UNIVERSITY OF OSLO

MASTER'S THESIS

Black-box performance modelling and analysis of microservice systems

Author:
Magnus Nordbø

Supervisor:
Hui Song

A thesis submitted in fulfillment of the requirements for the degree of Master of Informatics

in the

Faculty of Mathematics and Natural Sciences
Department Of Informatics

September 24, 2023

Declaration of Authorship

I, Magnus NORDBØ, declare that this thesis titled, "Black-box performance modelling and analysis of microservice systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITY OF OSLO

Abstract

Faculty of Mathematics and Natural Sciences
Department Of Informatics

Master of Informatics

Black-box performance modelling and analysis of microservice systems

by Magnus NORDBØ

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

Contents

List of Figures

List of Tables

AI	Artificial Intelligence
ML	Machine Learning

TABLE 1: Abbreviations

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Keywords

Observability
Microservices
Signal-to-noise ration

1.2 Abstract

This thesis project aims to explore the practicality and efficacy of multivariate time series classification to quickly diagnose performance bottlenecks in a microservice system. A simple microservice system running in Docker was forked from another project and minimally adjusted. Metrics were collected about the system using the Prometheus monitoring software, and stress testing was done at the API level with Pumba. The resulting data was fed into different MTS machine learning algorithms to judge their accuracy in predicting the source of the stress. Finally, the results of these predictions were compared and the methods for collecting, treating and training on the data were documented to benefit future implementations of a similar approach.

1.3 Research area and questions

The research area for this thesis is microservices and machine learning. An inevitable intersection appears between the two fields when the need for analysis of data from microservice systems arises. This is because these complex systems are capable of generating a vast amount of metric data about themselves. The amount of data is too large for a human to read, comprehend and reason about in any efficient capacity. This can make it very difficult to draw useful conclusions about the system. This is where machine learning comes in as a natural answer to the problem: While humans are limited in the amount of information they can comprehend, machines typically only benefit from absurdly large datasets to draw from, as long as the data is of acceptable quality. However, the field of machine learning applications on microservice system data is still immature as of time of writing. This thesis seeks to analyze a small subset of this field. The scope will be contained to identifying latency issues resulting from disproportionally high load in a part of the system, using only data from centralized logging.
Research question 1: How good are current machine learning classification algorithms at identifying causes for anomalous behavior in microservice systems?
Research question 2: What are the current main challenges to effectively utilize metric data from microservice systems?

1.4 Approach

To research these questions and try to create meaningful results that can have a real impact, I formulated a hypothesis to work as the base for the project. ***It is possible to use stress testing software and centralized logging to create a machine learning model of the system behavior to classify and identify sources of latency in a microservice system.*** To test this hypothesis, a fork of the open source demo microservice project "sockshop" by Weaveworks was used [1]. This system was hosted on a local machine and stress tested using various configurations. Locust, an API stress testing tool. The full spectrum of available Prometheus metric data was collected in 601 time series features into csv files and labeled. There were a total of five classes, representing different endpoints connecting to different underlying microservices. This data was then analyzed both manually and mathematically, and preprocessed to reduce noise and extract important features. Several methods of preprocessing and classifying were quantitatively compared. Finally, the results of the analysis were discussed and conclusions drawn.

1.5 List of important terms

- **Dynamic Time Warping**: A method of statistically comparing time series across time points to judge similarity.
- **Multivariate Time Series (MTS)**: Time series data with two or more variables.
- **Instance**: All the data collected from a system in a specific time frame, represented as an MTS.
- **Feature**: A unique variable in the time series that expresses some kind of information about the system. Collected at fixed intervals to form an MTS.
- **Time frame**: The period of time recorded in a time series. Each instance in this thesis corresponds to a specific time frame.
- **Centralized logging**: Umbrella term for various microservice info logging methods that collect logs from individual microservices and stores them together.
- **Distributed tracing**:

Appendices – this is the folder where you put the appendices. Each appendix should go into its own separate `.tex` file. An example and template are included in the directory.

Chapter 2

Background

This section will present the main concepts that are relevant to the thesis project and the reason why it's useful. It will first explain the main theory behind microservices and how it seeks to solve the main problems in legacy software architectures. Then it will detail the new issues that arise from microservices. Finally, it will paint a picture of today's landscape of distributed software and tie it all together to explain the value of the research in this project.

2.1 The monolith

In general when talking about both microservices and distributed computing in general, the concept of a **Monolith** often gets brought up. The monolith is this concept of a large, self-contained application where all the code and functionality is tightly bundled together, and is very hard to separate into individual parts. The definition has changed over time: According to the ITS back in 2001 [2], a monolith was "An application in which the user interface, business rules, and data access code is combined into a single executable program and deployed on one platform." However, most modern interpretations online refer back to a book from 2003 called *The art of Unix programming*. Usually referred to as "monster monoliths", it marks the beginning of the trend of using the term monolith as a bogeyman of unmaintainable, poorly planned code. This trend has been continued in modern days with software "gurus" and the like when needing something to compare to our lord and savior microservices [3]. It is therefore important to remember that:

1. The monolith is not a defined software architecture design paradigm. Rather, it is the default type of application that appears when not taking great effort and intentionality to split the code up in many distinct parts.

2. The monolith is not some damnable evil to be conquered: On software projects of a less-than-huge scale, it is very often quite optimal. It doesn't have to deal with inter-component communication. Keeping everything contained in one code base makes it easy to run and test on local machines, keeps all the code in one place for easy access, and so on.

3. The monolith mostly just exists as a concept when talking about microservices or other ways to break it up. Its purpose is mainly to demonstrate the benefits of code splitting in large software.

With that out of the way, let's discuss the problems with monolithic software that microservices seek to amend.
The core issue that with monolithic software is **entanglement.** In this context, it means how complex monolithic software will have too many interconnected, interdependent parts to manage effectively. This interdependence will mean that something that is a problem for one small part of the whole, will be a problem for the entire tech stack.

2.1.1 Scalability

Scalability refers to the ability to increase the workload capacity of the software [4]. If the amount of users, data, geographical reach or computation complexity increases, it becomes necessary to increase hardware capabilities. Scaling up monolithic software can be a time-consuming and costly affair, as it usually doesn't allow for only increasing the capacity of the parts of the software that are facing increased load.
An example:

> On the 24th of December 2004, Delta Air Lines' crew scheduling system crashed because of an integer overflow bug resulting from record system load. It took 24 hours of frantic work to implement a workaround that duplicated a database [5]. It used an old flight scheduling system called TRACK from 1986 to organize scheduling. Their database that handled pilots and flight attendants ran into a hard limit at 32000 entries. The hotfix for this problem was to spin up another server, and let one database only handle pilots and another only handle flight attendants. The crash caused widespread disruptions. According to the Department of Transportation of the US, approximately 269,000 passengers were affected by delays and cancellations [6].

Issues from heavy loads like this are always a risk. But entangled monolith systems are not easy to extend. In a microservices oriented system, implementing this extension fix would likely be on the scale of modifying a few lines of code and spinning up another instance of the database. I highlight this example because the fix for this huge problem ended up being a basic principle of microservices: Generating more instances of only the overloaded components.
When an application has reached load capacity, one realistically has two options [4]:

1. Improve the efficiency of the software, so it can serve higher loads with the same hardware
2. Upgrade the hardware

Naturally, the software improvement solution will only get you so far. It has the added issue of (in this example) being a monolithic application, so big changes to the software is a time consuming and costly project.
When upgrading the hardware capacity of monolithic applications, it often becomes a project of its own. One with expenses. Because of this, and the fact that monolithic application will have to be replicated in its entirety on the new server by default, a scaling project will try to calculate the future growth of the user base and invest in a solution that will be able to handle user load for the foreseeable future. This is a point of potential failure, as being wrong about future growth could turn out to be quite costly.

