new instance, and the service registry entry will be updated so that it shows where the service is running and what IP address was attributed to it. Service registry is a key part of service discovery. The registry is a database holding network locations of service instances. A service registry must be highly available and near real-time updated.

In conclusion, a service registry is a cluster of servers that use a replication protocol to keep consistency.

3.4.3 Communication between microservices in Kubernetes

Communication between microservices is a crucial process when it comes to orchestrating containers.

Client and services can communicate through distinct types of communications. Each one targeting a certain scenario, architecture, and goals. To supply such communications, we can consider four axes:

- **Synchronous**: Client sends request and waits for response from the service (client's code execution has nothing to do with this whether it is synchronous (thread blocking) or asynchronous (response will reach a callback event eventually)). Most known protocol for this is ***HTTP*.**
- **Asynchronous:** Client's code (or message sender in general) does not wait for a response. It just sends the message to a message broker queue (like ***RabbitMQ*** for example).
- **Single receiver:** One receiver only must process each request.
- **Multiple receivers:** Each request can be processed by zero to multiple receivers. This type of communications requires Asynchronous communication types. Based on message broker when propagating data updates between multiple microservices.

A micro-service-based application will often use a combination of these communication styles.

Most common tools for synchronous communication are ***HTTP REST APIs***, and ***RabbitMQ*** as an asynchronous message broker for asynchronous communication. [6]

3.4.4 Kubernetes & Observability

Distributed systems with many interconnected micro-services, are systems that require orchestration to ease its management. Although Observability pillars are relevant in Kubernetes; just collecting these data points and analyzing it; it is not enough to reach observability as we need it. It does not allow us to fully understand what is happening inside the containers.

A Kubernetes cluster is a complex multi-layered, ever-changing web of resources and services., finding the root cause of an error and fixing it would be a hard job, and a waste of resources.

For example, let us consider a 404 – Not found error. It was generated from a front-end service which runs on a certain pod of containers. The error might have been caused by a back-end service that runs on another pod of containers. In this case, finding exactly which pod is unhealthy among the large pool of pods, would be achievable by analyzing Kubernetes metrics. However, if the error does not stop the service from running, from Kubernetes point of view, the container would still be healthy. This is exactly why; Kubernetes metrics are not enough to reach the observability as we defined it above.

This can be avoided if we have deployed an observability framework. This framework will provide us with visibility (through collected metrics and spans visualization) and information (through collected logs) that will help us track the error and fix it.

4 Observability Tools

What is an observability tool?

An observability tool is a tool designed to gather information that is relevant to understanding system behavior. and application via monitoring metrics, logs, and traces. In contrast to individual monitoring tools, observability tools enable a system administrator to have constant insight and receive continuous feedback from their systems. This approach to monitoring and logging provides actionable information to system administrators faster than tools limited to monitoring or gathering logs.

Observability platforms help administrators understand system behavior and predict outages or problems before they occur. They use observability tools to act proactively and prevent system problems.

4.1 Prometheus

4.1.1 What is Prometheus?

Prometheus is an open-source monitoring and alerting toolkit originally built at SoundCloud. It is now an independent project since it joined the CNCF as the second hosted after Kubernetes. It is written in Go. It provides a multi-dimensional data model, a query language called PromQL, and integrates distinct parts of the application. [7]

Briefly, Prometheus collects and stores timestamped metrics alongside optional labels. To manage such a huge amount of data, it uses a time-series database and stores corresponding labels as key value pairs.

4.1.1.1 Components & architecture

Most of Prometheus's components are written in Go language, for its easiness of build and deployment as static binaries. Prometheus distributed components are as follows:

Prometheus server, this entity represents the main server of Prometheus. Its main activity is to scrape and store time series data.

Client libraries, these are the libraries that clients need to integrate in their code environments to make their applications expose metrics available for scraping. Metrics can be exposed via an HTTP endpoint on each service instance. CL[5] are available in multiple programming languages (Go, Java, Python, .NET, Node.js, Bash, C, etc.)

[5] CL: Client libraries

Push gateway, some components cannot be scraped, due to their short-lived jobs (batch jobs or service-level jobs), they might not live long enough to be scraped. So, we must push metrics making them available at some endpoint[6] for scraping.

Exporters, this component is what makes Prometheus widely used. There are libraries and servers that help third party systems export their metrics as Prometheus metrics, hence the name exporters.

Alert Manager, since Prometheus scrapes metrics, it also allows us to configure alerts on these measurements.

To manage data in time-series formats in an effective way, Prometheus dev teams developed PromQL. A Prometheus's original query language offering the possibility of aggregating and filtering data, thus providing a very important pieces of information. PromQL is designed to work with time-series databases. Therefore, it provides time-related query functionalities.

Prometheus uses Grafana to visualize its stored data. The couple provides the necessary insight to reach observability.

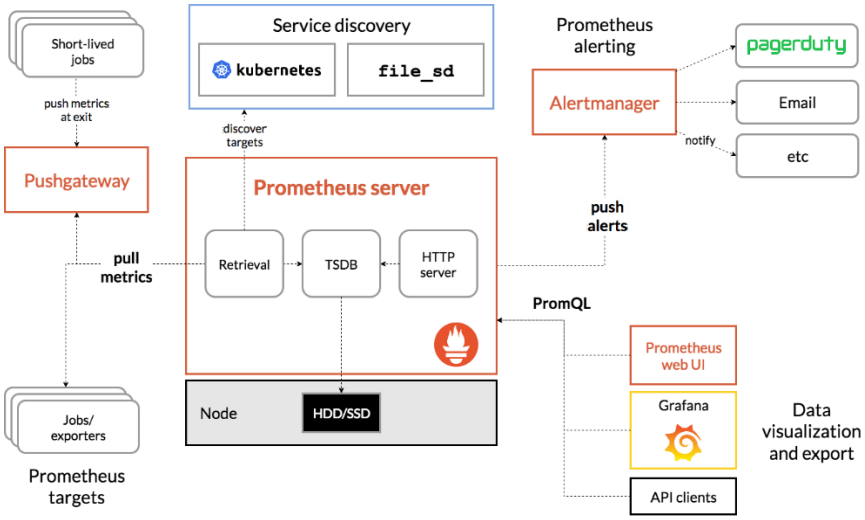


Figure 4—1 - Prometheus Architecture (source: https://prometheus.io/docs/introduction/overview/)

Prometheus scrapes metrics in two manners, directly through pulling metrics from instrumented jobs or from the push gateway. Then it stores scraped samples locally and applies rules over it, either to aggregate it and store new time series, or to generate alerts and push them to alert manager. Grafana or other visualization tools can visualize collected data.

4.1.1.2 Key Features

Initially, Prometheus uses multi-dimensional data model with time series data identified by metric names and key/value pairs. It stores streams of timestamped values of the same metric and the same labeled dimensions. Prometheus might also generate temporary derived time series as a response to users' queries.

Secondly, PromQL, a Prometheus flexible query language, invented to leverage the data dimensionality provides users with the possibility to aggregate and query over data to get more focused and enhanced visibility at of the desired metrics. Prometheus will retain a minimum of three write-ahead log files. High-traffic servers may retain more than three WAL[7] files to keep at least two hours of raw data. [8]

[6] Scrapable endpoint: Prometheus endpoint usually are named /metrics

[7] WAL: stands for write-ahead log. A mechanism used by Prometheus to secure memory against crashes.

Thirdly, Prometheus storage is based on an autonomous single server node. Thus, no distributed storage. In other words, time-series databases store data formatted under an optimized format for longer retention. Although the default Prom release uses local storage, it also provides interfaces to integrate remote storage systems. However, this approach has several drawbacks, such us, lowering data retention policy to avoid costly volumes; each Prom's server had to have a big enough dedicated volume to operate (increases costs) and any increase in metrics pushes central Prom's server to scrape more data which leads to timeouts.

Prometheus is effective in some use cases, but if the environment scales up high enough, managing it becomes challenging.

4.1.2 Contribution to Observability

As mentioned in Observability chapter, one of the three pillars of observability is metrics. Prometheus is the most widely used open-source metrics collection tool.

Prometheus client collects default metrics such us, CPU consumption, I/O throughput, HTTP requests rate, etc. But it also gives us the possibility to create our own metrics and register them to a registry. With this said, we can consider collecting metrics of a higher level, metrics that do not belong to the infrastructure level, but to the application level.

These metrics can be, the rate of registered users, the rate of dropped carts, etc. Prometheus does not only scrape these metrics, but it also visualizes them onto a variety of dashboards in near real-time. Furthermore, we also have the possibility to configure alerts on these metrics.

To conclude, Prometheus with the right configuration, customized metrics and set alerts, can be the right tool to cover the metrics pillar to reach system observability. In the last chapter, we will learn more about Prometheus configuration, how to add customized metrics, and how to visualize these metrics.

4.1.3 Glance at Prometheus' & Grafana's dashboard

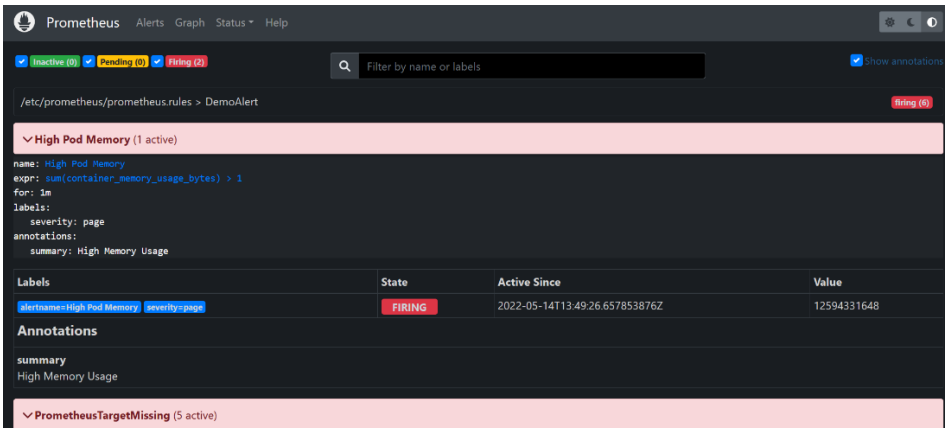


Figure 4—2 - Prometheus Interface for managing alerts

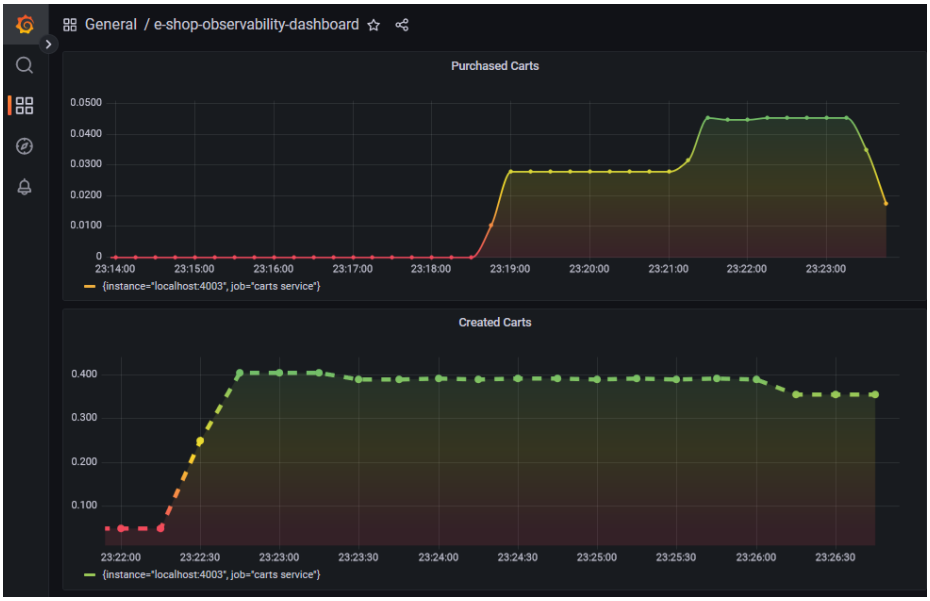


Figure 4—3 - Prometheus Interface for viewing endpoints status

4.2 Thanos

4.2.1 What is Thanos?

On its website Thanos is defined as an Open source highly available Prometheus setup with long term storage capabilities [9].

Thanos linguistically comes from Greek “Athanasios,” which means immortal. Thanos is not a monitoring and logging tools by itself, but it is a set of components that can be composed into a highly available Prometheus with unlimited storage capacity.

Thanos was brought to existence by the British gaming technology company “Improbable” as a CNCF project. On their blog, they revealed the project goal saying: *"to seamlessly transform existing Prometheus deployments in clusters around the world into a unified monitoring system with unbounded historical data storage”* [10]. With that said, it is safe to say that by adding Prometheus to Thanos, system administrators can build highly available metric systems with almost unlimited storage. Once deployed, Thanos provides global query vision, access to historical data and high accessibility as we can see on the following illustration (figure 4-4).

