MŰEGYETEM 1782

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Erdei Roland

Performance Measurements and Resource Provisioning of Cloud Native Metrics Ingestion and Service

Teljesítménymérések és erőforrásallokáció felhőalapú adattárolási szolgáltatáshoz

KONZULENS
Dr. Toka László

BUDAPEST, 2021

Abstract

With the continuous progress of cloud computing, many microservices and complex multi-component applications arise for which resource planning is a great challenge.
For example, when it comes to data-intensive cloud-native applications, the tenant might be eager to provision cloud resources in an economical manner, while ensuring that the application performance meets the requirements in terms of data throughput.

However, due to the complexity of the interplay between the building blocks, adequately setting resource limits of the components separately for various data rates is nearly impossible.

In this paper, we propose a comprehensive approach that consists of measuring the resource footprint and data throughput performance of such a microservice-based application, analyzing the measurement results by data mining techniques, and finally formulating an optimization problem that aims to minimize the allocated resources given the performance constraints.

We illustrate the benefits of the proposed approach on Cortex, a horizontally scalable, highly available, multi-tenant, long-term storage service for Prometheus monitored metrics as time series data. The data-intensive nature of this choice of illustrative example stems from real-time monitoring of metrics exposed by a multitude of applications running in a data center, and the continuous analysis performed on the collected data that can be fetched from Cortex.

We introduce Cortex and its key microservice components, then we present the data write and read paths along with the performance vs. resource footprint trade-off.
Finally, we build regression models to predict the resource consumption of the microservices, and we draw a linear programming formulation to optimize the most important configuration parameters.

Összefoglaló

A felhőalapú számítástechnika folyamatos fejlődésével számos mikroszolgáltatás és összetett, többkomponensű alkalmazás készült, amelyek esetében az erőforrás-tervezés nagy kihívást jelent.
Ha például adatintenzív felhő-natív alkalmazásokról van szó, akkor a bérlőnek fontos lehet, hogy a felhő erőforrásokat gazdaságosan biztosítsa, miközben biztosítja, hogy az alkalmazás teljesítménye megfeleljen az adatok átbocsátóképességével kapcsolatos követelményeknek.

Az építőelemek közötti kölcsönhatások összetettsége miatt azonban szinte lehetetlen a komponensek erőforrás-korlátjainak megfelelő beállítása külön-külön a különböző adatátviteli sebességekre.

Ebben a tanulmányban egy átfogó megközelítést mutatunk be, amely egy ilyen mikroszolgáltatás-alapú alkalmazás erőforrás-lábnyomának és adatátviteli teljesítményének méréséből, a mérési eredmények adatbányászati technikákkal történő elemzéséből, és végül egy optimalizálási probléma megfogalmazásából áll, amelynek célja a kiosztott erőforrások minimalizálása a teljesítménykorlátok mellett.

A javasolt megközelítés előnyeit a Cortex-en, egy horizontálisan skálázható, nagy rendelkezésre állású, több felhasználóra kiterjedő, hosszú távú tárolási szolgáltatáson mutatjuk be, amely a Prometheus által megfigyelt metrikák, mint idősoros adatok számára lett kialakítva. A választott szemléltető példa adatintenzív jellege az adatközpontban futó alkalmazások sokasága által kitett metrikák valós idejű megfigyeléséből, valamint a Cortex-ről lekérhető, összegyűjtött adatokon végzett folyamatos elemzésből ered.

Bemutatjuk a Cortex-et és annak legfontosabb mikroszolgáltatás-összetevőit, majd bemutatjuk az adatírási és olvasási útvonalakat, valamint a teljesítmény és az erőforrás-felhasználás közötti kompromisszumot.
Végül regressziós modelleket építünk a mikroszolgáltatások erőforrás-fogyasztásának előrejelzésére, és lineáris programozási formulát írunk fel a legfontosabb konfigurációs paraméterek optimalizálására.

Table of contents

1 Introduction

Cloud computing is widely used currently in the digital industry as a technology that enables cheap and easy means not only to online web services but also to big data processing, machine learning, and storing data. The cloud is backed by actual physical data centers across the world which host virtual machines which are offered to customers and users. The cloud concept empowers the user to replace their hardware with cloud server instances and creates a new economic model where the customer pays only for the usage and not for the entire hardware itself. However, decision-making in cloud environments can be difficult due to the diversity in pricing models and service offerings. There are no rules of thumb as each customer could have a specific set of constraints and requirements of their cloud application when it comes to selecting the perfect cloud environment [14].

Microservices have recently gained striking popularity due to being highly maintainable and easy to develop. Microservices-based applications allow the deployment of each microservice (or component) to be in physically separated virtual machines if needed. Microservices, being separately manageable, have a huge advantage in scaling which is one of the most important features of the cloud context. Instead of launching multiple instances of the whole application, there is a possibility to only scale in or scale out the specific microservice component [13]. Creating microservices with the help of container technologies can result in very powerful and easy deployment closely followed by the small footprint which could be easily reached with containers.

When creating microservices, the goal is not only to create a small footprint application but also performance-wise to match the Quality of Service (QoS) of the original monolith. When it comes to data-intensive applications, data throughput is the most important aspect. The components of the application communicate with each other via APIs. Creating a microservice where the data throughput is relatively slow can result in a bottleneck in terms of the whole application’s performance. Increasing the bottleneck components’ resources might mitigate the issue, but this approach contradicts the low footprint goals.

To create a resource-efficient microservice, the most important settings to tune are the CPU and memory limits that control the resource usage of a microservice instance. These limits can prevent components from using more resources than needed, and with the help of these limits the app provider or operator can make sure in a private cloud that concurrent applications use the cluster as efficiently as possible, whereas in a public cloud limit help to pre-estimate costs.

When it comes to microservices, setting the limits for several components at once raises a complex optimization problem. Application components have their tasks while working closely with other components and if one of the components lags, the whole microservices-based application will suffer. On the other hand, resource provisioning must consider the shared resources: over-provisioning one component can cause starvation at other components. Furthermore, each component may have several configuration parameters which need to be tested in terms of resource usage to find the optimal limit values making the specific component work as efficiently as possible.

Section 2 presents the related work; in Section 3 we introduce building blocks for cloud native applications and microservices. We pick an illustrative example for microservice-based applications, Cortex: Section 4 introduces Cortex's most important components, while in Section 5 we focus on the data ingestion pipeline as well as the data retrieval path of Cortex. In Section 6 we define the regression model built on the measurements and create a linear programming formula for resource optimisation and showcase the Cortex example with numerical analysis. Section 7 concludes the findings.

2 Related work

The authors in [16] argue that performance modeling of microservice-based applications allows us to determine the capacity distribution among each microservice. This enables planning for applications and the detection of the bottleneck in microservices. [16] proposes to apply statistical models, e.g., Theil-Sen estimator or Support Vector Regression for this purpose. After analyzing the data acquired by their approach on the example applications, it was identified that the microservices follow a common pattern in the performance versus workload relationship which suggests the performance degrading with the increase of workload up until a certain point when all the virtual resources are used. Commonly in all tests, the CPU utilization increased linearly with the number of requests sent to the microservices. The approach was tested on several test applications including a compute-intensive application, a database accessing application, and a web accessing application. Similarly, to [16] we also want to create a model to detect and avoid bottlenecks; however, we also want to provision the resources individually for each microservice while using as minimal resources as possible.

Yanqi Zhang et al. [17] presented Sinan, a Machine learning driven (ML-driven) resource manager for microservice-based applications. Sinan highlights the challenges of managing complex microservices and leverages a set of scalable and validated ML models to reduce resource usage while meeting the end-to-end Quality of Service (QoS). Sinan trains two models with the traces: a CNN (Convolutional Neural Network) model for short-term performance prediction, and a Boosted Trees model that can evaluate long-term performance evolution. The combination of the two models allows Sinan to be effective in both near-future and distant future resource management. Similarly, to [17], our goal is to meet the QoS requirements with an analytical model which can calculate and predict the expected average resource usage.

The detection of a bottleneck component is important if we want to increase the performance of a complex application deployed in the cloud. [18] presents an analytical model that can detect bottlenecks and predict the performance of a multi-tier application. The suggested approach consists of two resource provisioning steps: a predictive one for long-term scales and a reactive one for short timescales. Long-term prediction is useful when the load can be predicted e.g., a typical daily pattern, but only reactive provisioning can handle unexpected high workloads. Using both and combining them can create an effective provisioning scheme. With this approach in a scenario where the workload of a three-tier application has been doubled, the technique showcased in [18] was able to double the application capacity within five minutes while maintaining the QoS (Quality of service) targets. Unlike [18], the focus is to predict and create resource limits for the average resource usage of microservices of any cloud-native application.

3 Cloud native building blocks

Launching and managing multiple applications (or microservices) in the cloud can be a great task. However, with the help of containers, container orchestration tools and monitoring applications these tasks could be sped up, be more efficient and less time consuming. In this section we will take a closer look at these solutions.

3.1 Docker

Docker is a container virtualization technology. Containers are lightweight and contain everything required to run the application and provide isolation for apps from the infrastructure. Docker's popularity comes from the quick shipping, testing, deploying possibility which means the time between writing the code and running it in production is significantly reduced. Because Docker is more lightweight than the previously used/popular hypervisor-based virtual machines; with the same resources containers can utilize more of the resources for the actual application(s) and need less for the background tasks [11].

3.2 Kubernetes

Kubernetes is responsible for managing containerized workloads such as Docker containers. With Kubernetes scaling and health checking become easy. Deploying containers and pods only require a single line of command with additional support for options that can be set in a configuration file. It can restart containers when they fail, replace these containers with new ones, and kill containers when they are not responding [10].
Kubernetes allows the system owner to limit the containers' resource usage. In Kubernetes, containers are grouped into pods where they share the storage and network resources with other containers. For easy use, one can deploy pods without specifying any option this way Kubernetes will use the default options provided with the pod. The problem with this, e.g., the developer will not have control over the pods' resource usage. There is a huge possibility that a pod might utilize a lot of resources and deploying it next to another pod will result in leaving only a minimal resource for the other pod which can make it unreliable or even worse: the pod will not be able to do its task. The solutions are built-in for Kubernetes: developers can control the CPU and memory usage for a pod by setting resource limits and requests in the pod's configuration file. Limits could be used to maximize the pods' resource usage while with the request option Kubernetes provides at least the specified amount of resources for the pod. On a higher level, maintainers can create namespaces and give limits in the namespaces so the pods cannot use up all the resources [9]. The question is how should we approach this limit and request problem when we want to be as efficient as possible? What are the trade-offs in terms of resource limitations e.g., when we apply Cortex for storing Prometheus metrics?

