Tagenal: A Deployment Ready Distributed System Application Built for the Cloud

Florent Poinsard
Tsinghua University
Toulouse, France
moq20@mails.tsinghua.edu.cn

Malte Meng
Tsinghua University
Bonn, Germany
mengxr20@mails.tsinghua.edu.cn

ABSTRACT

In this work, we propose Tagenal, a scalable, deployment ready, distributed system. After introducing the concepts of distributed systems, distributed database systems and cloud-native development, we detail the requirements such systems need to fulfill. We then introduce multiple state-of-the-art open-source frameworks that Tagenal uses to fulfill these criterias. Detailling which components of these frameworks are implemented in Tagenal and how. In our evaluation we demonstrate Tagenal's performance on different project-relevant tasks.

1 INTRODUCTION

As the demand for digital services and the devices that can use them is constantly increasing, the requirements placed on the systems supporting these services is also growing. To deal with the issue of system scalability, the concepts of distributed systems and distributed database systems were created. By splitting up the functionality of a system or the database into multiple parts, the upper limit on scaling is increased. With cloud-computing providing a perfect partner technology to these paradigms, cloud-native development is essential to the creation of such systems.

To ensure that a system can take full advantage of all the benefits distribution in a cloud-native context entails, it needs to fulfill specific requirements. At it's core the distributed system needs to be capable of scaling to deal with increased workloads, in addition to allowing extensions to new functionalities post-initial-deployment. To ensure robustness the system needs to be able to recover from failures as well as provide clear observation and monitoring to engineers. Optimally all components of the system should integrate or be usable by other systems. Similar requirements apply to a distributed database management system. With the specific addition of needing to ensure efficient data insertion and querying.

To deal with the demands the development of such systems entail, many open-source frameworks provide solutions for the different aspects. Kubernetes helps automate deployment in multiple environments and handles container orchestration. Vitess provides fragmentation for relational databases, manages routing, performs request optimization and more. Grafana and Prometheus provide fully fledged monitoring even in a distributed context. Finally, Jaeger provides tools to track and visualise events occurring around the system.

2 PROBLEM BACKGROUND AND MOTIVATION

We are currently witnessing an era where the use of new information science technologies is in constant growth. However, this strive for innovation and better performance can come with major drawbacks. Systems are arriving at a point where the performance of a single host isn't sufficient to serve a large set of users that are distributed around the globe. For this reason distributed systems were introduced, followed by distributed database systems. The application of these principles makes the further scaling of services feasible. Distributed systems on their own however, aren't enough. The introduction of cloud computing and cloud-native approaches to develop applications has the potential to take distributed systems to a new level. In this section we discuss the reasons why a modern infrastructure needs distributed systems, distributed database systems, and why taking a cloud-native approach is essential.

2.1 Distributed Systems

Before the introduction of distributed systems and other related techniques, the norm was to perform vertical scaling on an infrastructure when current resources were insufficient. This meant increasing the CPU, RAM, and other physical components of the machine that was hosting the system[1]. Distributed systems try to overcome these challenges by taking advantage of horizontal scaling. This is done by splitting the functionality of a program over multiple inter-dependant nodes or containers.

The benefit of doing so extends far beyond the capacity to scale. By breaking down a single centralized system into multiple subcomponents, we inherently develop an architecture where multiple tasks can be run in parallel, leading to a significant speed-up. Additionally, the failure of a single sub-component in a larger centralised system can lead to a complete failure of the system. In a distributed context when a single sub-component fails, only the failing node is affected, whereas others are left unharmed.

2.2 Distributed Database Systems

Similar to distributed systems, storing the constantly increasing amounts of produced data in a centralised system is not feasible. By fragmenting a single database into multiple separate ones, distributed databases theoretically allow unlimited scaling. Through a proper database management system users should be able to access and edit the database as if it were a single centralised one.

To do so two main approaches have been proposed, horizontal and vertical sharding. Horizontal sharding focuses on sharding a single database into multiple ones based on the columns. Vertical sharding focuses on sharding a database by its rows. Either variants of sharding increase the capacity for scaling in addition to providing possibly higher throughput. Many benefits of distributed systems also transfer over to distributed database systems. Reliability and performance are also a big part of database systems and how they should be designed.

