Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Efficiency analysis of cloud native architectures on interdependent data stream processing

Scientific Students' Association Report

Author:

Márton Géza Pfemeter

Advisor:

Balázs Fodor
Dr. Balázs Sonkoly

2023

Contents

Kivonat

A felhő manapság egy kiforrott technológia, számos felhasználási területtel. Olyannyira a mindennapjaink részévé vált, hogy szinte már elképzelhetetlen olyan tárgyterület, ahol az ott alkalmazott szoftvermegoldás ne használná fel azt. Ilyen széleskörű felhasználás mellett bármilyen fejlesztés ezen a területen komoly hatást tud elérni.

Az egyik leggyakoribb feladata a felhőnek nagy mennyiségű adat feldolgozása. Ez az adat sokfajta formában lehet elérhető, azonban legtöbbször rekordok folyamaként érdemes rá tekintenünk, melyeket valamilyen módon módosítanunk, kezelnünk kell. Ennek a formátumnak köszönhetően lehetséges a folyamatos és párhuzamos feldolgozás megvalósítása, amire a felhőarchitektúrák tökéletes megoldást nyújtanak.

Bizonyos tulajdonságai a felhasznált adatforrásoknak azonban megnehezíthetik ezeknek az architektúráknak az implementációját. Nehézséget okozhat például amennyiben több, egymástól függő adatforrásból szeretnénk előállítani egyazon kimeneti adatrekordokat. Ezenfelül még amennyiben több lehetséges megoldási terv is születik, nem lehetünk biztosak benne, hogy az adott feladatra melyik milyen hatékony, de még abban sem melyik a leghatékonyabb az összes közül. A dolgozatomban egy konkrét implementációs feladaton szemléltetem a korábbi problémakört, ahol egy General Transit Feed Specification (GTFS) szabvány szerinti közlekedési adatforrást dolgozok fel. A GTFS szabvány jellegéből adódóan, a feldolgozás során több különböző, de egymástól függő adatfolyamot kell egyszerre kezelni egy közös kimeneti rekordhalmaz előállításához. A feladat példaként szolgál egy tipikus esetre, amikor is egy külső, számunkra módosíthatatlan formátumú adatforrást szükséges lefordítani egy általunk definiált belső reprezentációra.

Dolgozatomban több különböző architekturális és implementációs párosítást is megvizsgálok a hatékonyságuk szempontjából a GTFS szabványú adatforrás feldolgozására. Összehasonlítok egy mikroszolgáltatásokon, és egy függvényeken (Function-as-a-Service, FaaS). Elemzésemben egy általam kialakított vizsgálati környezetben végzem a teljesítménymérést a javasolt megoldási módszereken. A mérési környezetben különböző erőforráshasználat (CPU, memória) mellett hasonlítom össze a felhasználó számára legfontosabb metrikát, a teljes válaszidőt (a program teljes futásidejét). A mérhető eredmények mellett bemutatom az egyes megközelítések egyéb előnyeit és hátrányait.

Abstract

Cloud computing today is a well established technology, with numerous use cases. In fact, in recent years, it has crept into nearly every domain in which software solutions are being used. With such a wide range of applications, any advancement in this field is due to have a significant impact.

One of the most common applications of cloud computing is for the processing of large amounts of data. This data can come in many forms, though it can most commonly be considered as a stream of records which need to be manipulated in some way. Due to this format of the data sources, continuous and parallel processing is possible, for which cloud native architectures provide the perfect solution.

However, certain characteristics of the data sources being used can make the implementation of these cloud native architectures difficult in practice. Notably, challenges can arise if multiple interdependent data streams are used to produce a single set of outputs. Furthermore, even if multiple candidates for implementation are developed, the performance of each one for the task at hand, or even the relationship of their performance is unclear.

In my work, I present the aforementioned topic on a specific implementational problem, where I process a public transit data source that conforms to the General Transit Feed Specification (GTFS). Due to the format of the GTFS standard, multiple separate, yet interdependent data streams need to be handled simultaneously to produce a set of output records together. The task is a typical example where we have to use a data source that is in a predefined standard, which we would like to convert to a format we define.

In my paper, I explore multiple different architectural and implementational combinations to handle processing a GTFS standard conforming data source, and compare them based on their performance. Namely, I compare a microservice, and a Function-as-a-Service (FaaS) solution. I use a testing environment that I defined to conduct performance analysis on the aforementioned solutions. In this environment I compare the most important metric for the end user, which is the total execution time (total runtime), at different levels of available resources (CPU, memory). Apart from the measurable outcome of my tests, I present other advantages and disadvantages of the different approaches.

Chapter 1

Introduction

In recent years, the focus of application development seems to be shifting towards cloud computing. While the idea of centralizing resources, in this case processing power, and sharing it with others is not new, its scale definitely is. With the advent of advanced networking technologies, each data center can be accessed from essentially anywhere, making sharing them trivial. Albeit its relative novelty, even cloud computing is already becoming more democratic, for an increasing number of organizations demand to have their own private cloud infrastructure, further confirming that this is the next innovational trend, currently in the phase of being adopted by more and more people, and continuously being improved upon.

Although cloud computing has its potentials, it also has its limitations, especially if it is not used wisely. In general, information technology requires two separate aspects of it to work together, namely hardware and software engineering. Neither can exist, or deliver adequate solutions without the other. Therefore, no matter how much of an improvement the cloud may be on the hardware side, should it not be used wisely from the software side, its benefits diminish.

Moreover, the handling of large data sets can be challenging due to their scale, turning even small adjustment applied to every data record a monumental task. Additional challenges may arise, should the data to be processed is derived from not just one, but multiple sources, which are dependent on each other in some way, like to formulate the result records together.

Multiple cloud native architectural solutions exist for the latter data processing task, such as microservice based, or Function-as-a-Service based approaches. However, determining the performance of each architecture is difficult, if not impossible in practice without rigorous testing. Furthermore, although such testing can lead to practical results, it requires the implementation of both architectures, which is practically not feasible for most projects to undertake, and undermine the general usefulness of the results, as they will be heavily dependent on the underlying software implementation.

A great example of having to process multiple data streams is the General Transit Feed Specification (GTFS) (Section 3.6), which specifies multiple data sources for the description of a single type of information, which are public transit schedules. In the case of GTFS, the problem could be solved by simply uploading the normalized data structure to a relational database. However, that would result in the need for JOIN operations for nearly every query, as the data schema is normalized, which in turn would increase the response time of each query. On the flip side, by spending some processing time on denormalizing the data records according to the most frequent queries made to the database

before uploading them there, and storing the denormalized data, response times could be decreased, as JOIN operations would no longer be necessary. The reason this is a sufficient approach, is that GTFS data describes static schedule data, which only changes at most weekly, but usually monthly, or a few times a year. Therefore, due to the infrequent updating of the data stored, spending time denormalizing the data records and storing them that way does not seem to have the traditional downsides of storing denormalized data, like inconsistencies after an update operation.

In our work, we aim to address the uncertainty around the performance of the different kinds of architectures on data processing tasks. Specifically, we demonstrate the performance of multiple architectural solutions, by implementing them on the task of GTFS data processing, and measuring their response times.

The rest of this paper is organized as follows. In Chapter 2, we present the academic works published before our research, which are related to it, and which we used as a basis for our own work. The theoretical background required to understand the rest of the paper is presented Chapter 3. The fundamental concepts are explained, with references to more detailed resources for further reading. Chapter 4 is dedicated to explaining the design decisions we took during implementation and the reasoning behind them. The architecture of the different levels of the implementation are also detailed. Chapter 5 is devoted to the methodology of our experiments, the way we conducted tests, and the results of the tests. At the end of the paper in Chapter 6 we draw an overall conclusion of our work, and list a few possible future directions of research deriving from this article.

Chapter 2

Related work

The performance analysis of cloud computing solutions, and in particular containerized microservices, on data processing is a moderately studied topic, with correspondingly moderate understanding of the effects of the different parameters on the end performance of the system at hand.

As early as 2011, the authors of article [17] investigate ways of optimizing data processing workflows. They look at optimizing the processing workflow for the cost of the resources used, for the time it takes to process the data, and for a compromise between the two latter aspects.

More recently an overarching analysis of the different dedicated data processing frameworks was published [16]. Dedicated data processing frameworks offer a completely different approach compared to writing a custom data processing framework for the specific task at hand using containers [7] or functions [9]. The article presents the advantages and disadvantages of each framework, and consequently for what use cases they are best suited for.

Guaranteeing performance or response time in cloud computing is a very difficult task, however the authors of article [22] demonstrate a possible solution to this problem. They investigate which aspects of a microservice architecture determine the end response time perceived by the user, the most typical constraint formulated in Service Level Agreements (SLAs). The authors also present a prototype implementation of a possible algorithm for dynamically scaling the resources of the services running, for the achievement of the predetermined SLA, and conduct performance tests on their implementation.

In our work, we aim to compare containerized and Function-as-a-Service environments, like the authors of [16] did for dedicated stream processing tools. Additionally, we use our own software implementation, and conduct performance tests on it, similarly to the authors of [22]. Contrary to the latter article though, we do not try defining a specific algorithm for improving the performance of the two systems analysed.