3.3 Prometheus

Prometheus is a highly popular, open-source systems pull-based monitoring and alerting toolkit written in Go and a Cloud Native Computing Foundation (CNCF) project. It can collect and store metrics as time series data, i.e., metrics information is stored with the timestamp, alongside optional key-value pairs called labels.
Prometheus has several useful features like Prometheus Query Language (PromQL), a flexible query language that lets the users select and aggregate time series data in real-time. Prometheus provides a multi-dimensional data model with time series data identified by metric names and labels. The time series are being collected by a pull model over HTTP protocol. The query results can be a graph, viewed as tabular data. There is no distributed storage, so every single server node is autonomous [2].

Prometheus's goal is to run with as few dependencies as possible; due to this philosophy, it uses only the local network to gather the metrics and stores the received data on the local disk. Even though the storage format can store data for a long time, truly durable long-term storage needs to consider disk failure, replication, and recovery while Prometheus does not provide solutions out of the box [8].

3.4 Cortex

Cortex is a horizontally scalable, multi-tenant, long term storage for Prometheus written in Go which is also a CNCF project. The foundation's goal is to support and help the advancement of container technologies. Cortex can be run across multiple machines in a cluster, exceeding the throughput and storage of a single machine. More well-known CNCF projects include Kubernetes, Helm, Prometheus and Thanos.

Cortex is able to send the metrics from multiple Prometheus servers to a single Cortex cluster and run aggregated queries across all data in a single place which is useful when the operator wants to get an overview about all of the metrics and data collected by these Prometheus servers. Cortex can also isolate data and query from multiple different independent Prometheus sources in a single cluster, allowing untrusted parties to share the same cluster and being able to prohibit unauthorized access to this information. When it comes to long term storage support Cortex supports S3, GCS, Swift and Microsoft Azure. This allows durable data stores for later use [1]. Figure 1 shows a commonly used architecture for a data lake; Cortex can be used in this architecture as a data collector, ingestor and storage microservice.

Prometheus instances can scrape metrics and data from various targets and push them to Cortex with the help of Prometheus' remote write API. Prometheus' remote write API creates and sends batched Snappy-compressed Protocol Buffer messages inside the body of an HTTP PUT request.

Cortex requires that each HTTP request come with a header specifying the tenant ID for the request. These requests' authentication and authorization are handled by an external reverse proxy.
Incoming sample writes (from Prometheus) are handled by the Distributor while incoming reads (PromQL queries) are handled by the Querier or optionally by the Query frontend. These two cases will be introduced and presented in this paper [1].

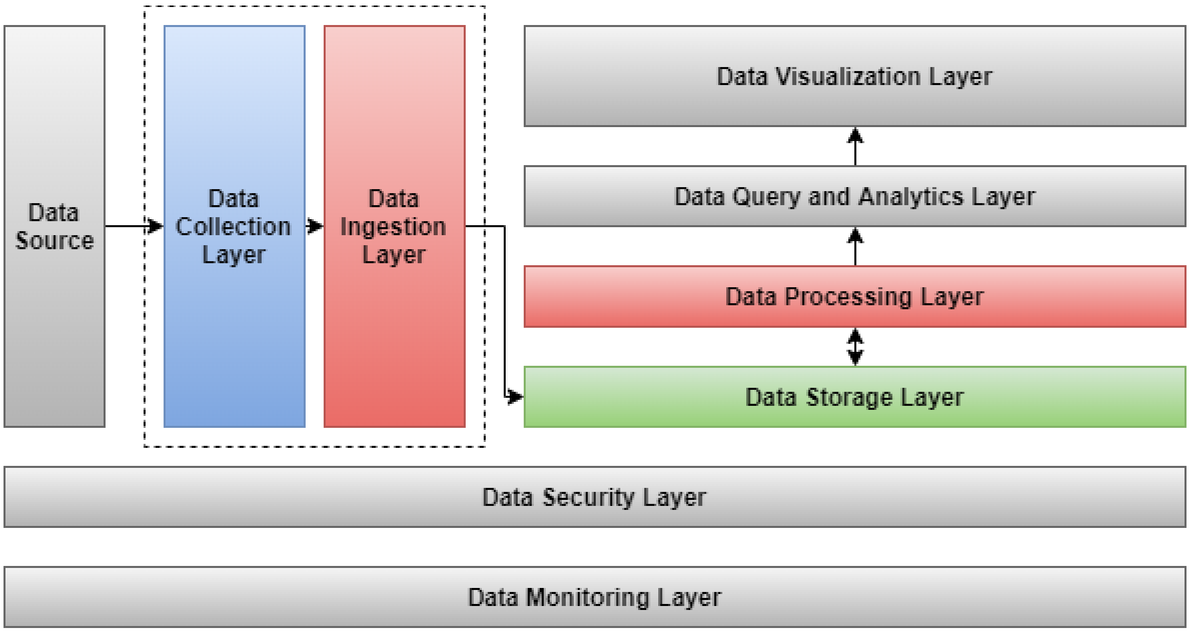


Figure 1: Common data lake architecture with colored extract, transform, load (ETL) procedures [19].

3.5 Thanos vs Cortex

In this section, we will compare Cortex and Thanos. Cortex and Thanos are two popular solutions used by many companies to scale out Prometheus in production. Cortex and Thanos are both in the CNCF Sandbox and initially started with different technical approaches and philosophies but over time they became similar when it comes to features and functionality.

Cortex has been designed for scalability and high performance since the beginning. When it comes to components Cortex has an all-or-none philosophy which means most of the components are not optional, there is a need to deploy most of the microservices. Cortex is using a push-based model which means the Prometheus server remote-writes to Cortex with the help of an API. Cortex uses both a NoSQL database and object storage.
Thanos on the other hand was originally planned for operational simplicity - Thanos can be rolled out incrementally. While Cortex uses a push-based model Thanos originally only supported a pull-based model which requires Thanos Queriers to pull out metrics from

Prometheus during the query. Thanos only uses object storage; there is no need for a database. Figure 2 shows an overview for both Cortex and Thanos when it comes to their used mode; Cortex's push-based model and Thanos's pull-based model.

Thanos' configuration is compact and easy to consume. With different design principles in mind, Cortex configuration allows the microservice operators to change every bit of the microservices, which turns out to be quite powerful when setting up a large production cluster but makes it more difficult to set up as most of these configurations have to be done before starting the microservices and even before making into the production pipeline.[7]

The Prometheus servers need to share the cluster with the target they are monitoring. This works extremely well when the application is deployed in one region but at a later stage the company starts deploying servers in multiple regions, therefore also needing to deploy Prometheus servers in each region which needs to be monitored. And when there is a need to run queries that cover data in multiple regions, there is no aggregated or specified place where these queries can be sent. This is the first problem that Cortex and Thanos set out to solve.

Thanos' solution is to reuse already existing Prometheus servers in the clusters. The Thanos Queriers spread apart queries to these existing Prometheus servers. In order to handle these queries and to help the Querier collect all of the metrics needed we need to deploy a sidecar (the Thanos Sidecar) next to the existing Prometheus servers. With the help of these Sidecars, the Querier can pull the data from these Prometheus servers.

In contrast with Thanos, in Cortex these requests go in the opposite direction – the existing Prometheus servers push data to a central, scalable Cortex cluster using Prometheus' built-in remote-write capability (API). The central Cortex cluster stores all the data and handles queries locally; if the Prometheus servers are not available all of the pushed data will still be available in Cortex.

Here we can see the first trade-off: the query's performance and availability. When using Thanos chunk data must be pulled back from the edge locations (from all of the Prometheus servers) to a central location where the Thanos Queriers are running. If there is an interruption in this wide-area network, the query performance and availability will be significantly slower and less reliable. In the worst case, the latest 2 hours of data may not be available for querying. In comparison to Thanos, Cortex relies on a push-based model to aggregate the data; when an edge location becomes unavailable, all data up to that point is still available to use. On the other hand, for this to work, we need to deploy a Cortex cluster and storage on top of the Prometheus deployment, while Thanos leverages the existing deployment.

Prometheus runs as a single process so if it’s needed to restart Prometheus, there is a possibility to end up with gaps in the data. The best and most common practice to avoid this issue is to deploy a pair of Prometheus servers in each monitored region, so if one fails, the idea is the other one continues to scrape the jobs and store metrics. But this does not solve the problem of the gaps – out of the box, Prometheus does not provide a way to merge data from multiple replicas.

When a pair of Prometheus servers are deployed the Thanos Querier will try to read from both replicas and combine these metrics into a single, no-gaps-included result. Thanos uses a heuristic to determine which results to show to avoid gaps. This step needs to be taken because there is a possibility that the two replicas are out-of-sync.

When it comes to this problem Cortex takes a different approach. As each replica is pushing samples to the cluster, Cortex deduplicates the streams internally and only store a single copy of it. Cortex also relies on tracking the last push from each replica connected to it and using a timeout to elect the other replica(s) as the master [8].

Comparing these two solutions there are no significant differences. Both can create gapless metrics which is the main goal when it comes to monitoring; to be able to monitor the applications without any issue or gaps and when these metrics need to be analyzed later, e.g., for troubleshooting all of the data need to be available to use.

Over time, the two projects started to influence each other, and these differences have been significantly reduced. Thanos started to support a push-based model with the receiver component, and many improvements have been made by the team both in terms of scalability and performance. Cortex’s operational complexity has been significantly reduced: the team introduced the single binary mode, removed some external dependencies. With these changes, Cortex has become more popular, easier to deploy and maintain [7].