2.3 Cloud Computing

Due to native distributed systems and distributed database systems being costly and difficult to manage, cloud computing came along in 2006 with Amazon Web Service (AWS) and its first version of the Elastic Compute Cloud (ECS)[2]. Cloud computing emerged parallel to the growing popularity of container-based environments, which gained even greater global popularity thanks to Docker in 2013. Cloud computing's major advantage of on-demand availability and low costs made it a top choice for IT teams. It allows teams to scale up and down infrastructure without paying for unused servers and resources, leaving all of the management to cloud providers.

2.3.1 Cloud Native. With the usage of cloud computing platforms becoming the standard, the term "cloud native" appeared. Cloud native is used to represent an approach aiming to develop and build scalable applications in flexible and modern cloud computing environments[3][4]. Usually, cloud native applications are built with multiple services packaged into containers and deployed as microservices. These microservices are being managed and monitored through agile DevOps processes.

2.3.2 DevOps. The term DevOps appeared not long after cloud computing's first platform, in 2007. It is now commonly used to describe a culture more than a physical person. We used to have two type of teams, developers (Dev's) and system administration engineers (Ops), these teams would work in a silo configuration, where the communication between the two teams was infrequent and frustrating. Examples of this would be, when an application running on a Dev's local machine is sent to the Ops to be deployed to production, but the application does not work anymore because the production environment is different from the Dev's. Instead, the DevOps culture tries to merge the two skill sets (Dev and Ops) into a single one. Automation and monitoring are some of the key characteristics being promoted by DevOps. Moreover, the introduction of the Agile methodology improves the overall development's life cycle, instead of having development cycles that are long and subject to a lot of unexpected bugs once deployed in production, DevOps prone the use of shorter development cycle, integration and unit tests, and continuous delivery.

2.4 Monitoring & Observation

In a world where a single second of downtime can cost millions of dollars, it is extremely important to be able verify how an application is doing and how it is handling different workloads. Most big applications or websites have a rule called the five nines. This rule tells us that a service must be available at least 99.999% of the time. If we follow this rule, we would be allowed only 5 minutes and 15 seconds of downtime per calendar year[5].

2.4.1 Monitoring. Monitoring enables us to observe the quality and performance of our system over time. Monitoring tools, such as the widely used open-source Prometheus and Grafana, can give us insights on the performance, health, and any other interesting characteristics. All services, and/or components of an infrastructure, can generate metrics, about their health, workload, occupancy, etc, all these metrics can be gathered and transformed into human-readable insights. Google's SRE (Site Reliability Engineering) book[6] discusses the questions a monitoring system needs to answer: "What is broken, and why?"[7].

2.4.2 Observation. Observability originates from mathematical control theory, it allows us to measure how well we can understand the internal states of our system based on external outputs (logs, metrics, and traces)[7][8]. Observability is the foundation of monitoring in a system. With observability we can answer questions such as: "Why did a request fail?", "How did a service process a specific request?"[7]. Famous frameworks for observability include OpenTracing, Jaeger, Zipkin. There is a clear distinction between monitoring and observability, the former will allow us to be aware that something is wrong, and the latter will tell us why it is wrong.

3 PROBLEM DEFINITION

Developing a distributed systems with a cloud native approach is no easy task. Before all, a cloud native application needs to be highly resilient, manageable, and observable. These three items represent a massive challenge, however many techniques and tools ease the development life cycle.

In this section we will be discussing the different aspects and components a successful distributed system needs to include. Additionally we will describe the data used in our solution and its details.

3.1 Distributed System

When we want to develop a distributed system application with a cloud native approach many things have to be taken into consideration.

3.1.1 Scalability. As the main motivator of the usage of distributed systems we need to ensure that our solution is scalable. The system must be capable of dealing with unexpected increases or decreases in the workload, by horizontally scaling up or down.

3.1.2 Accessibility. As the number of components in the system grows, errors are more likely to happen. If we want to follow the five nines rule, or any reliability rules, we need to guarantee that our application will be available to users regardless of the circumstances. One of the core concept of the DevOps culture is to always accept failures and errors, as they are required for a proper development cycle, when failures happen we can learn from them. However, the system needs to always be accessible in multiple ways.

