UNIVERSITY OF OSLO

MASTER'S THESIS

Data driven performance modelling og microservice systems

Author:
Magnus Nordbø

Supervisor:
Hui Song
Ketil Stølen

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

November 15, 2023

Declaration of Authorship

I, Magnus NORDBØ, declare that this thesis titled, "Data driven performance modelling og 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

Data driven performance modelling og microservice systems

by Magnus NORDBØ

This thesis presents a system for generating, collecting and preprocessing stress testing data on a microservice system. The intial goal of the project was to analyze the performance of a private company's microservice platform, but due to unforeseen circumstances and a lack of communication from the company, there was a change in scope. The new project direction was to create a framework for data creation and analysis. A microservice system running locally on Docker was created. A program automates stress testing with semi-random, adjustable parameters and collects the data in organized files. A data preprocessing pipeline performs exensive trimming, imputation, scaling, sorting and selection. This digests the massive amount of data into a manageable and sorted dataset that can further be used for visualization, analysis or machine learning. The entire process is deconstructed, explained and justified piece by piece.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

1.1 Keywords

Observability
Microservices
Preprocessing
Visualization

1.2 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 creating a framework for creating, collecting, aggregating and preprocessing data driven model data about a microservice system, using only centralized logging. The underlying idea is that one might be able to model the behavior of the system under heavy load beforehand, by using a stress testing tool and aggregating and preprocessing the data and understanding the system's behavior before it happens. **Research question 1**: Using stress testing, centralized logging and preprocessing, is it possible to gain meaningful information about the system's bottlenecks before they happen in production? **Research question 2**: What are the current main challenges to effectively utilize metric data from microservice systems?

1.3 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. 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 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.

1.4 List of important terms

- **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**: A more in depth, but also more resource intensive method for gathering information about a microservice system's runtime behavior, where it tracks the entire path of invocations through the system.

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

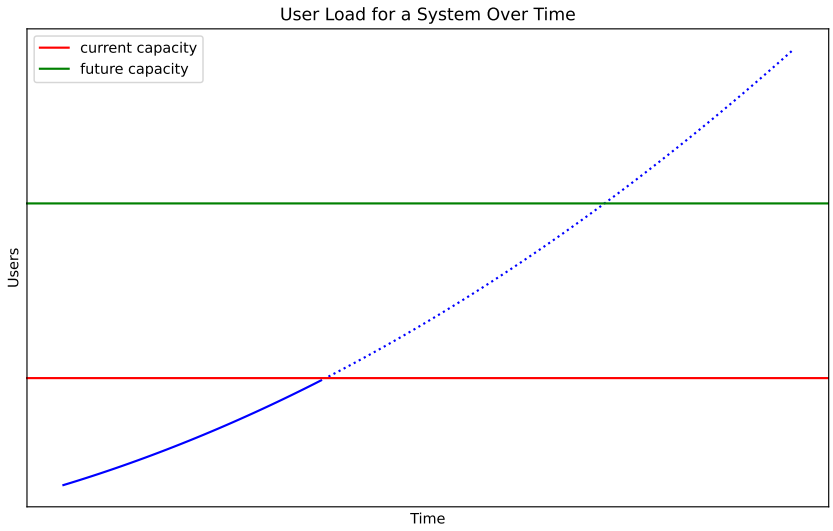

FIGURE 2.1: Scaling a monolith requires making assumptions about future growth

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 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.

2.2.1 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.

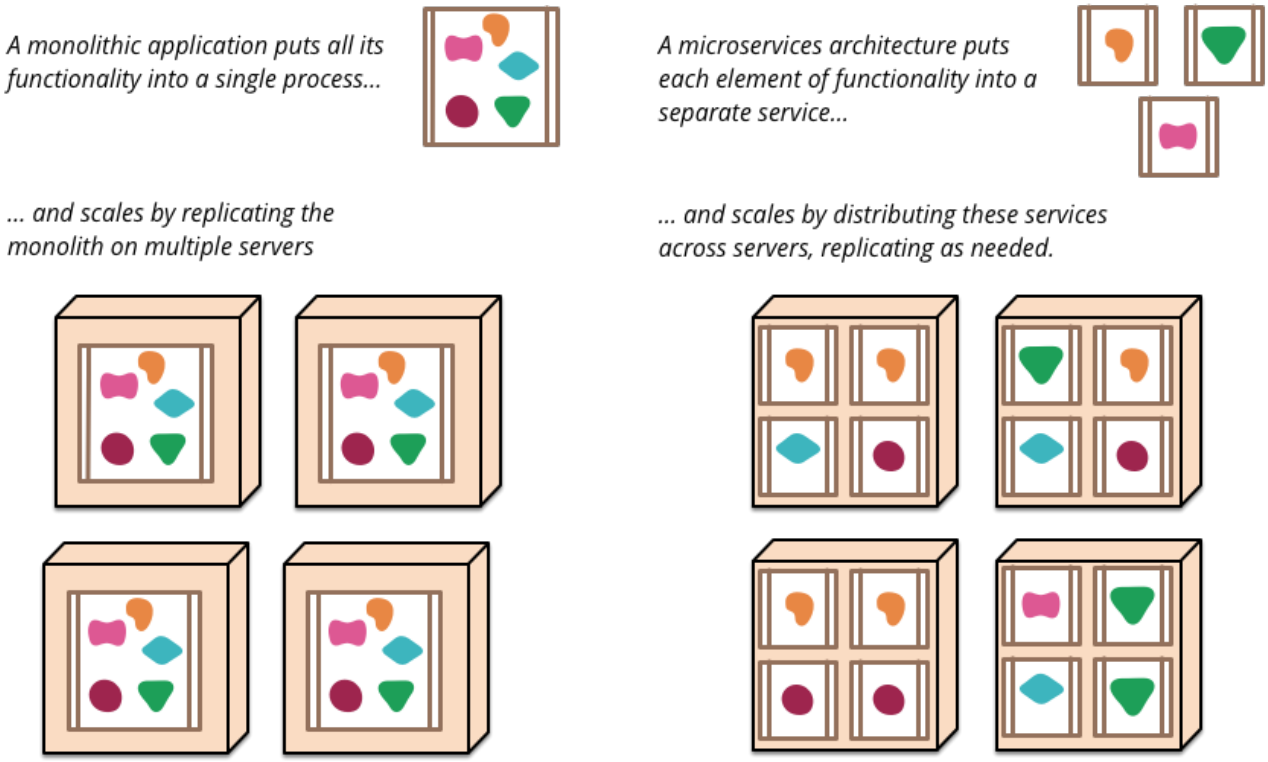

FIGURE 2.2: Illustrating how monoliths and microservices handle scaling. Credit: Martin Fowler [9]

2.2.2 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).

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.3 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, and are the dominant technology for implementing microservices architecture in modern systems. 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.

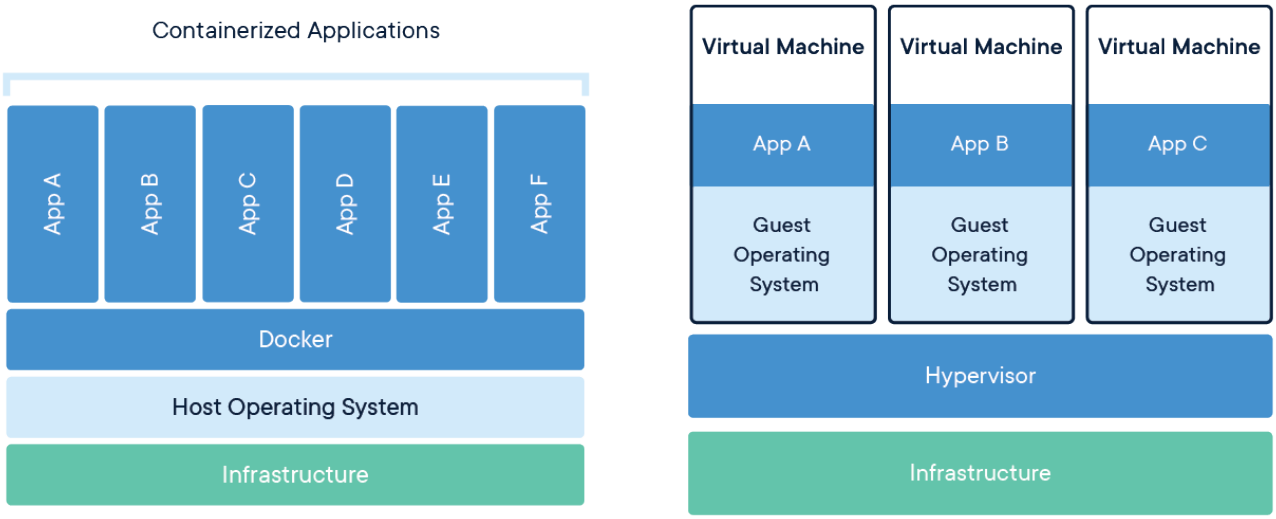

FIGURE 2.3: Containers vs VMs. Image credit: Docker inc

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 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.

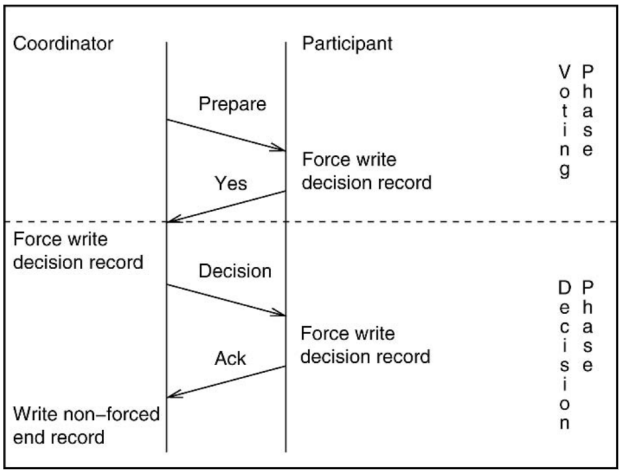

FIGURE 2.4: The two-phase commit protocol. Credit: [12]

Updating data can be a challenging affair. If, for example, there is a network of 100 microservices that need to be updated with 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].

2.4.3 Network issues

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. Network complexity and potential reliance on public internet and service providers exposes potential security threats and deferred responsibility, meaning parts of the system are out of the operator's control and can fail through no part of their own. If something like a partial network outage happens during operation, the complex nature of the system can make it extremely difficult to pinpoint exactly what went wrong where. It's extremely easy to make mistakes when orchestrating a microservices network. These mistakes can lead to a number of "impossible states":

- A message experiences an unexpected delay from one MS to another, leading to the message being recorded as not sent or missing before being received.
- Timezone issues or crashes/bugs in the sending or receiving MS or intermediate nodes cause a message to display as being received before it was sent.
- Messages being sent without knowledge of it being received: The classic two generals problem.
- Events that never successfully completed get stored as completed in the log
- Uncountable network and technology specific faults that can appear from the millions of possible states of a complex network.

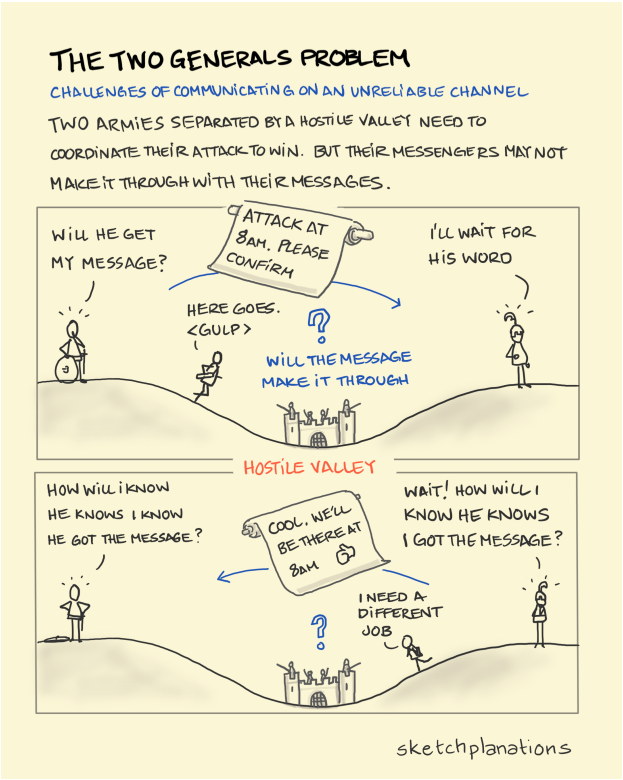

FIGURE 2.5: The two generals problem: The final sender of a message can never know that it has been received.