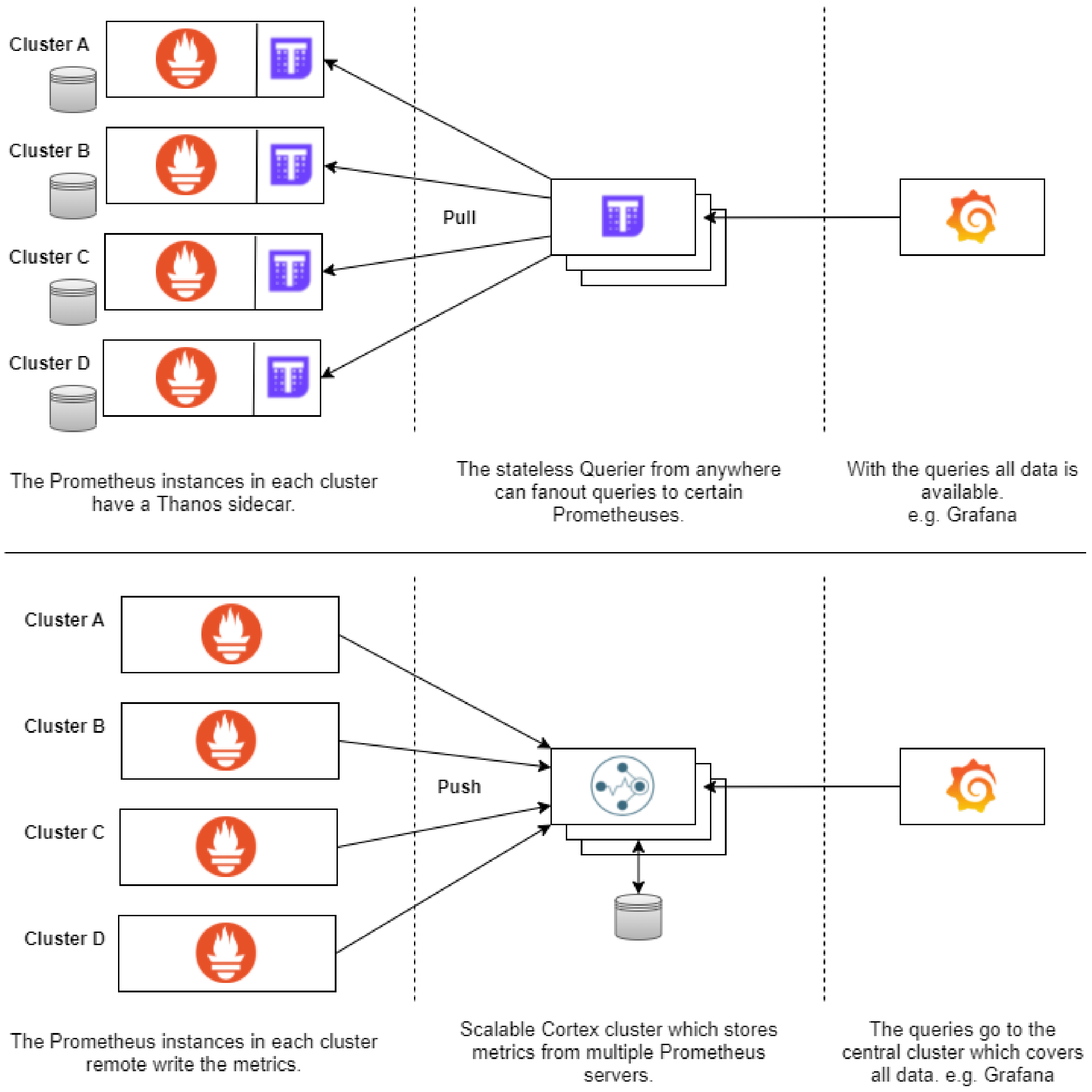

Cluster A
Cluster B
Cluster C
Cluster D
Pull
The Prometheus instances in each cluster have a Thanos sidecar.
The stateless Querier from anywhere can fanout queries to certain Prometheuses.
With the queries all data is available.
e.g. Grafana
Cluster A
Cluster B
Cluster C
Cluster D
Push
The Prometheus instances in each cluster remote write the metrics.
Scalable Cortex cluster which stores metrics from multiple Prometheus servers.
The queries go to the central cluster which covers all data. e.g. Grafana

Figure 2: Comparing Thanos and Cortex philosophy [8].

3.6 MinIO

Our long-term storage choice is MinIO, high-performance object storage. MinIO is S3 compatible, and it can handle unstructured data such as log files, backups, and container images with (currently) a maximum object size support of 5TB. The philosophy behind MinIO is to offer a high-performance infrastructure for machine learning, analytics, and application data workloads. The MinIO storage stack has three major elements which include the MinIO Server, the MinIO Client, and the MinIO Client SDK.

MinIO cloud storage server is designed to be as minimal and as scalable as possible. Designed with performance in mind, it offers many features including erasure coding, data decay protection [22], encryption, identity management, continuous replication, global federation, and multi-cloud deployments via gateway mode.

The MinIO Client, mc, provides an alternative to the well-known UNIX commands such as ls, cp, mirror, diff, etc. and adds support for S3 compatible cloud storage services.

MinIO Client SDK provides an API to access any Amazon S3 compatible object storage server. The API is available for Go, Java, Python, JavaScript, Haskell, and languages hosted on top of the .NET Framework. With the SDK all of the MinIO client commands and features are available [6].

4 Cortex components

Cortex consists of multiple horizontally scalable microservices from which we will introduce the most important ones in terms of performance and resource footprint in this section. Figure 3 contains most of the components used by Cortex including the external applications which vary on the use case. Write path is the microservice's data ingestion process where the collected and gathered data gets stored in the long-term storage after pre-processing. The read path on the other hand is the process when the data gets retrieved from the long-term storage. There are some components that are key components for both paths. When it comes to the write path, the key components are the Distributor and Ingester. The Ingester has a similarly important role when it comes to the read path. In addition to the Ingester, we will introduce the Querier, Query frontend (optional), Ingester, and the Store-gateway.

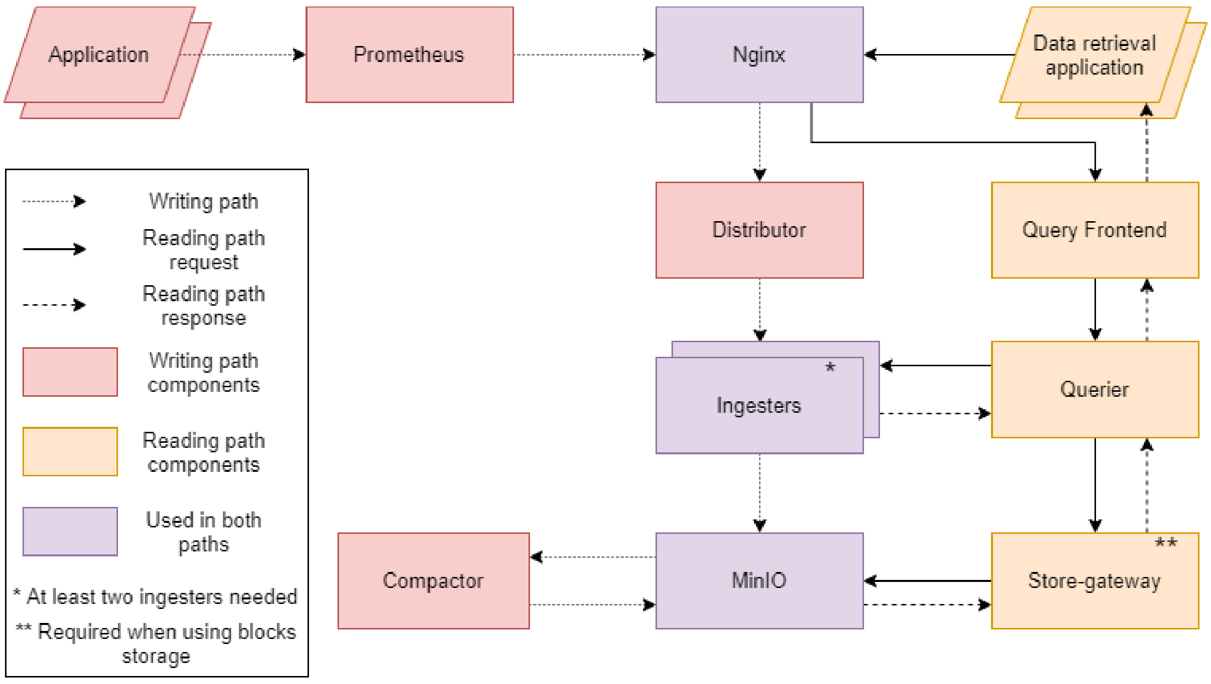


Figure 3: Cortex's architecture including the external applications [3].

4.1 Distributor

The Distributor microservice is responsible for handling the incoming samples of Prometheus. It's the first step in the writing path. When the Distributor receives samples from the Prometheus servers, each sample is checked and validated for the correctness and to make sure that it is within the preconfigured tenant limits and falling back to default ones in case limits have not been overridden for the specific tenant. The valid samples are then split into batches and get sent to multiple Ingesters in parallel by the Distributor.

The validation done by the Distributor includes checking the metric labels' names whether they are formally correct, checking the maximum number of labels per metric, the maximum length of label names and values whether they are being respected. In addition to these, the Distributor also checks if the timestamps are older/newer than the configured min/max time range [3].

4.2 Ingester

The Ingester microservice is responsible for writing incoming time series to a long-term storage backend (in our case into MinIO) on the write path and returning in-memory series samples for queries on the read path.
Incoming series are not immediately written to the long-term storage but kept in memory and periodically flushed to the storage (by default, 12 hours for the chunks storage and 2 hours for the blocks storage). For this reason, the Queriers may need to fetch samples both from Ingesters and long-term storage while executing a query on the read path.

If an Ingester process crashes or exits abruptly, all the in-memory series that have not yet been flushed to the long-term storage will be lost.
There are two main ways to handle this failure: replication and write-ahead log (WAL).

The replication is used to hold multiple (typically 3) replicas of each time series in the Ingesters. If the Cortex cluster loses an Ingester, the in-memory series held by the lost Ingester are also replicated at least to another Ingester. In the event of a single Ingester failure, no time series samples will be lost while, in the event of multiple Ingesters failure, time series may be potentially lost if failure affects all the Ingesters holding the replicas of a specific time series.

The write-ahead log (WAL) is used to write to a persistent disk all incoming series sample until they're flushed to the long-term storage. In the event of an Ingester failure, a subsequent process restart will replay the WAL and recover the in-memory series samples. [3]

4.3 Querier

The Querier microservice handles queries using the Prometheus Query Language (PromQL) query language. The Queriers fetch series samples both from the Ingesters and long-term storage: the Ingesters hold the in-memory series which have not yet been flushed to the long-term storage, of course, it will only fetch data from both when it is demanded because of the query's time range. Because of the replication factor of the Ingesters, there is a possibility that the Querier may receive a duplicated sample of time series; to resolve this, for a given time series the Querier will internally deduplicate these samples with the same timestamp [3].

4.4 Query frontend

The Query frontend is an optional microservice providing the Querier’s API endpoints and can be used to accelerate the read path. When the Query frontend is in place, incoming query requests should be directed to the Query frontend instead of directly to the Queriers. The Querier microservice will be still required within the cluster, to execute the actual queries the frontend does not query the required data.
The Query frontend internally performs some query adjustments and holds queries in an internal queue similarly to a database. Setting up like this the Queriers act as workers which pull jobs from the queue, execute them, and return them to the Query frontend for the aggregation if it's needed [3].

One more reason to use a Query frontend is that the frontend is capable of retrying a query on another Querier if the first should fail due to OOM or network issues [4].