Firstly, a system needs to recover from failures and disasters: when a node, service, or container fails, the system should be able to keep on providing service to end users. Secondly, deploying new features, containers, services, or adding new nodes to the system should not impact the running time of our system, the system needs to be running at all time even during deployment and maintenance. This trait is achieved by the conjugated usage of continuous integration/deployment (CI/CD) pipelines and orchestration.

3.1.3 Openness. An open distributed system is essentially a system that offers components that can easily be used by, or integrated into other systems. At the same time, an open distributed system itself will often consist of components that originate from elsewhere. Nowadays, a system should take advantage of the world-wide open source community, and gain benefits from it, optimally a system should also contribute new ideas and thoughts to this community.

3.2 Distributed Database Management System

While distributed database systems have concrete advantages over centralised ones, we need to ensure the effectiveness of an implementation. This section will outline what we view as the most important aspects of an implemented distributed database.

3.2.1 Efficient Data Insertion and Querying. Due to the fragmentation of our database, the possible throughput has increased significantly. To take advantage of this we need to ensure that the additional processing being done to the queries are efficient. Additionally the fragmentation of data that is being inserted into the database also needs to be efficient.

3.2.2 Database Monitoring. Since we're splitting a single database into multiple components, we need a monitoring solution that provides information on each of our fragments in a single place. Optimally the monitoring system can also automatically identify when a failure occurs at a certain site.

3.2.3 Fault Tolerance. Our distributed database system needs to be able to deal with faults at a single site or database. A way to deal with this would be to have replica's that are distributed across multiple sites. In the case of a site or database failure the replica's replace the databases and new replica's are generated.

3.2.4 Expansion and Deletion. The database needs to be able to deal with changing data specifications such as removal or addition of new columns. To enable this, dropping fragments or adding new ones dynamically should be possible.

3.2.5 Distribution Transparency. Although we want the system to be as distributed as possible, the system distribution should be completely transparent to the user. This should be the case for both hiding differences in data representation and ensuring that the location of objects in a system aren't be visible to the user either.

3.3 Monitoring & Observation

As mentioned earlier, being able to monitor the current status of our infrastructure and application is mandatory. This is done through implementation of both monitoring and observability techniques and tools.

Successful monitoring needs to provide DevOps a glance at the most relevant system metrics and insights, tools to search for certain nodes and containers, warnings when abnormal behaviour happens. Overall, and as we previously stated in the background section based on Google's SRE book, our monitoring solution must answer: "What is broken, and why?". While our observability solution must answer: "Why did a request fail?", "Where did my request go, where did we lose time?".

4 TAGENAL

To develop a truly cloud native system that solves all of the problems mentioned in Section 2., Tagenal relies on multiple frameworks. Each of the tools are open-source and currently being used in production environments by companies like GitHub, Slack, YouTube, Google, etc... The overall architecture and connection of these frameworks can be seen in Figure 1. Moreover, Tagenal's source-code is available on GitHub at: https://github.com/frouioui/tagenal.

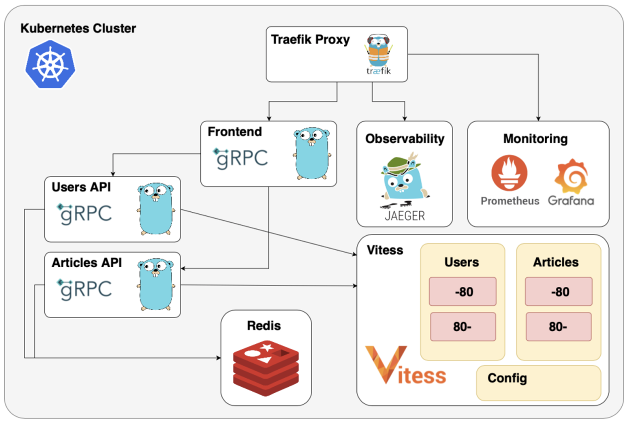


Figure 1: Tagenal's architecture

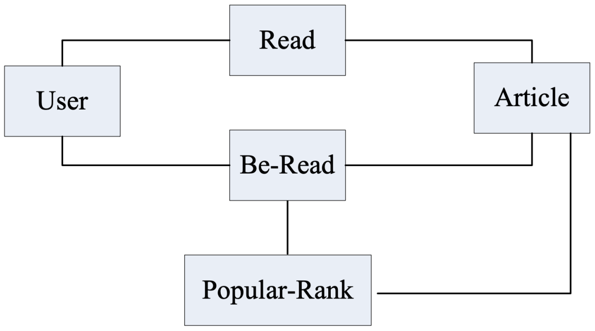


Figure 2: Relationship between data tables

4.1 Implementation Details

The data used for Tagenal's implementation consists of five relational tables in addition to images and video files. These tables hold different pieces of information about a theoretical application that hosts articles on different topics and allows users to interact with them. The tables are all interconnected as shown in the entity-relationship diagram in Figure 2.