2.4.4 Testing and debugging

The theme of tradeoffs is consistent when talking about upsides and downsides of microservices. It is mentioned earlier that testing and developing individual parts of a system is much easier in a decoupled environment. The tradeoff, then, is that testing and debugging behaviour in interaction between the various parts, and how the system behaves as a whole, is much harder. Consider, for example, a system that hosts parts of the application in different countries. A problem occurs when these do parts interact. To get to the bottom of the situation, one would ideally set up a testing environment that mimics the system. But what might be a case of renting server capacity for hosting and running a local instance of an entire application with a monolith, now becomes a case of coordinating test environments across teams and borders. Issues with time zones, language, communication and even local laws can make this a daunting prospect. Setting up a locally hosted test environment that mimics the circumstances of the cross border interactions might work, but it might also fail to capture the intricacies that are causing the issue in the first place. In general, setting up local testing environments that mimic the faulty system is difficult and potentially expensive.

2.4.5 Performance

While microservices lead to many benefits in runtime, like fault tolerance and auto scaling, it comes with several negative performance implication performance as well. The distributed nature of

microservice applications leads to increased physical and virtual distances that need to be traversed. There is a direct resource overhead from running progams to run programs. It can be hard to measure performance bottlenecks.

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. Microservices achieve great robustness and ease of extension in part by relying on API requests between each node. This typically means that time is spent serializing and deserializing data in addition to sending it over an internet connection. 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.

Resource overhead

A traditional application runs on an operating system that runs on hardware. Microservices run on an operating system running on virtualized hardware running on an operating system that runs on hardware. There have been great improvements in optimization and resource usage regarding containers and their orchestrating systems, but there will always be this fundamental difference between virtualized and non-virtualized hardware. Containers have to allocate system resources to do their virtualization. In addition, all larger microservice system need an orchestation tool like Kubernetes to be manageable. A service discovery service like Consul or Netflix Eureka is likely needed to manage network complexity and offload some responsibilites of data validation and routing. To be able to debug a system and get an idea of what is going on, a monitoring tool like Prometheus needs to run and log system metrics. On top of that you typically want to visualize the data with Grafana or a similar tool, and implement distributed tracing to trace root causes using something like OpenTelemetry.

Technology plurality

One of the greatest benefits of microservices is that different teams working on a large project don't have to conform to the same software technology stacks and standards. All containers come with their required dependencies. This means that different teams or individuals can do their work using the technology stack and programming language that they are most comfortable with, or judge to be the best for their specific case. This is fantastic from a software project management and developemnt perspective. The tradeoff is that one has to make all these different technologies and languages cooperate to deliver the final product. This typically means serializing output and input to a common format like JSON and sending that over the network to be deserialized and used in another component. This adheres to the black-box principle of decoupled programming: Components should not have to know or care about the internal workings of other components. However, this adds more steps to the already cumbersome communication process. It plays a part in making these API calls slower than internal system calls used in monolithic applications.

Bottlenecking

The data that a microservices system returns from some task is the result of a chain of microservices doing their own little task and sending the result further down the line until it is ready. This can easily create situations where one single overloaded or buggy or underperforming microservice causes the entire rest of the chain to sit idle and waiting. This is bottlenecking, and can be nasty problem for microservice applications. It is supposed be solved by the orchestration service noticing that a particular service is underperforming and initializing new instances of that service. However, this is not a catch-all solution.

- What if each instance is similarly slow, and initializing new instances does not solve the problem?
- What if the cause of the bottleneck is wrongly reported because of some other problem, and the wrong service is initialized?
- What if there are not enough available resources to create more instances?
- How long is the orchestration software supposed to wait before concluding that something has gone wrong in the first place?

These many potential issues of performance is what led this project to focus on deepening the understanding of performance and performance bottlenecks and analysis.

Chapter 3

Methodology

The general idea for answering the research question will be to create a system for generating stress test data, collecting metrics from the stressing period, and treating that data to be able to inform the user about bottlenecks or other performance quirks.

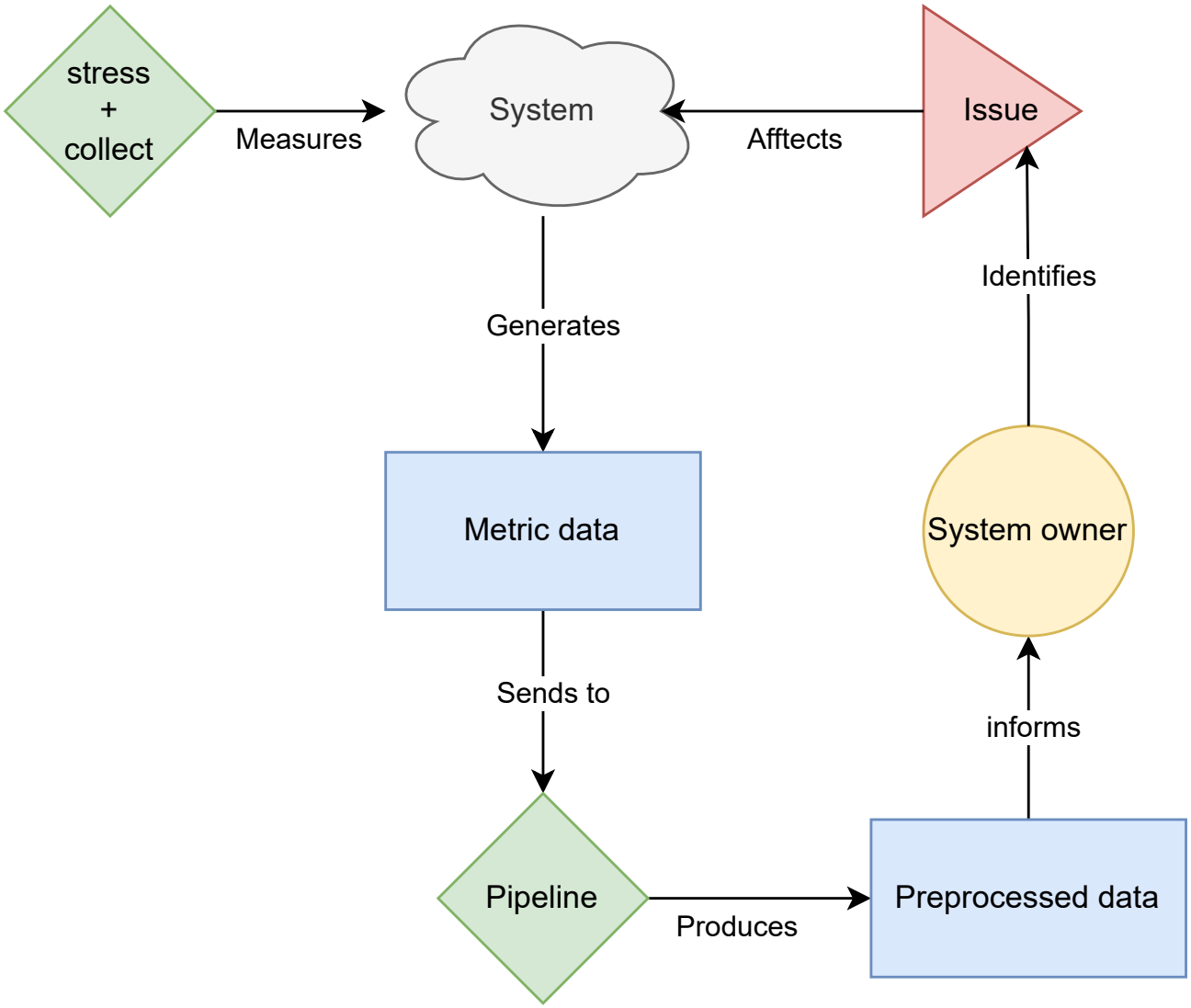

FIGURE 3.1: The proposed contribution to modelling and comprehension of performance issues in microservices: Contributions in green

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, difficulties arose with the company in question, and we were unable to get any data from them. We decided to go ahead with the initial goals of the project: To research a microservice system and try to make good use of collected data to say things about the system. 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.
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.

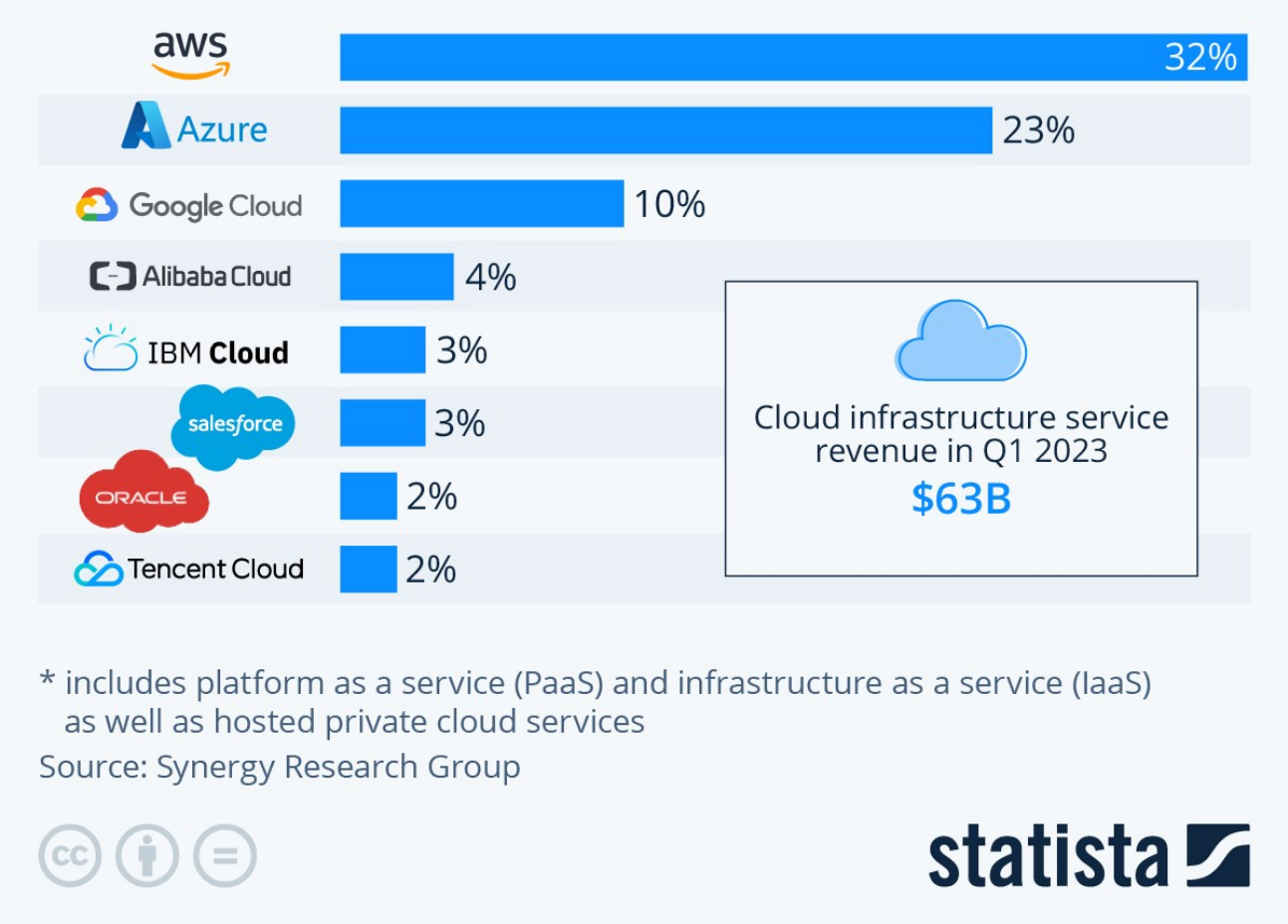

FIGURE 3.2: Worldwide market share of leading cloud infrastructure providers in Q1 2023. Source: Statista [14].