The flow of the query in a system where Query frontend(s) are being used starts with the query being received by a Query frontend, which can optionally split it or serve from the cache. (if Cortex’s caching options are being used) If it cannot serve it from the cache the Query frontend stores the query into an in-memory queue, where it waits for a Querier to pick it up and to execute it. After the execution, the last step is to send the results back to the Query frontend and then forward them to the client [4].

It is recommended to add a readiness/health check to the Query frontend to prevent it from receiving queries while it is waiting for Queriers to connect. HTTP health checks are supported by Envoy, K8s, Nginx, and basically any commodity load balancer. The Query frontend would not indicate healthy on its health check until at least one Querier had connected [4]. We decided to use Nginx in this project [23].

4.5 Store-gateway

The Store-gateway is the Cortex microservice responsible to query series from blocks; each block is composed by chunk files containing the timestamp-value pairs for multiple series as well as an index, which indexes the metric names and the labels. The gateway should have an almost up-to-date, valid view over the storage bucket(s), in order to discover blocks more easily. Bucket is a unit of storage; a bucket typically stores one disk block, which in turn can store one or more records inside. The Store-gateway can keep the bucket view updated with periodically scanning the bucket - this is the default option -, or with periodically downloading the bucket index.

When the bucket index is disabled - this is the default option - following can be experienced: At start-up Store-gateways iterate over the entire storage bucket to discover blocks for all tenants and download the meta.json and index-header for each block. While running, Store-gateways periodically rescan the storage bucket to discover new blocks (uploaded by the Ingesters) and blocks marked for deletion or fully deleted since the last scan (as a result of compaction). The blocks chunks and the entire index are never fully downloaded by the Store-gateway. The index-header is stored to the local disk, in order to avoid to re-download, it on subsequent restarts of a Store-gateway [5].

When bucket index is enabled, the overall workflow is the same but, instead of iterating over the bucket objects, the Store-gateway fetches the bucket index for each tenant belonging to their shard in order to discover each tenant's blocks and block deletion marks.[5]

The Store-gateway supports the following caches: index, chunks and metadata cache.
The index cache can be used to speed up lookups of postings and series from the TSDB clocks indexes. There are two supported options here: in-memory and memcached.
The chunks cache can be used to store chunks fetched from the long-term storage. The chunks contain samples and can be reused when the user queries the same series for the same time range.
The metadata cache can be used to cache bucket metadata using Memcached. It can store and cache the list of tenants, list of blocks per tenant and the block's meta.json content.

Although caching is optional, it is highly recommended to use by the official documentation in a production environment. With the measurements we experienced the same and we also recommend using all of the caching options if it is possible [5].

5 Cortex operations

In this section we present the data paths. Writing data into long-term storage as efficiently and rapidly is as important as being able to retrieve the stored data. Some of the Cortex components are being used in both paths but there are specific microservices which are only needed in one of the paths. First, we will present the general data path, afterwards we present the findings with some illustrative figures.

5.1 Data ingestion of Cortex

Our hypotheses on the write path include high disk usage while running multiple Ingester replicas. While the Ingesters storing and writing time series into the long-term storage the data which they carry is stored on a persistent disk for restoring purposes. This means more replicas result in higher disk usage. Next to the Ingester the Distributor is expected to be CPU load heavy just like the Ingesters. For the Distributor component it is crucial to run without any issue because without a Distributor the writing path stops and there is a possibility the microservice will lose data in the process. With multiple replicas of Distributors and Ingesters we are expecting stable run and load balancing between the components.

When it comes to the writing path the Ingesters receive incoming samples from the Distributors which are responsible for handling the incoming samples of Prometheus. Each push request comes from a specific tenant, and the Ingester inserts the samples to the specific TSDB stored on the disk. The received data is both kept in-memory by the Ingester and written to a write-ahead log (WAL) which can be used to recover the in-memory time series if the Ingester unexpectedly terminates. These logs should be stored on a persistent disk which can be easily used after the Ingesters failed and need to be restored to be able to work again. This high overview can be seen on Figure 4 which shows how the metrics and time series gets collected by Cortex and how they can be queried after the deduplication with a Grafana microservice [8].

The time series data stored in-memory is periodically written to a local disk when a new TSDB block is being created, which by default happens every 2 hours. Each created block is then inserted into the long-term storage and kept in the Ingester until the given retention period expires. The reason for this is in order to give Queriers and Store-gateways microservices enough time to discover these new blocks on the storage and pull the required data for later use during querying.

Due to Cortex’s replication factor X - typically 2 or 3 -, each time series is stored by X Ingesters. Since each Ingester writes its own block to the long-term storage and not communicating with the other Ingesters, it leads to a storage utilization of X times more than the minimum or expected. The Cortex compactor microservice solves this problem in Cortex by merging blocks from multiple Ingesters into one, single block, and deduplicates time series [12]. This introduced pipeline can be seen on Figure 5 where not only the components are presented but the flow of the data from the external application to the long-term storage.

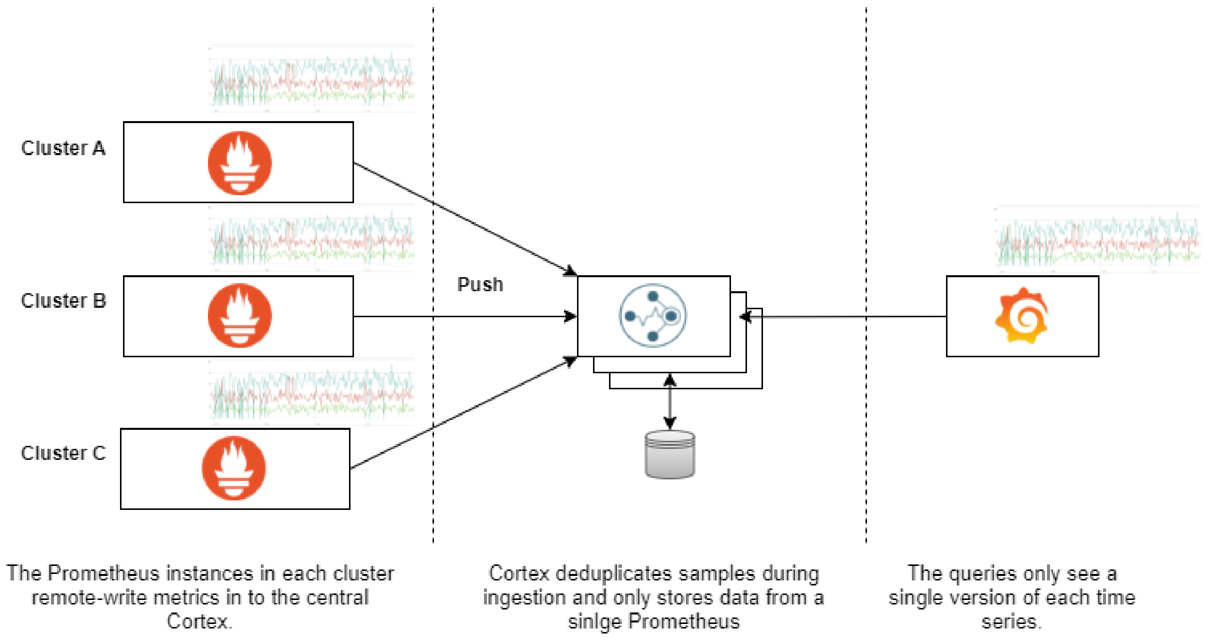

Cluster A
Cluster B
Cluster C
Push
The Prometheus instances in each cluster remote-write metrics in to the central Cortex.
Cortex deduplicates samples during ingestion and only stores data from a sinlge Prometheus
The queries only see a single version of each time series.

Figure 4: Cortex's writing path workflow [8].

This section presents the findings about Cortex's write path when we tested and measured the microservices with different configurations. It is very important to note that if the Distributor(s) crash or stop working, the collected monitoring data will be lost or not be written in the long-term storage which can have a negative impact when the user would like to query that specific time period. The compactor can save disk space with larger block ranges. Each block contains chunk files - which have the timestamp-value pairs for multiple time series - and an index, which indexes the metrics' names and labels.

Some rules of thumb to be taken into consideration when we are designing the writing pipeline are the following: every million time series in an Ingester takes approximately 15 GB of memory and consumes 15 GB of chunk storages while the indexes need ~4 GB of storage every day. The Distributor is capable of processing 100,000 samples per second with 1 CPU core while it does not require much memory during the process.

The Distributor's CPU consumption depends very much on the type of metric ingested by the pipeline. Prometheus and Cortex supports counters, gauges, histograms, and summaries. A counter can represent a single monotonically increasing value which can only increase or be reset to zero. Counters can be used to represent the number of requests served, tasks completed, or errors. A gauge can represent a single numerical value that can increase and decrease. They are typically used for measured values like current memory usage. A histogram samples observations and counts them in configurable buckets. They are typically used for things like request durations or response sizes. A summary is similar to a histogram, it samples observations while providing a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window. They are typically used for things like request durations and response sizes. The disk, CPU and memory usage of the Cortex write path components increases linearly with the number of time series given to Cortex. While the Ingesters can save disk usage when the WAL compression is turned on the CPU usage expectedly will increase however this increase is not relevant so using the WAL compression is recommended.