Chapter 3

Theoretical and technical background

3.1 Containerization

Containerization [7] is a software delivery and running architecture, which allows for the standardized handling of software products.

Writing programs which can be run on many hardware platforms is a difficult task, mainly due to two key issues: dependencies and runtime environments. The former means that even if our software runs as expected on one platform, it is not guaranteed that all of the other programs it depends upon do too, should they be available on that platform with the same version requirements in the first place. The latter problem arises from the fact that most programming languages are interpretable to humans, but not to machines, thus requiring compilers to “translate” between the two worlds. The problem is that "translating" to every possible hardware platform every version of our program is practically infeasible, and consequently rarely ever done.

Some programming languages tried solving this problem [31, 23] by inserting an abstraction layer between the programming language’s code and the machine readable code, introducing a cross-platform runtime environment between the two. While this does solve the latter issue to some extent, the former is still there, but now exaggerated due to the fact that the runtime environment is a brand new dependency that needs to be available on the host machine.

Historically, the evident solution was creating virtual machines [15], as they could be recreated on demand, packaged the software and all of its dependencies, and were guaranteed to run on the platforms they were designed for, as they included the operating system in them. The grave downside to this approach however, is the greatly increased size of the software package, which is often in the GBs, if not tens of GBs range.

Deriving from the ideas and benefits of using virtual machines, containerization [7] was developed as an alternative approach. Just like virtual machines, containers have every dependency needed to run the containerized software in them. Though virtual machines contain the entire operating system as well, containers do not, leading to a much reduced storage volume, in the hundreds of MBs or sometimes few GBs range. Furthermore, by not internalizing the entire operating system, containers have a much faster boot time, and greatly reduced processing resource usage.

The key difference of containerization from using a programming language's virtual machine, is that while the latter only works for a single or a few programming languages, the former is completely programming language agnostic.

The pioneer of containerization technology was the Docker Engine [4], essentially becoming the industry standard. Docker provides all of the necessary development and runtime management tools needed for successfully delivering containerized software, in a multitude of use cases. As the Docker Engine is an open source project [5], it can easily be built upon and integrated into new software solutions.

Over recent years, most container runtime platforms adopted the format of the Open Container Initiative (OCI) [42] for the description and running of containers. Due to this standardization of the technology, competition and alternative runtimes can be introduced, leading to further advancements in the field.

In our work, we use containerization extensively, using Docker for building and running the containers we create.

3.2 Container orchestration platform

3.2.1 Principal idea

A container orchestration platform [6] is a software environment, that allows for the handling of a large number of containerized software packages at the same time.

With the widespread adoption of containerization came the need for the standardized handling of containers. Albeit containerization was created as a standardization of software packages, it did not solve every problem. While the starting, running, and stopping of containers was standardized from a technical perspective (for computers), it was not standardized enough from the point of view of the operator (for humans). Even though the latter tasks were relatively simple to execute for a limited number of containers, the original solutions were not scalable enough. Taking into account everyday networking tasks, the providing of dynamic environment variables on the start of a container, or the scaling up and down of the number of containers, to cite a few examples, the handling of containers indeed needed a better solution.

Container orchestration platforms aim to solve the above problems, and many more, by standardizing, and automating the interaction with containers. Platforms like these allow for the handling of not just a few, but hundreds of containers at the same time, and can automate a variety of tasks, simple and complex alike.

The existence, and widespread use of orchestration platforms is very likely due to their use in cloud computing. Providing services to not just a few, but hundreds, thousands, or even millions of users at the same time is a monumental task, even if all of them are using the same basic software service. Even if one could somehow start that many software instances and let their users use them, without the use of container orchestration platforms, the handling of faulty containers, or the scaling of the number of instances would be a difficult task, that would need to be handled apart from starting all the instances in the first place.

Orchestration platforms help alleviate these challenges, by abstracting away and automating the basic issues. Such solutions allow the handling of many container instances at the same time, with minimal effort, and with the right configuration the automation of most tasks.

Platforms build on the standardization of container interfaces, and extend them by allowing the declarative description of deployments and tasks. The advantage of declaratively describing workflows, instead of doing so imperatively, is that it leaves room for optimization for the execution engine. Indeed, if every step is described imperatively, the software executing those steps cannot improve much on them. This is not a problem, should the programming engineer be able to comprehend the system being programmed. However, in the case of cloud computing, keeping in mind every aspect of the system at hand is usually a much too complex task. Therefore even though optimization engines are not perfect for every possible deployment scenario, they are still better than writing individual imperative software for each of those scenarios.

It needs to be said, that a container orchestration platform is still only a software environment, not a service. The crucial difference is that while the latter is available on demand from various providers through the internet, the former requires infrastructure to run on. Admittedly, the former can be run by a provider as a service, but strictly speaking the technology itself is only the software necessary to run such services.

3.2.2 Kubernetes

Kubernetes [32] is an open source container orchestration platform, with much of the same benefits as described above, and many more.

Kubernetes describes the environment comprising of itself and the software container deployed using it as a Cluster [33]. A Cluster needs computers to run on, which are called Nodes [36], and the containers being run are named Pods [38]. Kubernetes hides much of the complex logic behind operating a Cluster, allowing for one to focus on the logic and components strictly necessary for one's application.

Kubernetes provides an abstraction layer above the container level, allowing for easier management of Pods. The basic component of this layer is called a Deployment [34], although there are more fine grained options [41] as well. A Deployment describes an application, which uses a type of container (a container image) to run, and the instances of this type of container. It can have any number of Pods running in it, or even none at all. This abstraction layer allows for the standardized description and configuration of a software application, and assumes that it does not matter which container serves the requests to said application. In consequence, on their own, Deployments cannot have states, as it is not deterministic which container instance is going to handle a specific incoming query, and should a Pod restart, all of its stored information is lost.

Although Deployments are a great tool in managing clusters of containers, they are not enough on their own. Firstly, Deployments cannot manage stateful applications by themselves. Several solutions exist to this problem, for example by storing data for every Pod in the Deployment in an external storage, such as a distributed cache (Section 3.5) or a PersistentVolume [37]. Kubernetes' own solution to storing state or other persistent information, is a StatefulSet [40], where each Pod in it has its own dedicated PersistentVolume, and should a Pod fail, the replacement Pod is connected to the same PersistentVolume.

Secondly, while Deployments can run independently, and initiate IP requests, they cannot be the target of such requests. Kubernetes provides the Service [39] resource type for this use case. A Service allows a Deployment to receive requests from the IP network, and describes which domain names, IP addresses, and port numbers it can be reached on. Kubernetes has a complete IP network model implemented, allowing every Pod in the cluster to communicate with each other using it, and requests to be received from outside

the Cluster. Though most of the complexity of managing an IP network is configured automatically, various options can be specified manually. Kubernetes also provides built in load balancing capabilities, ensuring that Pods running in a Deployment, behind a Service, process roughly the same number of incoming requests.

We used Kubernetes as the backbone of our research. We chose it because it is widely used, supported with libraries and third party components, and open source.

3.2.3 Horizontal Pod Autoscaling (HPA)

Kubernetes Deployments allow for the standardized handling of containers which run the same software. Since Deployments assume it does not matter which Pod serves the incoming request, it is natural that more Pods in the Deployment can handle more requests.

This is called horizontal scaling [24], for an increasing number of the same kind of containers are deployed in parallel (i.e. horizontally) to handle the increasing number of requests. In contrast, vertical scaling describes the practice of providing more computational resources, such as processing power or memory capacity, to the same container to increase its capacity of handling incoming requests. In containerized cloud environments horizontal scaling is favored to vertical scaling, as while the latter is limited, for the processing capacity of a single CPU is limited, the former is theoretically limitless, for in theory any number of containers can be running at the same time, as long as a matching number of computers are available to run said containers.

Horizontal Pod Autoscaling (HPA) [35], as its name suggests, is Kubernetes' built in implementation of the principle of horizontal scaling. While the number of running Pods can be configured manually at any given time, this feature automates this task. The great advantage of this automatic implementation, is that the Cluster can automatically adapt its Deployments to the incoming workload, without the need for external intervention. HPA dynamically increases or decreases the number of running Pods based on the Pods' CPU and memory usage. Once the desired limit for the consumed resources of a single Pod, and the optimal percentage of said limit used are configured, the algorithm launches or stops Pods based on whether the other Pods are above or below the defined optimal percentage.

A known limitation of HPA is that it can only scale down to a single running Pod. This behaviour is due to the fact that a single instance of the container always needs to be able to instantly respond to incoming requests. For even though the launching of a container is far quicker than the launching of a virtual machine, its boot time is usually still not negligible compared to typical IP request response times.

3.3 Function-as-a-Service computing model (FaaS)

3.3.1 Principal idea

Function-as-a-Service computing model (FaaS) [9] is a cloud native architectural pattern, and the services implementing it, aiming at simplifying the development and deployment of cloud software.

Most advancements of the cloud introduce a new layer of abstraction above the existing options, thereby further simplifying the development and deployment of applications to

the cloud. FaaS is no exception in this regard, as it essentially provides a more convenient platform for developers to use when programming for the cloud.