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.

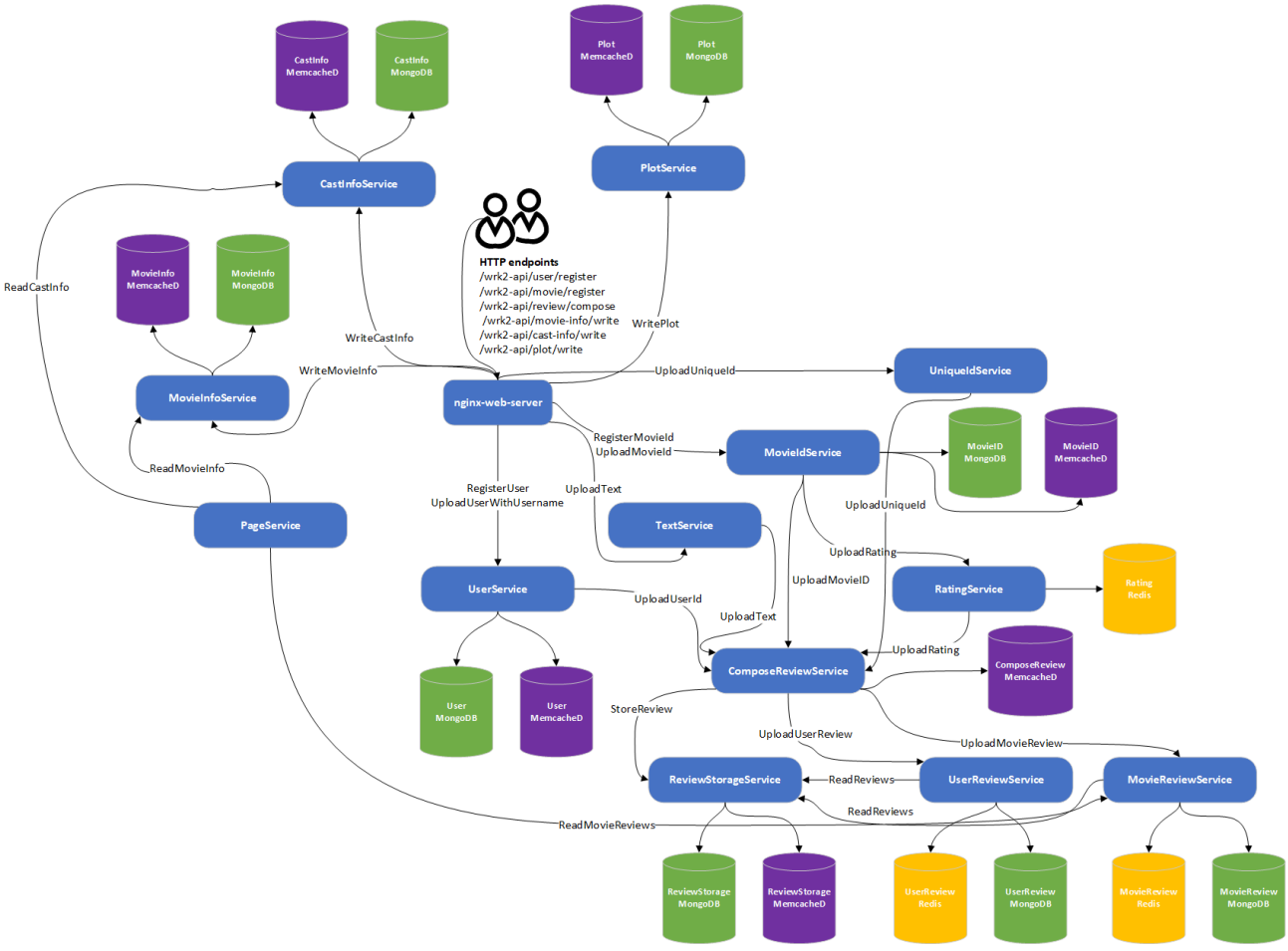

FIGURE 3.3: The workload architecture for DSB's media service [15].

- 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.

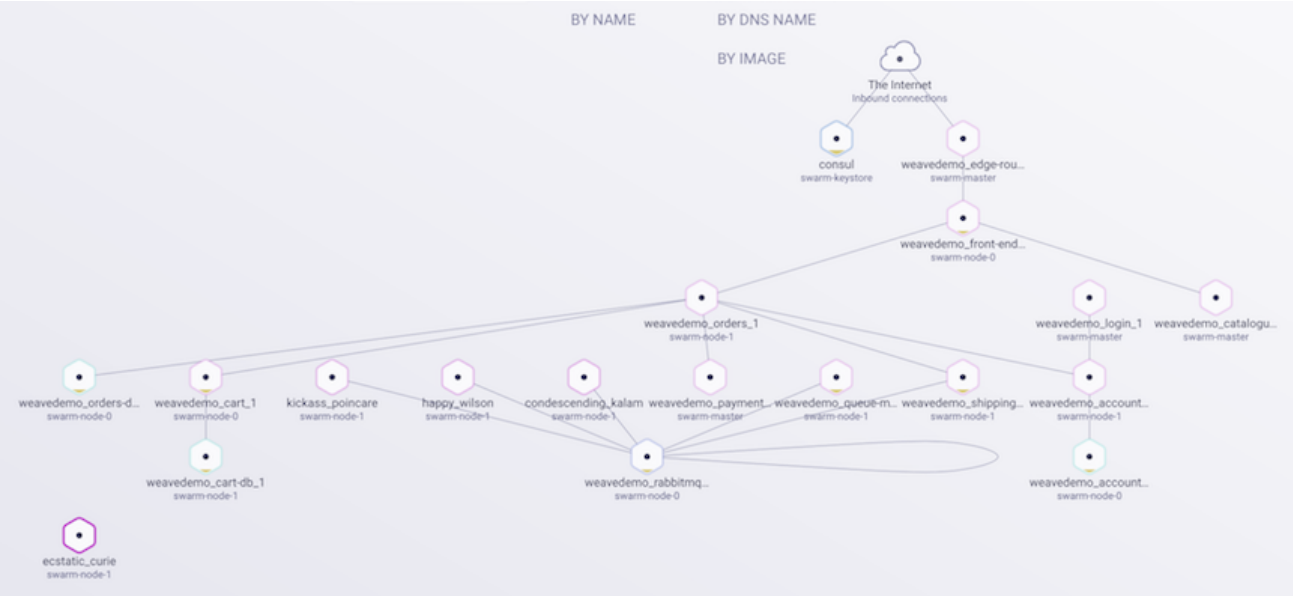

FIGURE 3.4: The architecture of the sock shop shown using WeaveScope, Weaveworks' own mapping tool. [19]

Sockshop comes with some strong benefits:

- Forgiving system requirements, enabling use of personal hardware instead of renting or borrowing
- Very thorough Prometheus instrumentation, with thousands of features to select and chose from
- Locust integration for stress testing out of the box

Of the strong candidates, this was the one most likely to not run into a hardware bottleneck on our computers. It also seemed to have a relatively easy installation process.

- 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
- Partly closed source
- Underlying issues with Docker implementation

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.
The project is also not fully open source. It implements packages hosted by WeaveWorks on the Docker hub, but does not make the source code for those elements public. As an addendum, it was later discovered that their implementation on Docker for some reason prevented Docker from assigning hardware bottlenecks on the system, making consistent stress testing and replication more difficult.

Spring petclinic

This is a project by the Spring petclinic community on Github [20]. 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 four main components. By "main components", I mean microservices that focus on delivering the service to the end user. As opposed to auxiliary components like monitoring and service discovery.

The sockshop from WeaveWorks was selected in the end on mainly pragmatic grounds: It guaranteed easy hosting on an available personal desktop computer. This way, there were no extra costs with renting a server to perform tests on.

Data generation

Data generation would consist of creating stress test data by overloading the running microservice system while collecting every available metric from Prometheus. There were many rounds of experimentation to achieve this. The basic workflow for incrementally approaching a good data generation setup went like this:

1. Start collecting system metrics
2. Run a Locust script with a given amount of user clients. The locust scripts would target the system in the following fashion:
    - Exclusively attack an HTTP API interface directly connected to a microservice, one such targeting per targetable microservice

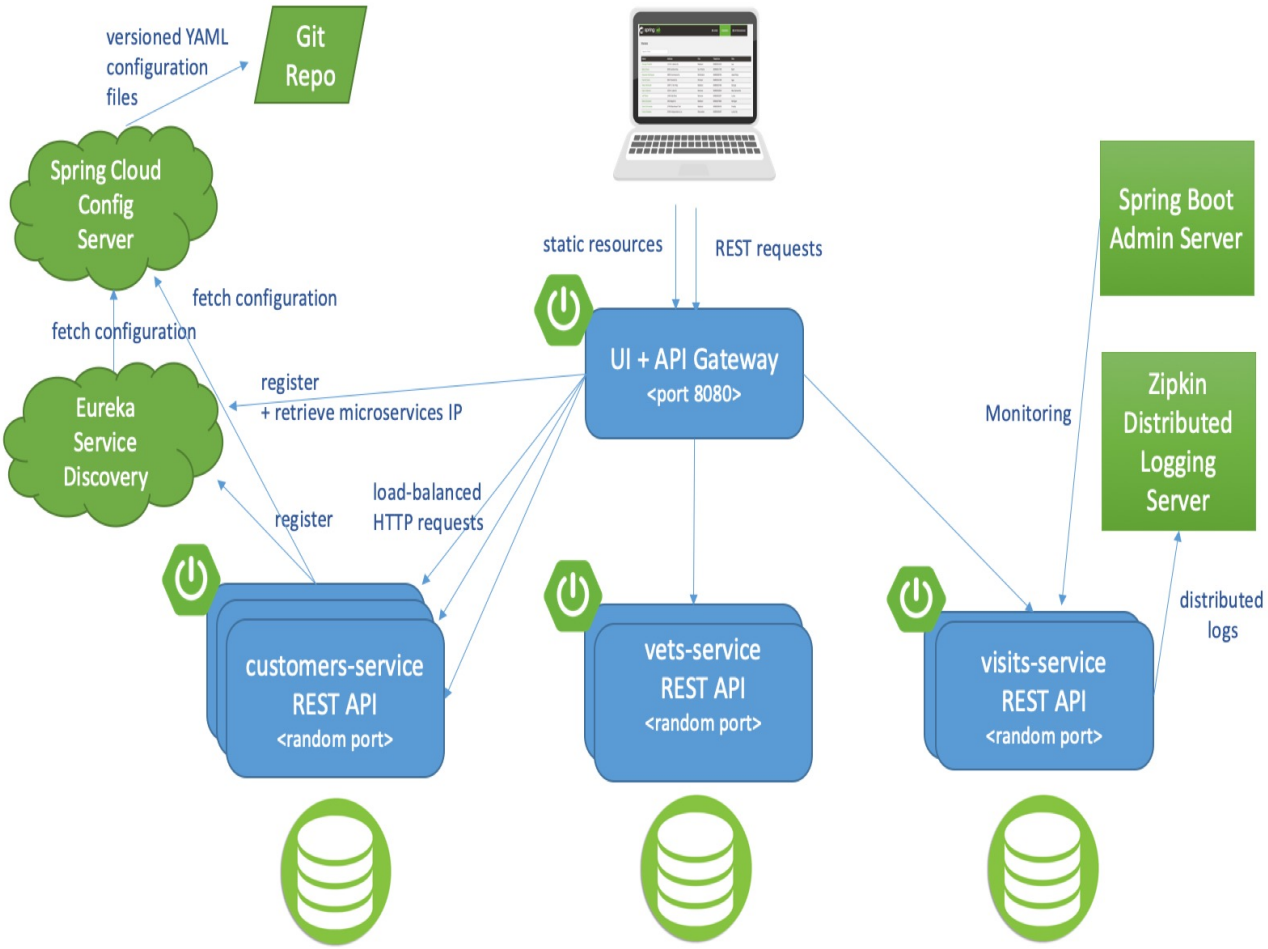

FIGURE 3.5: The architecture graph of the microservices spring petclinic

- Attack all services at once in an even pattern
- Simply collect data while not attacking the system at all, to serve as a control group