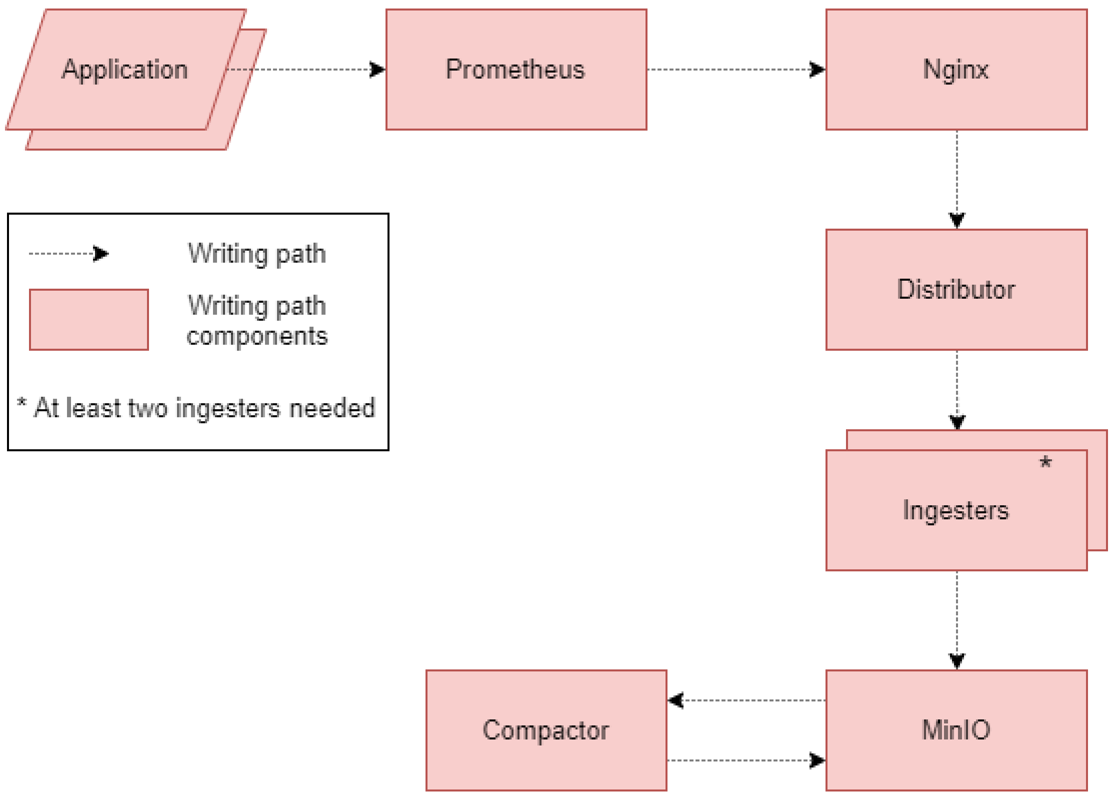


Figure 5: Cortex's writing path including the external, monitored application.

Overall, the write path depends on the specific parameter inputs: the number of time series written, the number of used Nginx, Ingester and Distributor components. The distribution between the number of Ingesters used by Cortex and the CPU consumption of the writing path when using the specific amount of Ingesters can be seen on Figure 6. The coloring shows the number of time series ingested by Cortex. On the writing path Cortex is using load balancing which aims for making the overall processing more efficient. As every replica needs a minimal resource to run the overall CPU and memory usage is higher when there are several replicas of each component. The disk usage is also higher when there are several Ingester replicas because every Ingester uses its own persistent disk. Using more replicas also provides more reliable operation which is important if we only use one Distributor and the component crashes, the writing pipeline stops and the writing process into the long-term storage also stops.

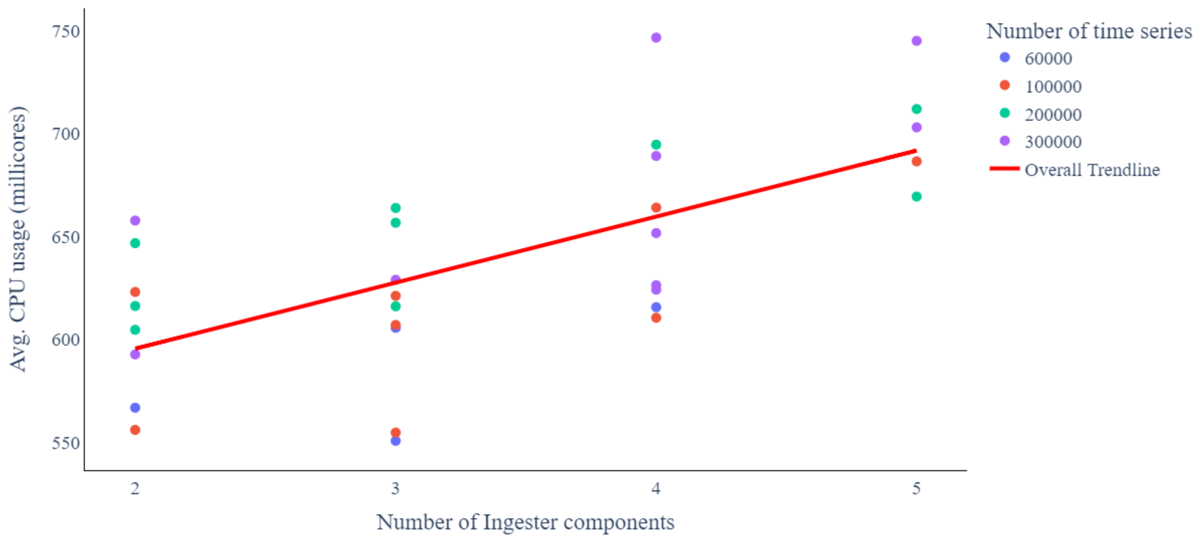


Figure 6: Writing path CPU usage by the number of Ingesters used with the coloring of the number of time series stored by Cortex.

5.2 Data retrieval from Cortex

Our hypotheses on the read (or query) path were a bit different. Comparing it to regular SQL databases we expected high CPU consumption with somewhat normal memory usage. The memory usage should depend on the component being able to retrieve the data in one step or need to create smaller batches and store the temporary data in memory. As Cortex components are scalable, we expected the Query frontend and the Querier to be able to work in parallel. This should result in higher CPU and potentially higher memory usage while speeding up the query time. Aggregating the data using the Cortex components should have a higher load on the whole microservice than the load when retrieving the data without any aggregation. We should expect longer query time and higher resource usage when we increase the number of time series retrieved by Cortex. From the initial first measurements, it was clear that the Querier, the Query frontend, and the store gateway will be resource-heavy while e.g., the Ingester and the long-term storage will not be affected as much as we expected.

Unlike querying a SQL database, with Cortex there is more steps to consider besides fetching data from the long-term storage (e.g., MinIO). The newest data (in the measurements the last 2 hours) are only stored in the Ingesters memory so if this data is needed, the Querier needs to think of these additional steps. Luckily with the Querier microservice Cortex does it on its own so there is no need for the developer to handle this step, however, later we will present a use-case where this problem needs to be tackled.

For optimization purposes, the Queriers and Store-gateways periodically scan through the storage bucket to discover blocks recently uploaded by Ingesters. The reason for this is to keep the stored statistics up-to-date. With the help of these statistics, the planner component can determine the most efficient execution plans for given queries. For every block, the Queriers download the block’s metadata file (which contains data about the min and max timestamp of samples within the specific block), while the Store-gateways download the metadata JSON as well as the index-header of the block, which is a small subset of the index used by the Store-gateway to search series at query time. The metadata blocks stored by the Queriers help to collect the list of blocks that need to be queried during query time and fetch the matching series of blocks from the Store-gateway instances holding the required blocks [12]. The reading path workflow can be seen on Figure 7. This path really depends on being able to serve the queries from indexes which need to be updated in every update window specified by the microservice.

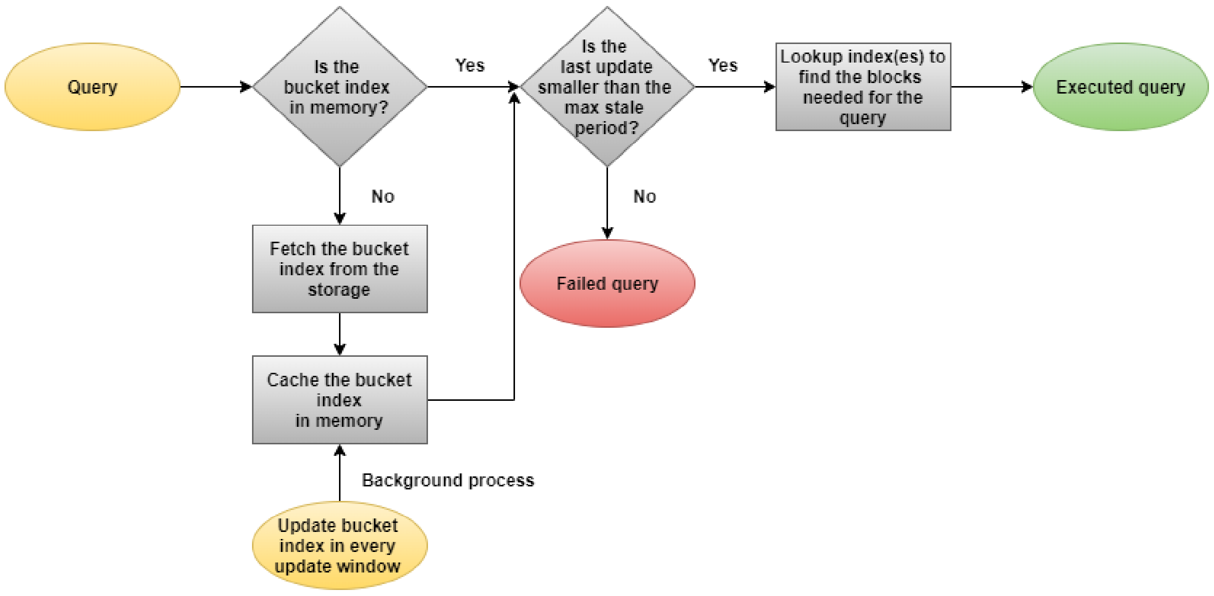

Query
Is the bucket index in memory?
Yes
No
Fetch the bucket index from the storage
Cache the bucket index in memory
Background process
Update bucket index in every update window
Is the last update smaller than the max stale period?
Yes
No
Failed query
Lookup index(es) to find the blocks needed for the query
Executed query

Figure 7: Cortex's reading path workflow.

This section presents the findings of Cortex's read path when we tested and measured the microservice with different configurations. We will focus on the Querier, the Query frontend, Store-gateway, and the MinIO. We started off by turning on all the caching options provided by Cortex. This decision was made by the documentation where the developers suggest that every production Cortex should utilize these caching options as they will help with load and query time. The Store-gateway supports the following caches: index, chunks, and metadata cache. The reading path components and their relation can be seen on Figure 8 including the external data retrieval data application.

As we tested multiple use cases, we found that querying huge amounts of data can make Cortex unstable and will increase the resource needs. As we used MinIO we tested the MinIO client from where we can access the stored data. This is the best method when an operator wants to pull all data stored by long-term storage as it only relies on the storage system and it will not influence the stable working of Cortex. If we want to retrieve the data with the help of Cortex, the resources needed will be 6-7x times higher compared to other queries. One thing to keep in my mind when querying this data manually from the long-term storage is that although Cortex provides deduplication and retrieves the data stored in the Ingester, when we use the MinIO client in this example we need to self deduplicate the data and also retrieve the stored data in the Ingester in a different query addressed to Cortex. MinIO supports two different methods for this data retrieval: sharing and mirroring. With sharing a custom URL can be generated which grants secured download access to the object. Mirroring can be used to synchronize data between file systems. Both of these methods are more lightweight than querying the data from Cortex.