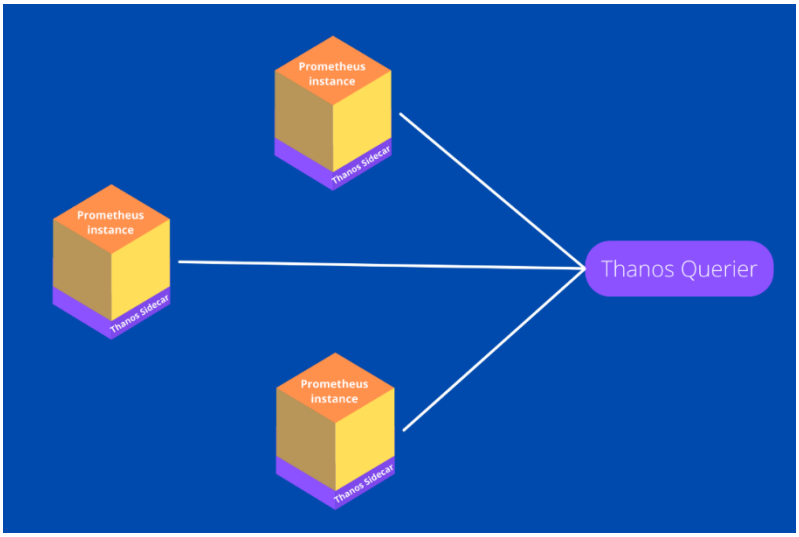


Figure 4—4 Thanos : a Prometheus federation

To explain better what Thanos does offer more than Prometheus we are going to detail the feature we mentioned above. First, global query view; Prometheus uses sharding[8] approach so it can access all our data in the same API or UI, Thanos does the same thing by collecting metrics from multiple Prometheus instances and allows us to visualize & query them. Second, historical data storage; one of Thanos's sidecar responsibilities is to watch new block scraped and added by Prometheus instances then adds the block in question to storage bucket to be uploaded to a cloud of our choice, this way the metrics persistence challenge is solved. Third, high availability; this is reached by Thanos sidecar and querier, if we watch closely the figure 4-4, we can notice that the querier can de-duplicate Prometheus metrics from each sidecar and merge them together.

4.2.2 Components & architectures

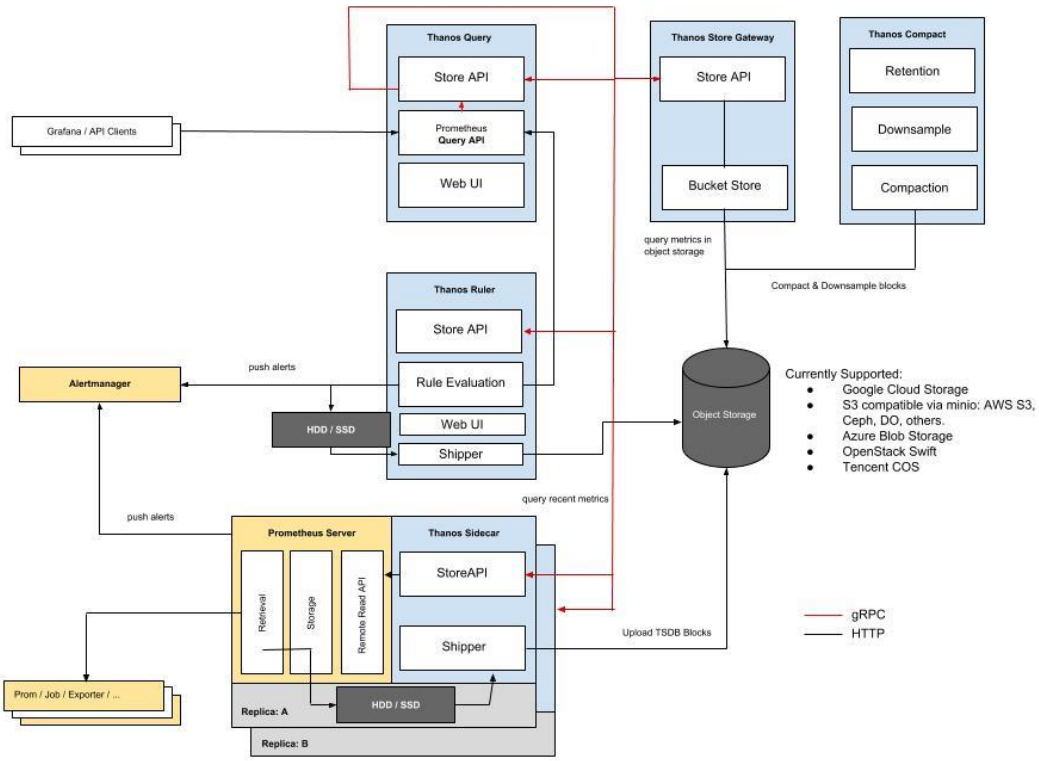


Figure 4—5 – Overview of Thanos architecture (Source: https://thanos.io/v0.6/thanos/getting-started.md/)

Thanos sidecar, the sidecar is the main component of Thanos. As we can see, Thanos is deployed alongside the existing Prometheus server. This component's task is to collect Prometheus data and upload it to a storage component.

Thanos ruler, the ruler's role is to push alerts to alert manager and ship logs to storage components. It provides a Web UI (same as Prometheus).

Thanos querier, it is a layer added upon Prometheus's PromQL API. This component has its own web UI. Also, it has a store API (gRPC API is used instead of HTTP API) to provide near-real time metrics.

[8] Sharding: It divides the targets Prometheus scrapes into multiple groups, small enough for a single Prometheus instance to scrape

Thanos Store Gateway, it is the official communicator for Object storage third-party components (it can be S3 buckets if AWS or under other bucket formats for other cloud storages).

Thanos compactor, this component controls three data aspects: Retention, compaction[9], and down sampling[10]. For example, depending on our application's nature and traffic, we can schedule retention configuration changes to optimize our storage use. Back to the compaction point, for the same situation, the process of compaction helps reduce the number of blocks to store and index indices.

4.2.3 Contribution to Observability

Thanos transforms Prometheus into a highly available metrics platform with unlimited metrics storage. Since the latter is a metrics collection and visualization tool, Thanos is, with no doubt, the same. To sum up what has been said before, both Prometheus and Thanos cover the metrics pillar of Observability getting us one step closer to a fully observable system.

4.3 Jaeger

4.3.1 What is Jaeger?

Jaeger is a German word that means Hunter. Jaeger is a distributed tracing system developed by Uber Technologies. It is used for monitoring and troubleshooting applications through collecting spans. [11]

The system focuses on five fundamental areas as follows:

- **Distributed context propagation**, to understand this, we need first to define what a context is. As we mentioned in previous chapters, spans are created when requests are made or answered to. As a result, a context can be a micro-service name or an operation type (database interaction, cloud interaction, business-logic). A request will cross multi micro-services to finally comeback to original sender, and with each micro-service it crosses, multiple spans will be generated with different context, so the context needs to be propagated with the request that we are able to trace it, hence distributed context propagation. The figure below is an example of distributed context propagation in an e-commerce application, based on three micro-services. The application will be explored in the next chapter.

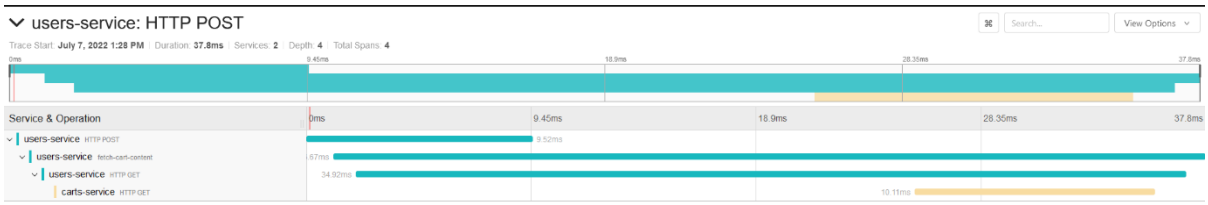


Figure 4—6 Illustration of distributed context propagation

[9] Data compaction: reduction of data elements.
[10] Down sampling: cutting the data to smaller chunk blocks.

- **Distributed transaction monitoring**, a transaction in our field is an end-to-end operation. As spans can contain a lot of information useful for understanding how the micro-services are doing their job, we can also add request response attributes (successes or errors), events, logs, and customed tags. Afterwards, we can query over our spans through all these options.

- **Root cause analysis**, as we mentioned above, a span can contain pieces of information that might be useful to understand an error if it happens. With that said, mixed with the two above, even if our system is scaled-up, localizing an error would take seconds thanks to distributed context propagation and distributed transaction monitoring. Analyzing what is wrong, becomes a matter of minutes thanks to the traces and the contents of the collected spans.

- **Service dependency analysis,** in a micro-services design pattern, the services are meant to communicate with each other. Any communication will issue a request and response, therefore a dependency. The following figures illustrate service dependency.

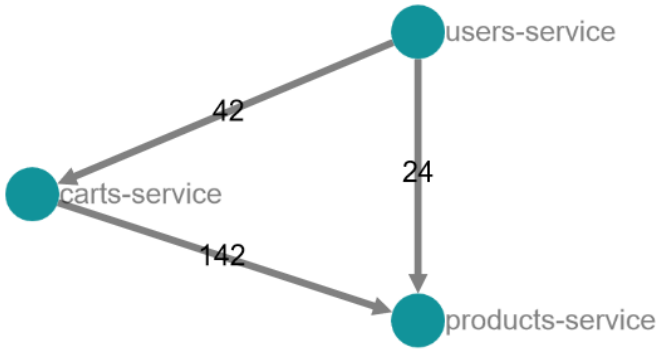


Figure 4—7 Service dependency analysis of e-shop minimal application

A typical Jaeger installation at Uber processes around several billion spans per day. So, the figure is really a 0,0001% of the real capacity of this Jaeger feature.

4.3.2 Jaeger components

Jaeger is composed of three main components. An agent, a collector, and a service for querying over collected data.

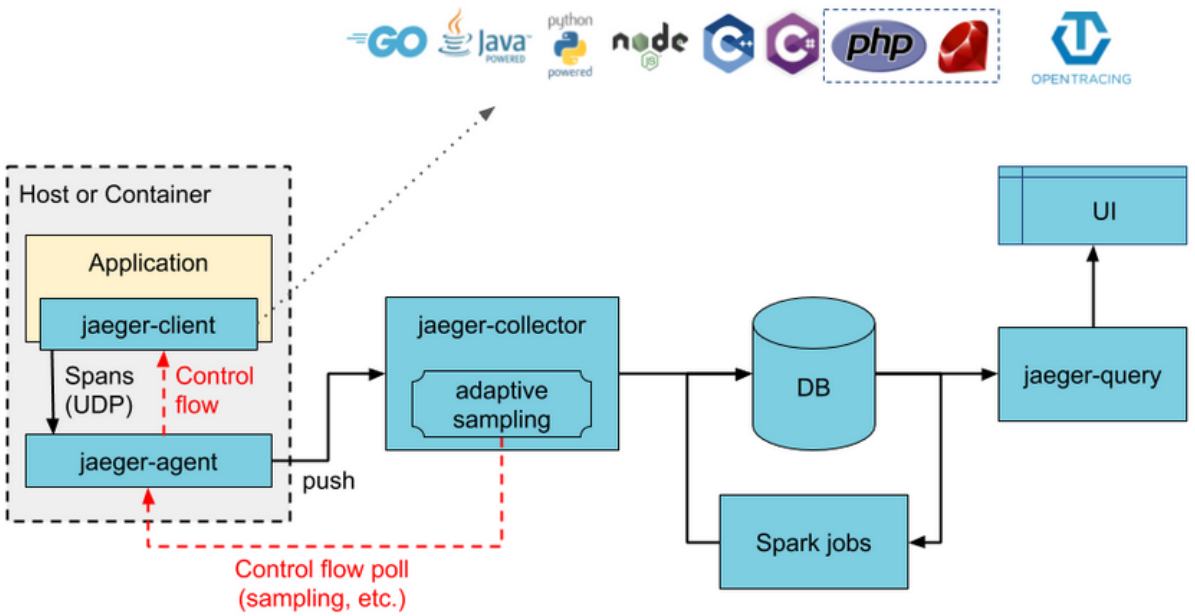


Figure 4—8Illustration of Jaeger components architecture. Source: https://www.jaegertracing.io/docs/1.36/architecture/

Jaeger-agent, client libraries included in the application's code to generate spans and build traces, expects a local jaeger-agent process on each host. The agent work is to expose endpoints for data collection. It also provides the ability to configure static list of collector addresses, as for the possibility to connect point to point to a single collector. [11]

Jaeger-collector, stateless components giving us the capability to run multiple parallel instances. Except for storage locations, collectors do not require any further configuration. Jaeger provides two possibilities for storage: local in-memory storage, as the name suggests, such a storage would not be able to hold for long in a production environment, however useful for testing environments. Second is the cloud persistent storage required by collectors. Jaeger supports multiple distributed storage backends like Cassandra[11] or Elasticsearch[12]. [11]

Jaeger-query, for this component has two tasks: Querying over data and visualizing it. This service is also stateless and so often run behind a load balancer such as NGINX. [11]

[11] Cassandra is an open-source NoSQL distributed, scalable and highly available database developed by Apache
[12] Elasticsearch is a scalable search and analytics engine. It is the heart of the Elastic-stack

4.3.3 Contribution to Observability

Jaeger is a distributed tracing system that collects and analyzes spans to provide us with enough information so that we can understand what the system is doing at every request and response. As we have mentioned, in the Observability chapter, spans are one of its three pillars. Jaeger is a tool that covers the spans pillar.

Designing our system to generate distributed propagated spans is crucial to reach full observability. For example, if we want to see the most followed paths by users so we can qualify the navigation of our web application, querying over spans can help us do this.

4.4 Datadog

4.4.1 What is Datadog?

Datadog is a software and SaaS proprietary to Datadog, Inc. that was founded by Olivier Pomel and Alexis Lê-Quoc in 2010.

Datadog is a monitoring, security, and analytics platform for cloud applications. It brings together the three pillars of observability: logs, metrics, and spans so it makes our application, infrastructure, and third-party services entirely observable. This software can be useful for any IT cloud application. In addition to its capacity to enable observability, it also provides analytics of all layers of an application, starting from infrastructure, arriving at real time BI[13].