3. Shut down Locust script while still collecting metrics for a while after, to catch the tail of the pattern and capture how the system stabilizes after

4. Analyze output data, Docker metrics on running containers and local system resource data to inform change to Locust parameters and stressing method

5. Repeat

This process eventually created an automated script setup that pushed the hosting machine to its limits. The total amount of stress testing and collection done on the machine until the final set of metrics was ready amounted to well over 60 hours of runtime at full capacity. The end result was five labels across 178 files, with hundreds of thousands of data points. After that came the process of making sense of the data.

Data preprocessing

The field of advanced time series analysis is quite underdeveloped. As mentioned in [21], much less consideration has been given to multivariate time series analysis than univariate, which is easier to

work with and reason about. As such, there was need for experimentation and use of a framework made to fit the problem. The Python library *sktime* was therefore a natural fit [22]. It is built upon the much more common Scikit-learn, with extensions specifically for working with time series data. The method of choice was to work with the various tools in sktime's toolbox and turn the data from a voluminous mess into something usable for either a user or a machine learning algorithm. To achieve this goal, one would have to figure out how to properly preprocess the data into something viable for either option. The built-in data processing and preparation functions of Scikit-learn and sktime were used liberally for this purpose. The methodology of this part just consisted of having some real data from the stress testing, and trying to refine it to have high variance, low redundancy, and retain as much information as possible in a more accessible manner than what was received from the Prometheus cleint.

Visualization

The Python package Seaborn was used for visualization purposes after preprocessing to try to make sense of the resulting data. It is an extension for the matplotlib library, which is a visualization library for Python. Seaborn just provides some helper functions on top of matplotlib to make it easier to work with [23]. The idea is that with sufficient data generation and preprocessing, it should with some visualization be evident if the data is varied enough and carries enough information to be useful to machine learning algorithms. It should also be possible to glean some relevant information about the system simply by looking that the resulting graphs.

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 [24].
- 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

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 4.3 are scaled. This means the variance value for each feature is proportional to the mean value of that feature. In practice, this just means that the variance calculation algorithm does not discriminate based on the average value of the features. An algorithm that does not do this scaling would assign much higher variance scores to features with high average values. 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.
Cases do exist of useful zero variance features, but they are very domain specific. For example, an online retailer could have a variable in ther database that simply tracks if an item is in stock. Such a feature would be important to keep in mind even if it is static for the entire training duration. However, no such feature exists in this dataset. This data is more "blind", and tracks system information instead of business relevant information.

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 exaggerate the variance of these important datasets, for example by squaring all values.

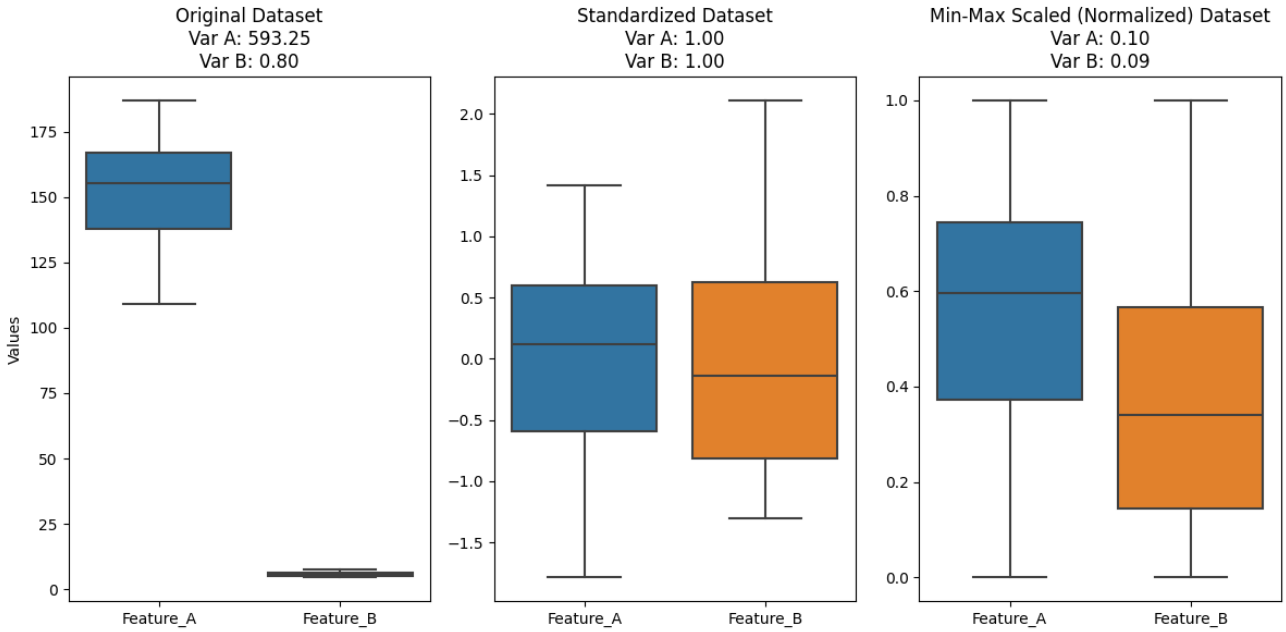

FIGURE 4.1: How standardization and normalization impacts variance in datasets of different scale

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 constantly growing. In the case for counters for example, that number might mostly just reflect how long the system has been running, or some other measurement of time. The dataset could then contain values that are consistently very low in the parts collected earlier, and consistently very high in the parts collected later in the same data collection workload. 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 and confuse the algorithm.

Here is a concrete example of the problem from the data: In one of the stress testing / collection runs, it ran for 12 minutes as usual while stress testing the "carts" endpoint. The collected data contains several monotonically increasing variables, including a variable called "go-memstats-mallocs-total". This Prometheus metric is from Google's Go instrumentation package, and shows

how many heap objects are allocated. It works as a counter, meaning it monotonically increases. Its value was 18387739 in the first instance of collection. 12 minutes later its value was 19284815. An unprepared machine learning algorithm will see meaningful change in this number. Some algorithms will also just see that it has a pretty high value, and assign importance to it. In reality, however, the correlation between system load and heap allocation is not well documented and understood in this system. More likely than not, it is as much a measure of time passing as it is a measure of system load.

Monotonically increasing variables can still be very useful in machine learning if treated properly. In many cases, these variables measure something that only increases because of some factor that is relevant to the analysis at hand. For these variables, it is often sufficient to measure the change in the value. This change from static value to a measure of difference is a generally strong preprocessing tool, as it often can provide more meaning to algorithms 4.2. However, using monotonic variables requires both good domain knowledge and good knowledge of how to transform the vairables into a format that can provide value for training. Some monotonic features do not provide good information even when differenced, and that differencing can make them more noisy. As is demonstrated in 4.2, the resulting data will often seem significant and valuable. If this is still not good data, then this will confuse the machine learning algorithm.
An alternative way for it to become noisy is if it is constant. In that case, differencing will turn it into a zero variance feature. As discussed earlier, this is usually noisy. Therefore, when performing differencing and potentially mean subtraction, one should calculate variance after, and remove zero variance results.

```
1
2  def select_by_variance(df:pd.DataFrame, amount:int):
3      """
4      Calculates variances, and returns the highest variance features
5      Params:
6      df: dataframe with time series data
7      amout: the amount of features to extract
8      """
9      variances:pd.Series = calculate_variance(df)
10     selection = variances.iloc[0:amount]
11     return selection.index
12 best_features = select_by_variance(trimmed_df, 5)
13 X = scaled[best_features]
```

Irrelevant features

Complex microservice systems are inherently very noisy. Even after removing low variance features, performing imputation of missing values, differencing monotonic featues and performing dimensionality reduction, noisy features will remain. The best tool for dealing with irrelevant features is domain knowledge, and intimate understanding of the data. This cannot always be expected when dealing with complex software. In the quite minimal example that is the dataset generated for this thesis, there are still 32 variables with 23674 datapoints for each of them. Beyond knowledge and mastery of the specific dataset, the best idea is to hope the data is mainly relevant, and irrelevant features constitute little enough noise that it gets filtered out. In theory, if the dataset contains features with solid correlations to the ground truth, they will override the noise from irrelevant features.

Non-ubiquitous features

There are cases where not all the features in a dataset appear in every instance. This will cause issues for several types of algorithms one might want to run on the data, like regression tasks, neural nets or kNN algorithms. There are several ways of dealing with this issue. The simplest of all is to remove all non-ubiquitous features altogether. This requires little processing and no guessing. It is only

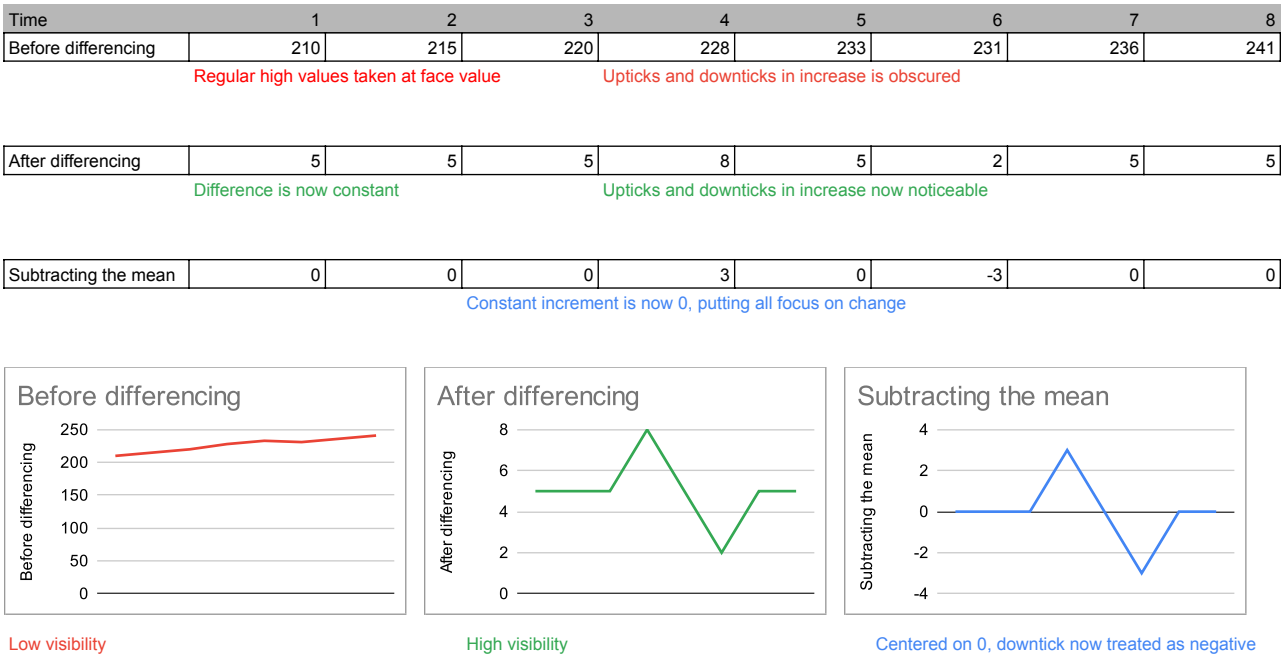

FIGURE 4.2: How differencing places emphasis on changes in increment

recommended on datasets with an abundance of features that will likely retain large amounts of information even after several features are removed. If removing features is ill advised, or the features are only missing from a small percantage of the total amount of instances, it is possible to perform imputation of the missing features instead. Mean or median imputation on numerical data or mode imputation on categorical data will be able to fill in missing features at the cost of some uncertainty. This generated uncertainty can be mitigated by performing dimensionality reduction on the imputed data. The idea is that by creating new, condensed features out of the old ones, relevant information will be brought forward and less relevant information will be obscured. Imputed features typically carry little meaningful information, since they by design try not to say too much about the data. In this project, Principal Component Analysis (PCA) is used to perform dimensionality reduction.