4.1.1 Base Tables. The User table has a single entry for every user that is registered in the application. The information stored about the user consists of personal information (name, gender, dept.), contact information (email, phone) and application relevant information (language, region, role, tag preferences, grade and obtained credits). The Article table has a single entry for every article that is featured in the application. The data stored for each article consists of article meta-data (category, tags, authors, language) and article content (title, abstract, text, image and video).

4.1.2 Information Tables. The Read table stores information of user activity on the different articles. Each row in the read table

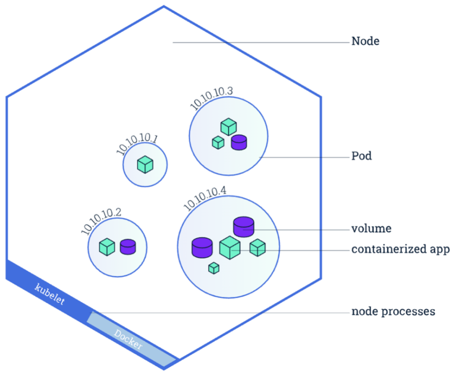


Figure 3: Nodes, Pods and Containers in Kubernetes

consists of an article user pair. The information stored consists of user read statistics for the article (if the user read the article, time spent reading and reading sequence) and user activity on the article (opinion, comments and if the user shared the article).

The Be-Read table stores information on article read statistics. Each row in the table corresponds to an article. The stored data consists of read statistics (number of reads, list of readers), opinions (number of agree's, list of agreeing users) and share statistics (number of shares, list of sharing users).

4.1.3 Aggregation Table. The Popular-Rank table stores ranking information on the most read articles. Each row corresponds to an article popularity ranking. The stored data consists of the rank granularity (daily leaderboard, weekly leaderboard and monthlys leaderboard), the point in time the ranking was created, and the list of articles in ranked order.

4.1.4 Data Generation. The data we used for the Base and Information tables was provided via three generation python scripts. After combining these into a single python script we generated data and loaded it into Tagenal.

4.2 Distributed System with Kubernetes

To enable scheduling and orchestration Tagenal uses Kubernetes. In this sub-section we will discuss why we used Kubernetes in Tagenal.

4.2.1 Kubernetes. Kubernetes is an open-source system that automates deployment on multiple environments, scaling, and management of containerized applications. In Kubernetes, each node contains one or more pods. A group of Docker containers is gathered inside each pod, forming a service, or part of a service. Pods are the smallest deployable unit in Kubernetes. Figure 3, taken from Kubernetes' website, describes the architecture of a Kubernetes node.

Kubernetes packages a lot of functionalities. The functionalities that we use the most in Tagenal and that brought our attention are: the storage orchestration, the automated and high performance horizontal scaling, the self-healing, and the strong load balancing. The conjugation of these features allow us to build a fully distributed systems in a cloud native way.

A Kubernetes cluster is highly scalable, simplifying the addition of new nodes. Scaling an service is also easy, Kubernetes makes expanding and reducing the number of pods for each service easy. Kubernetes' load balancing functionality allow us to scale up or down a service without impacting users experience, specially by using Canary deployments and automated roll-outs and rollbacks. Moreover, if a service or container fails after a deployment, Kubernetes handles it by self-healing. Meaning, new pods or containers can be created whenever one fails. All these features make Kubernetes a great option when it comes to container orchestration. The fact that Kubernetes answers challenges commonly found in distributed systems motivated our use of Kubernetes in Tagenal.

4.2.2 Minikube. When discussing Kubernetes, we talk about one or more "Kubernetes cluster". A Kubernetes cluster is composed of multiple physical or virtual nodes. However, since Tagenal is developed for local experimentation it does not have any physical or virtual clusters. Therefore, we rely on Minikube which allows us to set up a local Kubernetes cluster on any distribution.