More often than not, deployments in the cloud are already stateless, and therefore the instances of an application running at any given time are interchangeable. Moreover, combined with this interchangeability of the instances, their scalability is just as important, not to waste valuable resources. Indeed, should the infrastructure that one's cloud software is running on be a third party cloud service provider, every resource waste can be directly quantified as economic loss for one, derived from the price of the resource wasted at said provider.

One of the significant challenges of scaling cloud applications is the overhead of starting and of stopping an instance of the application. This overhead is quantifiable as the time it takes for the instance to start before, or to stop after it actually performs the task it was designed to do.

The obvious answer to reducing this overhead is to reduce the complexity of each instance of one's software. FaaS aims at doing just that, by reducing the amount of programming needed to deploy an application to just the code necessary for the application's primary purpose. Consequently, theoretically, no additional programming or configuration is required to run an application. In practice though it is greatly reduced, minimal amounts of it, like the configuration of the different application components' relation, is still required.

Behind the scenes, making FaaS *function* can be done in multiple ways. The most obvious method is to leverage existing containerized environments, and use their scaling mechanisms, essentially providing a wrapper software around existing solutions. This approach can be extended by providing a more fine grained control of the scaling, by making each container run many instances of the same application packaged as a function. Furthermore, provided the deployment and running of a function be standardized enough in the particular FaaS implementation, the same kind of containers can be deployed en masse, all of them capable of running any function written for that FaaS implementation. The benefit of the last approach, is that by standardizing the containers running the functions, many function instances can be aggregated on the same container instance, reducing the number and amplitude of scaling required on the container level.

An additional upside of using FaaS for a developer, is that by trying to simplify the deployments, a standardized, and usually automatic, way of declaring and including third party libraries and dependencies in one's code is provided. This feature makes the functions run by the provider more uniform, allowing for further optimizations while running them, and relieves the developer of yet another implementational overhead standing in the way of solving the actual problem at hand.

3.3.2 Knative Function-as-a-Service

Knative Function-as-a-Service (Knative FaaS) [19] is a practical implementation of the above listed FaaS principles.

Knative FaaS allows for the rapid development and deployment of simple function based applications. It automatically generates a starting programming project for one to work on, providing a basic framework for handling incoming HTTP requests. It supports several programming languages, though since its implementation is entirely open source and well documented, it can be extended to support any kind of languages.

It also helps tremendously in the deployment of one's applications. Knative handles the compiling of the application into a container, and the configuring of the container's pa-

rameters in the cloud. Should a change occur in the application's code, the redeployment of it is just as fast and as effortless, as it was the first time.

Knative FaaS builds upon Kubernetes, leveraging the latter's capabilities to improve upon them. Due to this dependency, it uses regular Kubernetes Pods behind the scenes [20], only being able to scale on the container level. Knative tries improving on this inherent limitation by creating starting programming projects that can be compiled into pure function running containers for that application. Consequently, a single container can run multiple functions in parallel.

Knative FaaS is part of the Knative library for Kubernetes [18], which is a software solution designed at facilitating the handling and communication between containerized applications in Kubernetes. Knative also provides its own autoscaling engine, the Knative Pod Autoscaler (KPA) [21]. It uses this engine in its FaaS implementation, allowing for dynamic scaling of applications based on either the incoming requests per second (RPS), or the number of parallel running functions in a single container (concurrency). Combined with the fact that Kubernetes HPA can be used as well, KPA provides great alternative metrics to scale by, while not sacrificing traditional resource consumption based approaches, should the particular application demand it, as this setting can be defined on a per application basis.

As a consequence of building upon Kubernetes, Knative FaaS itself is just a software environment, not a readily available service. In order to operate it, one needs to have an existing Kubernetes cluster, which in turn requires infrastructure to run on as well.

We chose to use it as a FaaS runtime, because it is easy to integrate with Kubernetes, which we were already using, and that because Knative is open sourced.

3.3.3 Comparison to containerization

From the features and characteristics described above, it is clear that FaaS is similar to regular containerized deployments. This similarity is no surprise, considering some FaaS implementations are built upon containerized environments, like Knative FaaS.

Containerized applications have the advantage in development and configuration flexibility. These days, there is practically no limit to what applications can be run inside a software container, and containerized environments allow for meticulous control over what goes into the container, and how the container behaves with its environment.

In contrast, FaaS applications have somewhat limited options in what can be deployed as a function. The reason for this is that FaaS environments try to manage as many aspects of the deployment procedure as possible, and do so automatically, for additional room for optimization when running the functions. Undoubtedly, having most aspects of deployment preconfigured is convenient for the developer, although it does become a burden should a very niche deployment case arise, which requires ample custom configuration.

On the other hand, by reducing the room for flexibility in the deployment procedure, FaaS environment allow for much more flexibility when running the applications, compared to pure containerized deployments. Indeed, while the latter can only be scaled at the container level, the former allow for a more fine grained control, as capacity can be increased or decreased on a per function basis. Moreover, in the case of a FaaS environment which enables functions to be deployed on homogeneous containers, resource allocation can be even more precise.

3.4 Microservice architecture

Microservice architecture [10] is an architectural pattern in cloud computing, with the goal of building complex software structures from simple, small, and decoupled components, that can be maintained and developed independently.

A significant challenge when developing software for a complex task, is that the solution is often very complex as well. The more complex a component is, the more likely it is that an error will occur during development, which is only made worse if changing said component may break several others as well, meaning that they are strongly coupled. The obvious answer to this problem is to divide the complex software into smaller, more manageable components, and decouple them as much as possible. Dividing helps with the complexity of each component, while decoupling enables said component to be handled independently of the others.

Traditional software architectures already use the principles of division and decoupling, however the decoupling of the components can only be so strong as long as all of the components are running in the same memory space. Microservice architecture mandates strong decoupling, by keeping the components in separate memory spaces, and only letting them communicate through the IP network, often using REST APIs [14]. Enforcing separation of the components in this way ensures that proper decoupling does take place, resulting in *micro* sized *services* (hence the name of the architecture). Furthermore, while it is possible for an inefficient division of the software to go unnoticed when every component is running in the same memory space, the inefficiencies are exaggerated when communicating over the IP network, therefore motivating one to develop a better solution to the problem. The reason that it is beneficial that these inefficiencies are made obvious, is because while they may be efficient enough when run in a single memory space, they still cause improperly decoupled components to exist, which negatively impacts the further development of those components.

Even though microservices warrant the use of IP network communication between them, this does not necessitate the use of containerized technologies, or any kind of particular technology for that matter. On the other hand, containerized or FaaS deployments can leverage the principles of microservices well, as they inherently provide the division of larger software into more manageable components. Therefore, microservices are not a technical specifications, but rather an overarching concept for the entire software service that one wants to provide.

The benefit of implementing the principles of this architecture are numerous. Firstly, they help software development, by being able to work on smaller, easier to understand components of it at a time, without having to worry about the consequences a change is going to have on the other components, as long as the REST API does not change of the service. Secondly, as the services can be developed quasi independently of each other, they can all use differing programming languages and technologies. Indeed, considering that the services only communicate using REST APIs, the only requirement is that the programming language and technologies used support that API. Thirdly, the independence of the components means that should a particular part of the application receive more requests than the rest, for example the authenticating component, it can be scaled up without having to scale the rest of the application with it, saving valuable resources.

In our work, we separate the components of our application using the principles of microservices, although we do not strictly use REST APIs for communicating between the components.

3.5 Distributed cache

A distributed cache [8] is a software solution facilitating the storage of simple state variables for the running deployments in a cloud environment.

A common problem for initially purely stateless applications is the eventual need to store some state information, for example session IDs. Traditional solutions to this problem, like storing it in memory, do not work, as each request may be serviced by another container which uses a separate memory context, therefore either losing the data or even worse, using outdated data. On the other hand, using entire database management systems (DBMS) for such a simple task would waste too many resources both in terms of CPU and memory usage to run the database, and in terms of unduly elongating the overall response time due to the overhead of searching in a complex database environment, which is persisted, further increasing the response time.

Combining the benefits of both worlds, distributed cache systems allow for the storage of small amounts of data, key value pairs more precisely, which are stored in memory, but in separate and dedicated containers for this purpose. Consequently, the data is stored securely without having to worry about which container serves a particular request from a deployment, and response times are still fast, as only simple keys need to be searched with simple values to return, and everything is stored in memory.

Apart from the basic usage of storing small amounts of data, using distributed caches can be used as the single source of truth. In distributed cloud environments, even if different parts of a cluster are consistent on their own, they may not be in sync even though they should be. Instead of using proprietary software for synchronization, distributed caches solve this problem with robust and fault tolerant algorithms tested at up to thousands of requests per second.

Various implementations of distributed cache systems exist, but the most widely used are the open source Redis [12] and etcd [8] tools. Redis allows for the storage of a greater variety of data structures than etcd, and stores its data in memory, providing faster response times as well. In contrast, etcd only allows for the storing of simple string key value pairs, and stores its data in persistent storage. As a result, while etcd may have to slightly increase its response time, in turn it can achieve much better fault tolerance.