Besides, Datadog is used also by DevOps teams that it enables them to configure alerts based on issue level. The alerts notifications can be received via a variety of tools such as PagerDuty, Slack, or e-mail. Likewise, it also enhances CI[14] visibility, event tracking and anomalies detection.

Moreover, Datadog uses a REST API allowing it to integrate with a variety of services, tools, and programming languages including Kubernetes, Puppet, Ansible[15], Ruby, NodeJS, .NET, PHP[16], etc.

Furthermore, the software centralizes everything into one highly customizable platform making the system administrators observe the entirety of the system with less clicks, in some cases, on one dashboard.

[13] BI: Business Intelligence
[14] CI: Continuous Integration
[15] Puppet & Ansible are automatization & configuration management tools
[16] Famous programming languages & frameworks

Datadog platform has a canvas of modules as the figure below shows:

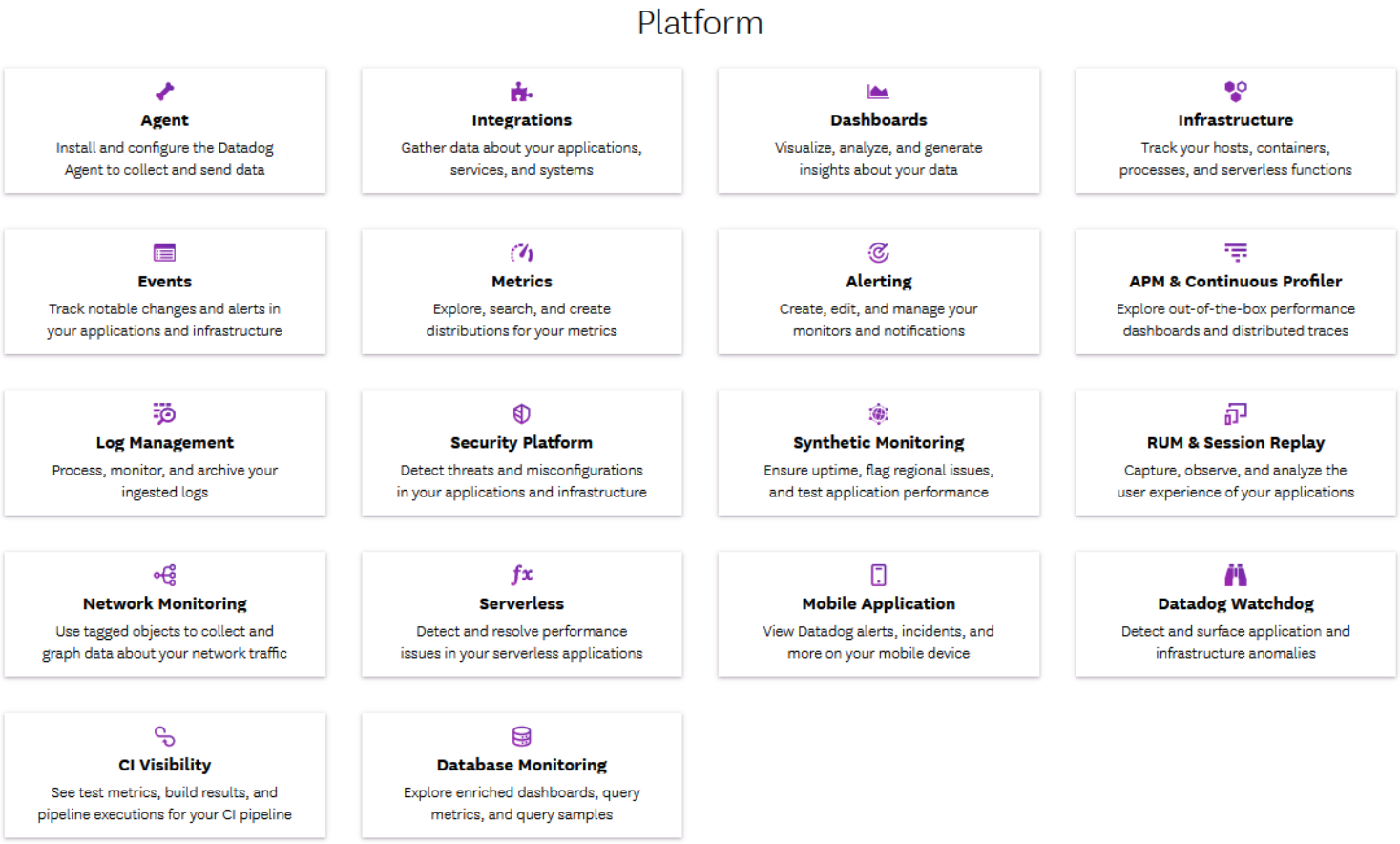


Figure 4—9 Datadog platform modules. Source: https://docs.datadoghq.com/

4.4.2 Deployment Strategies & Pricing

Datadog is available for deployment on premises as it supports multiple operating systems such as Linux based distributions, Windows, OpenStack, and Oracle cloud. As well as it is deployable as a SaaS[17]. When deployed as a SaaS it eliminates the tools maintenance, capacity scaling, updating or management operations. These benefits the team to focus more on the core product. [12]

Since Datadog is proprietary, it is interesting to detail pricing. In fact, Datadog does not charge for the whole software, but each module or integration might or might not be charged. This figure presents the plans as suggested by the company:

[17] SaaS stands for Software as a Service

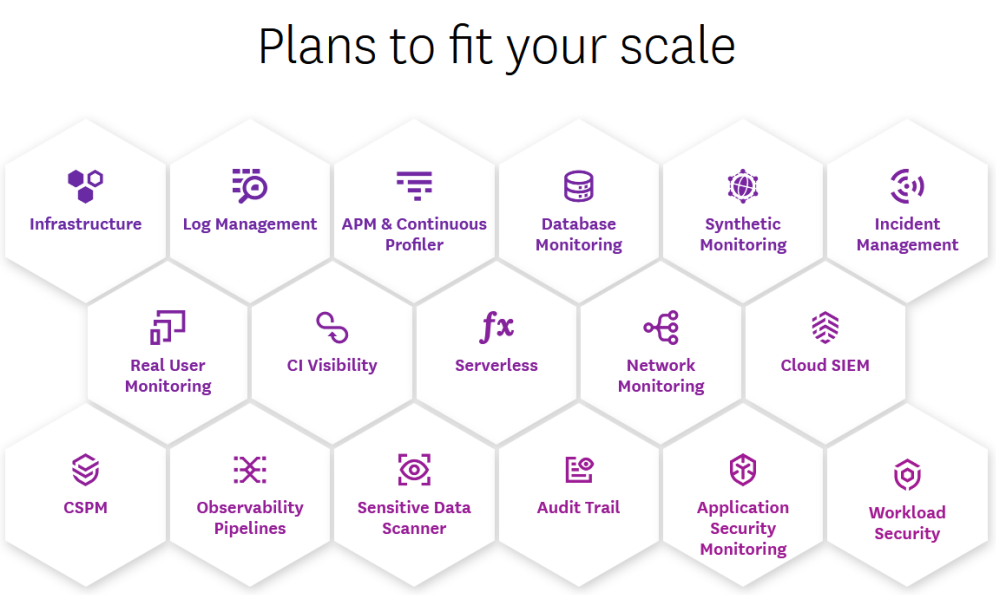


Figure 4—10 Datadog suggested pricing plans Source: https://docs.datadoghq.com/pricing

To give an overview of the pricing model followed, we will pick three modules. Starting with the infrastructure module, for $15 per host monthly, we have all standard features, plus 15 months of data retention, unlimited alerts, ten per host monitored containers (automatic selection), one hundred custom metrics, one hundred events. However, for the enterprise version of the same model, for $23 per host monthly, twenty per host monitored containers (custom selection), two hundred per host custom metrics, one thousand events.

For Log management module, Datadog charges $0.10 per GB ingested or scanned, $2.50 for a month period retention per million log event per month.

For APM[18] & Continuous Profiler, the company charges $31 per month per host for APM only. To benefits of both features the cost increases to $40 per month per host.

In conclusion, Datadog is a useful and affordable tool for big company with no financial pressure, on the other hand, it can be quite expensive for start-up companies. [13]

[18] APM stands for Application Platform Management

Datadog dashboard overview

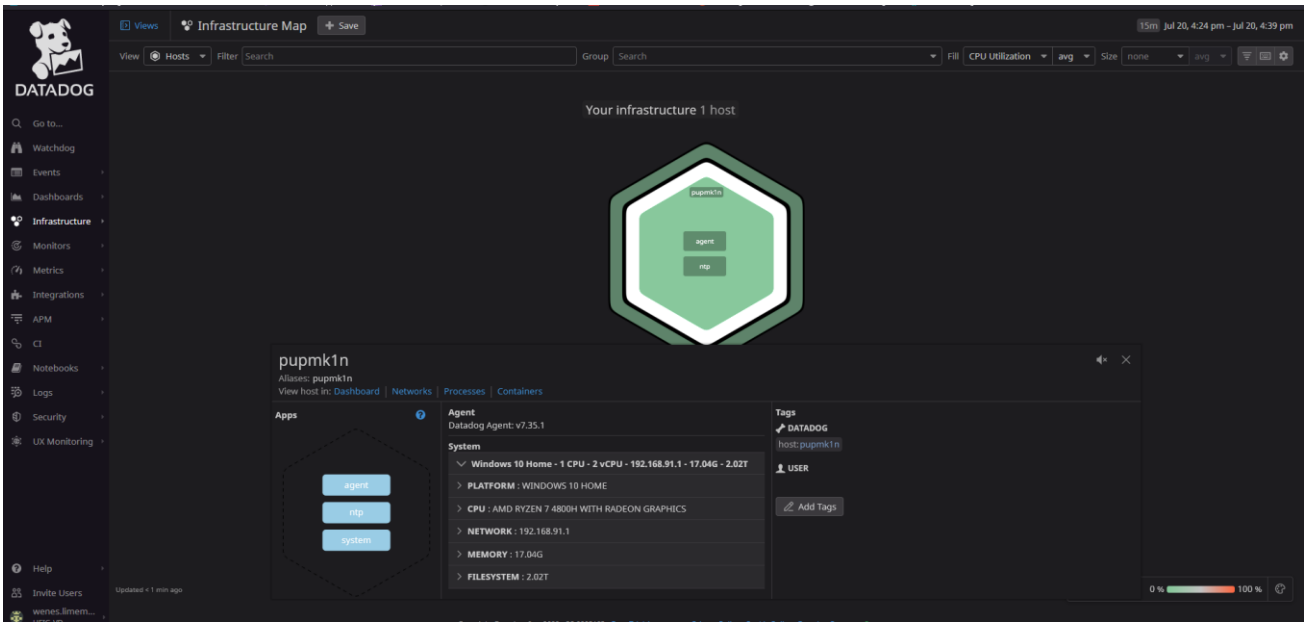


Figure 4—11 Datadog infrastructure dashboard

Figure 4—12 Datadog metrics dashboard

4.4.3 Contribution to observability

As we have said before, Datadog is a software and SaaS that enables observability on distributed systems deployed on-cloud or on-premises.

This software covers telemetric data collection, logs, metrics, and spans from various sources in an observed distributed system. The captured data is then, queried over, converted, and computed for different use-cases. Lastly visualized on a very customizable dashboard.

4.5 Comparing Observability tools

To wrap up the four tools & software we have seen so far, we have elaborated these two charts. The charts compare the previously presented tools and compare them based on many criteria.

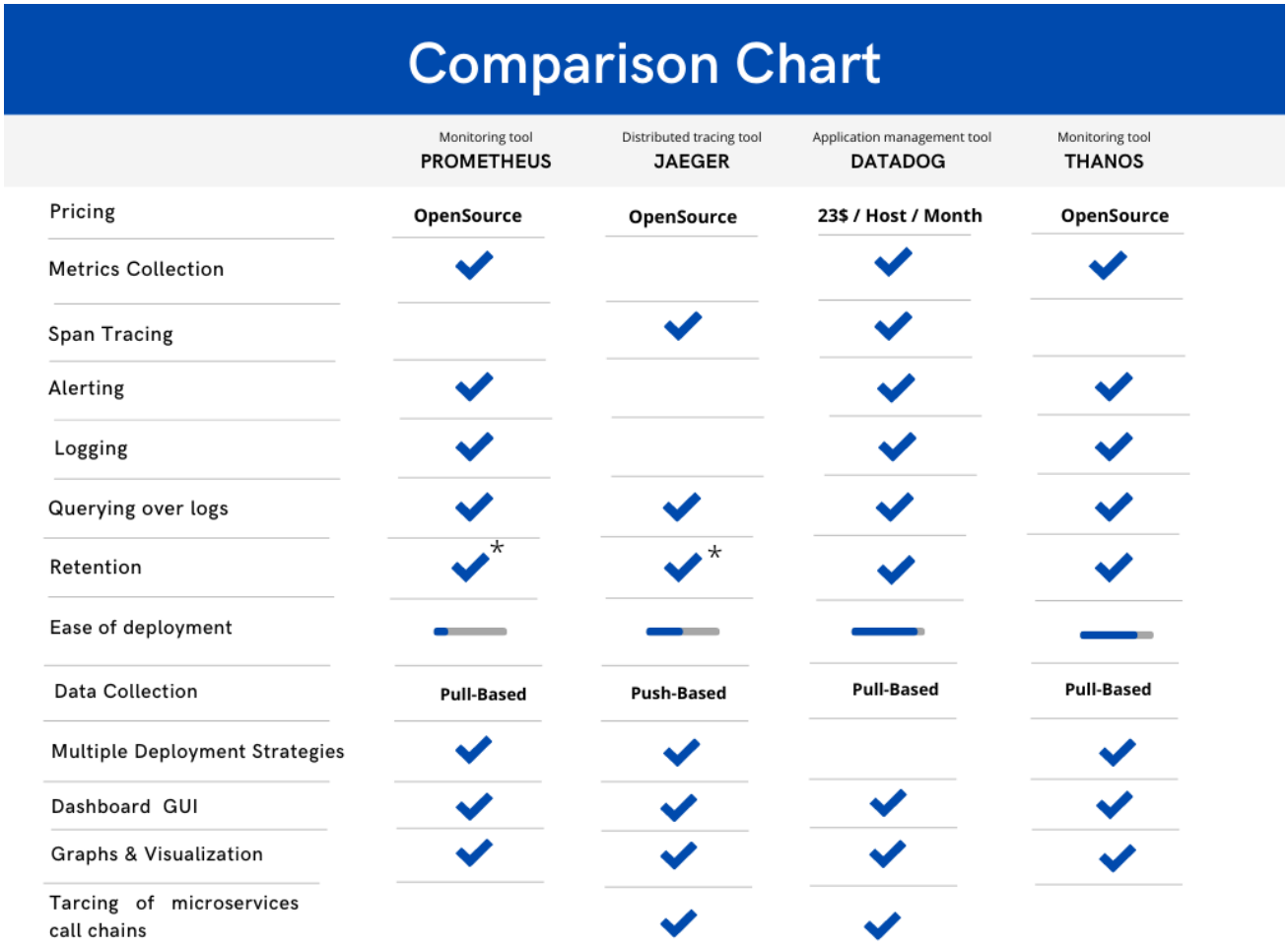


Figure 4—13 Comparison chart

Retention is a term that is usually related to data. Its value represents the maximum time a stockage service can hold to our data. In our case, Thanos & Datadog offer a variety of stockage integrations, allowing us to safely say that the retention for these two is almost limitless. In contrast, Prometheus only offers us a standard of six hours of retention time, and four weeks for Jaeger.