When it comes to the resources the Query frontend and the store gateway were the two resource-heavy components. Both of them require high resources to be able to retrieve the data. As we used the frontend optional component the queries were targeted to this component. As we tested the system, we could clearly see that the frontend needs to have enough resources to handle the incoming queries, or it will drop the queries which cannot be scheduled or served. It can also result in crashing the component and if this happens none of the requests will get back to the user. After restarting the component, the queries can be sent to the frontend once again.

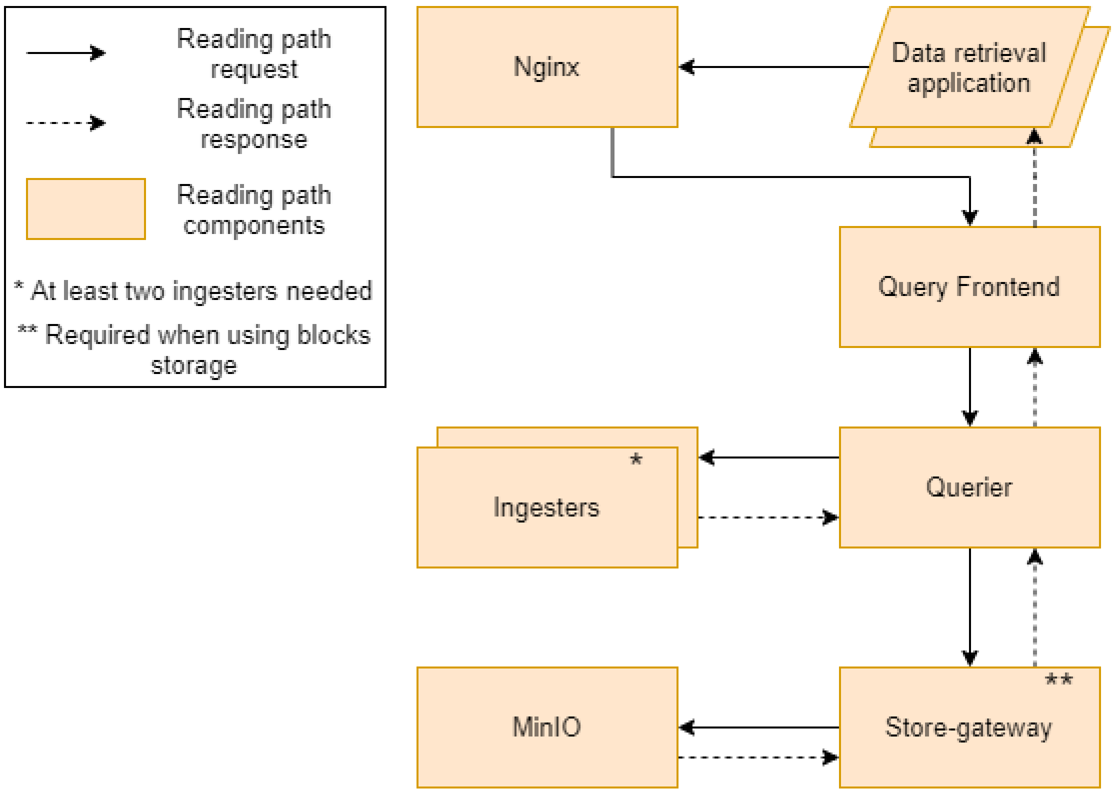


Figure 8: Cortex's reading path including the external, dara retrieving application.

The read path's components are efficient. When we created multiple replicas and queried data from Cortex, the microservice only used resources provided to one replica. Of course, when we use 3-4 replicas of Queriers the footprint is not the same as using 1. All of the components have a minimal required usage even in idle mode. Most of the measurements had a dominant replica which was selected by Cortex randomly and was used more than the other replicas and because of this used more resources than the others. The investigation suggests that using more frontend replicas can help reduce CPU consumption whereas with this method the overall memory usage will increase but this will only be useful and should be considered if there are several Queriers and the required data is big and for that, the queries will last long. The memory usage highly depends on the complexity of the query and the amount of data retrieved in that query. If the Querier can retrieve the data in one batch then there is no significant memory usage as there is no temporarily stored data in the component; the data is accessible in one piece and can be forwarded to the data retrieval application with the help of the frontend.

This can be helpful when the read path needs to serve resource-heavy queries 3-4 times a week, but it also needs to serve light queries e.g., every few minutes. With this insight, we know that Cortex will only use as much resource as needed for that specific lightweight query.

When we were looking for features for the regression model, we immediately started with the replica numbers as these are important features on the writing path. Unfortunately, as we wrote before, replica numbers are not important features when it comes to the model because Cortex is not doing any sort of load balancing; it will only use one replica per component if that one replica can handle the request. We did not want to test the difference between turning on and off caches and using these variables as model features because the documentation says for the production it should be always on.

Overall, the read path depends on the specific parameter inputs: the number of time series queried, the interval of the data, and the aggregation of the data. Prometheus and Cortex support built-in aggregations such as sum, min, max, average, quantile, etc. It is useful when the microservice stores the data for every second but for a specific use case the developer only needs the data every other minute. The number of time series and the interval of the data are the two inputs that have the highest correlation with resource usage. Using a bigger interval while aggregating the data can also reduce CPU usage and query time. Figure 9 represents this finding; the plot shows the distribution between the aggregation and the CPU consumption of the read path.

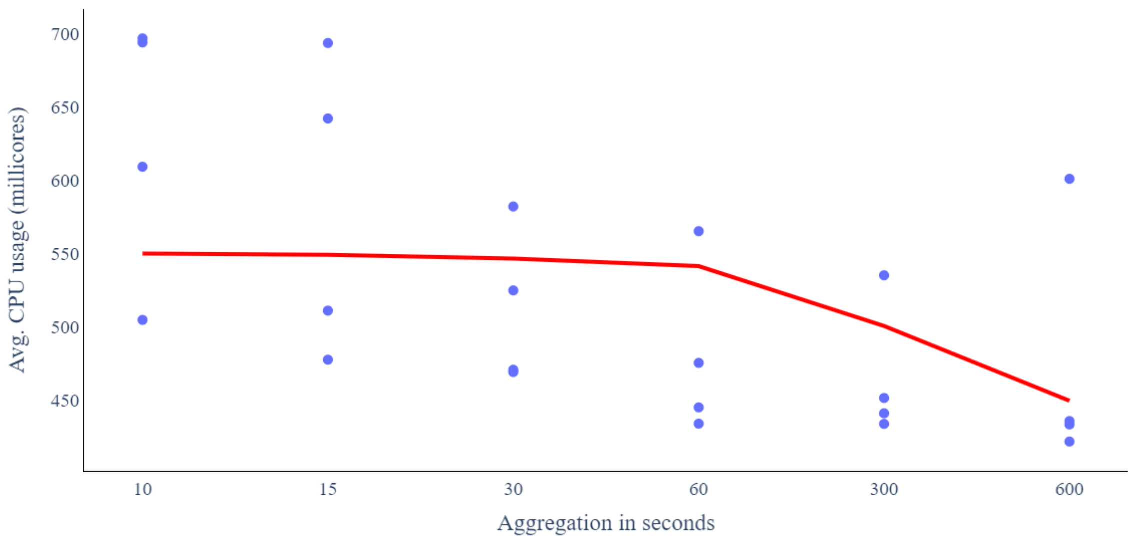


Figure 9: Reading path CPU usage vs. the level of data aggregation in seconds.

6 Methodology and model

We use Cortex to demonstrate the optimization methodology applicable to any microservice-based cloud-native application. We analyzed the performance vs. resource footprint trade-off of both the reading and the writing paths, then we applied data mining and optimization techniques to determine the necessary resource provisions for each microservice given a user-defined QoS demand.

6.1 Measurements and feature selection

We deployed Cortex in a Kubernetes cluster and monitored the CPU usage of each component. Along with the measurements, we gathered insights that are not documented at all. On the writing path, we ran every test for a couple of hours to check and verify the whole pipeline's CPU usage. This has to be done as there are components that are only active at the beginning or the very end of the pipeline. As for the reading path we tested our queries multiple times and created a data retrieval application that created different queries to avoid serving data from cache.
Based on the experiments and tests, we selected the most important parameters of the microservices. This is important to find the features which infer total resource usage. We selected the important features by first testing all potential features and after the initial tests, we went through all the tested features and selected the dominant ones.

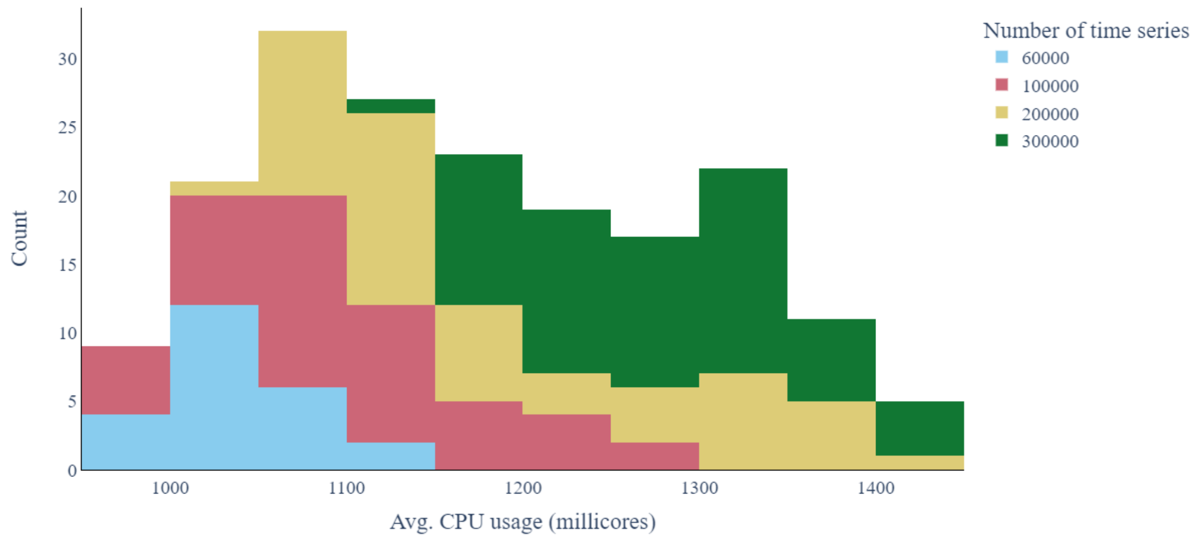
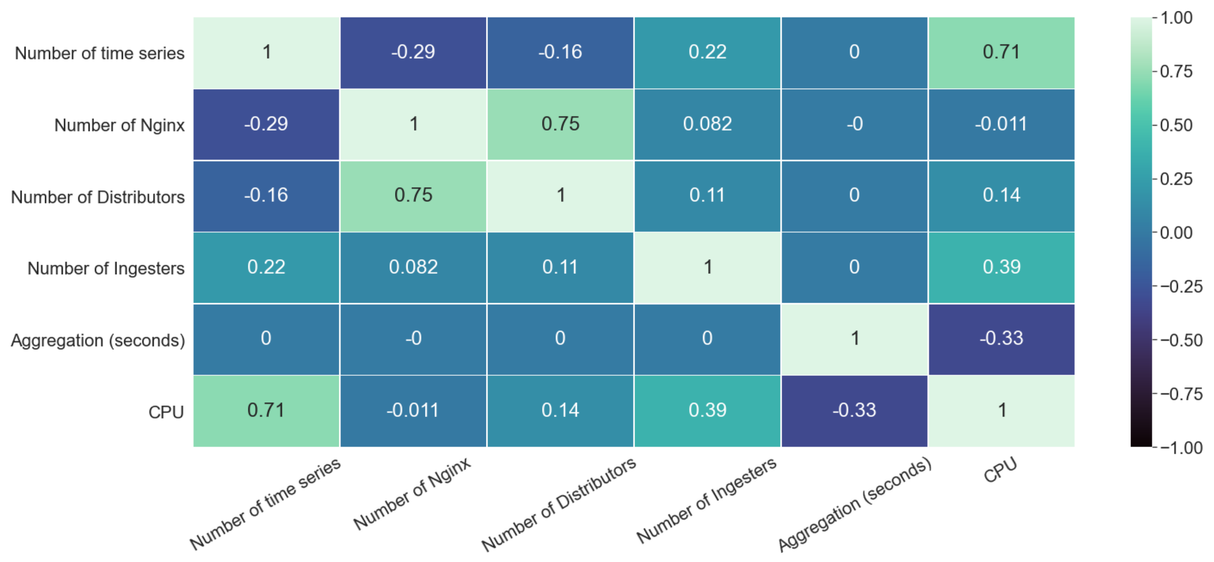