Kubernetes' control plane internally uses the etcd distributed cache. This is a crucial component of the orchestration platform, as it allows for the configuring software to know the state of the cluster and the Pods in it at any given time, and make decisions based on that information.

We use etcd in our cluster as well, for the tracking of the progress of the processing task accross multiple Deployments. The reason we chose etcd was its fault tolerance, and its widespread adoption in the industry, most notably by Kubernetes.

3.6 General Transit Feed Specification (GTFS)

The General Transit Feed Specification (GTFS) [25] is an open standard for digitally describing public transit schedules.

Originally named Google Transit Feed Specification, as it was developed by Google and was only later open sourced and renamed, GTFS helps transit agencies to publish their schedules online in a standardized format. This helps agencies by sparing them the effort

of coming up with a publishing format, and also helps promote their schedules, as any mapping provider can use the standardized data the agencies publish.

GTFS is a global standard, and therefore is designed to handle any and all kinds of transit agencies possible. Since each one is different, the standard contains numerous possible attributes, only a subset of which are required and therefore guaranteed to have a value. Consequently, parsing GTFS data requires robust algorithms, which can handle many missing fields in the data set.

While the main purpose of GTFS data sets is the publication of public transit schedules, other related information can be described with it as well, for example the fares required to use the described transit routes. Additionally, GTFS can contain GPS coordinates for the routes described in the schedule, enabling mapping applications to display them. Moreover, GTFS Realtime [26] allows for the real time publication of vehicle locations and delays compared to the original schedule, giving riders further information on their planned trip.

The standard specifies a data structure which is normalized, meaning there is no redundancy in the storing of data. Each agency is required to publish a ZIP file, containing TXT files with CSV formatted data in them. Each file contains data for a specific entity in the standard, where the rows of the file are the records conforming to that entity. The data files can also be regarded as data streams, when processing them line by line, each stream providing a flow of records to be processed.

In our project, we use GTFS data as the basis of the processing task that we implemented.

3.7 Document based database

Document based, or NoSQL databases [11] are database implementations which allow for more flexibility in the storage of records compared to relational databases, especially regarding the handling of optional record fields.

Document based databases developed as an alternative for relational databases [13]. While the latter are great for normalized, predetermined database schemas, they handle the storage of optional fields poorly, as the fact that an attribute does not hold data needs to be stored as a NULL value. Due to having to store NULL values explicitly, should the schema of a table change, for example be extended by another attribute, that attribute's value needs to be given for every already existing record.

In contrast, document based storage does not define schemas in advance, but rather allows any kinds of records to be stored in a collection, and let the applications using the records handle if a field does not have a value. This approach is better for today's software development patterns, as most applications are continuously developed with incremental changes, whereas always updating the corresponding database can be a monumental, and mostly unnecessary task. Indeed, in a the case of already handling optional fields, from the client's perspective, receiving the field with a NULL value or not receiving the field at all are practically equivalent scenarios.

Another paradigm shift in document based storage compared to relational based storage is the abandoning of the normalization of the data stored. While normalization is great for keeping a database consistent, and reducing redundancy, in some applications these aspects are much less important than the overall response time to a database query. Indeed, if all the data needed for a particular query are stored in the same record, a JOIN operation's execution can be saved compared to storing the data normalized.

One of the industry leading document based databases is MongoDB [27], implementing all of the above described characteristics. A significant advantage of using MongoDB is that it has a freely available community version with most of the core features of the proprietary product, and that application libraries, or database drivers, are available for a great number of programming languages, allowing for flexibility in the choice of programming language to access the database.

We chose document based storage to facilitate faster response times, and we chose MongoDB in our project due to its support of many programming languages.

Chapter 4

System architecture

4.1 Design considerations

As with any research project, we had many ideas along the way that are still open for debate. In this chapter, we would like to present a few of them, and explain our decisions regarding them. Additionally, we are also going to present some of the challenges that we had to consider when implementing this project.

4.1.1 At the edge of the memory context

Originally, we had the idea to analyze the difference between a monolithic, a microservice, and a Function-as-a-Service (FaaS) based implementation of the same data processing problem. The reason we wanted to investigate these particular architecture patterns, was that at a high enough level of abstraction, they essentially work the exact same way with one key difference: where the edges of the memory contexts are, and where IP network communication is necessary.

Consider that a particular data processing task has a software written for solving it using a monolithic architecture, meaning every component of the application runs and communicates in the same memory space. If the interfaces of the internal components of said application are well designed, and provide a sufficient level of decoupling, then converting the monolithic architecture to a microservice based one is rather simple. One has to move a component's original implementation into a separate service, and fill the gap at the original location of the component with a mock implementation (Section 4.3.3). The mock implementation's only purpose is to fulfill the original interface contract, and forward any internal function calls over the IP network to the newly separated original implementation. This procedure has to be repeated for every component to achieve a microservice based implementation instead of the monolithic one.

The above procedure can also be adapted to move from a microservice based approach to a Function-as-a-Service based one, executing each step described above, but this time for the functions of the implementation instead of its components.

The two main challenges that would need to be overcome using this approach are how to write monolithic software with sufficient decoupling, and how to deal with the characteristic of FaaS environments not inherently supporting returning values to the caller like in memory function calls do. The former may be difficult, but not impossible, as most

standard, object oriented programming design patterns aim at doing just that, creating components with a low level of coupling between them.

The latter problem arises from the fact that while in memory function calls can use the stack to call one another, then return and keep processing in the caller function, FaaS functions do not have that luxury. While simple, single level, calls can be made asynchronously in a function, nesting these calls can lead to significantly increased overall response times, potentially leading to timing out at the original caller function. Therefore, most FaaS environments prefer a linear calling structure, meaning the processing task moves from one function to the next, without returning to the starting function. Adapting this linear calling structure for in memory calls however, would lead to very deep, almost infinitely long stacks, with the potential of a stack overflow error. Even if the returning problem were to be solved, using multiple threads for example, the basic logic of the monolithic software would need to be rewritten, as this is a very rare pattern to use when using a single memory context for execution. Moreover, using such an uncommon approach in the monolithic implementation of the program would annul any results of this decomposition, as the monolithic version would not be at all authentic.

To summarize, while it does seem like an interesting topic to investigate, it still requires more research to properly evaluate its value.

4.1.2 The right metric

Deciding upon using the final response time as our key performance indicator was not evident at the start of our research either. This approach had to be selected from multiple other possible metrics to measure, some of which we list here.

The overall CPU or memory usage can be measured as well of the running containers, alongside the number of containers and/or functions invoked. The reason these are particularly interesting metrics, is because most often these are the basic components by which our application running in the cloud is billed. Consequently, perhaps combining the results with the running time of a processing task, a price estimate could be given using a linear combination of all of these parameters. In our case though, we are not using public cloud providers to run our programs, meaning that firstly we can only estimate, and not verify, the final price of a single running of our tasks, and secondly that we do not have access to infinite amounts of available resources to provide our containers with. Due to these reasons, pursuing these metrics did not seem relevant in our case.

Other than the resource usage and performance of the individual containers, the IP network between the containers can cause a bottleneck too. To find out if this is the case, most likely because of the components and APIs used emit too many messages, one could measure the overall volume of traffic going through the network. Additionally, the database engine can be the bottleneck in a data processing task as well, therefore using a metric describing the throughput of the database, like its input/output operations per second (IOPS) can argued for. While these latter two approaches have potential use cases, they also require further research in order to verify their value.

All in all, while several other metrics have arguments for them to be used, we decided upon using the end-to-end response time of a processing task, for this is the final and only metric the user of an application is going to perceive, and is therefore the most important one, in our opinion.

4.1.3 Dedicated data stream processing tools

Apart from writing custom programs for a data processing task, dedicated tools exist for it too, such as Apache Kafka [1]. Such tools are optimized for the processing and transforming of data streams, therefore have the potential for much faster processing times than the microservice or Function-as-a-Service based approaches. The main downside of using such an approach, and comparing it to writing custom code for the task, is how different the two can be. Considering how much more complex the internal logic of such a tool can be than the custom implementations one can write, it can be argued that the performance difference between the two could simply be down to the efficiency of the custom implementation, and not to the frameworks like microservices or Function-as-a-Service used.

We decided against including a dedicated tool like Apache Kafka because comparing its performance to our custom implementations would be irrelevant.

4.1.4 General Transit Feed Specification (GTFS) considerations

We chose General Transit Feed Specification (GTFS) (Section 3.6) as a data source format to be processed because of our personal interest in public transit, and because we find it has some unique challenges in the way it stores its data.

As we mentioned in the introduction (Chapter 1), the GTFS standard is a very general specification, aimed at any and every kind of transit agency possible, therefore containing many optional fields, which are difficult to handle. As an optimization to our processing algorithm, we only handle the strictly necessary fields of the specification, of which almost all are required to not be empty, and we only handle the smallest set of data streams that are still enough for meaningful output. As an example, we do not handle the possible GPS coordinates of the routes, and we do not handle possible exceptions on specific dates in the schedule, only the regular service hours. These and the other optimizations that we made when processing the data streams, in our opinion, do not contribute enough to the output records that it would be worth the effort of implementing their processing as well, especially not in a time constrained project like this is.