There are several methods of dimensionality reduction. PCA is one of the simplest and least computationally expensive ones, as well as very common and well understood algorithm. The drawback of PCA is mainly that it is a linear transformation: It will always optimize for linear vector representations of the higher-dimension input data, because it uses computed eigenvalues to optimize an eigenvector to represent information in the data. This can be a drawback if one is working with data whose information can not easily be represented by a linear vector. This project still uses PCA because the data is theroretically linear in nature, being a standardized representation of the linear flow of time in a mostly static system. A future improvement to the project would be to compare various means of dimensionality reduction, but it was given a low priority in this project and not included in the scope.

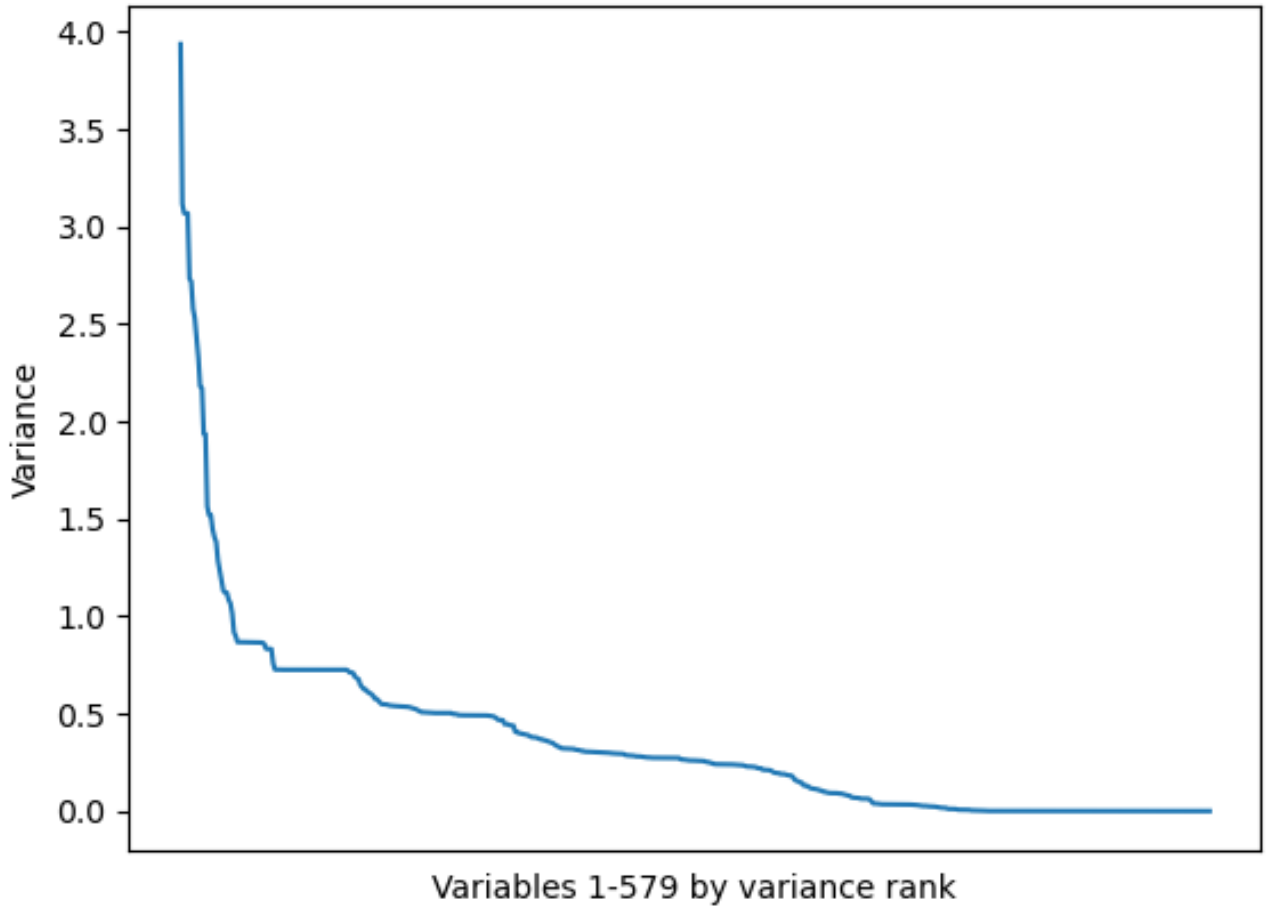

FIGURE 4.3: Scaled variance of all the variables

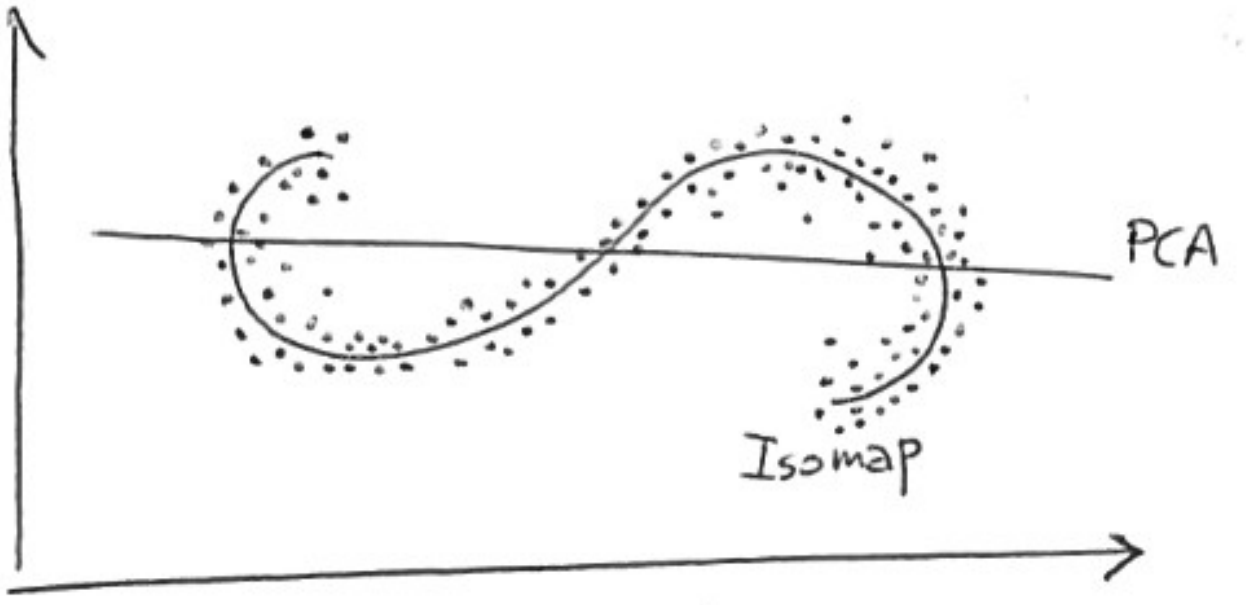

FIGURE 4.4: Demonstrating how the linearity of PCA makes it bad at approximating non-linear data. Here compared to isomap, a non-linear dimensionality reduction algorithm. Credit: https://stats.stackexchange.com/a/124545

Collection and cleaning

Stressing

This chapter will detail the step-by-step process of generating, collecting and cleaning data.

The first issue is to generate the data to be studied. The sockshop service is instrumented with 566 Prometheus metrics. This means that each time data is collected, it can collect 566 variables for each time point. The first part of the program is the Locust stress testing framework.

Locust is used as a Python package that can be simply installed with pip. The exact behavior of a Locust program, like what data to set to what endpoints, is defined in a python file that imports the Locust class, called a locustfile. The locustfile is then initiated through the command line. A custom program written in Python generates semi-random parameters for the locustfile. The parameters are: Amount of simulated users, between 10 000 and 12 000. Spawn rate of users, meaning how many new users per second are instantiated, between 30 and 60. These numbers are a result of experimentation. The goal was to find an amount of users and spawn rate that put the system under considerable stress without crashing the computer entirely. These numbers depend entirely on the capabilities of the system it was run on. The system specs for this round of testing was as follows:

1. Windows 10 Home edition
2. 32 GB 3200MHz DDR4 memory
3. AMD Ryzen 7 5800X CPU
4. Palit GeForce GTX 1080 GameRock Premium

The program that automates stress testing cycles through each individual locust task once, each locust task stressing one endpoint on the microservices system. Between each run (a run meaning the stress testing of one endpoint, corresponding to one tag), there is a 15-second delay to let the system cool down after stressing.

Let's first look at the function buildCommand, which returns a command line string that initializes Locust.

```
def buildCommand(users, spawnrate, runtime, tags, fileLocation) -> str:
    """
    Build locust command

    :param str/int users: amount of users
    :param str/int spawnrate: How fast the threads are spawned
    :param int runtime: How long it runs, in minutes
    :param str tags: space separated string of tags to execute
    :param str fileLocation: path to directory with the relevant locustfile
    """
    base = "docker run -p 8089:8089 --rm --name {} --mount type=bind,source={},target
    =/mnt/locust locustio/locust -f /mnt/locust/locustfile_complete.py --headless -u
    {} -r {} --run-time {}m --host http://host.docker.internal --tags {}"
    #name =  str(users) + str(spawnrate) + str(runtime) + str(tags)
    name = buildName(users,spawnrate, runtime)
    return base.format(name, fileLocation, users, spawnrate, str(runtime), tags)

def buildName(users, spawnrate, runtime):
    return  "" + str(users) + "U_" + str(spawnrate) + "R_" + str(runtime)
```

These functions are used in the following setloop_random function, which selects the random parameters to be used in a stress testing and collection run.

```
def setloop_random(tags_and_amounts, runtime):
    now = datetime.now()
    now_unix = time.mktime(now.timetuple()) #Making a prometheus compatible timestamp
```

```
4
5       for tag in tags_and_amounts["tags"]: #tags are references to locustfile tasks,
        that stress a given endpoint corresponding to that tag.
6           amount = random.randint(tags_and_amounts["lower_bound_users"],
        tags_and_amounts["upper_bound_users"]) #amount is number of users, between 10000
        and 12000
7
8           command = buildCommand(amount, DEFAULT_SPAWNRATE, runtime, tag,
        LOCUSTFILE_COMPLETE_LOCATION)
9           print("Running Locust with tag " + tag + " with " + str(amount) + " users")
10          print("command: ", command)
11          result = cmd(command)
12          if result.stderr:
13              with open("Print_output.txt","a") as outputfile:
14                  outputfile.write(str(result.stderr))
15                  outputfile.close()
16
17          time.sleep(15)
18          collection.collection(RUN_TIME+1, DEFAULT_STEP, "./metrics.json", "rand " +
        str(now_unix) + "_" + str(amount),tag, now_unix)
```