4.3 Database Sharding and Management with Vitess

Although a considerable amount of databases provide support for NoSQL, we decided not to use it. While NoSQL databases provide great support for unstructured data, we wanted to ensure that the relationships between the tables were clearly defined. Therefore, we decided that and SQL database would be a better fit.

The framework we found that allowed us to perform relational database fragmentation was Vitess, a database solution for deploying, scaling and managing large clusters of database instances. Vitess is currently used in production by many tech companies. In a recent report[9] written by Slack's development team stated that Vitess allowed them to support 2.3 millions queries per seconds at peak load. To provide an additional speedup we also decided to use Redis to provide a caching layer to our application.

4.3.1 Vitess. The architecture of a sample Vitess deployment can be seen in Figure 4. Powered by Kubernetes, each fragment of our database resides in an individual pod as a stand-alone instance of MySQL. These instances are managed by Vitess' VTTablet, which performs request pooling and query rewriting to optimise performance. Application queries are done through VTGate, which routes traffic to the corresponding VTTablet's and returns the consolidated results. Since VTGate can use SQL queries as an entrypoint, applications communicate with it as if it was a MySQL server, providing us with system transparency. Vitess clusters are divided into multiple keyspaces. A keyspace represents a single and unique MySQL logical database server. For each keyspace we have a specific replication topology. The global topology service stores all information relating to keyspaces, shards and tablet alias's for each shard. The Topology Service is a key-value datastore powered by etcd enabling separate tablets to coordinate with each other, provides VTGate with the information required to route queries and stores custom configurations that need to persist between server restarts.

Using Vitess, we split the 5 tables into two separate keyspaces, namely: articles, and users. In our application we divide the data models into two big categories, article and user. The two keyspaces

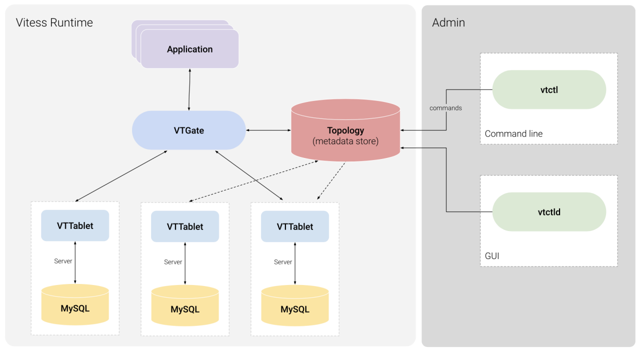


Figure 4: Vitess's architecture

solely manage their own category. For instance, the articles keyspace will store tables and data that is only part of the articles-business-logic. This choice of splitting our 5 tables into distinct MySQL logical database server follows the microservice architecture pattern, which is commonly used nowadays. For each of our keyspace we can define a specific replication topology deciding how many replicas, backups, and read-only VTTablets we want. In the case of a fault or failure of one of the master VTTablet, Vitess automatically replaces the faulty master VTTablet with one of the replicas VTTablets. Automatically rerouting all relevant requests to the new elected master VTTablet.

For each table in these keyspaces we have not only each relevant table, but also replica's. In the case of a fault or failure of one of the pods, Vitess automatically replaces the tables with those on the replica. Automatically rerouting all relevant requests to the new master VTTablet.

Updating the information stored in the be-read table is achieved thanks to powerful replication streams, provided by Vitess. These streams are installed between the different shards and tables. Concerning the popularity rank, a set of Kubernetes CronJobs defined in our Kubernetes cluster allows us to automatically update the popularity ranking. The regularity of execution of the jobs is based on multiple CRON schedules.

4.3.2 Redis. As an open-source framework for in-memory key-value storage Redis was the perfect solution to provide us with caching. Redis can serve frequently requested items at sub-millisecond response times, and enables us to easily scale for higher loads without growing the backend. Redis can enable multiple different types of caching, query result caching, session caching, web-page caching and object caching (images, files or metadata).

Added as a layer between our APIs and our Vitess cluster, we are using Redis to do query result caching. This leads to a significant reduction in processing time for identical requests, which will be discussed in the evaluation section. Our Redis cluster is composed of 3 Redis master, and 3 Redis slaves. Scaling the number of pods that forms our Redis cluster is an easy task thanks to Kubernetes.