The real challenge of handling GTFS data is its multiple data streams, combined with our aim of denormalizing its data structure. As we are denormalizing the data structure of the standard, multiple data streams include the data for a single type of output stream, the order of the upload of which input streams matters as well. The reason that this order matters, is because naturally, the input stream containing the primary key of the output stream is required to be processed first, to create the records of the output stream, where the other input streams can upload their own data. This is the interdependency of the GTFS data structure referenced in the title of this paper, which dependencies are contained in the order in which our algorithm processes the various data streams, as illustrated in Figure 4.1.

4.1.5 Alternative processing flow topologies

The main difficulty of the processing task is the interdependency of the data structure, as we explained earlier in this paper (Section 4.1.4). There are most likely numerous solutions to this problem, of which we thought of three, and decided upon implementing only one. We are going to explain the other options that we were considering in this chapter, that could form the bases of further research projects.

One of the options that we were considering was quite similar to our final implementation. Both solutions built upon the idea, that by deciphering the exact dependencies that created an output record stream, and those that only provided additional information to that stream, a graph could be defined, with directed edges pointing in the direction of the dependencies (the edges pointing towards the dependent nodes). A topologic order can be defined in this graph, as shown in Section 4.3.1. A simpler solution however, is first creating each output record from the corresponding input stream which contains its primary key. This can be done all in parallel for each input stream, as each output record is only dependent on one input stream for its primary key. Afterwards all of the additional information can be uploaded from all of the input streams in parallel, as all of the output records will have been created at that point. While this is a simpler option, we decided against it because we think that the option we did implement holds more potential and provides better performance, however this does need further research to be confirmed.

A completely different approach is to upload all of the input records as-is to the database all in parallel, and let the database deal with creating proper output records from them. The key component in this solution is using database trigger functions [29], and setting them to trigger updates in the database at each new insertion. This methodology essentially moves all of the processing to the database itself, which is while an interesting concept, we believe is not likely to be an efficient one as well. The reason for our scepticism regarding this solution, is that even when the database does not have any processing to do, only to insert and update records, it turns out to be the main bottleneck in the system. Putting additional processing in the database is probably going to make this bottleneck worse, and hence why we did not implement this solution.

4.1.6 Further parallelization

During our implementation, we tried parallelizing the processing task in as many ways as possible, however we did omit it in one possible scenario, which is the initial parsing of the raw data files. we could have parallelized this step as well, since the records in a data source file are independent of each other. On the other hand, by keeping this component as a single instance, the tracking of the progress of the processing of a particular input stream was much simpler, as every upload process responded with its status to this single component. This simplification in the development convinced us to accept this minor inefficiency in terms of upload performance.

4.1.7 Choice of programming languages

Most of our development experience is in the Swift programming language [2], and therefore we started implementing this project in it as well. However, since the Knative Function-as-a-Service (Section 3.3.2) environment does not have built in support for Swift, we decided on taking the opportunity to deepen our knowledge in the Node.js [30] programming language, as it did already have built in support.

4.1.8 FaaS and containerization

We decided upon running the row-distributor (Section 4.3.3) component as a container even during our Function-as-a-Service based experiments, which arguably hinders their results. We believe this choice to be well founded however, considering our choice of limiting parallelization in the row-distributor component (Section 4.1.6), and how we

wrote that component in Swift which is not supported by the Knative Function-as-a-Service environment (Section 4.1.7).

4.1.9 Database constraints

During our implementation, we constantly experienced performance issues, where requests sent from the row-distributor (Section 4.3.3) component would time out. After a lot of investigation of the possible cause of the issue, we noticed that scaling up our other custom implemented components did not help the problem, only the scaling up of the database engine. We initially only tried running the database engine as a single instance, but even after vertically scaling that instance up, the timeout issues still persisted.

A probable next step in properly solving the issue would be to try MongoDB's database sharding technology [28], essentially the horizontal scaling of the database. Another, simpler approach however, is to replace the database with a mock implementation (Section 4.3.3), that returns HTTP status code 200 to every request.

The main purpose of our experiments still holds up with this simplification too, as even with the mock implementation all of the network traffic that would take place between the components and the database does take place. Therefore, we saved implementational time while not sacrificing the goal of our research.

4.2 Problem definition

We now define the exact scope of the project, what it tries to achieve.

The program handles a single, predefined GTFS ZIP archive, reads each file line by line in the defined dependency order (Section 4.3.1), and uploads the data according to a newly specified database schema.

The city of Szeged's local public transit schedule [3] is used as a data source. Most data files in the data source have record numbers in the couple hundreds or couple thousands range, but the data file including the planned departure times at each stop for each trip (stop_times.txt) include as many as 138000 records. Evidently, the latter data file required the greatest amount of resources to be processed, and took the longest to do so on average. The database itself is a mocked implementation that only simulates the saving of the data.

Almost all of the containers, with the exception of the row-distributor (Section 4.3.3) component (Section 4.1.7), are implemented using the Knative Function-as-a-Service (Section 3.3.2) framework, allowing each container to run in both a purely containerized environment, and a FaaS based environment as well.

4.3 Architecture

In this Section we present our implementation, going into detail on its structure at different levels of abstraction.

4.3.1 Global design

Globally, the main determining factor of how the data can be processed, is the data itself. In particular, the interdependency between the different data streams of the input impacts greatly the order of their processing.

The dependencies between the data streams can be divided into two categories. Either an input stream creates the output records, or it just provides additional information for them. In the former case, the input stream contains the primary key of the output records, therefore it is necessary for the creation of those records. In the latter case however, the input stream only contains additional information, with which it can update already existing output records.

Consequently, during processing, each stream of output records needs to be created before it can receive updates from additional information providing input streams. Drawing the input and output data streams as nodes, and the dependencies as directed edges, where each edge is pointing towards the dependent node, a directed graph can be defined. This graph can be seen in Figure 4.1, where the dashed lines represent the additional information providing dependencies, the thin solid lines the creating dependencies, the blue boxes the output streams, and every other colored box the input streams.

Regarding the above defined directed graph, it clearly does not contain any cycles, therefore it is a Directed Acyclic Graph (DAG). We know that DAGs have a topological order, and using that topological order an order for the data processing can be defined. Since only the input data streams need to be processed, we only need to define their order for the processing task. The only requirement is that no additional information is uploaded to any output record streams before that stream was created. This means, that in the order we define, iterating over each node, none of them can have a creating dependency towards an output stream that has an additional information dependency towards a node that preceded the node at hand.

The thick blue edges define an order of the input streams that satisfies the criteria defined above, and therefore is the precedence order that we use in our implementation.

The colored input nodes all represent an input file in the GTFS ZIP data source. They are aptly named stop_times.txt for RawStopTime, stops.txt for RawStop, and so on. The second row of blue nodes represent the main output streams, corresponding to the three main entities in the data source. A Route is a transit route with particular stops, a Stop is a transit stop or station, and a Trip is an instance of a Route at a particular time and date. The first row of blue nodes represent embedded information in the three primary entities. Their names are self explanatory, for example RouteTrip represents Trip instances that use that Route, StopDeparture represents departure times for that Stop, etc.