2.1.2 Comprehension

Comprehension of large software is impossible. There is a human limit to how much one can keep in one's mind. In large software, the amount of functions and interactions can balloon into the tens or hundreds of thousands. Fully comprehending such a system is impossible for human brains, and it becomes necessary to abstract it down to a manageable mental model. Without good comprehension of the system, it is very difficult to make good changes, adjustments or additions. A highly entangled monolith can very difficult to abstract, without clear boundaries for which parts do what.

2.1.3 Modern software development

Modern software development is characterized by autonomous teams working separately and collaborating through continuous integration. This means: build often, test often, merge often. According to Agile practices, one should write tests first, then write code that passes the tests. But

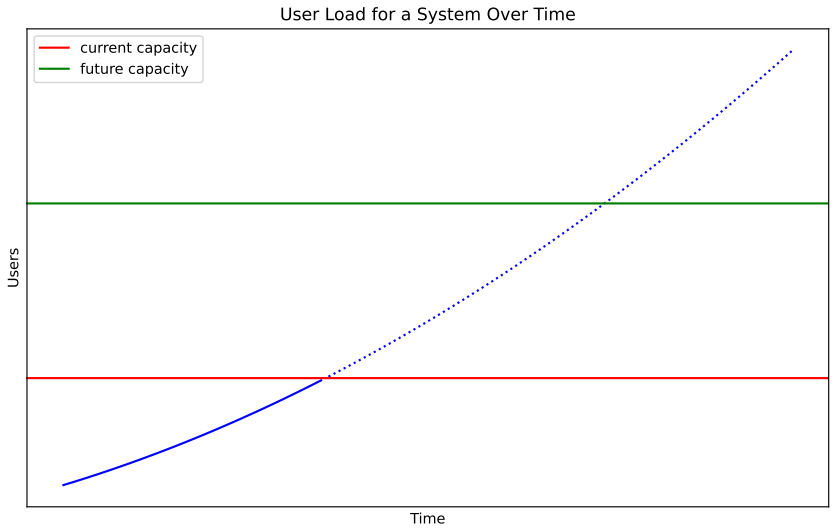

FIGURE 2.1: Scaling a monolith requires making assumptions about future growth

writing tests for highly entangled monolithic software is challenging, as even small changes can have consequences far down the chain. Building often will also become costly: Monolithic software will typically need to be rebuilt from scratch each time. A costly affair, and entirely unfeasible if the goal is to build several times a day.

2.1.4 Resilience

Resilience has several definitions in the context of software. [7]. In the context of this thesis, we can define it as "the ability to keep running correctly in spite of failures". Often called fault tolerance. The entanglement of the monolith once again becomes a problem here. A single function running into an unexpected situation and throwing an error will, by default in most technologies, abort the execution of the program entirely. It can be circumvented by try/catch statements and other error handling methods, but an entangled system is extremely difficult to make fault tolerant.

2.1.5 Technical debt

Technical debt is a somewhat loosely defined term. It is defined by Steve McConnell as *A design or construction approach that's expedient in the short term, but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time) [8]."* As a monolithic application grows, making any change becomes a larger project. The more entangled everything is, the more changes will have to be made to accommodate updated dependencies or changes in behavior. These changes to accommodate the change can also cause other things to be changed. This is often called the "ripple effect" or "cascading changes". This can lead to mounting technical debt: As updating components is such a project, it is delayed, or jury rigged to work in the short term. But each delay or subpar implementation leads to an increase in how big of a project it would be to properly update dependencies. Technical debt will also make regular maintenance more difficult and time consuming over time. Technical debt is not a problem that is unique to monoliths. But it is hard to pay off that debt through refactoring and maintenance in an entangled system that handles change poorly.

2.2 Containers

Containers are a technology that packages a unit of software and all its required dependencies. They make use of the Linux kernel's namespaces feature. Namespaces partition system resources in a way that processes only "see" the resources in their namespace, instead of the total system resources. This provides a level of abstraction and security, as processes can only access their own namespace. By utilizing namespaces, containers can provide isolated workspaces for processes, essentially allowing applications to run in their own "virtual" system, oblivious to other processes running on the same host. This makes containers completely dependency and OS agnostic, since each container comes with the OS and packages they need to run their process. Containers are similar to virtual machines, but much faster and more lightweight. VMs virtualize the hardware and entire OS, like RAM and the kernel, and then run their OS on the virtualized hardware. Containers simply virtualize the OS, running it on their assigned namespace on the host's kernel. This takes far fewer system resources at a slight cost to isolation. This makes them viable as a way to organize code segments. Containers talk to each other using regular network calls. Because of this, an application can freely be split up between several machines.

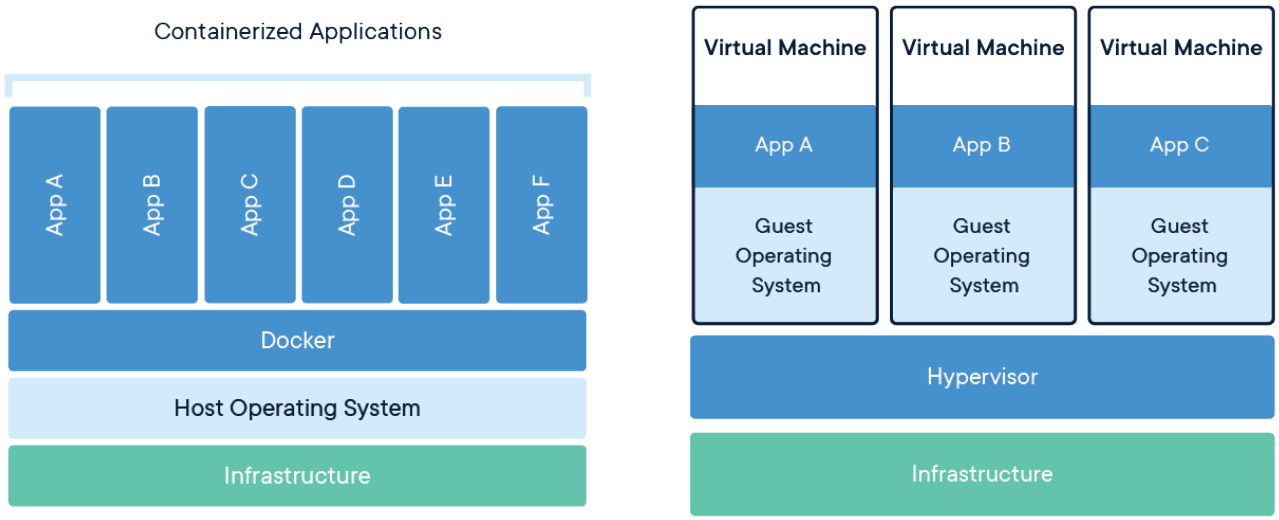

FIGURE 2.2: Containers vs VMs. Image credit: Docker inc

2.3 The microservice

The core idea of microservices is modularization, which has been a concept in software engineering since its inception. Microservices are a modern spin on the concept with cloud and DevOps in mind. The defining feature of the microservice is the container. By splitting the code of a piece of software into several autonomous containers that talk to each other using web requests, one can counter a majority of the problems with large software projects.
Let's examine how microservices tackle the main problems with monolithic software.