Figure 10: Distribution between the number of time series used by Cortex and the average CPU used.

Our target function is the microservice’s overall average CPU usage. The CPU consumption includes every component of Cortex and also Prometheus on the writing path to get an overview on not only the active components but on all components working in the background while running the tests. The average CPU usage varies between 900 and 1.600 millicores.
The two paths can be connected by the number of time series and their average CPU consumption can be summed.

From the writing path we will be using the number of time series continuously written into Cortex and the number of Nginx, Distributor and Ingester components. These components are used in the writing path, scaling other components will not change the CPU usage of the write path. WAL compression could have been a feature but as introduced before actually turning on this option is recommended as it saves disk usage while increasing CPU consumption, but this latter is marginal.
On the writing path we wrote several blocks into MinIO. This way we could make sure we tested the whole writing pipeline numerous times. We wrote 60,000-300,000 time series and deployed 1-3 replicas of each featured component. The only exception is the Ingester which should not be deployed in a single instance, instead we deployed 2-5 of them. The distribution between the number of time series ingested by Cortex and the average CPU usage can be seen on Figure 10.

From the read (query) path we will be using the number of time series queried in Cortex and the aggregation (or step as Prometheus calls it) as important features. As we previously wrote, on the reading path the replica numbers cannot be used as the query side of Cortex is not using load balancing and the microservice will only use as many components as needed.
We queried time series between 60,000-300,000 (to match the write path’s numbers) and aggregated the data inside Cortex. The aggregation was between 10 seconds and 1 hour with various aggregation levels. With every test we queried the last 4 hours from Cortex while writing data inside it to ensure we always have the latest data in the long-term storage and in the Ingesters. The requests were targeted towards Nginx; it forwarded it to the Query frontend which could optionally transform the query into smaller queryset batches and send it to the Querier which was doing the aggregation and the data retrieval.



	Number of time series	Number of Nginx	Number of Distributors	Number of Ingesters	Aggregation (seconds)	CPU
Number of time series	1	-0.29	-0.16	0.22	0	0.71
Number of Nginx	-0.29	1	0.75	0.082	-0	-0.011
Number of Distributors	-0.16	0.75	1	0.11	0	0.14
Number of Ingesters	0.22	0.082	0.11	1	0	0.39
Aggregation (seconds)	0	-0	0	0	1	-0.33
CPU	0.71	-0.011	0.14	0.39	-0.33	1

Figure 11: The two paths' correlation matrix.

From the correlation matrix on Figure 11, we can see that the most dominant feature is the number of time series. This comes from mostly the writing path as the reading path is being influenced by the query aggregation. This can vary as the queries retrieval can be created by different approaches. From the writing path the number of Ingesters is also an important feature in contrast to the number of Distributors and Nginx.
The correlation's conclusion is the following: the most important thing to plan when creating a data ingestion pipeline is the number of time series stored and later queried. Using several Ingester can balance the writing path and can reduce the possible crashes which is important because this component stores the metrics and time series before it gets written into the long-term storage. On the reading path using higher aggregation can lower the CPU usage as well as only querying as much time series as needed.

6.2 Linear regression models

After all of the tests and measurements were done, we created different regression models for each component and trained them. We used linear regression models because the chosen features showed linear dependencies with the target dependent variable, i.e., CPU consumption. We only needed to focus on the components which have an important role in any of the paths. Most of the other components' resource usage could be seen and taken as constant values; not significantly depending on any of the test settings.

When it comes to statistical or machine learning algorithms the most well-known is probably linear regression. Linear regression is a linear model which means that from a linear combination of input variables (x) an output variable (y) can be calculated. If the input is a single variable, it is most often referred to as simple linear regression. When there are multiple input variables, we can call the method as multiple linear regression. Today linear regression is often used to examine if a set of predictor variables can precisely predict the dependent variable. If this is possible with the data regression analysis can use this dataset to forecast different effects.

In a simple regression problem (where the is a single x and a single y), the form of the model would look like this:

$$y = A * x + B,$$

where y = predicted variable, B = constant (called as intercept), A = regression coefficient [15].

Our linear regression model comes from the two paths we introduced before: write path and read (query) path. The reason we are creating one aggregated model is that for a microservice which is used for data ingestion not only will it be used for writing or reading interchangeably; it will be used continually by both paths so creating an aggregate model which takes features and information about the different paths at the same time is important. With this philosophy we can also give a higher view which can be useful when designing data processing pipelines and microservices like this.

Features	**Coefficients**
Number of time series	9.28
Number of Distributors	32.54
Number of Ingesters	24.21
Number of Nginx	1.91
Query aggregation	-0.182
Intercept	129.3

Table 1: The regression coefficients.

Table 1 shows the coefficients of the features used in the model. The performance of the regression model is best reflected by R^2 - the coefficient of determination - which was around 0.75. The Mean Absolute Error (MAE) is around 58.2. As the microservice requires a minimum CPU and of course memory resources we created an offset of 750 millicores. This has to be added at the end when this model is being used. For better understanding the coefficients we also divided the number of time series with 10,000. With this data transforming step the coefficient (9.28) is for every 10,000 time series used. Figure 12 shows all the data points used for this model and the distribution between the overall average CPU consumption and the stored and queried number of time series.

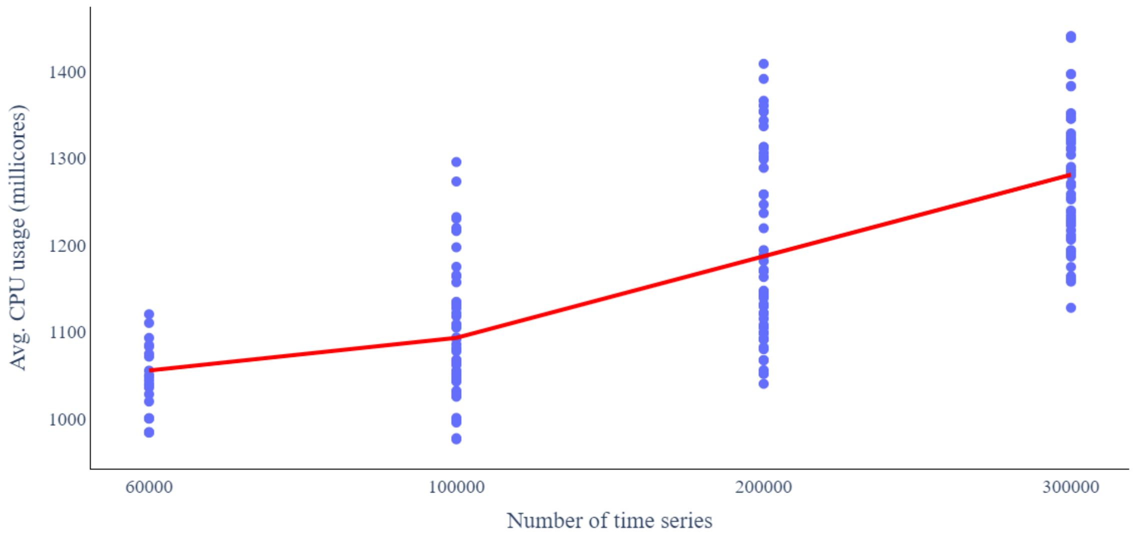


Figure 12: The two paths combined CPU usage by the number of time series used by Cortex.

6.3 Integer linear programming for optimization

With a complex regression model including both the read and write path, we created integer linear programming optimizations for each component with specific constraints. With this in mind, we also configured the microservice with the findings and the results of the optimization.

Linear programming (LP) is a method that can be used to achieve the best possible outcome in a mathematical model whose requirements are represented by linear relationships. Linear programming is a special case of mathematical optimization. An integer programming problem is a linear programming method in which some or all of the variables are restricted to be integers only. In many settings, the term refers to integer linear programming (ILP), in which the objective function and the constraints are linear.

Integer programming is nondeterministic polynomial-time complete (NP-complete). This means that the problem for which the correctness of each solution can be verified quickly, and a brute-force search algorithm can find a solution by trying all the possible solutions and when the problem can be used to simulate every other problem for which we can verify quickly that a specific solution is correct [20].

We minimize CPU usage for the user inputs, i.e., the number of time series used in the system, the available memory for the microservices, and the data aggregation given in seconds for the query part. The output is the average CPU utilization expected by components; the number of Ingesters, Nginx, and distributors suggested to use, and whether the given memory is enough for the microservices. As we previously discussed the number of time series is one of the most dominant features and the amount of resources needed for the microservice heavily depends on this number. As for the aggregation, the average CPU usage on the reading path can be lower while using a higher aggregation window.