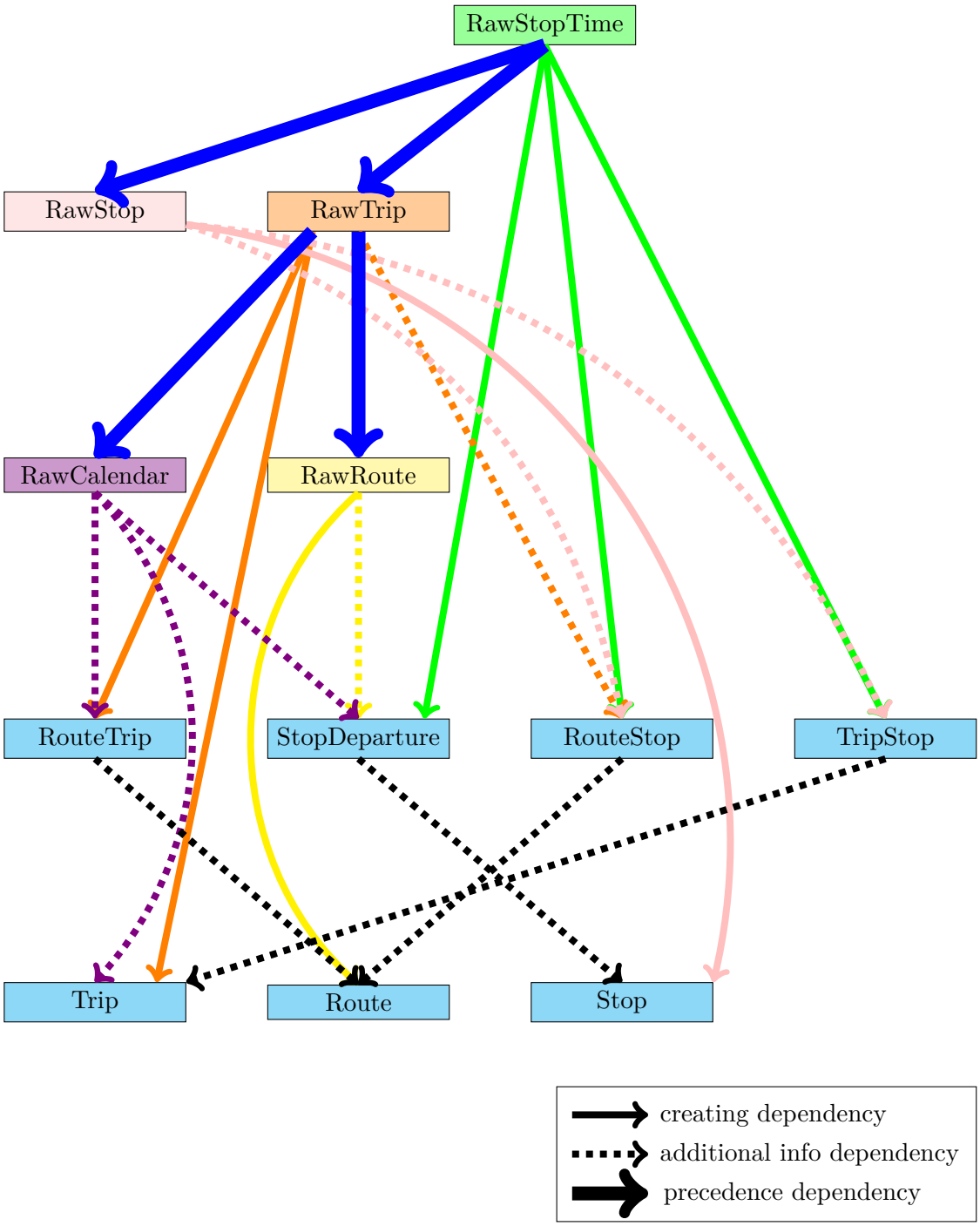

Figure 4.1: Data dependencies between the different GTFS data streams.

4.3.2 Data flow design

The processing of a single input data stream has multiple steps as well, which Figure 4.2 illustrates. The solid lines represent the flow of the data records as they are transformed and then sent to the next component for further processing, whereas the dashed line means the constant progress reporting from the processing components to the distributed cache. The thick black line at the top and bottom of the Figure means input or output communication outside the scope of the processing of the single data stream at hand. The green colored nodes represent custom implemented components that were written in

Node.js, the orange node stands for the custom component implemented in Swift, the blue node represents Kubernetes' Service type as a load balancer for the requests between the first two components, and the pink node represents etcd distributed cache (Section 3.5) as a directly installed component.

Both the final input and output of the processing flow are requests to start the processing of the next data input stream, but do not directly contain said input stream, as that data is already packaged into the row-distributor (Section 4.3.3) component's container. The output is generated once all of the records are processed from this data stream.

The row-uploader (Section 4.3.3) component is shown multiple times in the third row of the Figure 4.2 to show that it can be horizontally scaled up or down.

The progress tracking is done in a push model, meaning the row-distributor (Section 4.3.3) component uploads the progress of the upload after every x records were either uploaded or failed to upload, where x is a predefined constant.

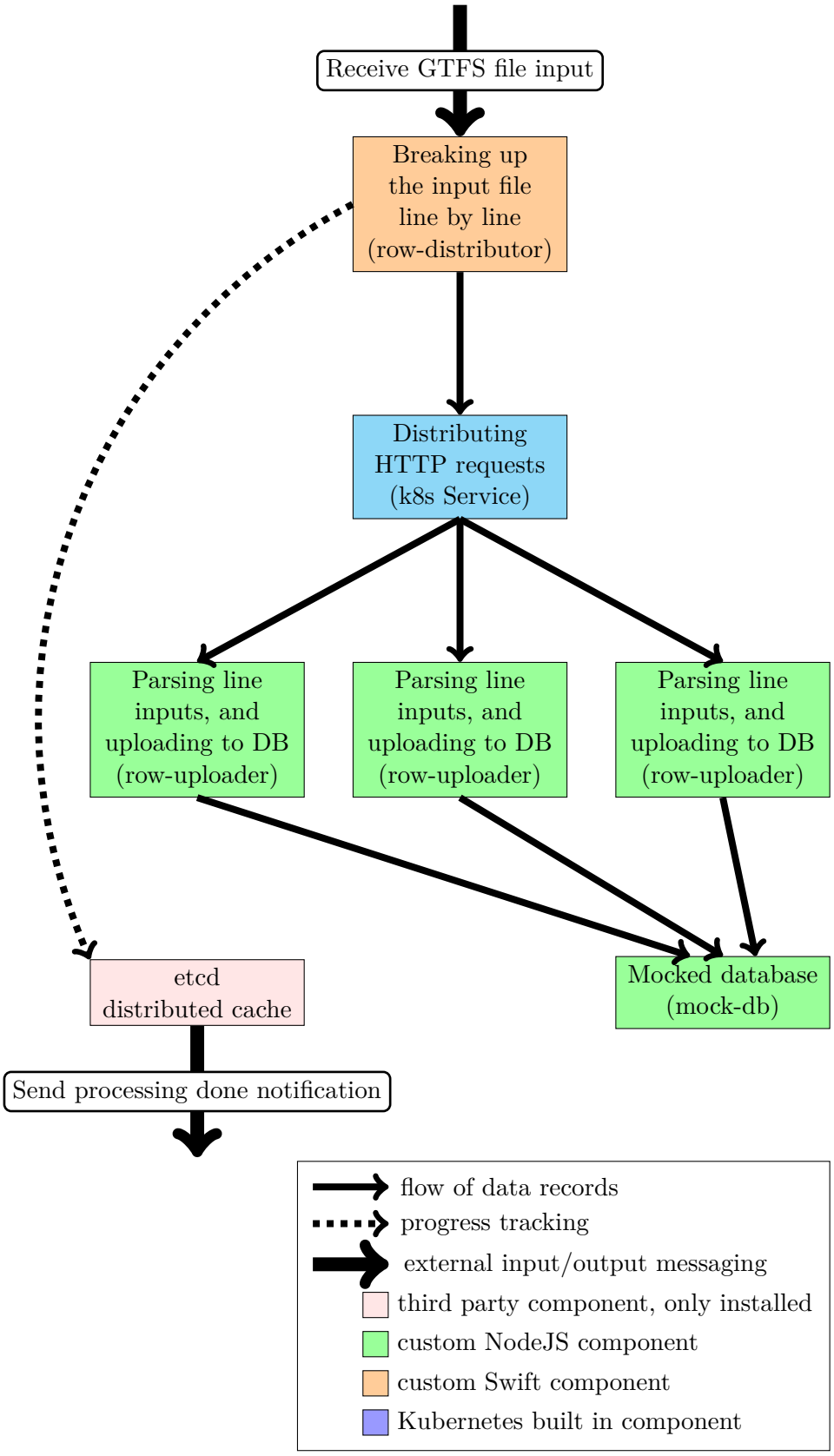

Figure 4.2: Flow of the data records while processing a single GTFS data stream.

4.3.3 Component design

Figure 4.3 shows the flow of records and requests on the component level, and the types of the components involved. The solid black lines represent the flow of the data records as they are transformed and uploaded to the database, or in this case its mocked version (Section 4.3.3). The dashed lines represent the route of the progress tracking requests, which ensure that the progress of the processing is saved to the distributed cache (Section 3.5). The solid red lines on the other hand represent simple HTTP requests without any data, that control the starting of the processing of a new data stream once the previous one is done. The green and orange nodes represent custom implemented components in Node.js using the Knative Function-as-a-Service (Section 3.3.2) template, and in Swift respectively. The pink node represents etcd distributed cache as a directly installed component.

While data uploading packages do contain data, and therefore their size is relatively bigger, progress tracking requests only contain a key-value pair, and the new process starting requests only a few request parameters.

It is important to note that while the Figure 4.3 does not emphasize it, Node.js components implemented using the Knative FaaS (Section 3.3.2) template can be scaled in both microservice and FaaS environments, whereas the Swift component could be but is never scaled (Section 4.1.6) in either environment.

Additionally, even though the only external component currently listed is the etcd cache, the database would be an external component as well, had everything gone as planned during implementation and preliminary testing (Section 4.1.9).

Resource limits and autoscaling options are explained in the Chapter (5).

request-entrypoint This Node.js component serves as the starting point to all data stream processing. It receives simple HTTP requests with only a few parameters, and decides which processing to start next based on those parameters.

This component communicates with the distributed cache to upload the initial starting time and the final ending time of the processing tasks.

etcd-connection In order to access the etcd cache from the row-distributor (Section 4.3.3) component written in Swift, a custom adapter was required, as the standard etcd accessing library was not available in this language. Since it was required anyway, other components use this Node.js one as well as a proxy to the real etcd cache.