4.4 File Storage with NFS

The generated data we use in Tagenal contains text files, images, and videos. These medias need to be stored and made accessible to the applications needing it, such as the front-end. To answer this requirement, we went over a few options.

4.4.1 Hadoop HDFS. We initially implemented a Hadoop HDFS cluster inside our Kubernetes cluster, and added Spark and Hive on top of it. However, we decided to not deploy the Hadoop HDFS cluster implementation for the following reasons: First, Hadoop is not built for real-time applications and our needs were for real-time. Secondly, Hadoop's filesystem HDFS is not POSIX compliant. This means that it does not support files smaller than the block size used in the name and data nodes properly. In our case since we need to store text files that can be smaller than the default block size (usually 64MB) HDFS would not be optimal.

4.4.2 Ceph and Rook. Following our implementation of Hadoop HDFS, we decided to implement Rook with Ceph in our Kubernetes cluster. Rook is an open-source, cloud-native storage for Kubernetes, it eases the management for file, block and object storage. Ceph is a highly scalable distributed storage solution for block storage, object storage, and shared filesystems. Ceph was a very good candidate for our file storage. However, since we were running Tagenal on a local environment using Minikube we were not able to test these tools. Ceph uses OBS (Object Base Storage) but the support for OBS on Minikube is still isn't complete. We thus decided that Ceph was not worth all the modifications that would have to be done, to get it running.

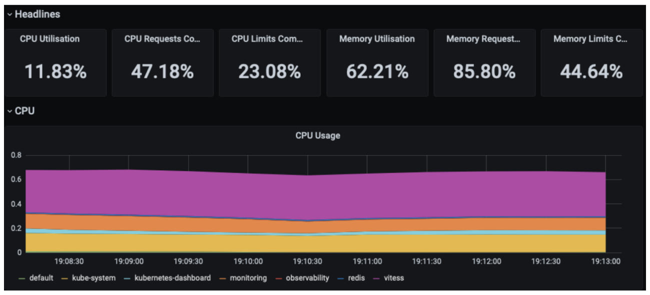
4.4.3 NFS. Eventually, we decided to set up NFS between our local machine and Minikube's VM to mount a local volume. We then created volumes inside our Kubernetes cluster, and mounted them onto the applications that required access to all or part of the files and data.

4.5 Monitoring with Grafana, Prometheus, and Alertmanager

Monitoring sits at the core of our implementation. To enable monitoring we decided to opt for the open-source projects Grafana, Prometheus, and Alertmanager which comes from Prometheus. The combination of these tools is popular and well known. Many solutions exist in terms of monitoring, however the ease and popularity of these tools seemed to be a great fit for Tagenal.

4.5.1 Grafana. Grafana is used to provide interactive visualization of a system's metrics and health. Grafana can be accessed by its Web UI that we host inside our Kubernetes cluster. The Web UI provides graphs, charts, figures, and alerts about the system's state. These displayable components form dashboards that can be created, imported, exported, and modified. In Tagenal's Grafana we use a variety of dashboards to monitor the cluster's health and metrics, each pod's status, the network utilization, and so on. Everything can be seen from this Web UI. To feed itself, Grafana fetches informations from Prometheus at a given time interval. An example of the CPU Usage visualisation of different clusters can be seen in Figure 5.

4.5.2 Prometheus. Prometheus is one of the leading open-source monitoring solution. It scrapes data from the different pods, systems, services, and store them efficiently into a custom format that easily scales by using functional sharding and federation. Prometheus


Headlines

CPU Utilisation 11.83%
CPU Requests Co... 47.18%
CPU Limits Com... 23.08%
Memory Utilisation 62.21%
Memory Request... 85.80%
Memory Limits C... 44.64%

Figure 5: Snapshot of Grafana in Tagenal

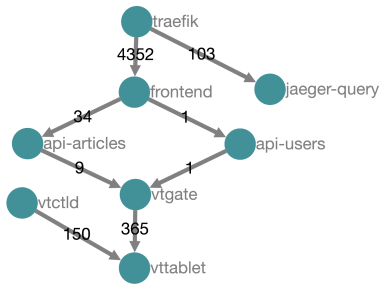


Figure 6: Span tree of Tagenal visualized in Jaeger

provides a powerful query engine that can be used to generate tables, graphs, time series, and alerts.