The decision variables are the following:

$$\begin{aligned}
&T\ =\ number\ of\ time\ series\ used\ [10{,}000]\\
&I\ =\ number\ of\ Ingester\ instances\\
&D\ =\ number\ of\ Distributor\ instances\\
&N\ =\ number\ of\ Nginx\ instances\\
&Q\ =\ level\ of\ data\ aggregation\ [sec]\\
&M\ =\ memory\ resource\ available\ [MB]
\end{aligned}$$

Our CPU offset is 750 millicores and the intercept is 129.3 millicores. These two values are important to keep in mind because they are not presented in the optimization equation because these are constant values but need to be added to the overall CPU usage.

User input: T, Q, M.

Minimize $\quad 9.28 \cdot T + 32.54 \cdot D + 24.21 \cdot I + 1.91 \cdot N - 0.182 \cdot Q \qquad (1)$

Subject to

$$I > 1 \tag{2}$$

and

$$D > 1 \tag{3}$$

and

$$N > 1 \tag{4}$$

and

$$T > 0 \tag{5}$$

and

$$Q > 0 \tag{6}$$

and

$$128 \cdot N + 512 \cdot D + 1500 \cdot I + 1000 \cdot T / 10 < M \tag{7}$$

and

$$T/100 - I > 0. \tag{8}$$

Our objective function is (1) where we want to minimize the CPU resource given by the user's input.
We are using (2), (3), (4) constraints for high availability between the components. As previously discussed, multiple replicas are useful for load balancing on the writing path but also important for availability; for example, a Distributor failure can result in a temporary stop on the writing path and can potentially create data loss if not handled correctly. As for (5) and (6) the constraint is for nonnegativity. When it comes to memory and memory usage we previously showed the memory usage cannot be modeled with the same approach we used for the CPU. From the measurements, we managed to formalize the important component's memory needs and tested these approaches to find the minimal memory needed for these components, this is Constraint (7). There is a constant need for all of the components which are not in the model; these components need approximately 1,500 MB (megabyte) memory when only one replica per component is available in the microservice. As for the 3 components which are used in the model an Ingester needs approximately 2,000 MB memory when used by both paths parallel, an Nginx only needs 128 MB memory while the Distributor needs approximately 512 MB. From the tests, we can also confirm that

not only the CPU usage is being influenced by the number of time series but also the memory usage, memory needs. From the measurements, we can state that for every 100,000 metrics there is a need for another 1,000 MB by the microservice. This additional resource is being divided between the components proportionately. Constraint (8) is also for high availability. From the measurements, we found that launching an additional Ingester replica for every 100,000 time series used can help load balance the writing path and provide safer operation.

After the optimization model calculates the average CPU usage and uses the appropriate I, D, N numbers it is useful to be able to translate the results to every component. From the number testing, we can state that Table 2 indicates the distribution of the CPU among the microservices.

Component	**% of total CPU usage**
Ingester	32%
Distributor	26%
Querier	14%
Query frontend	13%
Store gateway	9%
Nginx	6%

Table 2: The components' CPU consumption distribution.

6.4. Illustrative user input examples and optimization results

To demonstrate the optimization, we created 3 examples where we gave input to the model and we got back the optimized solution. We used PuLP [21] for these demonstrations. Many libraries offer the option to pick the solver which will be used. The used solver is the COIN-OR Branch and Cut Solver (CBC) which is a pen-source mixed integer linear programming solver written in C++. Other solvers include GNU Linear Programming Kit (GLPK), LP Solve, Coin-or linear programming (Clp), etc.

With the first example, the inputs are 300,000 time series with 12 GB (gigabyte) memory available and the aggregation should be 10 seconds.
The model suggests using 3 Ingesters and 2-2 Distributors and Nginx. Our expected average CPU usage will be around 1,300 millicores and the memory usage should be around 11.8 GB. This test shows that the expected memory usage will be around our available amount.

With the second example, the inputs are 400,000 time series with 16 GB (gigabyte) memory available and our aggregation should be 60 seconds.
The model suggests using 4 Ingesters and 2-2 Distributors and Nginx. Our expected average CPU consumption will be around 1,405 millicores and the memory usage should be around 14.8 GB.
This test shows that the expected memory usage will be enough and the microservice will still have some available amount of memory if needed.

With the third example, the inputs are 120,000 time series with 9 GB (gigabyte) memory available and the aggregation should be 15 seconds.
The model suggests using 2 Ingesters and 2-2 Distributors and Nginx. The expected average CPU usage will be around 1,105 millicores and the memory usage should be around 8 GB. This test shows - just like the first one - that the expected memory usage will be around the available amount.

With the fourth example, the inputs are 250,000 time series with 8 GB (gigabyte) memory available and the aggregation should be 120 seconds.
The model suggests using 2 Ingesters and 2-2 Distributors and Nginx. The expected average CPU consumption will be around 1205 millicores and the memory usage should be around 11,3 GB. Here the model shows that using 8 GB memory will not be enough so we should solve the problem by using more memory or reducing the time series to e.g., 100,000.

We summarized the examples in Table 3. These tests show that the Distributor and the Nginx replica should be 2. This is because of high availability but using more replicas for these components will not give any benefit so the minimization results in the lowest number of instances possible. The Ingester number depends on the number of time series used but also on the available memory.

Time series used [1,000]	Memory available (GB)	Aggregation (sec)	Ingester replica number	Distributor replica number	Nginx replica number	Predicted CPU usage (millicores)	Cortex planned CPU (millicores) [23]	Relative saving
300	12	10	3	2	2	1,300	6,000	78%
400	16	60	4	2	2	1,405	8,000	82%
120	9	15	2	2	2	1,105	2,400	54%
250	8	120	Infeasible				5,000	-
250	12	120	3	2	2	1,218	5,000	76%

Table 3: Illustrative examples summary table with Cortex's recommendation.

The paper's most important takeaway message can be found in Table 3. Although Cortex's documentation recommends capacity planning [23], these resources are not optimal when we are trying to reduce and keep the resource usage as low as possible. The illustrative examples show not only the documentation's recommendation but also the relative saving which we managed to achieve with the regression and optimization models. From the 4 examples this saving ranges from 54-82%. If we translate these numbers to cost, i.e., we use these microservices in the cloud and paying for being hosted, then we can save 54-82% of the total operational cost.

7 Conclusions & future work

In this paper, we argued that resource provisioning is important, and in fact possible to perform in a scientific manner when using cloud native microservices. We proposed an approach that we demonstrated on a data ingestion and storage cloud native application, Cortex. After measuring the performance vs. resource footprint trade-offs with several different configuration settings, and creating a regression model for resource usage, we showed the most important features and their quantitative effects on CPU utilization. Then we introduced an integer linear programming formulation that can be solved to minimize the expected CPU usage for given user inputs.

The main focus of further work is to create a model for memory usage as well. For that, we need to take a different approach as the current one can only predict constant memory usage. An example can be the read path of Cortex where the memory usage is expected to be constant as the Querier can serve the query and for that reason, there is no data stored in the memory.

8 References

[1] Cortexproject/Cortex [Online code repository], Available:
https://github.com/cortexproject/cortex (accessed: 2021-10-25).

[2] Prometheus/Prometheus [Online code repository], Available:
https://github.com/prometheus/prometheus (accessed: 2021-10-25).

[3] Cortex Architecture [Online code repository], Available:
https://cortexmetrics.io/docs/architecture (accessed: 2021-10-25).

[4] Joe Elliott. 2020. Scalable Query Frontend.
https://cortexmetrics.io/docs/proposals/scalable-query-frontend

[5] Blocks Storage/Store-gateway [Online code repository], Available:
https://cortexmetrics.io/docs/blocks-storage/store-gateway (accessed: 2021-10-25).

[6] MinIO/MinIO [Online code repository], Available:
https://github.com/minio/minio (accessed: 2021-10-25).

[7] Marco Pracucci. 2020. How the Cortex and Thanos projects collaborate to make scaling Prometheus better for all
https://grafana.com/blog/2020/07/16/how-the-cortex-and-thanos-projects-collaborate-to-make-scaling-prometheus-better-for-all

[8] Tom Wilkie. 2019. Two Households, Both Alike in Dignity: Cortex and Thanos
https://grafana.com/blog/2019/11/21/promcon-recap-two-households-both-alike-in-dignity-cortex-and-thanos

[9] Peter Arijs. 2018. How to use resource requests and limits to manage resource usage of your Kubernetes cluster. Available:
https://jaxenter.com/manage-container-resource-kubernetes-141977.html

[10] What is Kubernetes? Available:
https://kubernetes.io/docs/concepts/overview/what-is-kubernetes (accessed: 2021-10-25).

[11] Docker overview. Available:
https://docs.docker.com/get-started/overview (accessed: 2021-10-25).

[12] Blocks Storage. Available:
https://cortexmetrics.io/docs/blocks-storage (accessed: 2021-10-25).

[13] Chris Richardson. What are microservices? Available:
https://microservices.io

[14] Faiza Samreen, Yehia Elkhatib, Matthew Rowe, Gordon S. Blair. Daleel: Simplifying cloud instance selection using machine learning
NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium
https://ieeexplore.ieee.org/document/7502858

[15] Jason Brownlee. 2016. Linear Regression for Machine Learning. Available: https://machinelearningmastery.com/linear-regression-for-machine-learning (accessed: 2021-10-25).

[16] Anshul Jindal, Vladimir Podolskiy, Michael Gerndt. 2019. Performance Modeling for Cloud Microservice Applications.
ICPE '19: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering
https://dl.acm.org/doi/10.1145/3297663.3310309

[17] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, Christina Delimitrou. 2020. Sinan: Data-Driven Resource Management for Interactive Multi-tier Microservices
https://www.csl.cornell.edu/~delimitrou/papers/2020.mlarchsys.sinan.pdf

[18] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal. 2005. Dynamic Provisioning of Multi-tier Internet Applications
Second International Conference on Autonomic Computing (ICAC'05)
https://ieeexplore.ieee.org/document/1498066

[19] Navdeep Singh Gill. 2021. Blogs and Insights on Cloud, DevOps, Big Data Analytics, AI-XenonStack.
https://www.xenonstack.com/blog/big-data-ingestion

[20] Mirko Stojiljković. Hands-On Linear Programming: Optimization With Python. https://realpython.com/linear-programming-python (accessed: 2021-10-26).

[21] PuLP. [Online code repository], Available:
https://coin-or.github.io/pulp (accessed: 2021-10-25).

[22] Data degradation [Wikipedia], Available:
https://en.wikipedia.org/wiki/Data_degradation (accessed: 2021-10-26).

[23] Nginx. Available:
https://www.nginx.com (accessed: 2021-10-26).

[23] Capacity Planning.
https://cortexmetrics.io/docs/guides/capacity-planning (accessed: 2021-10-26).