It exposes basic etcd operations as HTTP endoints, namely the inserting, getting, and deleting of specific values, the resetting of the cache. Additionally, it also supports watching the values of a specific key's value, and sending an HTTP request to the specified endpoint with the specified parameters once the value of the watched key reaches the given value.

row-distributor The only component implemented in Swift, and that does not support FaaS deployment. Its purpose is to process the data files stored in its container, and create separate HTTP requests from the header and each of the rows of those files.

This component is also responsible for the tracking of the progress of each request it sent out, by systematically waiting for a reply, and tracking the request as an error should it reach the predefined timeout limit. This progress is sent to the etcd-connection (Section 4.3.3) component after every x number of requests, where x is a predefined constant.

The reason for not updating the etcd cache on every request's response, is that from preliminary tests it hinders the performance of the data processing task.

In order to avoid overwhelming of the system, a sliding window approach is used for the sending of requests. It only allows for a configurable number of requests to be in the "unknown" status, meaning they were sent out already, but did not receive a response. Consequently, as replies to the requests sent out come in, the counter of "unknown" requests decreasing, allowing for other requests to be sent, essentially sliding the window along to the next requests in the queue.

row-uploader This Node.js component is responsible for taking the parsed rows of the data streams it receives, and uploading it to the database in the correct format by generating the corresponding database queries.

The uploading procedure is implemented using MongoDB queries, however the component does have a mock driver as well, which only sends basic HTTP requests to the mock-db (Section 4.3.3) component.

mock-db Instead of using a real MongoDB instance, this component provides a mocked version of it, replying with HTTP status code 200 to every request it receives.

The reason for using such a mocked version is related to performance issues experienced during preliminary testing of the system (Section 4.1.9).

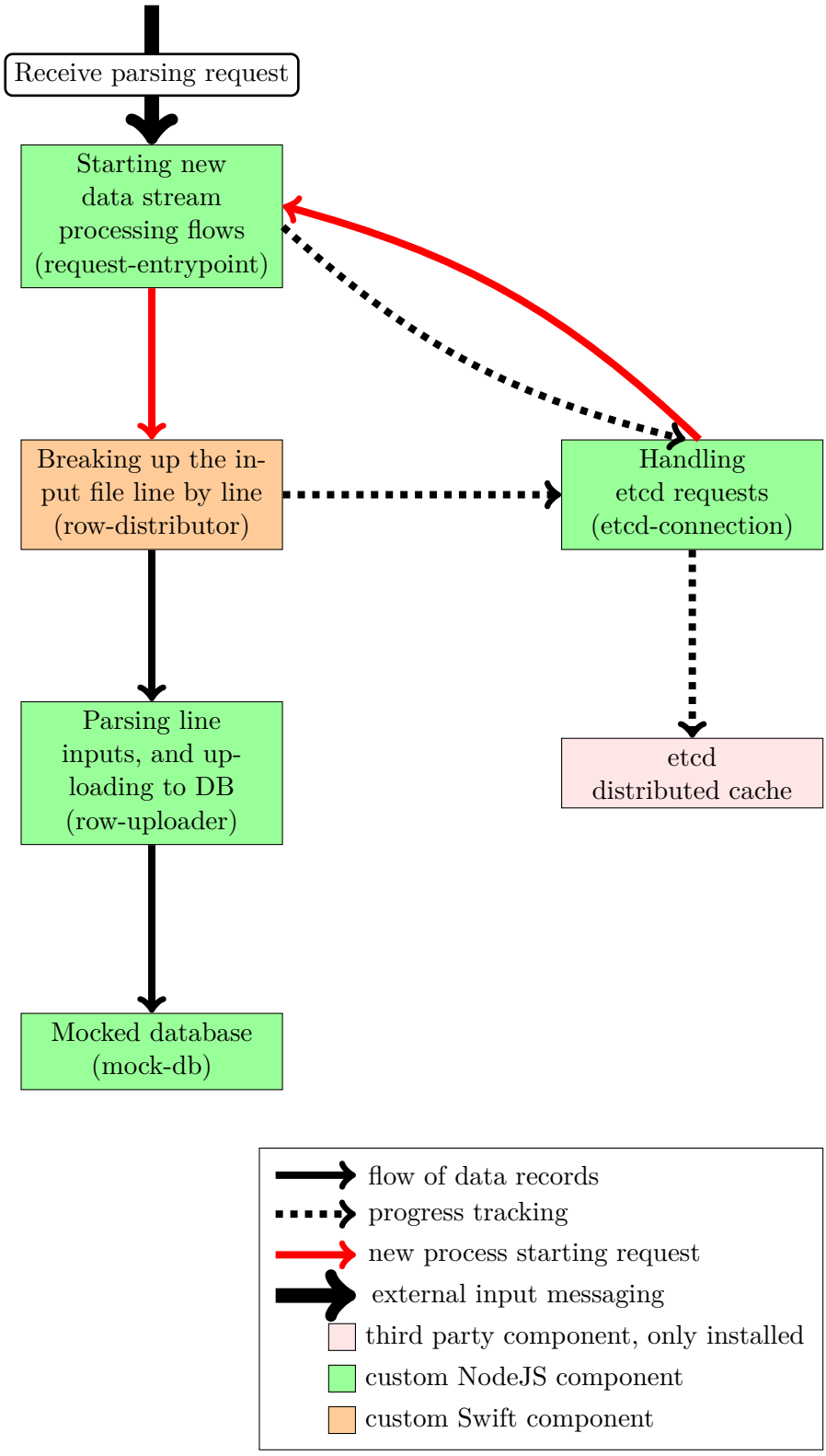

Figure 4.3: Data and control flow between the application's components while processing data

Chapter 5

Results

5.1 Test design

As we have seen in Section 4.1.2, the key performance metric is the end-to-end response time experienced by the user which is the target of our comprehensive analysis. The end-to-end response time means the time between the request is sent out, and the time the last record was processed from the last data stream. Both of these timestamps can be read from the etcd-cache in the cluster once all of the processing is done, as the progress of the processing is uploaded there (Section 4.3.3).

We designed these tests to measure the performance of our implementation using the Knative Function-as-a-Service (FaaS) (Section 3.3.2) development template deployed in a containerized environment. This is possible due to the characteristic of the Knative FaaS environment, where it builds containers from its function templates, and runs the containers as functions. Thanks to this characteristic, we are able to run these containers ourselves, without the help of the FaaS environment.

5.1.1 Testing resource tiers

In order to evaluate the performance our implementation we defined three resource tiers which can be found in Table 5.1, marked by the letters of the alphabet. To differentiate between the tiers, we specified different CPU and memory requests, as well as limits for them. Additionally, we defined two different scaling parameters for the Horizontal Pod Autoscaler (Section 3.2.3), to measure its effectiveness as well. The limits specified here were set for all of the FaaS based components, however not for the purely containerized row-distributor (Section 4.3.3), which was given its own generous resource limits due to its standalone deployment. The latter received a limit of 20 cores of CPU and 20GB of RAM, so that this component does not become the bottleneck.

5.2 Test results

Figure 5.1 shows the results of our experiments side-by-side of each other. The y axis shows the total response time of the test instance, whereas on the x axis, under each bar, its name can be seen that indicates which similarly named resource tier found in Table 5.1 was used for that test instance. The number values above the bars indicate the bar's value on the y axis, the total response time of that instance.

Table 5.1: Different resource tiers for experimentation.

Tier	Resource request		Resource limit		Autoscaler
	CPU (m)	RAM (m)	CPU (m)	RAM (m)	HPA (%)
A1	750	190	1000	250	50
A2	750	190	1000	250	75
B1	1500	375	2000	500	50
B2	1500	375	2000	500	75
C1	3000	750	4000	1000	50
C2	3000	750	4000	1000	75

Looking at the results, we can determine that extremely strict resource limits greatly increase the processing time, as seen using resource tier A1. On the other hand, it seems that greatly increasing the available resource limits does not automatically improve the total response times. On the contrary, even when quadrupling the available resources in resource tiers C1 and C2 compared to the levels seen in tiers A1 and A2, the response times did not improve significantly. To find out the root cause of this behaviour, more experiments are needed, using perhaps other variables as well to influence the outcome of the tests.

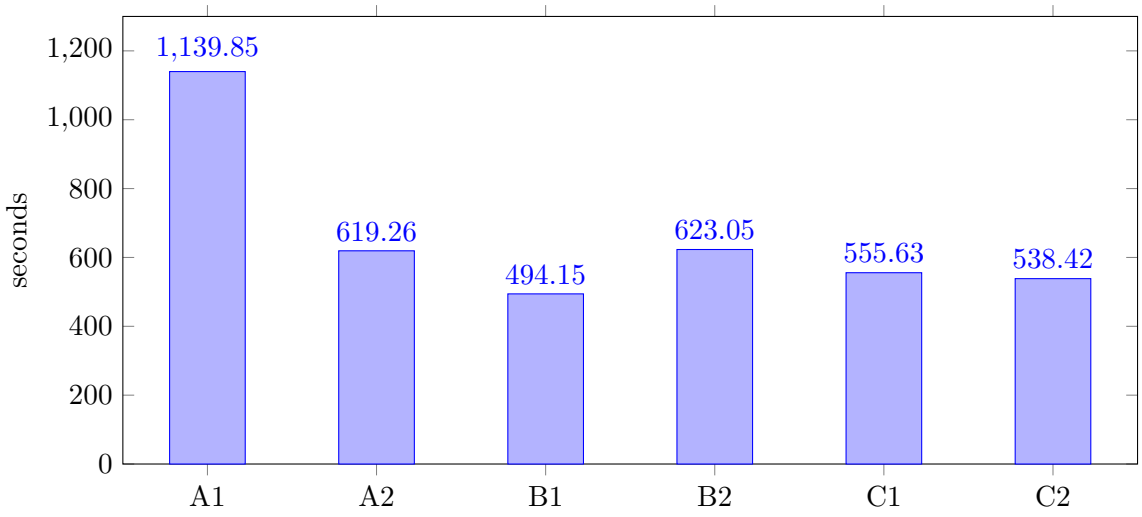

Figure 5.1: The total response times in seconds of running the data processing at the different resource limit tiers.