2.3.1 Scalability

Microservices are designed to be very easy to scale. Because the containers in a microservice system talk to each other using network calls anyway, a microservice system does not particularly care *where* the containers are located. This means you can gracefully distribute the software across servers and spin up only however many instances of each microservice you need. This works well with modern server renting solutions, that let you pay for only what you need. In this way, the aforementioned

scaling project for monoliths that try to calculate future growth becomes unnecessary. One can simply scale out only what one needs, when one needs.

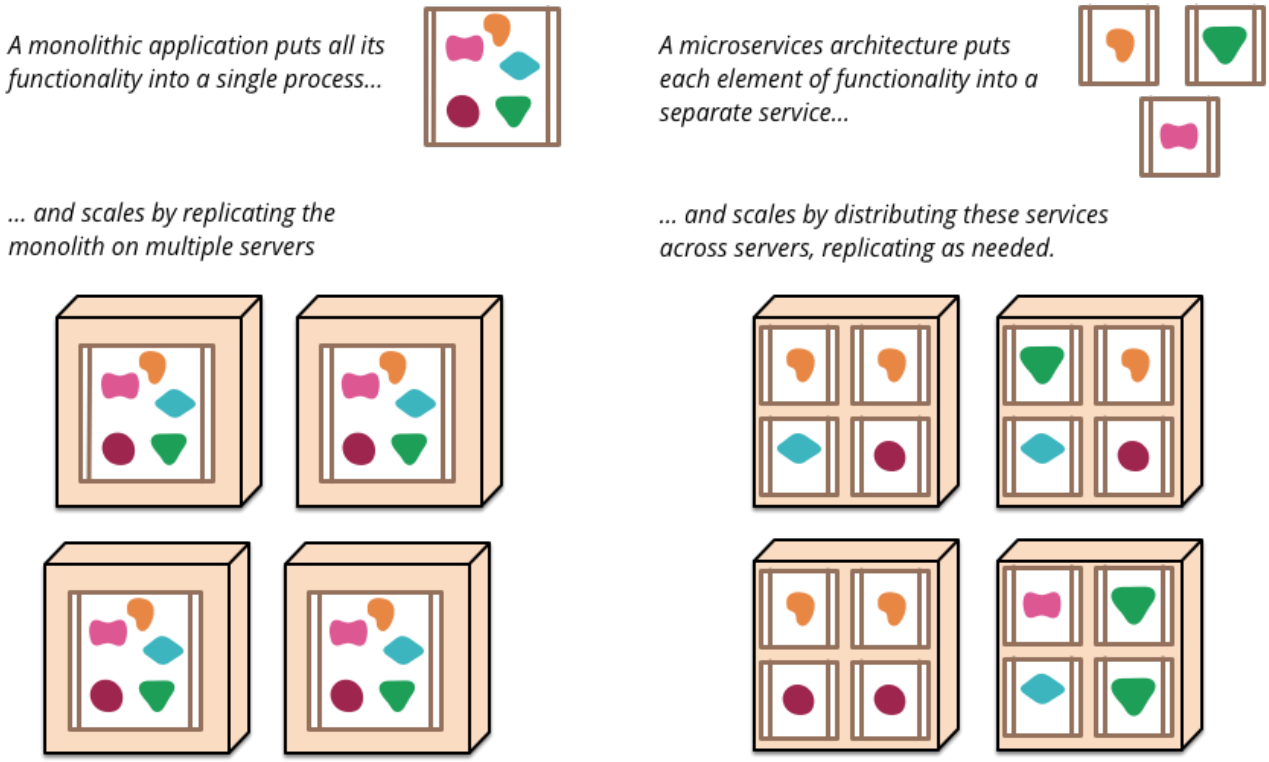

FIGURE 2.3: Illustrating how monoliths and microservices handle scaling. Credit: Martin Fowler [9]

2.3.2 Comprehension

Microservices promise to make complex software easier to work with. It does not typically do this by reducing complexity, but by providing real, traceable boundaries between units of software, and by use of interfaces. By software boundaries, I mean that developers have an objective rule to follow for breaking code up into pieces that can be mapped out and abstracted. Without microservices one can still map out a program, but the boundary lines between units can often be hard to nail down in a satisfying manner. Functions are a natural "unit" to break a program up into, but functions vary a lot in size, scope, and utility. Mapping out a program or its states, for example in UML, requires reasoning about a level of abstraction to use in the current map context. This decided level of abstraction will likely need to represent some functions by themselves, while grouping others together. While this is not necessarily a bad thing, it makes the process of mapping out a program require skilled, subjective reasoning. Microservices, on the other hand, provide an objective boundary that makes sense in the context of mapping out a program to comprehend it. Thus, as long as the microservice architecture itself is well thought out (which is not guaranteed at all), no skill is necessary to create a logical abstraction of the system that is guaranteed to be useful for comprehension.

Interfaces are the intended inputs and outputs of each microservice. They help comprehension by letting developers think of the rest of the program as a black box, and only focus on comprehending the parts of the program that interact with the microservice they are currently working on. They don't have to know what the entirety of the system does to make good changes (in theory).

2.3.3 Modern software development

The modern day is a time of DevOps and agile development. The core principles of DevOps is up for some debate [10], but can be roughly summarized to be a set of guiding principles to facilitate communication and collaboration between developers and operators (hence the name DevOps). It places emphasis on continuous integration, delivery, and deployment (CI/CD). Microservices are often regarded as a core component of DevOps [11]. The independence of microservices aligns well with DevOps principles. Each service can be developed, tested, deployed, and monitored independently, allowing for more frequent updates and releases. This plays into the CI/CD model central to DevOps: updates are continually integrated into the software, tested, and then deployed. By breaking down the application into smaller services, teams can manage and maintain their CI/CD pipelines more effectively, each focusing on their own service.

Microservices also fit well within Agile methodologies. Agile focuses on iterative development, where software is developed in small pieces, with stakeholder feedback incorporated along the way. Microservices support this iterative approach by allowing teams to work on different services simultaneously.

2.3.4 Resilience

The ability to handle faults, crashes and unexpected occurrences is crucial to modern software platforms that provide necessary services to large parts of the population. Modern microservice technologies and services, like Kubernetes, OpenShift, Rancher, Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), and many more, provide automatic load balancing and self healing. This means that they will detect faults and spin up new instances of the microservices to take over the workload of the failing service. This makes microservice applications much more robust than monoliths, which are constantly at great danger of letting a single fault bring the whole service down.

2.3.5 Technical debt

The way microservices help deal with technical debt is by making it easier in general for developers to make changes and adjustments to the system. That is to say, technical debt will accumulate in microservice architecture just like it does in monoliths. It is just that the more robust and changeable microservice system is easier to refactor individual parts of the system. This is achieved through the total separation of components that talk to each other through **interfaces**. The interface in this context just means the expected inputs and outputs of each microservice. Each individual microservice is "blind" to what goes on in the others, and treat them as black boxes. All they care about is the interface: If what goes into a service and what comes out of it has the same properties as before, a microservice system will not notice a change even if the entire technology stack of the microservice is replaced. This makes updating to new technology a much simpler process: A developer just has to make sure it can receive and send the same data as before, and not worry any further about ripple effects.