Deployment strategies mean the diverse ways we can integrate the tool with an instrumented system. In fact, Jaeger offers diverse ways to deploy its agent. We can choose between an all-in-one ready-to-use docker container, or deploy the agents to Kubernetes using Helm, or run it on our host directly. Same goes for Thanos & Prometheus. On the other hand, Datadog provides us with a highly customizable agent to be installed either as a cluster-agent or as a software.

Tracing of microservices call chains is the capacity to follow each request across the micro-services and gather its related spans and traces.

5 Observable systems

5.1 Sock-shop: Existing sample application

5.1.1 Overview

Sock-shop is, as the name suggests, a user-facing part of an e-commerce website that sells socks. It is a micro-services-based web-application developed by Weaveworks in 2017 as a demonstration application for micro-services design architecture and cloud-native technologies. It is maintained by Weaveworks and Containers Solutions. It is registered under the Apache License, Version 2.0 [14]. The figure below is a screenshot of the user-interface of the sock-shop application while it was deployed on local Kubernetes node.

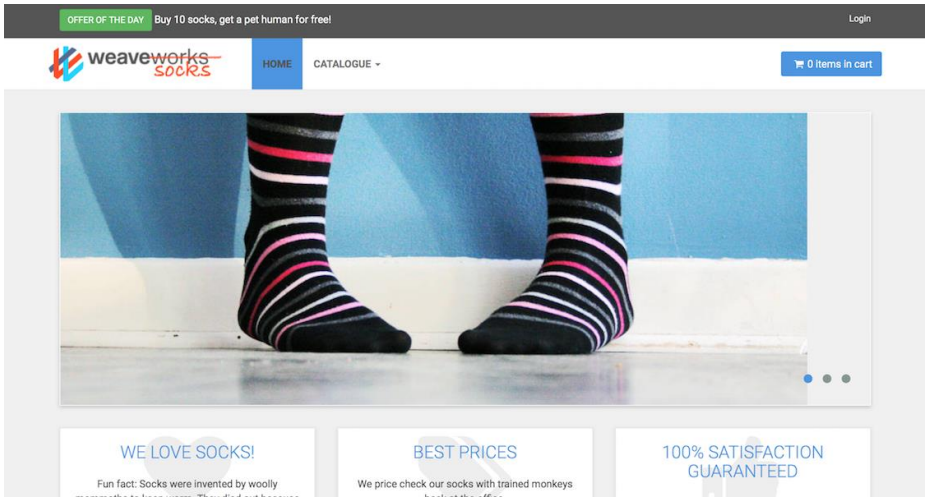


Figure 5—1 Sock-shop user-interface screenshot

5.1.2 Design & Layers

The application was modularized to micro-services, as the next figure shows.

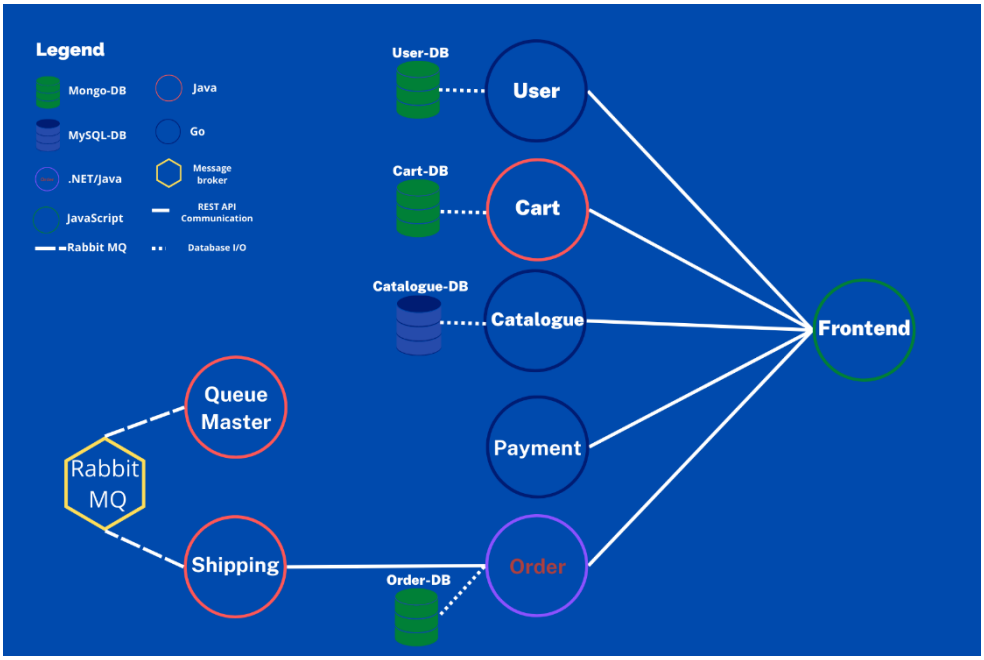


Figure 5—2 Sock-shop design

Micro-services are not written in one language, and that is the interest behind this design pattern. In fact, Frontend micro-service is written in JavaScript, whereas Order & Catalogue micro-services were written In Go-Lang.

Some microservices are connected to databases, such us Order, User, Catalog and Cart. However, the Sessions database is a standalone micro-service that a session is stored and treated independently of the user. The services communicate with each other using REST

APIs over the HTTP protocol for synchronous communications and RabbitMQ for asynchronous communications.

5.1.3 Infrastructure & Deployment

Sock-shop is deployable over Kubernetes. The deployment files can be found at their official GitHub repository. We are using Docker Desktop node for Kubernetes, but it should work on any Kubernetes cluster.

5.1.3.1 Sock-shop namespace deployment

A namespace is a Kubernetes unit used to isolate containers from other namespaces. Kubernetes own containers run in an isolated namespace named *kube-system* and *kube-node-lease.*

The namespace is created as follows:

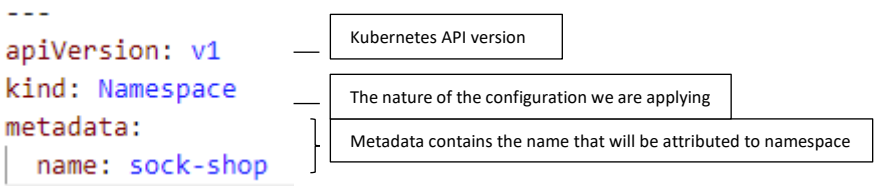


Figure 5—4 Sock-shop namespace deployment YAML file

5.1.3.2 Sock-shop micro-services deployments

To deploy a micro-service, we need to create a pod and then expose it with a service at a certain port. In Kubernetes, deployments allow us to describe a desired state and let the deployment controller manage the state changes. We can use deployments to create replica-sets, declare new state of pods, scale-up the deployment. First, let us start with the deployment file:

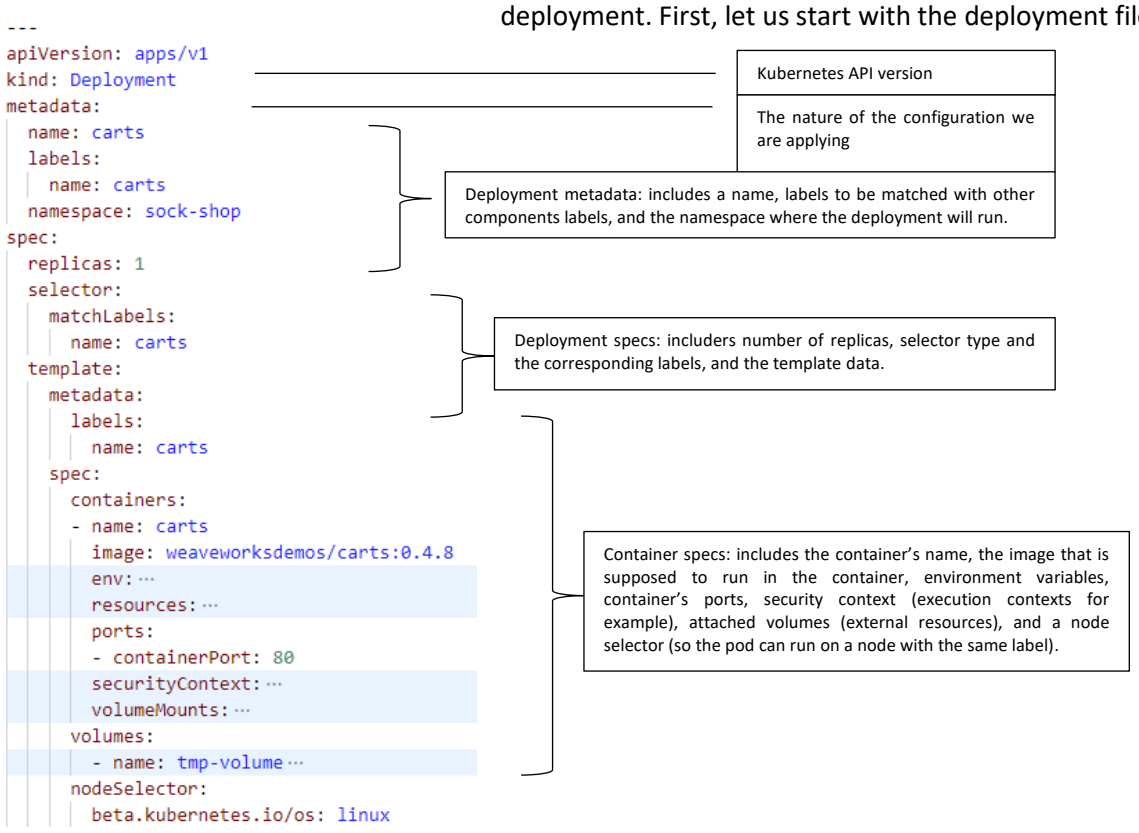


Figure 5—5 Deployment file for carts micro-service

With that applied to Kubernetes, the next step is to expose the deployment with a service as the following file shows:

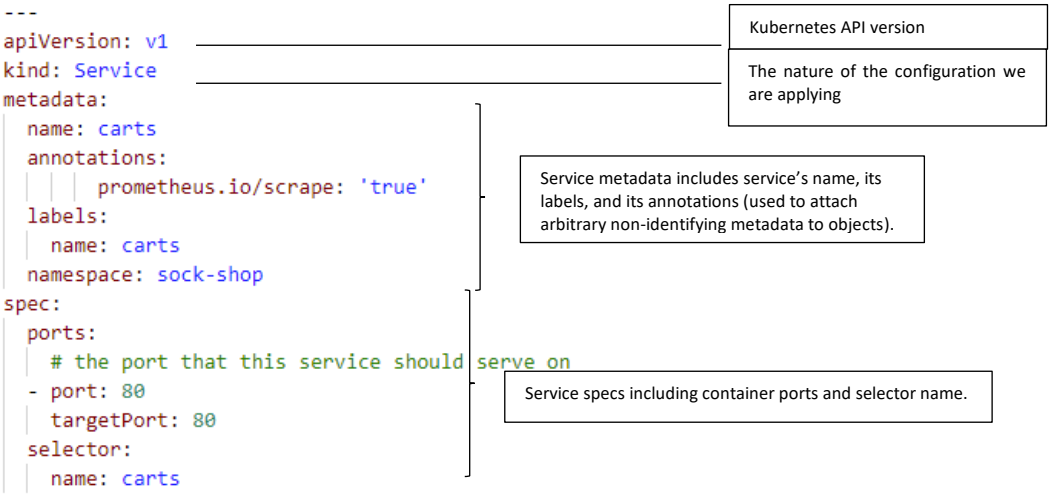


Figure 5—6 Carts service YAML file deployment

As we have seen in the previous figure 5-2, some micro-services are connected to a database, such as the carts micro-service. For that, we need to deploy a container containing the database related to the service. The following file contains a deployment example of carts database.