LISTING 4.1: The function that generates a locust command for semi-random stressing. This function generates one set of metrics for each stressing type

After each stress job on a tag/endpoint, the program runs the collection module to store the system data that was just generated. The collection module is a Python program that collects and sorts the data into separate folders and files. It works by pinging the locally running Prometheus Docker service that comes with the sockshop service. The stressing period lasts 10 minutes, and the collection module collects 11 minutes of system data. This is to catch the head and tail of the testing period with the state of the system both before and after stressing.

```
1  def collection(time, step, source, filename, tag, now_unix):
2      """
3      time: time in minutes
4      step: query interval within the timeframe in seconds
5      source: json file with the metrics to collect
6      filename: created csv file will have this name
7      tag: locust tags to execute
8      """
9      file = open(source)
10     metrics = json.load(file)['data']
11
12     #quick'n dirty bool to add timestamps to only first iteration
13     first = True
14     for metric in metrics:
15         response = requests.get(
16             'http://localhost:9090/api/v1/query_range',
17             params={
18                 'query': metric,
19                 'start': now_unix - 60 * time,
20                 "end": now_unix,
21                 'step': step
22             }
23         )
24
25
26         results = response.json()['data']['result']
27
28
29
30         labelnames = set()
31         for result in results:
32             labelnames.update(result['metric'].keys())
```

```
        labelnames = sorted(labelnames)

        with open("generated_csvs_4/" + str(tag) + "/" + str(filename) + ".csv", mode
    = "a") as f:
            writer = csv.writer(f, lineterminator='\n')
            if(first):
                first = False
                timestamps = []
                for x in results[0]['values']:
                    timestamps.append(x[0])
                writer.writerow(["identifier"] + timestamps)

            for result in results:
                vals = [x[1] for x in result['values']]
                identifier_string = ""
                for label in labelnames:
                    identifier_string += result['metric'].get(label, '') + '&'

                writer.writerow([identifier_string[:-1]] + vals)
```

This collection module collects each instance in a file, in a folder structure corresponding to instances. Each feature, corresponding to a Prometheus metric, is annotated with a string containing all identifying information separated by *&*. This information is in x parts: The name of the metric itself, the name of the instrumentation providing it (if this name is provided), other sub-tags, the microservice that reported the metric, and which port the metric's API was running on. It is collected in the following folder structure:

```
    @tag('even_load')
    @task
    def even_load(self):

    self.client.get("/index.html").close()
    self.client.get("/category.html").close()
    self.client.get("/category.html?tags=" + random.choice(filterList)).close()
    self.client.get("/detail.html?id=" + random.choice(productList)).close()
    for _ in range(random.choice([1,2,3,4,5,6,7,8,9])):
    self.client.post("/cart", json={"id": random.choice(productList)}).close()
    self.client.get("/basket.html").close()
    self.client.get("/orders").close()
    self.client.get("/customer-orders.html").close()
    self.client.get("/index.html").close()

    def on_start(self):
    self.client.get("/login", headers={"Authorization":"Basic bG9jdXN0OmxvY3VzdA=="})

    def on_stop(self):
    self.client.close()
```

LISTING 4.2: The "even_load" tag, which tells Locust to attack every available http endpoint. One of the five tags used to generate stress data.

–

This entire process is automated through a PowerShell script. The script does 6 things:

1. Initialize Docker
2. Wait 3 minutes for the sockshop services to be up and running
3. Start the Prometheus container
4. Run the stressing and collection program
5. Close Docker

6. Repeat

It has to close and reopen Docker because of crashes or errors that eventually occur after multiple rounds of stressing and collection. The crashes seem to occur from some memory leaks somewhere, and the local network can also get congested with thousands of unclosed connections from the Locust stress testing. The locustfile containing the stress scripts implement attempts at manual cleanup, but some connections always seem to leak through.

After sufficient experimentation and several test runs, the final version of the program was left running for approximately 48 hours. The end result was a total of around 250 CSV files, but several of them were incomplete due to bugs or partial crashes. After removing incomplete or corrupted files, what remained was 178 CSV files, each with 120 time points and around 600 columns. The amount of columns, corresponding to Prometheus metrics, is not consistent across all the files. This can also be assumed to be a result of the stressing of the system interrupting or corrupting some collection or storage process. The final collection of files has a total of 13707246 values from 23674 rows x 579 columns. Not all of these variables have a valid value, and register as NaN. They are removed in the Preprocessing stage. While this might seem like a large amount of data, most of it will disappear or be reduced in preprocessing.

Preprocessing

The data collected from the system is initially of very low quality. The data collection program makes no distinction between the various metric it collects, and most of these metrics end up having little to say about the system. Most obvious is the overall variance of the dataset: As seen in 4.3, most of the data has such low variance that it is impossible to gain any useful information from it. The data is also populated by NaNs that need to be dealt with: If the affected feature has a low percentage of NaNs, one can impute the missing values. If not, the entire feature must be removed. This step can be skipped if one knows they will either only look at the data visually, or are going to use algorithms that tolerate missing values.

The first part of the preprocessing code retrieves the stored CSV data and transforms it into a pandas DataFrame. Pandas is a data analysis and manipulation library for Python, and the DataFrame is a way of representing complex data in a readable format with helper functions for formatting and manipulation. It is widely used in data science and analysis and enjoys extensive support in data science libraries like Scikit-learn and sktime. The timestamps of the collected data were stored as the first row of the CSV file in UNIX format. The names of the Prometheus metrics were stored as the first column. These together form the index and column names of the final dataframe product.

```
    def _readcsv_modified(csv_loc:str) ->pd.DataFrame:
        csv:pd.DataFrame
        try:
            csv = pd.read_csv(csv_loc, skip_blank_lines=True)
        except pd.errors.ParserError:
            return
        except pd.errors.EmptyDataError:
            print(csv_loc)
            return
        metrics = csv["identifier"].to_list()
        timestamps = csv.columns[1:].to_flat_index()
        timestamps = timestamps.to_numpy().tolist()
        timestamps = _make_datetime_index(timestamps)
        vals = csv.drop(labels="identifier",axis=1).to_numpy().transpose()
        s = pd.DataFrame(vals, index=timestamps, columns=metrics)
        return s
```

LISTING 4.3: Loading a csv file as a dataframe in memory

Perceptive readers will notice this loads a single dataframe. The function is called individually on each CSV file using a list of file locations. The next step is to remove monotonically increasing features. As mentioned in 4.3.3, there exist possible ways of gaining useful information from them. However, it was out of scope for this project.

During parts of the stressing and collection phase, the system is unable to gather data from the system entirely. This creates pockets of missing data, in the form of NaNs. While imputation can solve a lot of those issues, there are cases where the majority or a significant minority of the values is missing from a feature column in the data. In such datasets, there is a proportionally higher risk of imputation creating misleading imputed data from an insufficient information base. Therefore, one of the first steps performed in preprocessing is to remove feature columns with percentage of NaNs over a threshold. This prevents them from poisoning future preprocessing steps that make assumptions about the validity of the data.

```
def reduce_NaNs(df:pd.DataFrame, treshold:int):
    #Calculate the percent of NaNs in each columns (corresponds to feature in the
    time series)
    length = len(df)
    nan_counts = df.isna().sum()
    nan_percentages = (nan_counts / length) * 100
    #Put it neatly into a dataframe
    percentdf = pd.DataFrame({'Column':nan_percentages.index,'nan_percent':
    nan_percentages.values })
    #Create a boolean mask that corresponds to NaN percent being under the treshold (
    True=good)
    removal_mask = percentdf["nan_percent"] < treshold
    #trim the dataframe to only include only truthy values
    percentdf = percentdf.loc[removal_mask]
    #Get the names of the columns that pass
    okay_cols = percentdf["Column"]
    #Return a trimmed dataframe with only the listed columns
    return df[okay_cols]
```

LISTING 4.4: The code that removes columns(features) with a given percentage of NaNs

This code also runs on one dataframe at a time. There is some room for discussion here on whether this step could be implemented on a cross-instance level. Cross-instance level would mean to merge every dataframe first, and then compare them after. The upside would be to most likely preserve more data: Columns with missing values in parts of collection could very well be only missing in parts of the data, and fully present at other phases of collection. Under that assumption, the future removal of non-ubiquitous datasets could facilitate the loss of valuable data. For example, if a given feature ran into some problems with its instrumentation code during stress testing because of resource strain in only one part of the code, the removal of non-ubiquitous features would lead to near-complete datasets being removed entirely when imputation would suffice. It was implemented this way anyway, because of the setup of the stressing algorithm: Locust tags corresponding to class labels are processed in batches, so the data on one instance is likely to mirror the next. This creates a potential problem where missing data issues are localized to one class label, which would then get its data imputed based on information from the corresponding columns in another label. This would potentially poison the data by making it more uniform for little gain. The judgement was made that there was enough data already present to worry too much about making as much of every column as possible, and instead focus on what can be easily worked with. A future project with less data to work with would have to work in some solution to implement imputation only within the same class label or some other solution.

The program also removes monotonically decreasing features. This is not strictly required, as there are no monotonically decreasing features in the dataset. It was added for the sake of completeness;

monotonically decreasing features create the same problems as increasing ones. It uses the functions supplied by Pandas to create a list of features to remove and then drops them from the dataset. This can create issues of its own: A feature can be monotonic in one instance and non-monotonic in another. Features would then be present in some instances and missing in others, potentially causing breaking issues for algorithms.

```
def _remove_monotonically_increasing_rows(df_list:list) -> list:
    x:pd.DataFrame
    returnlist = []
    for x in df_list:
        dropme = []
        for column in x.columns:
            if x[column].is_monotonic_increasing or x[column].is_monotonic_decreasing
    :
                dropme.append(column)
        returnlist.append(x.drop(dropme,axis=1))
    return returnlist
```

LISTING 4.5: The code for removing monotonic features from the dataset.

Non-ubiquitous features would be a potential problem either way, as this dataset already columns that do not appear in every single instance. The simplest solution to such features is to simply remove every non-ubiquitous feature from the dataset entirely. While this can indeed solve the problem, it risks loss of data when for example only a small subset of the instances lack a certain column. One can instead impute the missing column values using something like a kNN median or mean imputer. This is explored more in 4.3.3. How this step of preprocessing data is done can depend on what one uses the data for. The program allows for a high degree of choice in how one chooses to preprocess the data: If one wants to remove non-ubiquitous features, removing monotonic features, removing columns with NaNs over a given threshold, whether to fill missing data points with imputed values.

```
def remove_non_ubiquitous(frames:list) -> list:
    template:pd.DataFrame = frames[0]
    #Round 1: Reduces the first frame to be the smallest size to match with everyone
    for frame in frames[1:]:
        common_columns = template.columns.intersection(frame.columns)
        template = template.reindex(columns=common_columns)
    returnlist = [template]
    #Round 2: Reduces all the other ones to the same minimum
    for frame in frames[1:]:
        common_columns = template.columns.intersection(frame.columns)
        returnlist.append(frame.reindex(columns=common_columns))
    return returnlist
```

The code works by using the Pandas intersection() function to build a "template" dataframe with only columns found in all dataframes. Reminder that in this part of the code, the input of the function is a list of dataframes, each dataframe representing an instance. After constructing the minimal template, it reduces every other dataframe to only those features present in the template. This is much less computationally expensive than imputation or dimensionality reduction, as can be demonstrated by the following graph:

Remove NU features	Impute NaNs	PCA(10)	Processing time (s)	Notes
Yes	No	No	7.5	Incomplete dataset (NaNs present)
Yes	No	No	10.5	32 Features in final dataset
Yes	Yes	Yes	9.9	123
No	Yes	Yes	120.4	Imputation on large dataset is expensive
No	Yes	No	109.2	69 features in final dataset. Many incomplete.