2.4 The problems with microservices

Microservices solve many problems with legacy software, but it also introduces issues of its own. The problems that microservices create have largely to do with complexity. The reality is that microservices aren't less complex than monoliths, they just exchange one type of complexity for another. One that, it is argued, is easier to deal with. It leads to issues nonetheless. This section will give a detailed overview of the problems that microservices introduces.

2.4.1 Complexity

Microservices do not reduce complexity: They simply exchange one type of complexity for another. The issues with complexity in microservices come from the architecture itself, and from the fact that it is so loosely coupled that it allows for any technology stack and programming language.
Microservice has to orchestrate and manage many, many nodes. The scaling method of creating more instances of microservices that are struggling to handle load is great for easy scaling, but it can easily become a nightmare to manage. When multiple instances of the same service are to cooperate, care has to be taken to split the workload in a good manner. Careless management can lead to race conditions, data getting lost, or tasks being done several times in a redundant fashion.
Larger systems will have hundreds or thousands of microservices that need to communicate with each other in a complex web that is difficult or impossible to comprehend. Orchestrating and managing this complex web of web requests is a field still in development.

2.4.2 Communication overhead

The decoupled nature of microservices comes at a concrete cost in communication overhead. If one compares two equally well made services of equal function and size, one running on a monolith and one on microservices, the monolith should have less latency and better performance. The cost of communication is twofold: Complexity (as mentioned above), and resource usage. There are ways to organize connection complexity, like event-driven architecture (EDA), service mesh solutions (Istio, Linkerd, Cosnul) or serverless architecture solutions like AWS Lambda, which abstract the network away from the developer entirely. But all of these solutions need computing power and bandwidth to work, which means they come at a cost of performance.
There is also the issue of the various units of code communicating with each other through network requests. This is, in many ways, the secret sauce that makes microservices so reliable and scalable. But function calls are faster and less resource intensive than network requests by an order of magnitude. Internal function calls just deals with local memory, and their speed is measured in nanoseconds. Meanwhile, network requests are typically measured in milliseconds. The distributed nature of the microservices also mean that a service could be needing to communicate with another server on a different place on the planet. This is also a significant factor for latency, and can be a bottleneck for service quality.

2.4.3 Data consistency

Data consistency (and the lack thereof) is another complexity trade-off in microservices architecture. While implementations vary, it is common for microservices to each have their own little database for storing data relevant to their work. This leads to resilience, but also means that changed values may take time to propagate to all relevant microservices, leading to microservices potentially disagreeing on the value of certain variables. This can lead to inconsistent data, where an end user might get a different result depending on time of day or where they are pinging the service from.
Updating data can be a challenging affair. If, for example, there is a network of 100 microservices that need to be updated with

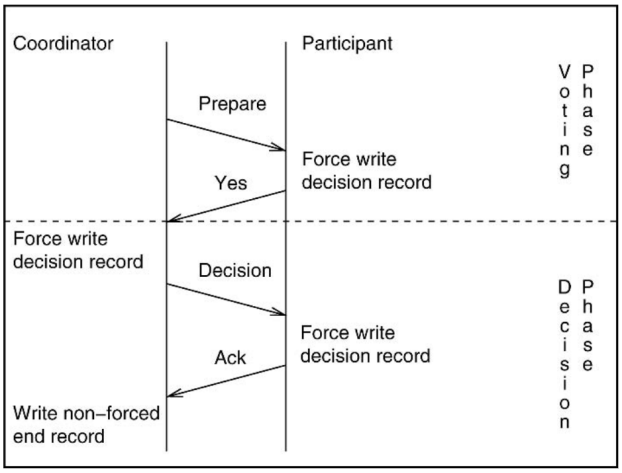

FIGURE 2.4: The two-phase commit protocol. Credit: [12]

new data, that is 100 potential points of failure.
If data consistency is important, changes would need to be rolled back if any of the 100 transactions fail. That would require noticing a fault occurred, then notifying all the services that changed successfully that they must roll back, introducing more potential points of failure. The most common implementation of this is the *two-phase commit protocol* (2PC). It works by assigning one node as the coordinator/master, who sends a query to commit message to all other nodes. All nodes then vote, by performing some internal check to see if it could successfully commit the proposed change. This is the first phase of 2PC, the voting phase. If all nodes voted yes, then a commit message is sent from the coordinator/master to perform the change. This is phase two of 2PC, the decision phase. This process does a pretty good job at ensuring data consistency, but has many potential issues. While the core algorithm itself is pretty simple, the actual implementation ends up being complex because it has to include protocols for many types of issues [12].

2.4.4 Testing and debugging

Microservice architectures can become very complex. As stated earlier, it is an intentional complexity trade-off that should be more manageable in theory. Their specific brand of complexity gives rise to a specific brand of headaches for testers and developers. The complex network interactions between independent nodes create new potential points of failure that can be hard to track and debug.

Chapter 3

Methodology

In this chapter, I will describe the decision-making process behind the system used for generating and collecting test data. I will then describe the system itself in more detail.

3.1 Background

The project was initially started by Sintef in collaboration with a company that runs a microservices hosting platform. This company had requested Sintef to do research on their platform, to look at ways to anticipate issues like version conflicts and performance drops.
Another student and I were brought onto the project as master thesis project tie-ins. The other student was to focus on technical lag, and I was to focus on performance. However, soon after we had been brought onto the project and gotten the go-ahead from the faculty and everything stamped and sealed, the company that hired Sintef simply stopped responding. No e-mails, no meetings, no phone calls. So we were left stranded, and would have to come up with something for ourselves.
We decided to go ahead with the initial goals of the project: To research a microservice system and try to make predictors that could see performance and compatibility issues ahead of time. But instead of simply being supplied information about an existing microservice system, we had to create our own systems and our own tests.

3.2 System selection and creation

Building a microservice system can be a very complex affair, and subject to a master thesis project on its own. So making one from scratch would be far outside the scope of the project. As such, a list of requirements for the project were devised:

- The system would need to be relatively quick and easy to get up and running.
- It must be possible to generate faults in the system.
- The system must be able to collect and provide data about itself and its performance.
- It must be feasible to run and stress test this system on a student's budget.

3.2.1 Cloud solutions

Cloud computing is in several ways a natural fit for this project. Microservices as a concept is formulated with cloud computing in mind. Cloud service providers tend to provide some microservice-specific services. The three biggest players in cloud services are Amazon, Microsoft and Google [13]. They all offer some level of free access, and student deals can be negotiated for more credits. They all integrate Kubernetes while adding their own ease-of-use features, so launching some premade system on them should be feasible.