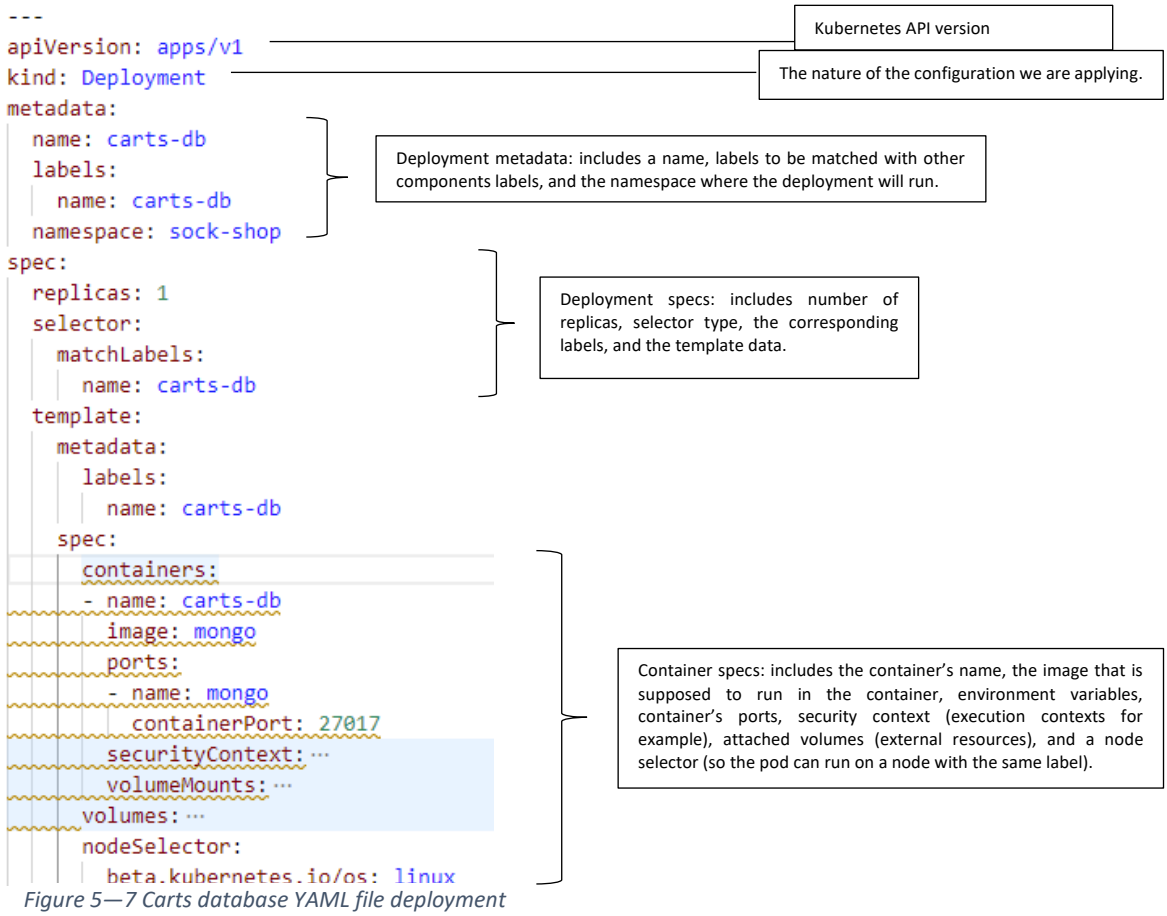


Figure 5—7 Carts database YAML file deployment

Following the same strategy, we can deploy the rest of the application. To do that, we have two possibilities:

First, deployment files can be found in *sock-shop* => deploy => Kubernetes => manifests to deploy components separately as the next tree shows:

Figure 5—8 Tree of sock-shop repo folder part 01

Second, using one file that has all the configuration summed up together. The file can be found at sock-shop repo => deploy => complete-demo. yaml.

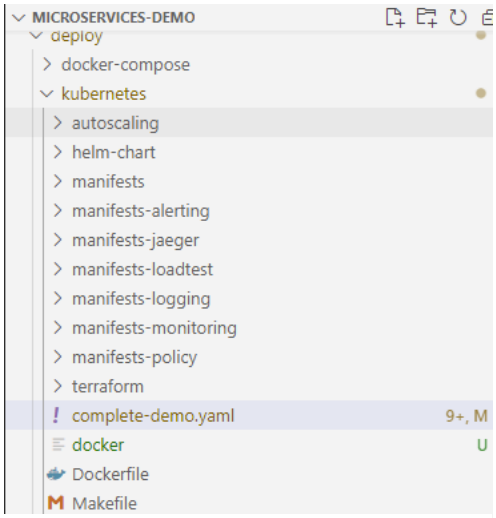


Figure 5—9 Tree of sock-shop repo folder part 02

Once the application is deployed, we can check on the deployment status using a terminal. The output should look like the following figure:

```
└>>ı kubectl get all -n sock-shop
NAME                                  READY   STATUS    RESTARTS        AGE
pod/carts-7bbf9dc945-6np67            1/1     Running   1 (8m55s ago)   24h
pod/carts-db-67f744dd5f-lct48         1/1     Running   1 (8m55s ago)   24h
pod/catalogue-5847ff7b9d-cq56p        1/1     Running   1 (8m55s ago)   24h
pod/catalogue-db-554b5765cf-5kgw5     1/1     Running   1 (8m55s ago)   24h
pod/front-end-566bc5cb45-4qqg8        1/1     Running   1 (8m55s ago)   24h
pod/orders-759d77885f-qrf45           1/1     Running   1 (8m55s ago)   24h
pod/orders-db-bcc65d8b7-688rx         1/1     Running   1 (8m55s ago)   24h
pod/payment-bf57b494f-klszt           1/1     Running   1 (8m55s ago)   24h
pod/queue-master-5c568f48f4-bb5gz     1/1     Running   1 (8m55s ago)   24h
pod/rabbitmq-c9c9d6567-9qxmj          2/2     Running   2 (8m55s ago)   24h
pod/session-db-77fb5f98fc-sk779       1/1     Running   1 (8m55s ago)   24h
pod/shipping-76b9956679-vfv96         1/1     Running   1 (8m55s ago)   24h
pod/user-b8c4c9799-8rrcx              1/1     Running   1 (8m55s ago)   24h
pod/user-db-59bc4d8969-zffnd          1/1     Running   1 (8m55s ago)   24h

NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)              AGE
service/carts          ClusterIP   10.108.149.119   <none>        80/TCP               24h
service/carts-db       ClusterIP   10.97.250.83     <none>        27017/TCP            24h
service/catalogue      ClusterIP   10.97.12.26      <none>        80/TCP               24h
service/catalogue-db   ClusterIP   10.96.36.245     <none>        3306/TCP             24h
service/front-end      NodePort    10.103.134.109   <none>        80:30001/TCP         24h
service/orders         ClusterIP   10.100.185.140   <none>        80/TCP               24h
service/orders-db      ClusterIP   10.99.69.25      <none>        27017/TCP            24h
service/payment        ClusterIP   10.110.58.20     <none>        80/TCP               24h
service/queue-master   ClusterIP   10.110.196.121   <none>        80/TCP               24h
service/rabbitmq       ClusterIP   10.110.121.29    <none>        5672/TCP,9090/TCP    24h
service/session-db     ClusterIP   10.109.156.32    <none>        6379/TCP             24h
service/shipping       ClusterIP   10.98.169.211    <none>        80/TCP               24h
service/user           ClusterIP   10.97.241.206    <none>        80/TCP               24h
service/user-db        ClusterIP   10.108.140.106   <none>        27017/TCP            24h

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/carts             1/1     1            1           24h
deployment.apps/carts-db          1/1     1            1           24h
deployment.apps/catalogue         1/1     1            1           24h
deployment.apps/catalogue-db      1/1     1            1           24h
deployment.apps/front-end         1/1     1            1           24h
deployment.apps/orders            1/1     1            1           24h
deployment.apps/orders-db         1/1     1            1           24h
deployment.apps/payment           1/1     1            1           24h
deployment.apps/queue-master      1/1     1            1           24h
deployment.apps/rabbitmq          1/1     1            1           24h
deployment.apps/session-db        1/1     1            1           24h
deployment.apps/shipping          1/1     1            1           24h
deployment.apps/user              1/1     1            1           24h
deployment.apps/user-db           1/1     1            1           24h
```

Figure 5—10 Sock-shop deployment status - Terminal view

We notice that the command prints out the various information depending on the object kind. For example, for pods, we have the pod's readiness, the number of restarts, and the status. For services, the service type, cluster attributed IP and port mappings. For deployments, we have the readiness & update status and its availability.

5.1.4 Sock-shop evaluation: Amazing… but hardly customizable

Sock-shop is a good application to demonstrate the micro-services design pattern and deployment over Kubernetes. However, this is not the only reason behind choosing it as a demonstration application; it is also due to the instrumentation it has.

Since the application was for demonstration purposes, the instrumentation was not configured across the whole collection of services, but only some of them.

Let us start with micro-services generating traces. As we said before, Jaeger is a trace collection tool, so it is going to be used to capture the traces generated by the services. To enable the sock-shop traces, the Kubernetes manifests can be found at *microservices-demo => deploy => manifests => Kubernetes => manifests-jaeger.* The content of this folder is as the figure 5-10 shows. Once we apply these files to Kubernetes, we notice that only three micro-services deployments (catalog, payment & user) will be updated, and one deployment will be added (Jaeger all-in-one deployment).

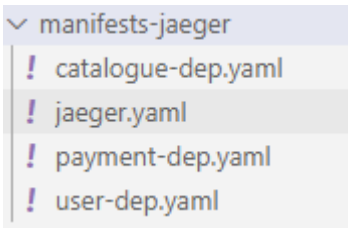


Figure 5—11 manifest-jaeger content

To explain what happened, we will compare the two manifests of catalog micro-service deployment. Figure 5-11 presents the catalog deployment file without the Zipkin exporter as environment variable name.

```
spec:
```

Figure 5—12 Catalogue micro-service deployment with Zipkin to export Jaeger traces

```
    env:
    - name: ZIPKIN
      value: http://zipkin.jaeger.svc.cluster.local:9411/api/v1/spans
    resources:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
```

```
spec:
  conta
  - nam
    ima
    command: ["/app"]
    args:
    - port=80
    resources:
      limits:
        cpu: 200m
        memory: 200Mi
      requests:
        cpu: 100m
        memory: 100Mi
    ports:
```

Figure 5—13 Catalogue micro-service as deployed previously

Once that is added, the application will know where to push these spans using Zipkin exporter to the correct URI (http://zipkin.jaeger.svc.cluster.local:9411/) using the first version of the spans API at port 9411.

At the same time, the Jaeger deployment manifest has the same port configured to receive those spans as the following figure shows.

```
spec:
    containers:
    -   env:
        - name: COLLECTOR_ZIPKIN_HOST_PORT
          value: ":9411"
        image: jaegertracing/all-in-one
        name: jaeger
```

Figure 5—14 Jaeger deployment file

Both figures show Jaeger user interface capturing traces generated by the user micro-service. The figure 5-14 shows the health check spans, however the figure 5-15 shows the login spans.