Note: The option of only doing PCA is not included, because PCA does not compute on missing values. The computed datasets are available on the GitHub repository.

The final steps are to optionally scale and standardize the data, and potentially perform dimensionality reduction with PCA to concentrate the information in the data. These two steps are quite trivial with the supplied preprocessing tools from the Scikit-learn libary:

```
def scaledf(df:pd.DataFrame) -> pd.DataFrame:
    scaler = StandardScaler()
    return pd.DataFrame(scaler.fit_transform(df), index=df.index, columns= df.columns
    )

def perform_pca(df:pd.DataFrame, pca_components) -> pd.DataFrame:
    pca_scaler = PCA(n_components = pca_components)
    scaled = pca_scaler.fit_transform(df)
    return pd.DataFrame(scaled, index=df.index)
```

The scaler is the Scikit-learn StandardScaler, which performs z-score standardization on the dataset feature-wise. For each feature, it calculates the mean μ and the standard deviation σ. Each feature is then transformed using this formula: $z = \frac{x-\mu}{\sigma}$. This scales all features such that the mean of all features is zero.
The PCA algorithm is similarly simple to implement. The one thing to be mindful about it is that one should tune the data to an appropriate dimensionality. This depends entirely on the data and the task at hand.

Finally, all these functions are collected in a wrapper function that accepts arguments deciding the exact nature of preprocessing, like how many dimensions PCA should aim for, if any, or whether to remove monotonic features etc.

```
def readcsvs(csv_loc_list:list, remove_monotonic_increasing=True,
    reduce_NaNs_treshold=50, remove_unique_cols=True, scale=True, pca_components=0,
    impute=False):
    individual_dataframes = []
    for i in range(len(csv_loc_list)):
        df = _readcsv_modified(csv_loc_list[i])
        #The read_csv pandas function sometimes fails when something went wrong with
    the metric collection from prometheus. These are simply skipped without throwing
    errors
        if df is not None:
            individual_dataframes.append(df) #time series
        else:
            print(i)

    if remove_monotonic_increasing:
        individual_dataframes = _remove_monotonically_increasing_rows(
    individual_dataframes)
    #Here: Go through the loop once again, start trimming. compare everything to
    element at 0, trim with it so it stays as the leanest version.

    if remove_unique_cols:
        individual_dataframes = remove_non_ubiquitous(individual_dataframes)

    concated = convert_to(individual_dataframes,to_type='pd-multiindex' )
    #NaNs can appear when concating dfs so this is done after the conversion to
    multiindex df
    if reduce_NaNs_treshold:
        concated = reduce_NaNs(concated, reduce_NaNs_treshold)

    if impute:
        concated = impute_nans(concated)

```

```
    if scale:
        concated = scaledf(concated)

    if pca_components != 0:
        concated = perform_pca(concated, pca_components)

    return concated

    return concated
```

This will build a semi-preprocessed file, with options for the user about what sort of preprocessing steps they want to apply to the data.

Selection

Even after preprocessing, it can be difficult to make sense of the reduced dataset. There are still many features and data points. There is a need for selection of the most useful and telling data. The simplest and most effective measure of data usefulness is variance: It measures how much change there is in the feature. Features with high variance are much more likely to carry useful information for both observation and training models. The first part of the selection and visualization process is therefore calculating the variance of all the features and selecting the top ones.

```
def calculate_variance(df:pd.DataFrame) ->pd.Series:
    #The coefficient of variation
    cv = df.std() / df.mean()
    return cv.round(10).sort_values(ascending=False)
def select_by_variance(df:pd.DataFrame, amount:int):
    variances:pd.Series = calculate_variance(df)
    selection = variances.iloc[0:amount]
    return selection
```

The code calculates the dataframe's standard deviation divided by its mean. It divides by the mean to take into account poorly scaled data. While this is not typically necessary on the already scaled data that most likely returns from the preprocessing function, it is included for standardization purposes as it covers all possible formats that can return from preprocessing. It rounds to maximum 10 decimal places to deal both with overly specific values and problems arising from binary representation of numbers (1.000000000003 instead of 1 and so forth). The second function sorts the features of the dataframe by variance in descending order, and returns a given slice of the most varied ones. This way, one can reliably look at only data that has a higher chance of carrying relevant information about the system.

Annotation

Usually, the labels are stored separately from the data to make sure that included labels do not inform any models being trained on it. For visualization, however, it is pertinent to know which part of the testing process the data actually comes from. The next functions therefore work to simplify and bring together the data into an annotated dataframe that can easily compare data with different labels.

```
def annotate_label(df, labels):
    df["label"] = 0
    for idx, row in fivebest.iterrows():
        indexvalue= idx[0]
        df.at[idx, "label"] = labels[indexvalue]
```

The first step in the process is to assign every time point in the dataset a label corresponding to its instance. It initializes an empty column with zero values, then populates that column for every

time point for every instance. After this begins the process of visualization of the data to gain a better understanding of the underlying system. This process also makes it simpler to feed the data directly into graphing packages: This project uses Seaborn, which is a Python package built on top of Matplotlib [23]. To achieve this ease, the specific timestamps are removed in favor of generic index increments from 0. This is done because the precise timestamps in the data are unique, so they would not get compared directly on the same x-axis otherwise.

```
def reindex_timepoints(df):
    grouped_by_instances = df.groupby("instances")
    level0 = df.index.get_level_values(0)
    level1 = grouped_by_instances.cumcount().values
    newindex = pd.MultiIndex.from_arrays([level0, level1], names=["instances", "
    timepoints"])
    df.index = newindex
    return df
```

The final step for making it easier to visualize the behavior of the system over time is to collapse all column values for each instance into their mean. One could potentially also use their median to lose some overall precision but be more resistant to outliers. The code works by grouping every label in the dataset together. After getting the generic time points from the previous function (on each label individually to preserve data independence), it calculates the mean of each column per time points across instances.

```
def meanvalues(names, df:pd.DataFrame):
    frames = []
    grouped = df.groupby("label")
    for name in names:
        newdf = grouped.get_group(name).drop("label",axis=1)
        reindexed = reindex_timepoints(newdf)
        means = reindexed.groupby("timepoints").mean()
        means["label"] = name
        frames.append(means)
    return frames
```

After this entire process is finished, the data will have gone from a massive, incomprehensible dataset of seemingly random values to a highly curated and simplified dataset that will hopefully be of great use to a potential end user trying to make sense of their system. The simplified data is well suited for visualizing the results of the process.

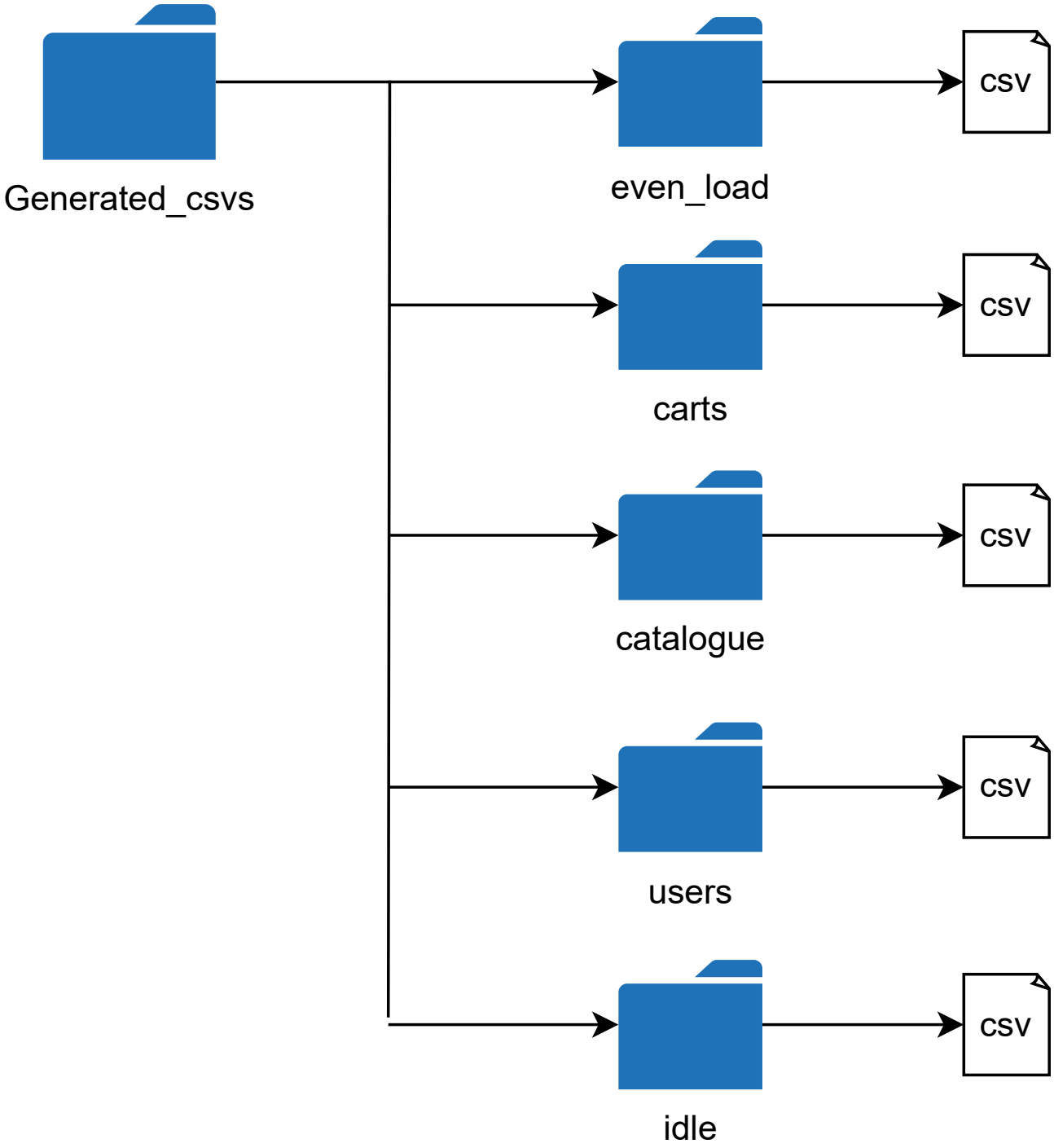

FIGURE 4.5: The resulting data structure

Results

The resulting output from the pipeline created in the project is able to visualize fluctuations in the system as it handles incoming loads. Surprisingly, it uncovered patterns that appear across testing parameters seemingly in the middle of the stressing process. The process went from having nothing to go on, to generating massive amounts of test data, to refining and cleaning the data in an automated pipeline into something that has much higher use for machine learning algorithms or simply observing the behavior of the system. During the research process, several possible necessary preprocessing steps were uncovered and provided for in a comprehensive system that covers many use cases. There are 3 main results from the project:

1. A standardized way of performing stress testing of a microservices system
2. A pipeline for data processing
3. Visualization of the most relevant metrics for tracking important changes inside the runtime of the system in question

Locust stress testing suite and open source testing data

The stress testing suite is able to automate stressing and data collection of any locally hosted microservices system using accessible API endpoints. It has been iterated upon four times, and the development process of the testing and data collection suite has helped contribute to the Locust stress testing suite itself, with a contribution to the documentation of the tool on correct implementation on Windows machines [25]. The collected data, which has been tuned to be from a system undergoing near-overwhelming load, has been uploaded to the open source data repository Zenodo [26]. There it is publicly available for other researches to perform further learning and experimentation. It features 6 MB of compressed CSV data from the stress testing and collection.