4.5.3 Alertmanager. As a part of Prometheus, Alertmanager receives alerts that are raised by Prometheus. It takes care of grouping, de-duplicating, and then transfers the alerts to a chosen platform (Slack, Email, etc.).

4.6 Observation with Jaeger, OpenTracing and EasticSearch

To ensure observability of Tagenal, we also decided on using a combination of multiple open source tools. To generate, store and visualise actions being done throughout our whole system we us a combination of Jaeger, OpenTracing and ElasticSearch.

4.6.1 Jaeger. Originally developed by Uber prior to being made open-source, Jaeger is a complete distributed tracing solution. With Jaeger we generate traces when different events occur across the system. These traces are then stored by Jaeger and made accessible. Through a Web-UI Jaeger provides seperate visualisations and information on these traces. This provides DevOps with the tools required to do service dependency analysis, root cause analysis and even performance / latency optimization.

4.6.2 OpenTracing. The trace model we used in Jaeger to represent the events / traces are defined by the conventions in OpenTracing. Single traces using the OpenTracing convention are defined as directed acyclic graphs where vertices are individual spans and edges represent the call-chain. These spans contain valuable information to the execution of individual tasks. Concretely this information consists of, start and finish timestamps, event-specific data that is pulled from Prometheus and the connection to other spans. As seen in Figure 6 a tree can be used to represent the causal relationships within a trace additionally a time-axis can also be used to display the temporal relationships.

4.6.3 ElasticSearch. To store the trace data in Jaeger we use ElasticSearch. When ingesting trace data into the storage, the information is pre-processed and indexed by ElasticSearch. This enables faster performance for simple searches and more complex queries. This speedup is essential to time-critical tasks like fault localisation.

4.7 Sample Application & API

To fully complete our application, we implemented a front-end service and multiple APIs. The APIs and front-end are interconnected using gRPC and HTTP queries, and have enabled instrumentation for tracing throughout all the queries.

4.7.1 Front-end. The front-end integrated in Tagenal is brief and is mostly used, for now, as a way to interact with the data. The front-end is developed using Echo which is a web development framework developed in Golang by LabStack. For now, the front-end contains trivial features such as the ability to see all the users of a given region, the details of a specific user, see all the articles of a given category or storage location, and see the detail of a specific article. Moreover, the front-end service is made accessible to outside the Kubernetes cluster by using Traefik Proxy as a reverse-proxy and Kubernetes ingress.

4.7.2 APIs. As mentioned earlier, Tagenal follows the microservices architecture pattern. We thus have 2 APIs, namely: users and articles. Each responsible either for the users or articles related actions and data. Both APIs support both HTTP and gRPC queries, and respect the REST architecture style.

5 SOLUTION EVALUATION

We built Tagenal, a fully distributed application designed and developed with a strong cloud native approach. To evaluate how Tagenal performs and verify that we respect cloud native application guidelines, we ran multiple tasks and recorded the system's performance. Since we don't have access to a cluster of machines, all of the tests were performed locally using Minikube. Although this means that the benefit of the database and system distribution will not be apparent, we can test for functional completeness.

5.1 Data Specification

Tagenal is shipped with a python script allowing automatic and random generation of data. During our evaluations we used a static configuration of 5500 users, 4000 articles, and 20000 read records. For each article a random number ranging from zero to ten of pictures and videos are generated and added onto the article's property.

5.2 Inserting data

Inserting all our data into our system is one of the most important metrics we would like to know to measure the performances. We use a technique called bulk load aiming to load a big chunk of data into the database. We insert all of the rows from the 3 generated tables (user, article, read), inserting these rows will automatically

Table 1: Query Times

	Without Caching	With Caching
Single Row	21ms	8ms
1.7k Rows	589ms	484ms
4k Rows	2518ms	1788ms

create new rows to the 2 other tables (be-read and popularity), summing up to a total of 35257 rows, that were inserted in 5.035 seconds.