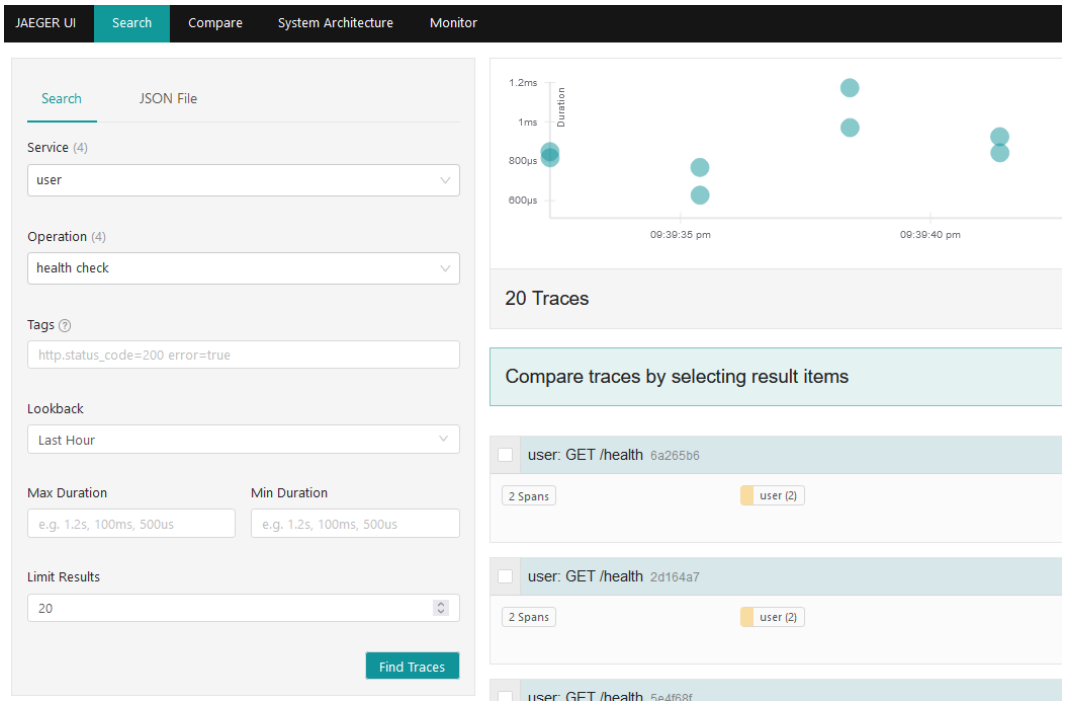


Figure 5—15 Jaeger UI capturing health check spans from user micro-service

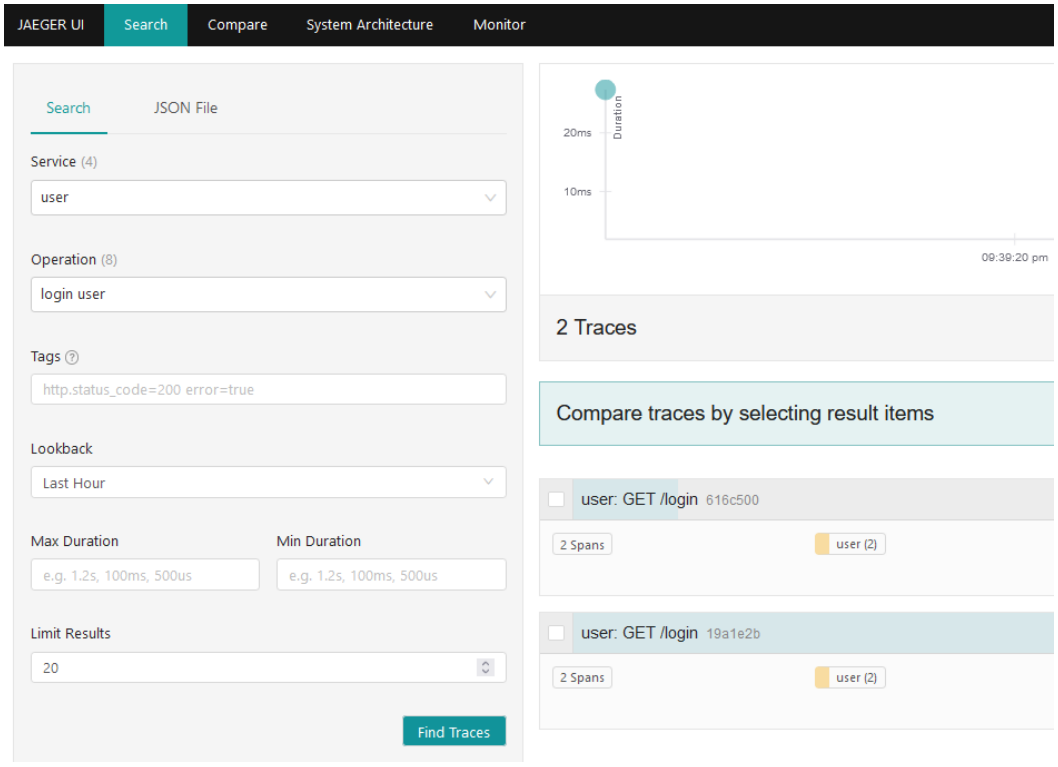


Figure 5—16 Jaeger UI capturing login spans from user micro-service

The previous figures show spans of operation happening in the same micro-service. An operation as a login, does not require to go out of the user micro-service to login the user. In fact, the login function reads the username and password from the front-end micro-service, issues the authentication token, and passes it along to other micro-services, but the login itself does not go out of the user context.

As the title of the section suggests, sock-shop application is an amazing application for a lot of demonstrations. However, the application's instrumentation does not show the real power of traces, such us: distributed context propagation, and system dependencies analysis.

Furthermore, the metrics exposed by the Cart micro-service are only the default ones (CPU consumption, opened connections count, etc.). Even when we tried to tweak the micro-service so that it exposes our custom metrics, the whole application stopped working. At first, we tried to debug the errors, but the errors were originated by a previous version of Springboot framework.

In conclusion, to demonstrate micro-services, tracing technology, monitoring tools over a cloud-type infrastructure, Sock-shop is the go-to application. But when it comes to demonstrating the effect of observability, we decided to develop our own micro-services-based application named E-shop minimal that will be analyzed in the next chapter.

5.2 E-shop minimal: Custom sample application

5.2.1 Overview

E-shop minimal is an e-commerce basic application. The application is modularized to three micro-services: Users, Carts, and Products, hence the name minimal. It was developed for the purpose of demonstrating the span distributed context propagation allowing us to track every request across the system, and default & custom metrics exportation providing us with real-time insights of different layers of the application. Each of the micro-services has its own REST API to communicate with other micro-services.

The application was decided, designed, and developed in two weeks. To demonstrate the observability principles, we opted to show it on a back-end component that they will be a very good case-study. As a result, the application does not have a front-end, only API calls are available to interact with the system. Even though the micro-services design pattern allows us to use different languages and frameworks in one application, we decided otherwise. In fact, the totality of micro-services was written in JavaScript with Express[19] framework and run on NodeJS[20].

Furthermore, the application can run either on a bare-OS or over Kubernetes. Kubernetes manifests are found in *K8s-Infra*. This will be discussed more in the last section of this chapter.

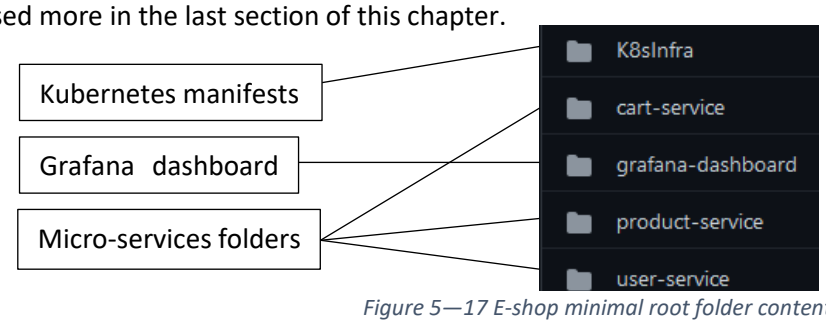


Figure 5—17 E-shop minimal root folder content

[19] Express is a JavaScript is a back-end web application framework for Node.js.
[20] NodeJS is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser, which was designed to build scalable network applications.

5.2.2 Design

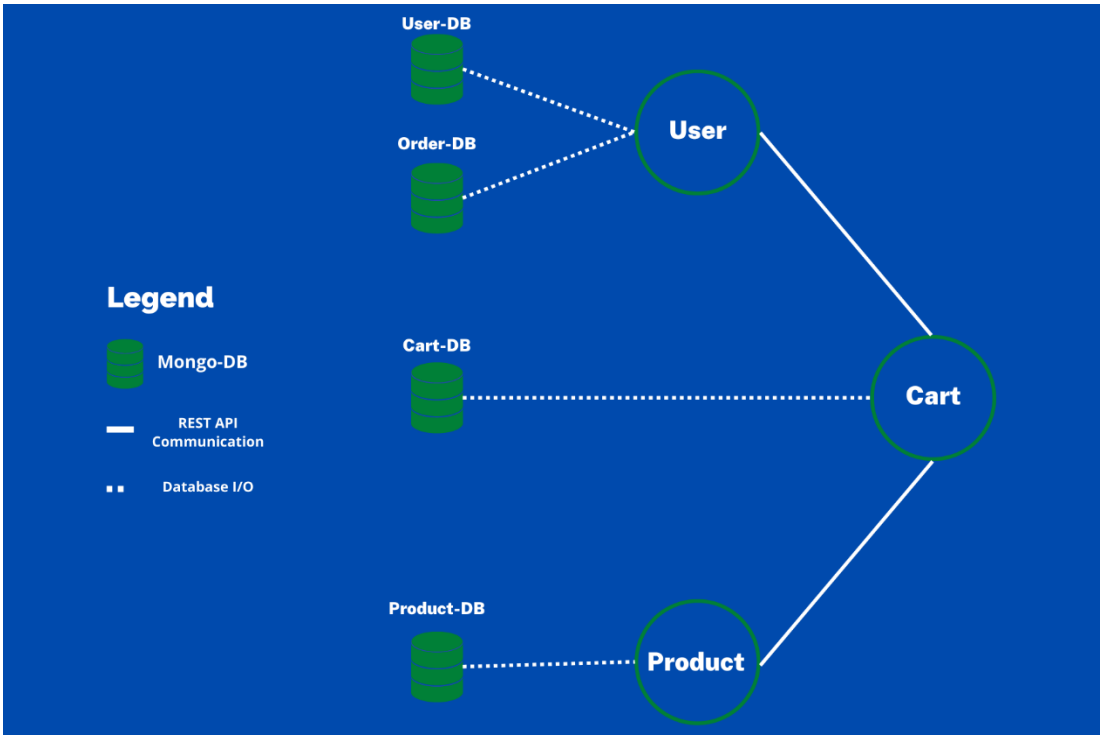


Figure 5—18 E-shop minimal design

As we mentioned before, the application is micro-services based. Each of the micro-services is developed as a standalone application and uses a REST API for communication.

5.2.2.1 Users micro-service

Users micro-service is the service that executes operations related to user account, such as, registration, login, edit profile information and place an order. Each of these operations has a route defined in the application API. In addition, once these operations are executed, spans and metrics will be generated, providing us with real-time events happening in the users-service.

The following figure represents the possible operation for user micro-service as seen from Postman[21] interface.

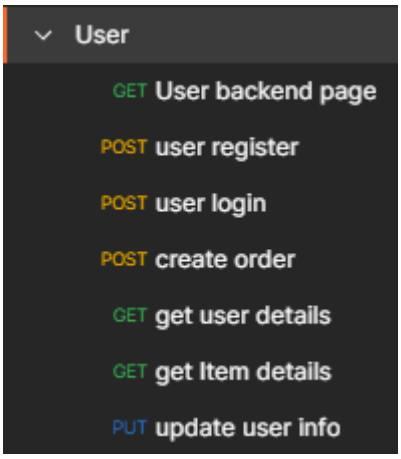


Figure 5—19 User Micro-service API from Postman UI

User backend page, used as a health-check endpoint, to verify whether the application is running or not.

User register, it registers users, requires an email and a password. The response should include an ID will be used afterwards to create orders or get user details.

User login, this is the login endpoint, it requires an email and a password. The response should include a JWT token required as a bearer token for some of the cart's micro-service operations.

Create order, this operation requires a payment method to be added to the user account info. To do so we need to use the *update user info* endpoint. It also requires a cart Id and user id.

[21] Postman is an API platform for building and using APIs

This micro-service has two MongoDb[22] databases (users and orders) on an Atlas[23] cluster. The user database is as follows:

```
const User = new schema(
    {
        email: {type: String, required: true, unique: true},
        password: {type: String, required: true},
        shippingAddress: {type: String},
        paymentDetails: {type: CardSchema},
        cartId: {type: Number},
        orders: [{type: String}]
    },
    {
        timestamps: true,
    }
);
```

Figure 5—20 User database code snippet

The Card Schema is a nested database to hold payment card details such us: name of the card holder, card number, cvc number and expiration date.

The following figure represents the orders database.

```
const Order = new schema(
  {
    content: {type:String},
    paymentDetails: {type:CardSchema},
    value: {type:Number},
    order_date: {type:Date},
  },

  {
    timestamps: true,
  }
);
```

Figure 5—21 Order database code snippet

[22] MongoDB is an open-source cross-platform document-oriented database program.
[23] Atlas is the database cluster for MongoDB

In this part, we are going to explore the metrics and the spans and how we instrument these micro-services. First, as we analyzed some of the tracing and metrics collection tools in the first part of this document, it is only logical to use them in this demonstration.

OpenTelemtry is a collection of tools, APIs and SDKs used to instrument, generate, and collect telemetry data [15]. The OpenTelemetry is the standard for spans creation and formatting. For that reason, we are going to use it in the micro-services to generate the spans. Furthermore, we need to create a tracing module and export it so that it can be used throughout the micro-service. The module file is *tracing.js* and it can be found in the root folder of the micro-service.

The following figure is a screenshot of the Postman interface registering new user.

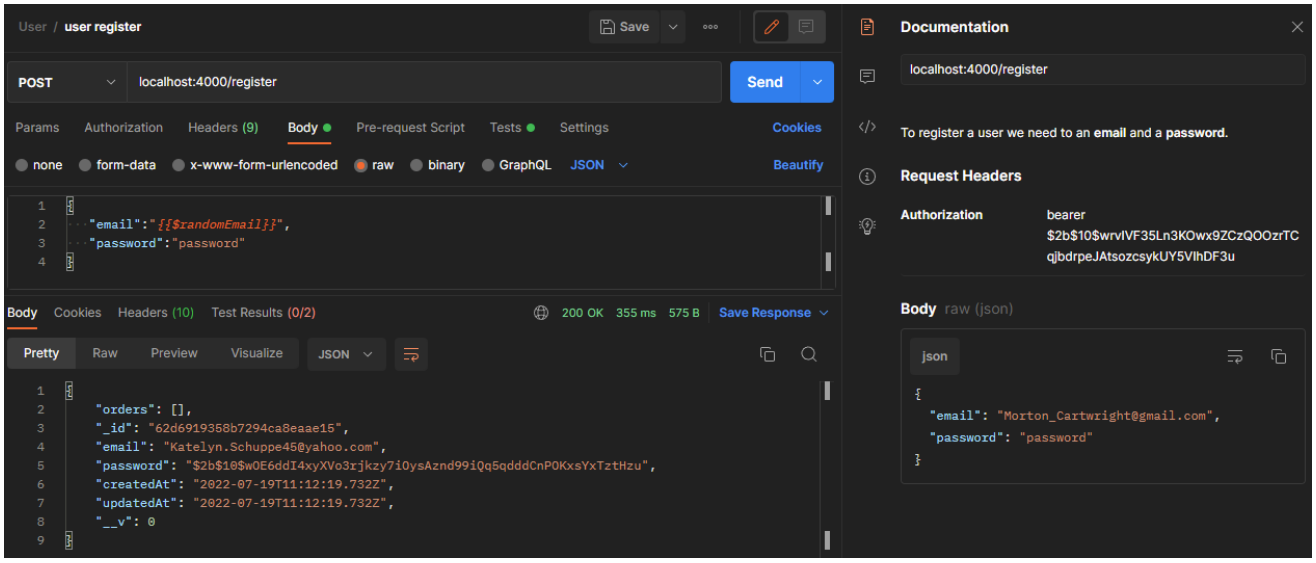


Figure 5—22 Screenshot of Postman registering new user

To illustrate what have been said, the figure 5-21 represents the spans generated by the application at the user registration moment and captured by Jaeger.