5.3 Development experience

Apart from the quantitative results of our experiments, on the way to implementing the data processing workflow and conducting the experiments we learned quite a few lessons in the development challenges of a task like this (Section 4.1). Notably, we analysed the difficulties of providing similar implementations in monolithic, microservice, and FaaS based environments, in order to measure and compare their performance. We also found out that Knative FaaS uses containers in its implementation, leading to the unique scenario where providing the same data processing implementation for microservice and FaaS based deployments trivial.

Chapter 6

Conclusion

In our work we present the challenges of data processing tasks, and many of the options to solving them. As demonstration, we implemented a data processing workflow for the handling data in the General Transit Feed Specification (GTFS) (Section 3.6) format. We implemented that workflow using templates provided by the Knative Function-as-a-Service (FaaS) (Section 3.3.2) environment, which allowed us to create components that can be run in containerized and FaaS contexts as well.

We tested our implementation using multiple tiers of resource limits, and measured its response time when using these tiers. It can be concluded that extremely low resource limits implicate a severe increase in the total response time. However, increasing those limits above a certain threshold does not seem to allow for significant improvements, apart from the drop in response time after the resource limits specified exceed very scarce limits.

6.1 Possible future work

The experiments can be expanded by running the workloads in the Knative FaaS environment, using Knative's own autoscaler solution. This would most likely produce different results, as this autoscaler uses different metrics than Kubernetes' own Horizontal Pod Autoscaler (Section 3.2.3).

Additionally, it would be interesting to analyse the performance of the data processing implementation when considering other metrics than the total response time. The results of such experiments would especially be interesting when conducted on a public cloud provider's infrastructure, as the price paid for running those experiments would be a clear indicator of the implementation's resource efficiency.

Acknowledgements

I would like to thank my advisors, Balázs Fodor and Dr. Balázs Sonkoly, for helping me develop this project from an idea into this work. I am especially thankful for their help provided not just in the early stages of the project, but throughout its course, and the significant extra energy they put in to helping me conclude this project.

Bibliography

[1] Apache Software Foundation. Kafka documentation. https://kafka.apache.org/documentation/, 2023. (Last checked: 2023-10-29).

[2] Apple Inc. and the Swift project authors. The Swift Programming Language. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/, 2023. (Last checked: 2023-10-29).

[3] Citizens of Szeged. Szeged local public transit GTFS data source. http://szegedimenetrend.hu, 2023. (Last checked: 2023-10-29).

[4] Docker Inc. Docker documentation. https://docs.docker.com, 2023. (Last checked: 2023-10-29).

[5] Docker Inc. Docker Engine documentation. https://docs.docker.com/engine/, 2023. (Last checked: 2023-10-29).

[6] IBM. What is container orchestration? https://www.ibm.com/topics/container-orchestration, 2023. (Last checked: 2023-10-29).

[7] IBM. What is containerization? https://www.ibm.com/topics/containerization, 2023. (Last checked: 2023-10-29).

[8] IBM. What is etcd? https://www.ibm.com/topics/etcd, 2023. (Last checked: 2023-10-29).

[9] IBM. What is FaaS (Function-as-a-Service)? https://www.ibm.com/topics/faas, 2023. (Last checked: 2023-10-29).

[10] IBM. What are microservices? https://www.ibm.com/topics/microservices, 2023. (Last checked: 2023-10-29).

[11] IBM. What are NoSQL databases? https://www.ibm.com/topics/nosql-databases, 2023. (Last checked: 2023-10-29).

[12] IBM. What is Redis? https://www.ibm.com/topics/redis, 2023. (Last checked: 2023-10-29).

[13] IBM. What is a relational databases? https://www.ibm.com/topics/relational-databases, 2023. (Last checked: 2023-10-29).

[14] IBM. What is a REST API? https://www.ibm.com/topics/rest-apis, 2023. (Last checked: 2023-10-29).

[15] IBM. What are virtual machines (VMs)? https://www.ibm.com/topics/virtual-machines, 2023. (Last checked: 2023-10-29).

[16] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access*, 7:154300–154316, 2019. ISSN 2169-3536. DOI: 10.1109/ACCESS.2019.2946884. Conference Name: IEEE Access.

[17] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 289–300, New York, NY, USA, June 2011. Association for Computing Machinery. ISBN 978-1-4503-0661-4. DOI: 10.1145/1989323.1989355. URL https://dl.acm.org/doi/10.1145/1989323.1989355.

[18] Knative Authors. Knative documentation. https://knative.dev/docs/concepts/, 2023. (Last checked: 2023-10-29).

[19] Knative Authors. Knative Function-as-a-Service documentation. https://knative.dev/docs/functions/, 2023. (Last checked: 2023-10-29).

[20] Knative Authors. Knative Function-as-a-Service build process documentation. https://knative.dev/docs/functions/building-functions/, 2023. (Last checked: 2023-10-29).

[21] Knative Authors. Knative autoscaling documentation. https://knative.dev/docs/serving/autoscaling/autoscaler-types/, 2023. (Last checked: 2023-10-29).

[22] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, pages 62–77, New York, NY, USA, December 2022. Association for Computing Machinery. ISBN 978-1-4503-9915-9. DOI: 10.1145/3567955.3567964. URL https://dl.acm.org/doi/10.1145/3567955.3567964.

[23] Microsoft. What is "managed code"? https://learn.microsoft.com/en-us/dotnet/standard/managed-code, 2023. (Last checked: 2023-10-29).

[24] Microsoft. Design applications for scaling. https://learn.microsoft.com/en-us/azure/well-architected/scalability/design-scale, 2023. (Last checked: 2023-10-29).

[25] MobilityData. GTFS documentation. https://gtfs.org/schedule/, 2023. (Last checked: 2023-10-29).

[26] MobilityData. GTFS documentation. https://gtfs.org/realtime/, 2023. (Last checked: 2023-10-29).

[27] MongoDB Inc. MongoDB documentation. https://www.mongodb.com/docs/, 2023. (Last checked: 2023-10-29).

[28] MongoDB Inc. Sharding in MongoDB. https://www.mongodb.com/basics/sharding, 2023. (Last checked: 2023-10-29).

[29] MongoDB Inc. What are Database Triggers? https://www.mongodb.com/features/database-triggers, 2023. (Last checked: 2023-10-29).

[30] OpenJS Foundation and the Node.js contributors. Node.js documentation. https://nodejs.org/en/about, 2023. (Last checked: 2023-10-29).

[31] Oracle. Java Virtual Machine documentation. https://docs.oracle.com/en/java/javase/17/vm/java-virtual-machine-technology-overview.html, 2023. (Last checked: 2023-10-29).

[32] The Linux Foundation. Kubernetes documentation. https://kubernetes.io/docs/home/, 2023. (Last checked: 2023-10-29).

[33] The Linux Foundation. Kubernetes Cluster. https://kubernetes.io/docs/concepts/architecture/, 2023. (Last checked: 2023-10-29).

[34] The Linux Foundation. Kubernetes Deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2023. (Last checked: 2023-10-29).

[35] The Linux Foundation. Kubernetes Service. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/, 2023. (Last checked: 2023-10-29).

[36] The Linux Foundation. Kubernetes Node. https://kubernetes.io/docs/concepts/architecture/nodes/, 2023. (Last checked: 2023-10-29).

[37] The Linux Foundation. Kubernetes PersistentVolume. https://kubernetes.io/docs/concepts/storage/persistent-volumes/, 2023. (Last checked: 2023-10-29).

[38] The Linux Foundation. Kubernetes Pod. https://kubernetes.io/docs/concepts/workloads/pods/, 2023. (Last checked: 2023-10-29).

[39] The Linux Foundation. Kubernetes Service. https://kubernetes.io/docs/concepts/services-networking/service/, 2023. (Last checked: 2023-10-29).

[40] The Linux Foundation. Kubernetes StatefulSet. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, 2023. (Last checked: 2023-10-29).

[41] The Linux Foundation. Kubernetes workload types. https://kubernetes.io/docs/concepts/workloads/controllers/, 2023. (Last checked: 2023-10-29).

[42] The Linux Foundation. Open Container Initiative documentation. https://opencontainers.org, 2023. (Last checked: 2023-10-29).