5.3 Execution of Insert, Update and Queries

To evaluate how our system performs in real life situation, we execute multiple queries. We want to be able to perform basic insertion, update, as well as complex queries with joins. We can execute these queries inside a MySQL client connected to VTGate. The performance of our cache layer implemented with Redis can easily be observed. We ran select queries of varying size: single row query, 1700 rows, and 4000 rows. The query were made on Tagenal's front-end application, and then displayed in a web interface. The time measured takes into account the whole query lifetime: from the reverse-proxy to the final front-end rendering. The average response time without the caching takes about 20ms, whereas with caching we orbit at an averaging 8ms per request.

5.4 Fault Tolerance

To evaluate how resistant to failures our system is, we practiced disaster recovery and observed how the system behaved. Doing so is common practice in industry. These tests showed us that we are able to shutdown any pods of our Vitess cluster and keep our application running and serving at all times. Once one of our Vitess pods stops due to a failure, it takes only a few seconds for the scheduler and Vitess to initialize a new pod and elect a new master in the keyspace or shard.

6 CONCLUSION

In this report we provided a brief introduction on the background of distributed systems, distributed database systems and the importance of cloud-native development for both. We define the specific data that was used for the project and the different requirements our solution needs to fulfill. We then introduce Tagenal, a solution comprised of multiple state-of-the art open-source frameworks. After introducing the concepts required to understand the functionality of the different frameworks, we detail how we implemented them into Tagenal and which prior requirements they allow Tagenal to fulfill. To evaluate our solution we run multiple different tasks in a local environment to record it's performance.

Components that could still be added in future include additional support for logging with FluentD, a Kubernetes powered distributed file-storage using Ceph and Rook, implement more functionalities in the front-end application to demonstrate Tagenal's capabilities. Because of hardware limitations, it also wasn't possible to run tests and evaluate the system in a true distributed context over multiple nodes. Doing so in the future would provide valuable insights into Tagenal's performance in a production environment.

7 PROJECT DETAILS

7.1 Manual

Tagenal's documentation main component is a quick-start whose goal is to setup Tagenal from A to Z. It is made of 7 distinct steps. To complete all the steps, only 15 commands need to be executed. All the commands are gathered into a global Makefile, easing the deployment and experimentation process. The 7 steps are defined below:

- Setup the Kubernetes cluster
- Setup Jaeger
- Setup Traefik Proxy
- Setup Vitess cluster
- Setup Redis cluster
- Setup monitoring
- Setup APIs and Frontend

Beforehand, the user need to respect a few requirements including the installation of a few packages that are listed at the start of the quick-start.

The documentation is accessible at https://frouioui.github.io/tagenal.

7.2 Distribution of Labour

As the team-leader Florent Poinsard was in charge of the project infrastructure, development and versioning. In constant contact, Malte Meng did continuous testing, integration and aided in tool / framework discovery. Despite the COVID-19 pandemic situation, the team members closely collaborated on each of the project's components.

8 ACKNOWLEDGMENT

We would especially like to thank Prof. Ling Feng, for all the support she provided over the course of this project. We would also like to give thanks to the team working on maintaining Vitess for their support and help in solving problems that occurred over the course of the project. And finally we would like to thank the open-source community for making sharing and cloud-native such a beautiful universe.

REFERENCES

[1] A. Bhat, "Distributed systems (horizontal vs vertical) scaling." https://medium.com/@aparnabhat10/distributed-systems-horizontal-vs-vertical-scaling-6901b8136798, 2020.
[2] Wikipedia, "Cloud computing." https://en.wikipedia.org/wiki/Cloud_computing, 2020.
[3] Wikipedia, "Cloud native computing." https://en.wikipedia.org/wiki/Cloud_native_computing, 2020.
[4] CNCF, "Cncf cloud native definition." https://github.com/cncf/toc/blob/master/DEFINITION.md, 2020.
[5] "The holy grail of five-nines reliability." https://searchnetworking.techtarget.com/feature/The-Holy-Grail-of-five-nines-reliability.
[6] B. Beyer, "Site reliability engineering." https://www.oreilly.com/library/view/site-reliability-engineering/9781491929117/, 2016.
[7] I. Egilmez, "Monitoring vs. observability: What's the difference?." https://thenewstack.io/monitoring-vs-observability-whats-the-difference/, 2020.
[8] M. Raza, "Observability vs monitoring: What's the difference?." https://www.bmc.com/blogs/observability-vs-monitoring/, 2019.
[9] "Vitess." https://slack.engineering/scaling-datastores-at-slack-with-vitess/.