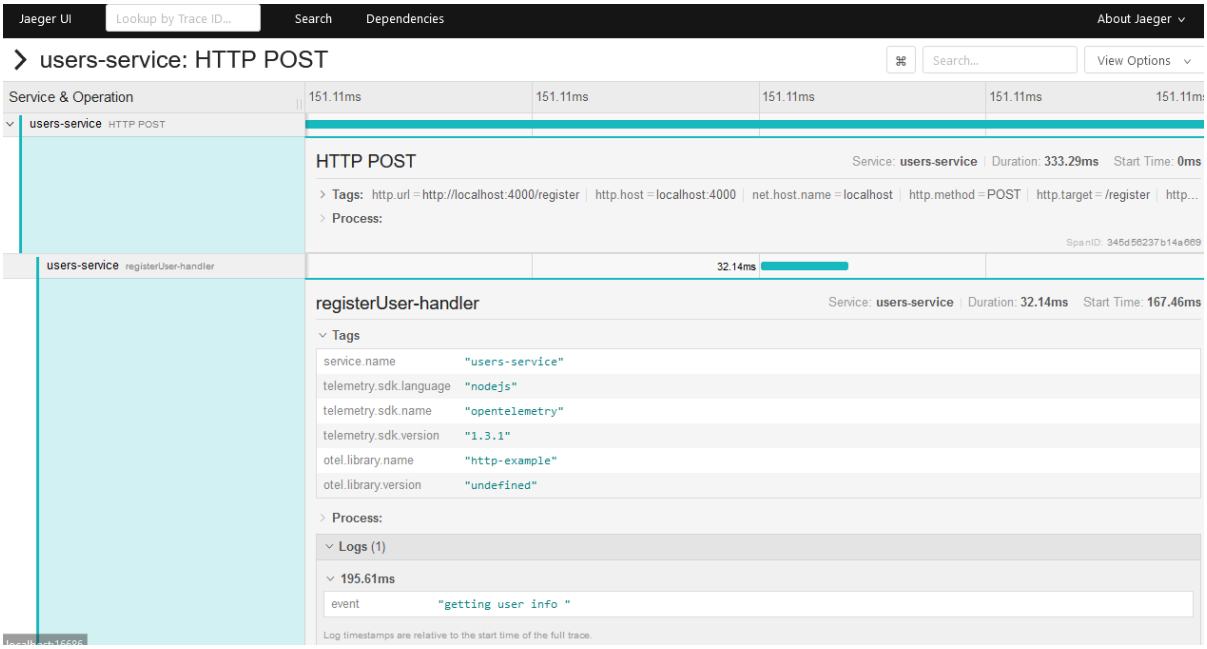


Figure 5—23 Span captured during registration operation

It is only logical that the same operation will increment a custom counter-metric (named users) that we defined to count the number of registered users. Such metric provides us with real-time insight of user's interaction with the micro-service. The following figure represents the metric visualized by Grafana and exported to Prometheus.

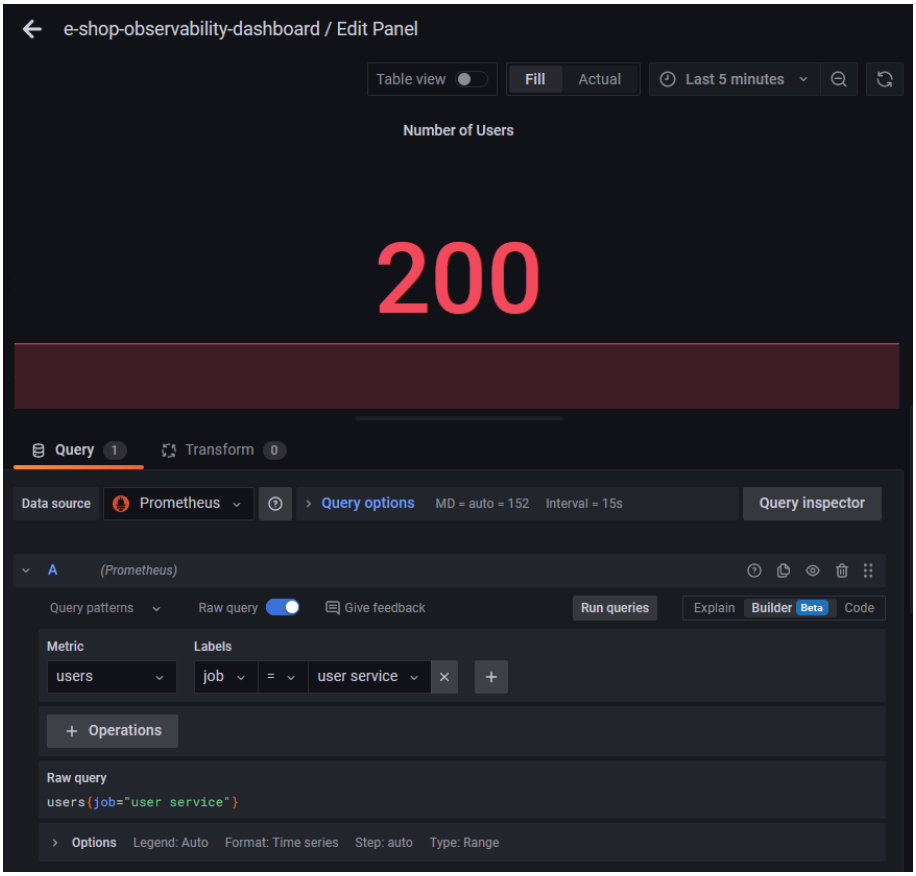


Figure 5—24 Grafana visualizing users metric

5.2.2.2 Carts micro-service

The carts micro-service is the service responsible for executing the carts related operations, such as, cart creation and deletion, adding an item to cart, updating cart content. This micro-service usually communicates often with Products micro-service, so it gets the item details and User micro-service, so it gets the user ID and verifies authentication tokens. Same as in the Users micro-service, the carts also provide a REST API to interact with the application. The following figure represents the service's endpoint. Some of the interesting endpoints will be explored afterwards.

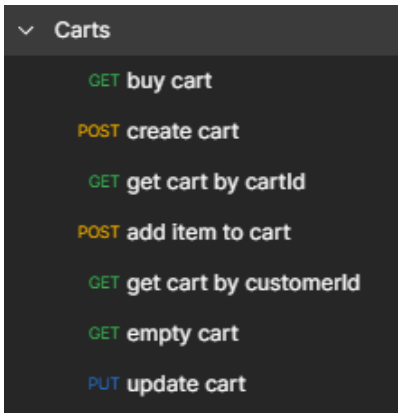


Figure 5—25 Postman view of Carts micro-service's API

This micro-service has a MongoDB database on an Atlas cluster. The following figure represents the code snippet for the database.

```
const Cart = new schema(
    {
        customerId: {type: mongoose.Schema.Types.ObjectId, required:true},
        items: {type: [ItemSchema]},
        subtotal: {type: Number, default: 0}
    },
    {
        timestamps: true
    }
);
```

Figure 5—26 Cart schema code snippet

Most of the application endpoints were instrumented the same way as in the previous micro-service. The following figure illustrates how we can add an item to a cart.

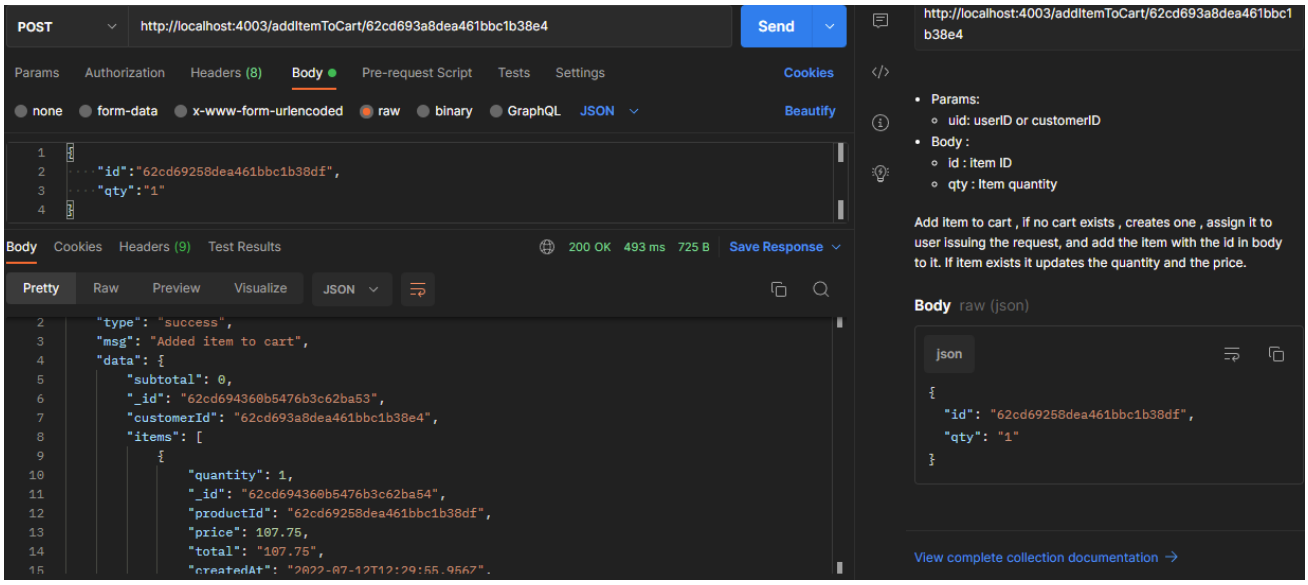


Figure 5—27 Add item to cart endpoint, Postman view

As we can see, the Item id and the desired quantity will be inserted in the request body. However, the user Id and the cart Id (optional) must be in the request params. The fact that cart Id is optional, it is because it a user has no active cart, we simply create a new one and add the item to it.

The interesting part of this operation is the traces it generates. Since the micro-service will get the item metadata from the Products service, we must observe the distributed context propagation and the service dependencies analysis. The figures 5-27 & 5-28 illustrate the spans captured by Jaeger.

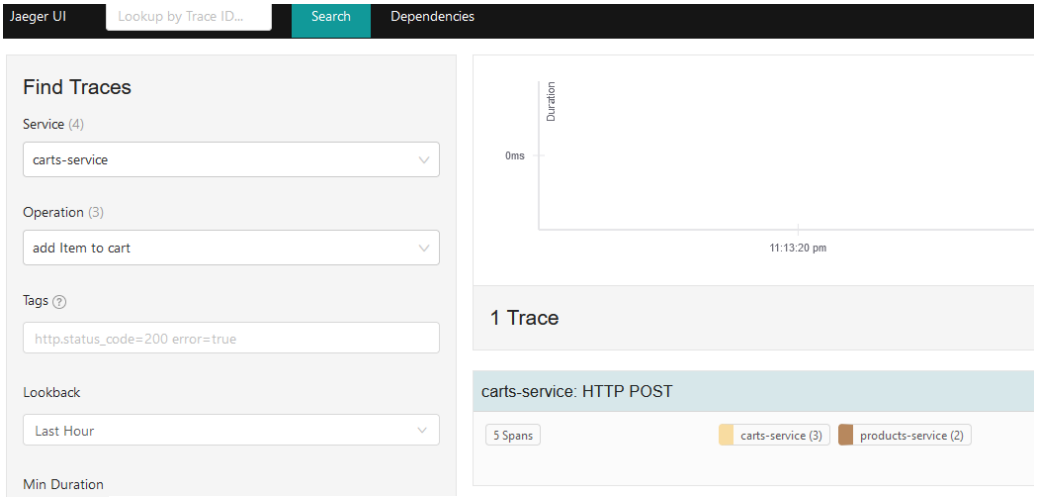


Figure 5—28 Traces generated by Cart micro-service when adding an item to cart

The next figure is a deeper view of the previous span.