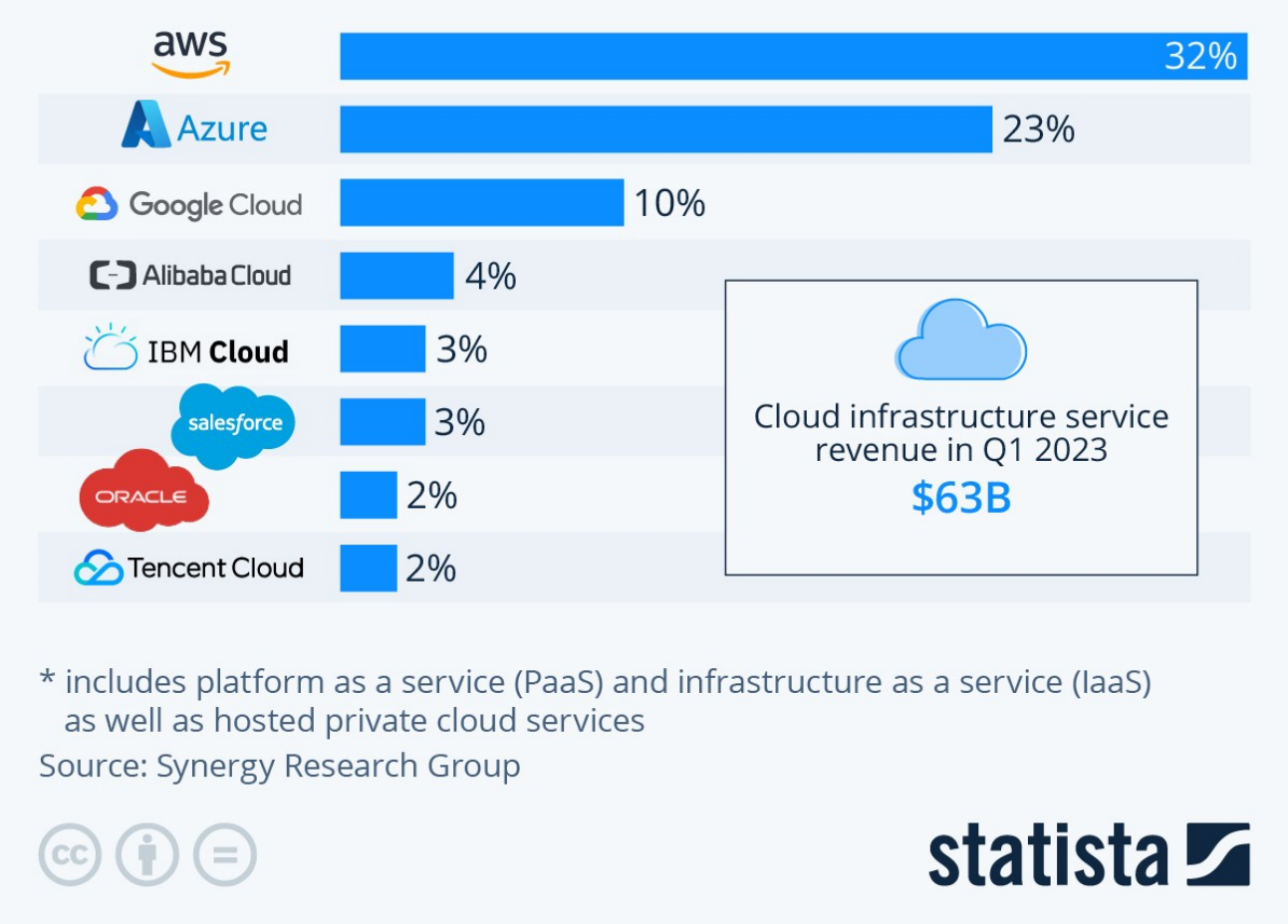

FIGURE 3.1: Worldwide market share of leading cloud infrastructure providers in Q1 2023. Source: Statista [14].

However, they come with some issues. The biggest issue is cost. In the course of my work, I was bound to do quite a lot of stress testing. This would be bound to burn through a ton of credits. Another issue would be dealing with their own built-in load balancing and DDoS protection. Getting accurate results from stress testing could turn out expensive as well as unreliable.
I set up free accounts for both Google Kubernetes Engine (GKE), Google's distributed cloud service, and Amazon Web Services (AWS), and reached out to apply for a student grant. However, response was slow, and it looked unlikely that I would be granted the necessary amount of credits.

3.2.2 Local machine

With the prospect of using cloud infrastructure seeming slim, we turned to doing what we could with the resources available to us. We both possessed personal desktop computers with decent processing power. Running on a local machine would give much better control over the system, and the project would not be beholden to the whims of a cloud service provider. It does come with some drawbacks. Microservices are a cloud focused architecture type. Data generated from a local machine it is much less likely to be directly relatable to real world systems that might make use of the research.

There are a variety of tools that help launch containers for microservices locally. Most of them are a subset or extension of Docker.

3.2.3 Candidates

DeathStarBench

DeathStarBench is an open-source benchmark suite developed by Cornell University for research purposes. It features a total of five complete services: A social network, a media service, hotel reservation, an e-commerce site, a banking system, and a drone coordination system for drone swarms. If chosen as the system for this project, we would have focused on one of the first three services. They are in a more complete state than the rest according to the project's GitHub page. DeathStarBench makes a compelling choice as it seems to be quite complete and sophisticated, as well as being made specifically for testing [15].
However, there were two main issues with it.

- System requirements and complexity

The complex nature of the project made it unclear if we would be able to run it on our available hardware. There were also doubts as to how simple it would be to get up and running in the first place. We would run the risk of spending a lot of time getting it up and running, only to find that it doesn't have enough resources to run properly.

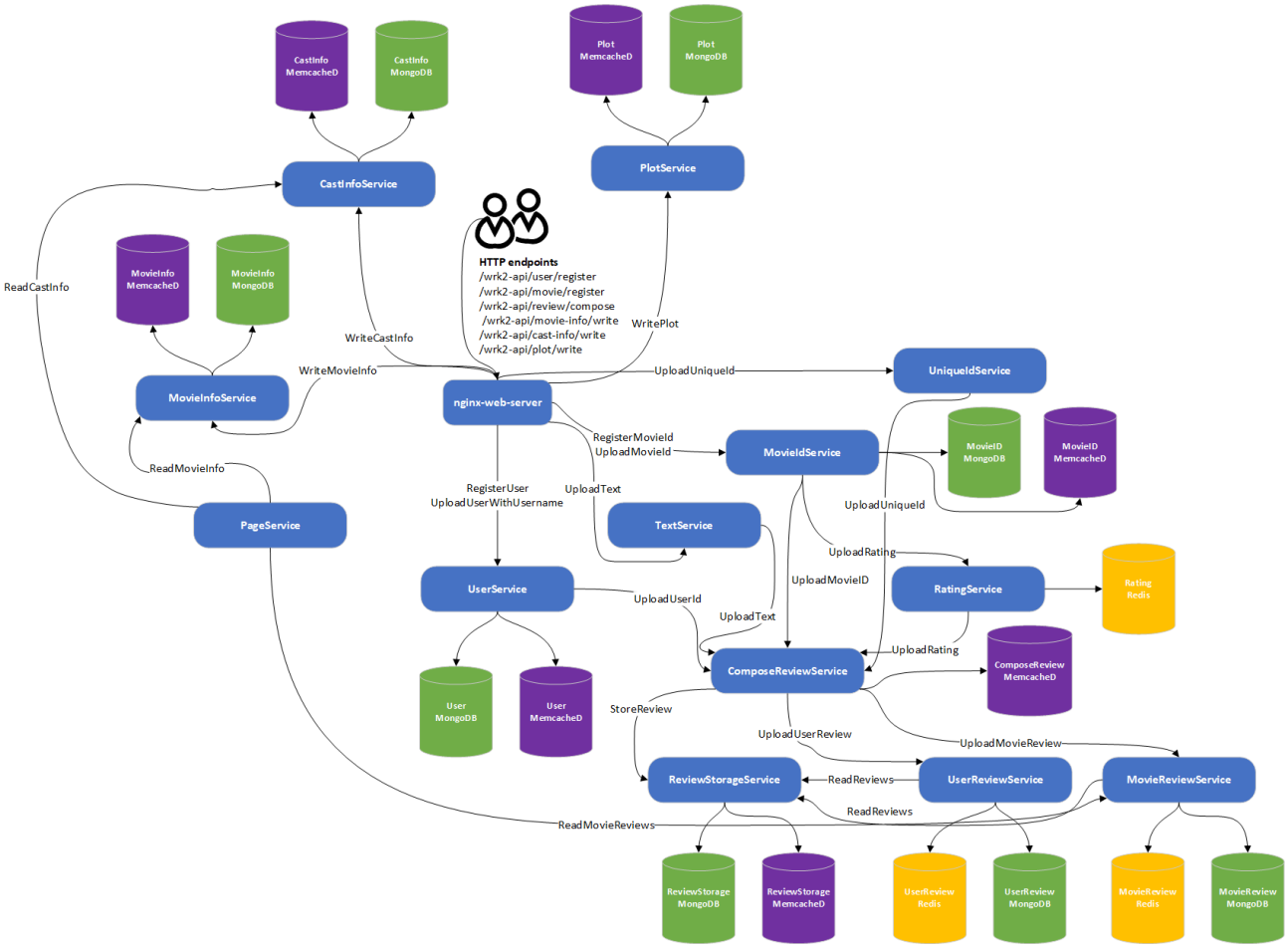

FIGURE 3.2: The workload architecture for DSB's media service [15].

- Distributed tracing

DeathStarBench uses Jaeger to provide distributed tracing of the system. This provides detailed information about the behavior of the system during operation. Higher quality data lets ML models

train more efficiently and provide more accurate and detailed results. Working with this would put the focus of the project work into the realm of making use of the data for more detailed prediction purposes, something that is already quite thoroughly studied by people with more skills and resources [16], [17], [18].

Sockshop

Sockshop is a lightweight microservices demo created by Weaveworks inc. It is "intended to aid the demonstration and testing of microservice and cloud native technologies" [19]. It simulates an online sock retailer, complete with users, carts, catalogs etc. It is an older, smaller project, and parts of it were used to build the larger DeathStarBench [15]. The smaller, less complex nature of this project means that it could feasibly run on our available home machines. It is composed of 14 services running in individual containers. The messaging between them is handled by RabbitMQ, which runs as its own container as one of the 14.

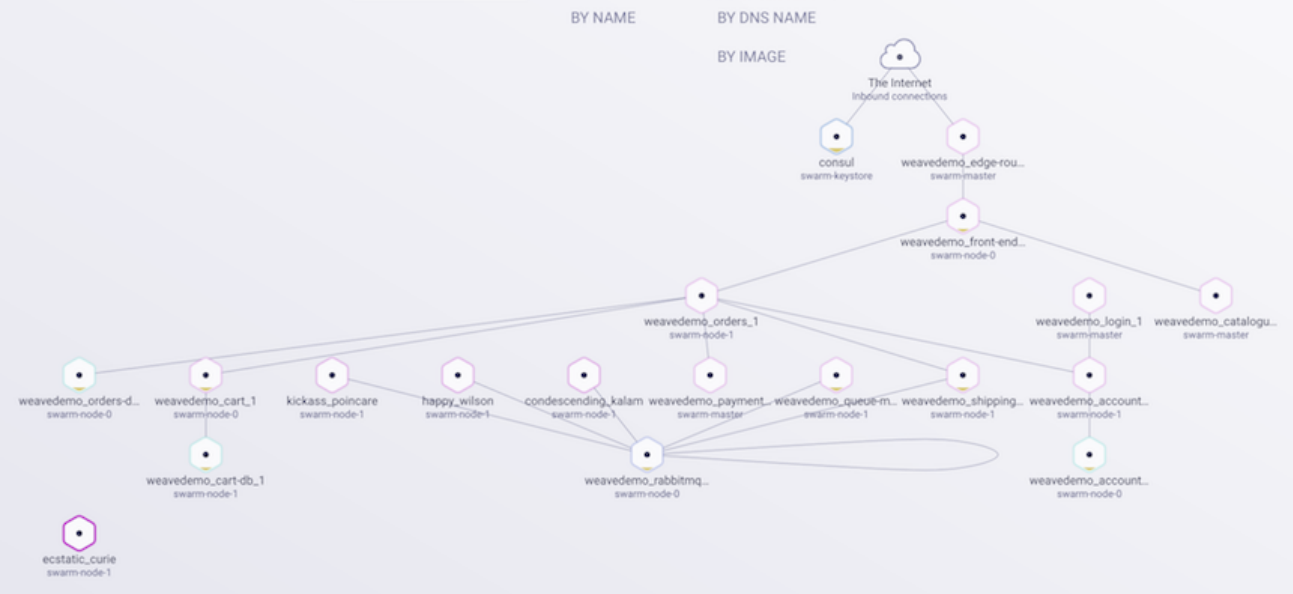

FIGURE 3.3: The architecture of the sock shop shown using WeaveScope, Weaveworks' own mapping tool. [19]

Sockshop comes with some strong benefits:

- Forgiving system requirements

Of the strong candidates, this was the one most likely to not run into a hardware bottleneck on our computers.

- Centralized logging

Sockshop comes instrumented with Prometheus, and publishes an API endpoint for retrieving metrics from the Prometheus microservice. Centralized logging provides less insight than distributed tracing, but carries less resource overhead and is easier to add to an existing system. This provides an angle to give a unique value proposition to the project: A method for learning about and optimizing an already existing system, while minimizing time and effort spent instrumenting the microservice system and training the models. Results would not be as accurate or reliable as more thorough projects using data collected from distributed tracing, but it could provide a "good enough" result for a fraction of the investment.

There are also some major downsides to the Sockshop.

- Poor documentation

The website for the project [19] has documentation, but it gives few details about the underlying structure of the system. For example, it has Prometheus instrumentation. But it gives no information on which client libraries are installed where. Finding out requires digging through the source code and metrics. The project has been dead for some time. The last commit on GitHub as of time of writing was on August 17, 2021.

Spring petclinic

This is a project by the Spring petclinic community on Github. Spring petclinic was a demo application created all the way back in 2004 to showcase the then brand new Spring framework. Since then, it has gone through several changes, but was put up on GitHub in 2013. It has since been developed into several forks that take the demo application in different directions. One of those directions is the *spring-petclinic-microservices* fork, which showcases how one can split a monolithic application into microservices. This fork is currently by far the most popular of the forks. The petclinic demo application is quite simple, as it is essentially just a database system. It consists of only three main components. By "main components", I mean microservices that focus on delivering the service to the end user. In addition to the main components, it comes with Prometheus, Grafana

Chapter 4

Data processing

4.1 Imputation

4.1.1 Background

Most relevant algorithms for time series classification do not accommodate missing values. Both the .csv data format utilized for storing the data and some internal data representations employed by sktime and sklearn for processing do not account for missing values. These missing values are represented as "NaN", which stands for "Not a Number". In this document, we will refer to these missing values as NaNs.

To address this issue, several methods can be employed:

- After collecting all the data, remove any columns containing NaNs. This approach results in data loss but ensures the accuracy of all remaining values, as it avoids estimation.
- Limit algorithm usage to only those capable of handling NaNs. A considerable number of algorithms in sklearn and sktime can work in this manner, but it still imposes a constraint [20].
- Implement imputation of missing values, which entails using an algorithm to make an informed guess about a plausible value based on existing values in similar positions.
- Perform dimensionality reduction on the data. Reduction algorithms like Singular Value Decomposition (SVD) are good for removing dummy data like NaNs.