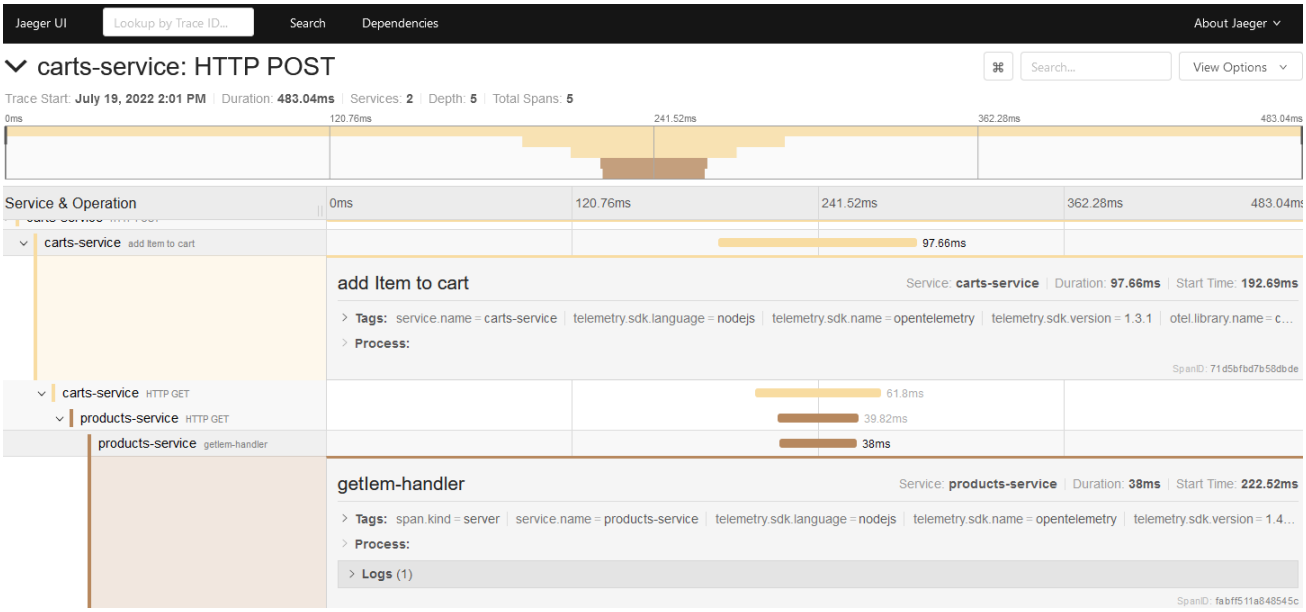


Figure 5—29 Span details of the span generated by Cart micro-service when adding an item to cart

The figure 5-28 represents the details of the captured span. The span shows multi-nested spans. Since the Cart service is the one that originated the request, it is normal that it is the root span. The products span is nested into the other span. This figure shows exactly what it is meant by distributed context propagation.

The figure 5-29 represents the modules dependency analysis graph.

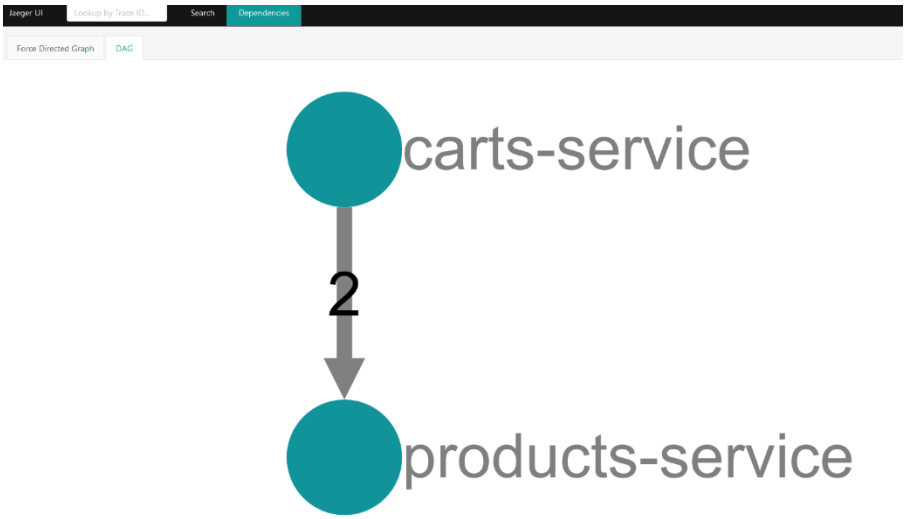


Figure 5—30 Modules dependencies analysis graph as seen from Jaeger

An interesting metrics at this level would the number of created carts and the number of purchased ones, that both would provide us with Business Intelligence real-time insights, such us, cart dropping rate. The following figure shows the number of created carts increasing when we issued the request.

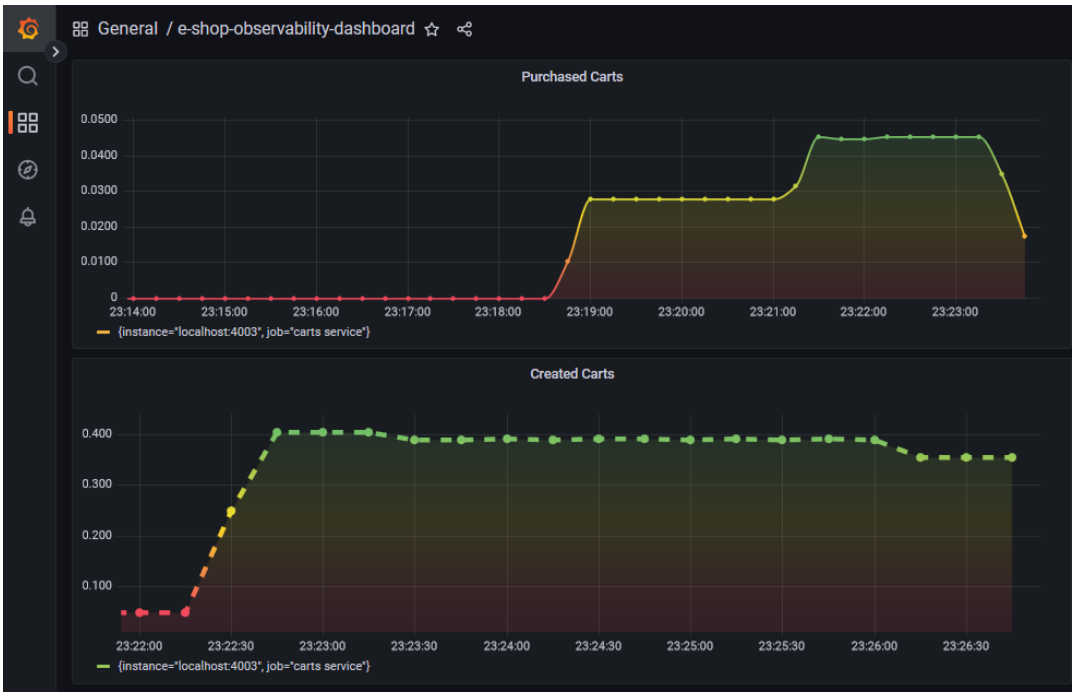


Figure 5—31 Cart created & purchased metric, Grafana view

The following figures resumes this whole section.

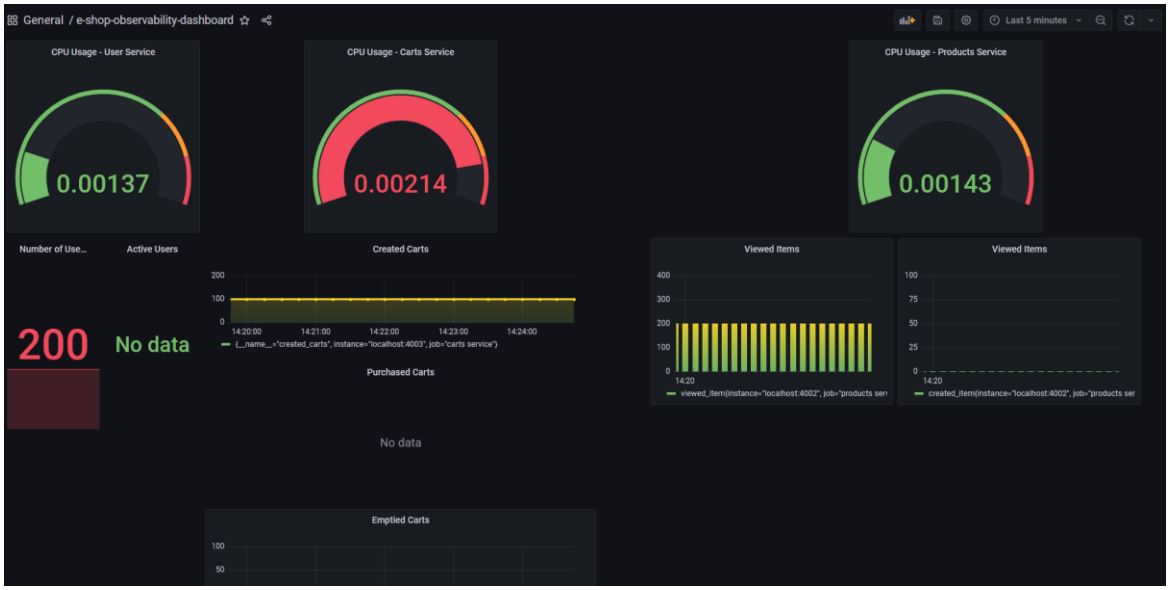


Figure 5—32 Suggested Grafana's dashboard overview

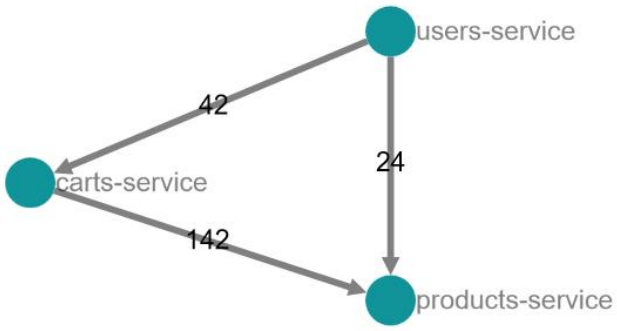


Figure 5—33 Module dependency graph after some traffic generation

5.2.3 Running the application

There is three ways to run the application.

First, we can run each micro-service as a standalone application. To do so, we need to change to the root directory of every micro-service and run the command *npm start*.

Second, we can run each micro-service as a docker container. We have pushed ready-to-start images to the docker hub under the name *w3n3s/$service_name: latest*. There is also the possibility to build new images from the sources using the command *docker build -t $container_name.* after changing to the root directory of each micro-service.

Finally, we can deploy the application to a Kubernetes cluster. Kubernetes manifests are found in K8s-Infra folder. There is two ways to deploy it, by running the command *kubectl apply -f full_deploy. yaml or* by applying each manifest apart.

6 Conclusion

Since the launch of cloud computing technologies and services, coupling it with micro-services design patterns has become a favorite design for modern application development. Traditional monitoring, logging and data collection tools could not keep up with the scalability and distribution of modern systems, hence the need for modern tools suitable for these systems. To answer this need, engineers' teams are racing to provide us with modern tools that support system distribution and scalability.

Micro-services architecture design is beneficial for cloud-based applications. However, this design, even beneficial, can be quickly transformed into a burden if the system is scaled up and no instrumentation layer is added to it. To make the design bearable and to decrease its failures, we must observe any change happening, whether at the infrastructure layer or the application layer. We have reached an era, where availability is a vital indicator for modern web applications. In fact, observing the course of events after a period will cause our application to crush more, and users are most likely to use it less. Therefore, to preserve acceptable availability rates, we must observe events in real-time and act accordingly.

Observability does not allow us to monitor the system in case of failures only, but also provides us with an overall insight of the running application and what is happening inside of it. It resides on three basic pillars: metrics, spans, and logs. These telemetric entities can include pieces of information valuable in understanding the system, or the failure, or in localizing a faulty service across a scaled-up distributed system. What is more interesting, is that the use of these metrics can be extended to reach the application layer. If we enable observability in our application, we can obtain business intelligence insights in real-time, observe errors impossible to reproduce, and track the user's most followed paths to enhance the system's navigation, with custom telemetry data.

Finally, observability tools usefulness does not stop at observing the infrastructure layer, its insights, and related failures, but can also be extended to the application layer, its insights, and failures.

7 References

[1] J. Livens, “What is observability? Not just logs, metrics and traces,” Dynatrace, 1 10 2021. [Online]. Available: https://www.dynatrace.com/news/blog/what-is-observability-2/. [Accessed 18 05 2022].

[2] G. Singh, “Observability Best Practices and its Benefits,” Xenonstack , 03 March 2022. [Online]. Available: https://www.xenonstack.com/insights/what-is-observability. [Accessed March 2022].

[3] Docker, “Docker,” Docker, [Online]. Available: www.docker.io.

[4] S. Newman, Building Microservices: Designing Fine-Grained Systems 1st Edition, O'REILLY'.

[5] Kubernetes, “Kunbernetes,” CNCF, [Online]. Available: https://kubernetes.io/.

[6] Microsoft Docs, “Communication in microservices architecture,” Microsoft, 13 04 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[7] Prometheus, “Overview,” Prometheus, 2014. [Online]. Available: https://www.prometheus.io/overview.

[8] Prometheus.io, “Storage,” Prometheus, 2014. [Online]. Available: https://www.promehteus.io/docs/latest/storage. [Accessed Mars 2022].

[9] Thanos , “Thanos,” CNCF, 2016. [Online]. [Accessed Mars 2022].

[10] Improbable, “Introducing Thanos: Prometheus at scale,” Improbable, 18 05 2018. [Online]. Available: https://www.improbable.io/blog/thanos-prometheus-at-scale. [Accessed April 2022].

[11] The Jaeger Authors, “Jaeger features,” The Linux Foundation, 2022. [Online]. Available: https://www.jaegertracing.io/docs/1.36/features/. [Accessed April 2022].

[12] Datadog, “Datadog SaaS monitoring,” Datadog Inc., 2022. [Online]. Available: https://www.datadoghq.com/saas-monitoring/. [Accessed April 2022].

[13] Datadog, “Datadog Pricing,” Datadog Inc., 2022. [Online]. Available: https://www.datadoghq.com/pricing/. [Accessed May 2022].

[14] Weaveworks, Inc, “Weavesocks,” Weaveworks, Inc, 2017. [Online]. Available: https://microservices-demo.github.io/. [Accessed Mars 2022].

[15] The OpenTelemetry Authors, “OpenTelemetry,” 2022. [Online]. Available: https://opentelemetry.io/.

8 Table of figures