Initially, the first method was effective for the project. This was due to the simulation being poorly configured and not subjected to significant stress. Additionally, there were insufficient time points collected and an inadequate number of instances generated. This meant that there were few opportunities for NaNs to appear in the collected data, so few columns had to be discarded. When the stress testing improved to be more stressful and more data points were gathered, significantly more NaNs appeared and quite a few columns would have to be discarded, inflicting significant data loss. The second method was briefly considered, but since the project's primary goal was to compare multiple classification methods, this idea was quickly discarded.

Ultimately, imputation emerged as the most suitable solution. Imputation of univariate datasets is typically straightforward: Missing values are assigned the mean, median, or most frequent value for their respective column. Multivariate imputation is more challenging. The primary issue is that each column may exhibit significant variation between instances. Simply using a statistic about the entire column would dilute the data, thereby worsening the signal-to-noise ratio.

4.2 Series length

The data collection program that collects the Prometheus data and saves it as .csv files strives to obtain the same number of time points for each instance. This is achieved by using consistent timeframes and collection intervals. However, this is not always possible.

Factors such as the test machine's poor performance under heavy load or data loss during the cleaning process may cause slight variations in the number of data points or time points between some instances. This issue presents a challenge for statistical classifiers, as most of them are designed to work with datasets of equal length.

There are two main ways to deal with this problem:

- Use only algorithms that can handle series of unequal length.
- Perform

In sklearn/sktime, exactly two classifiers are able to handle series of series length: A KNN classifier and an SVC (Support vector classifier). They will be used for comparison, but having more options to compare would be better.

4.3 Feature selection

The raw data collected from the Prometheus service has 579 features. Most of this data is useless noise. I have identified the following main factors that make data into noise:

- **Variables with zero or very low variance between instances.** These are very common because the Prometheus instrumentation of the test system are generic, i.e. they simply expose as much information about the system as they can. Many parts of the system go fully or relatively untouched during the (quite superficial) stress tests.
- **Subsets of data with large amounts of NaNs.** These datasets can still be useful if the non-NaN data points contain useful information. The problem is that the missing data points will have to be imputed, potentially diluting the usefulness of the data.
- **Variables whose changes are unrelated to the stress testing.** These include counters that track how long certain threads have been running, new instances of unrelated subprocesses, etc.

Noisy data will lead to poor performance of the machine learning model because the noise will mask the real underlying function it tries to learn. To get good predictions out, the main task is going to be separating noise from information.

4.3.1 Zero variance features

The variance numbers shown in figure 4.2 are scaled. This means the variance value for each variable is proportional to the mean value of that variable. Of the 579 variables in the dataset, 110 of them have a variance of exactly 0. This means they are entirely unaffected by both time and stress testing. Such variables are entirely useless and simply dilute the information in the data.

4.3.2 Low variance features

Low variance features can be tricky to make good use of. Deciding which features say a lot about the system with small changes, and which features simply have little to say requires good domain knowledge and expirementation. Once one has a solid idea about the relevance of the various low variance features, one can make sure they matter more in the data set. One approach to this is simply model selection: Tree based models like random forest will naturally treat features of varying variance equally or near-equally. This can be a great choice if one does not want to change the data too much for whatever reason. However, this same attribute means one has to prune low-impact low-variance data, lest they contribute a lot of noise. Another approach is to solve this low-variance high-impact problem through preprocessing. Normalization and/or standardization will naturally even out the

variances between features in an MTS. However, the way this evening out occurs is different. As demonstrated in 4.1, standardization forces all variances to be exactly 1, while normalization is more dependent on the dataset. It is worth noting that the variances around 0.1 in the graph is not a rule, and dependent on the range and variance on the original dataset. To make of low variance features in datasets that are going to be normalized, it might be important to

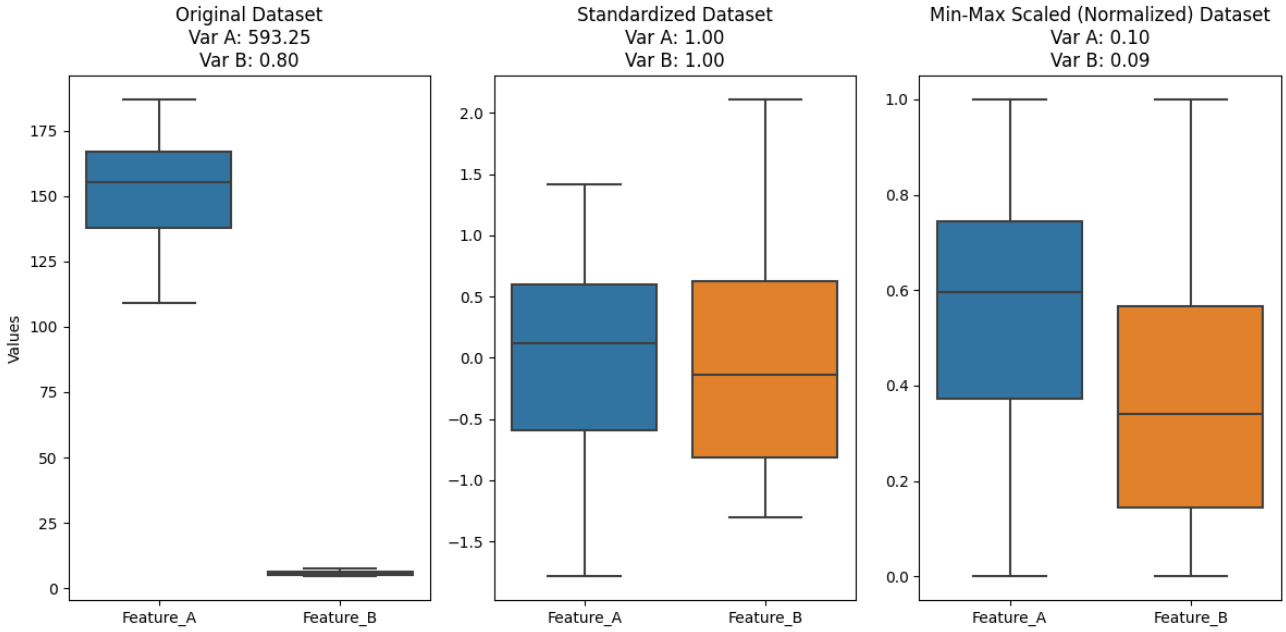

FIGURE 4.1: How standardization and normalization impacts variance in datasets of different scale

does stan

4.3.3 Monotonic features

Another issue is monotonically increasing features. In this dataset, these variables tend to be some form of counter. Simply throwing them in with the regular variables would add a lot of noise, as the algorithms would not know to treat them differently. They would have to be treated differently to not be noise because they are consistently changing, i.e. growing. Most machine learning algorithms would only see the change in value, and try to associate the static changes with some part of the data. This would dilute the actual information in the dataset an confuse the algorithm. Monotonically increasing variables can still be very useful in machine learning if treated properly, but it is out of the scope of this project. This is mainly because using monotonic variables require both good domain knowledge and good knowledge of how to transform the vairables into a format that can provide value for training. For the sake of scope, they are simply removed from the dataset when training in this project.

```
def select_by_variance(df:pd.DataFrame, amount:int):
    variances:pd.Series = calculate_variance(df)
    selection = variances.iloc[0:amount]
    return selection.index
best_features = select_by_variance(trimmed_df, 5)
X = scaled[best_features]
```

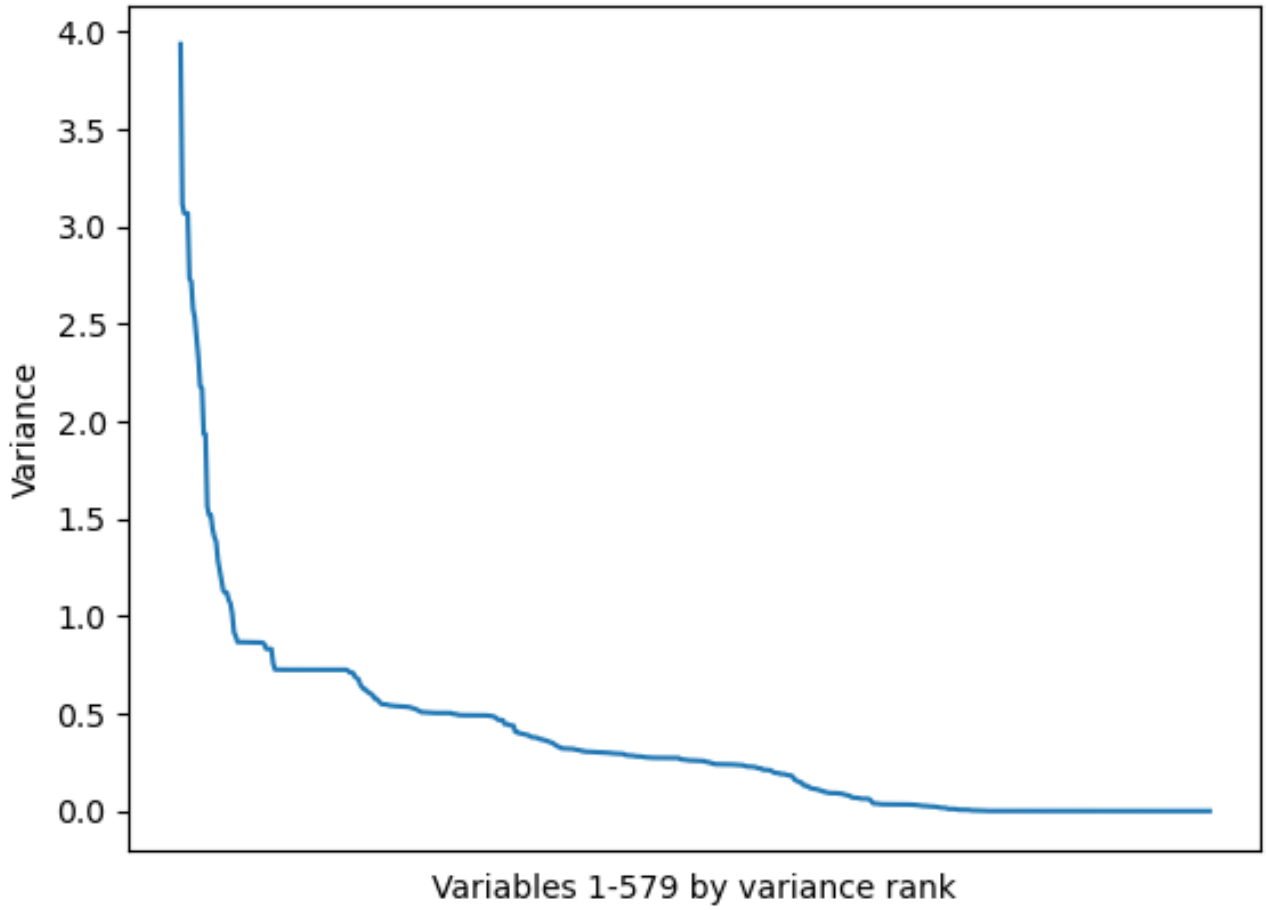

FIGURE 4.2: Scaled variance of all the variables

4.3.4 Variable vs feature

In time series learning, the terms variable and feature are often used interchangeably, which can cause confusion. Crucially, variables are a type of feature.

Bibliography

[1] Weaveworks, "Sock shop," 2021. [Online]. Available: https://github.com/microservices-demo/microservices-demo.

[2] ITS, *Glossary*, 2001. [Online]. Available: https://web.archive.org/web/20070902151937/http://www.its.state.nc.us/Information/Glossary/Glossm.asp (visited on 03/27/2023).

[3] J. P. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pp. 62–65, 2017. DOI: 10.1109/ICSAW.2017.35.

[4] C. B. Weinstock and J. B. Goodenough, "On System Scalability," *U.S. Ministry of defence*, 2006. [Online]. Available: http://www.sei.cmu.edu/publications/pubweb.html.

[5] J. Schmidt, *USATODAY.com - Comair to replace old system that failed*, 2004. [Online]. Available: https://web.archive.org/web/20210126095521/https://usatoday30.usatoday.com/money/biztravel/2004-12-28-comair-usat{_}x.htm (visited on 07/04/2023).

[6] DOT, *Microsoft Word - Final2-28.doc | Enhanced Reader*, 2004. (visited on 07/14/2023).

[7] B. Curtis, "How Do You Measure Software Resilience?," [Online]. Available: www.it-cisq.org.

[8] S. McConnell, *"Managing technical debt (slides)" in Workshop On Managing technical debtt (part of ICSE 2013)*, 2013.

[9] M. Fowler, *Microservices*, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 01/31/2022).

[10] R. Jabbari, N. B. Ali, K. Petersen, and B. Tanveer, "What is DevOps? A systematic mapping study on definitions and practices," *ACM International Conference Proceeding Series*, 2016. DOI: 10.1145/2962695.2962707. [Online]. Available: https://dl.acm.org/doi/10.1145/2962695.2962707.

[11] Amazon, *What is DevOps? - DevOps Models Explained - Amazon Web Services (AWS)*. [Online]. Available: https://aws.amazon.com/devops/what-is-devops/ (visited on 07/17/2023).

[12] G. Samaras, "Two-Phase Commit," 2009. DOI: 10.1007/978-1-4899-7993-3_713-2. [Online]. Available: https://www.researchgate.net/publication/275155037.

[13] D. Infra, *Top 10 Cloud Service Providers Globally in 2023 - Dgtl Infra*. [Online]. Available: https://dgtlinfra.com/top-10-cloud-service-providers-2022/ (visited on 06/28/2023).

[14] F. Richter, *Cloud providers market share*, 2023. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

[15] Y. Gan, Y. Zhang, D. Cheng, *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 3–18, 2019. DOI: 10.1145/3297858.3304013.

[16] J. Bogatinovski, S. Nedelkoski, J. Cardoso, and O. Kao, "Self-supervised anomaly detection from distributed traces," *Proceedings - 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC 2020*, pp. 342–347, 2020. DOI: 10.1109/UCC48980.2020.00054.

[17] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," *Proceedings - 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, pp. 241–250, 2019. DOI: 10.1109/CCGRID.2019.00038.

[18] G. Zhou and M. Maas, "LEARNING ON DISTRIBUTED TRACES FOR DATA CENTER STORAGE SYSTEMS," 2021.

[19] Weaveworks, *Microservices Demo: Sock Shop*. [Online]. Available: https://microservices-demo.github.io/ (visited on 06/30/2023).

[20] Scikit-learn, *6.4. Imputation of missing values — scikit-learn 1.2.2 documentation*. [Online]. Available: https://scikit-learn.org/stable/modules/impute.html (visited on 04/13/2023).