A pipeline for data processing

The data generated from the stress testing suite is verbose, obtuse, and carries huge amounts of static or irrelevant information. During the course of several months, the pipeline has gone from a simple attempt to make sense of the data to an automated and configurable package that optionally reduces the dataset to less than one percent of the original.

Non-PCA metrics visualization

Let's look at the results from the pipeline in the case of not performing dimensionality reduction, for the sake of finding good data to help inform visually about the system. The total process for this data is as follows:

1. Remove monotonically increasing features.
2. Remove columns with more than 50% missing values.

3. Remove non-ubiquitous columns.
4. Impute missing values in the rest of the dataset using the kNN imputer, with the n_neighbors argument set to 1.
5. Standardize the values using the StandardScaler.
6. Select the 5 highest variance features.

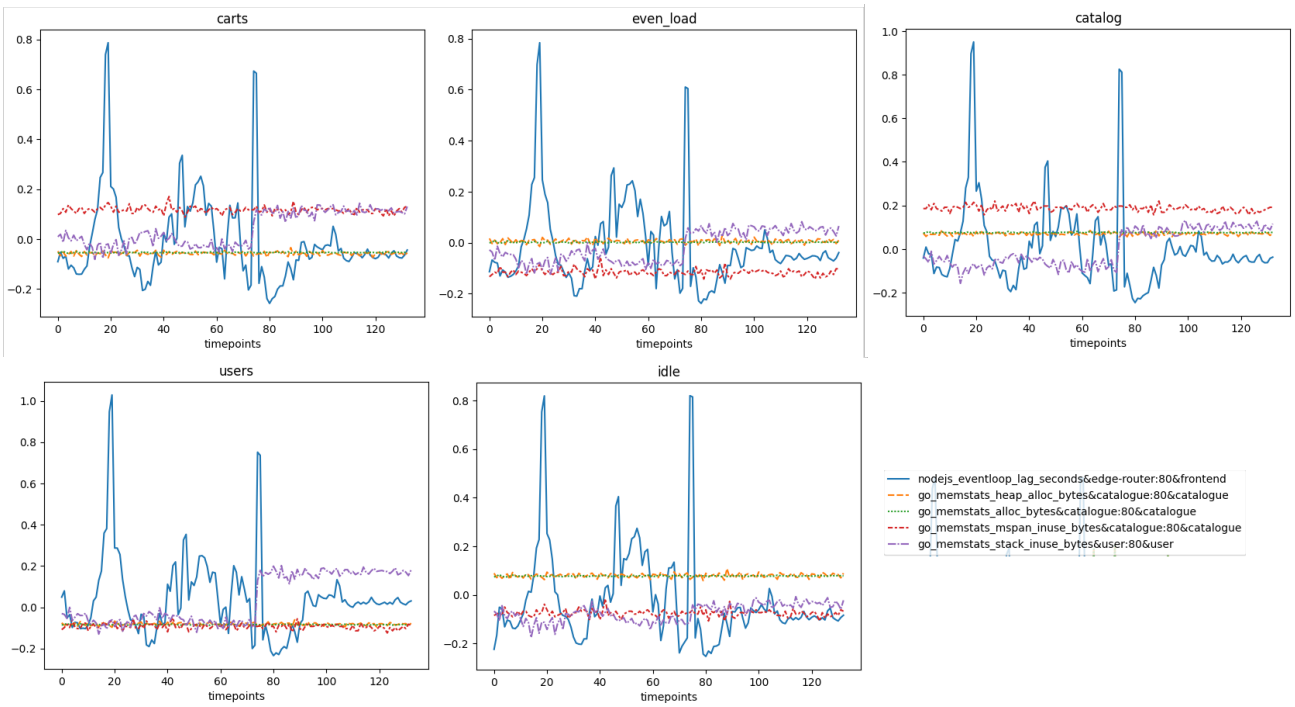

FIGURE 4.6: The mean metric readouts for each time point during stressing. The shown metrics are the top 5 highest variance metrics

The figure shows several surprising attributes. For one, each type of testing, including the idle control group, exhibit consistent fluctuations in the *nodejs_eventloop_lag_seconds&edge-router:80&frontend* metric. Not only this metric, but several of the other metrics exhibit similar behavior around the same time points. *Nodejs_eventloop_lag_seconds* is from the basic supplied NodeJS Prometheus instrumentation package [27]. It measures the NodeJS event loop lag. The event loop is the queue that non-blocking operations are put in to wait for their turn to run. It is overall a decent measure of how busy the system is. The "edge-router" part designates that it comes from the load balancing edge router, which in the case of the sockshop demo is Traefik [28].

One can also observe that the metrics measuring allocated bytes in heap and memory for the catalog microservice are consistently low or around zero during all runs except the "catalog" tag and the "idle" tag. The "catalog" tag makes perfect sense and is a good indicator for useful data, but the fact that the "idle" tag gives the sames results is concerning.

Another standout measure is *go_memstats_stack_inuse_bytes&user:80&user*. It sees a sharp increase around timestamp number 75: With each timestamp being 5 seconds apart, that corresponds to happening about 6 and a half minutes into the testing. It does not do the same spike during idle running.

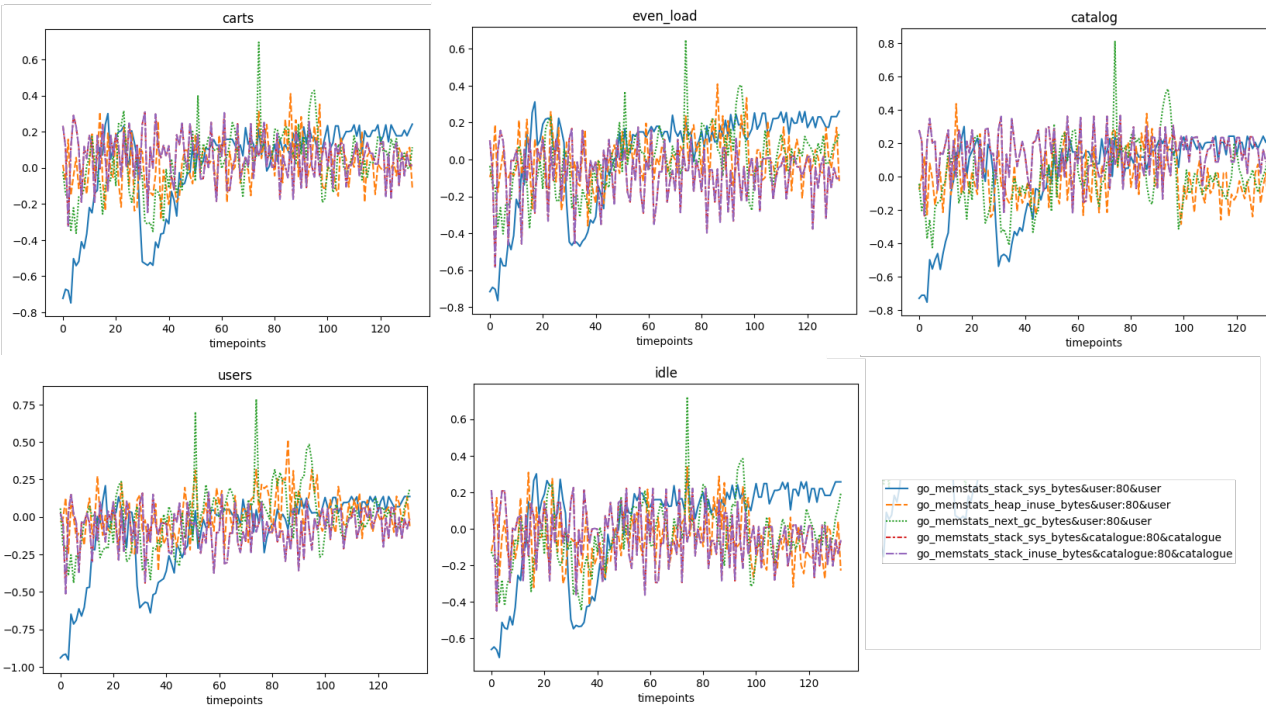

FIGURE 4.7: Metric 6-10 by variance

Visualization - keeping non-ubiquitous features - no PCA

Keeping non-ubiquitous features has a profound impact on the data supplied. It gives a different view of the system entirely, and shows more natural curves. Most notably, it follows the same perplexing pattern observed in the previous data: A spike followed by a falloff around the 75th time point. The average system load for the shipping model is also strange: It shows a consistent decline in its own perceived system load during stress testing. The "shipping" microservice should in theory not suffer much in the stressing in the first place, as it does not have much defined behavior in the sockshop system.

PCA features

The following data was created by performing dimensionality reduction with PCA on the data. This removes direct observability from the dataset, as the original features are lost. However, all individual features are optimized to carry as much information about the system as possible. This dataset demonstrates much of the same metrics that are observed in the directly represented features: A gradual shift in the system that converges around the middle of the testing.

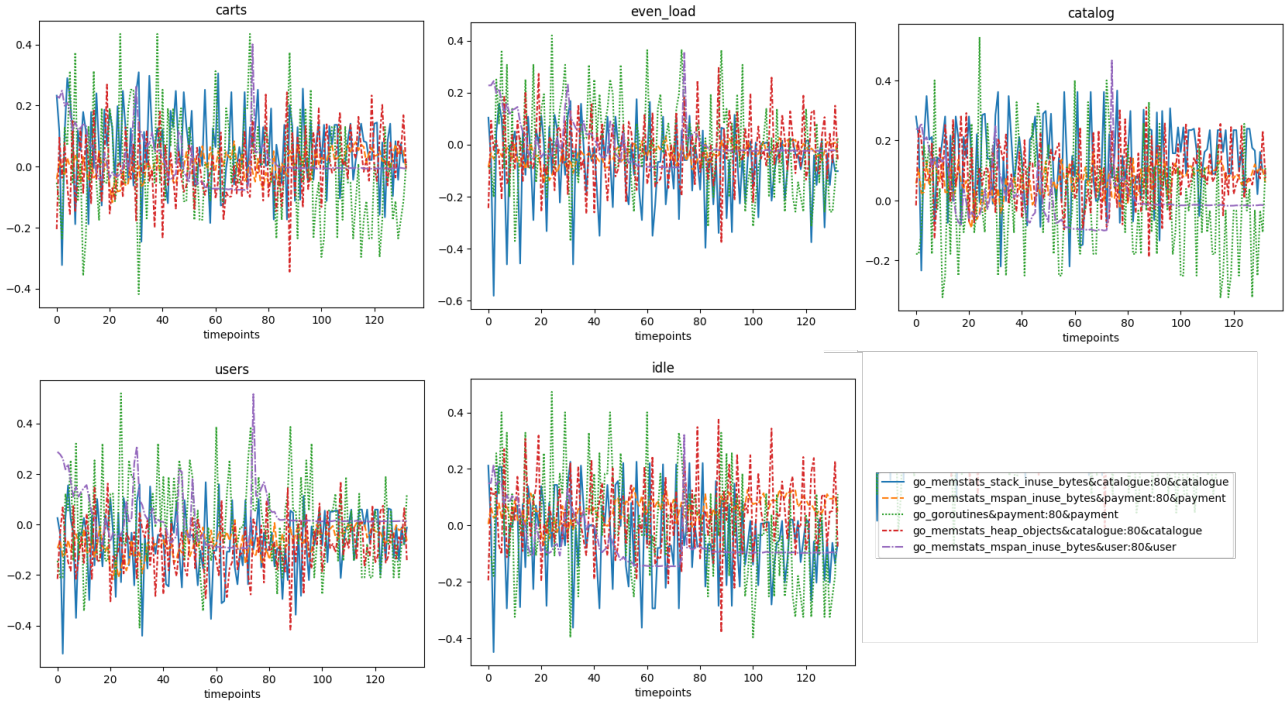

FIGURE 4.8: Metric 10-15 by variance

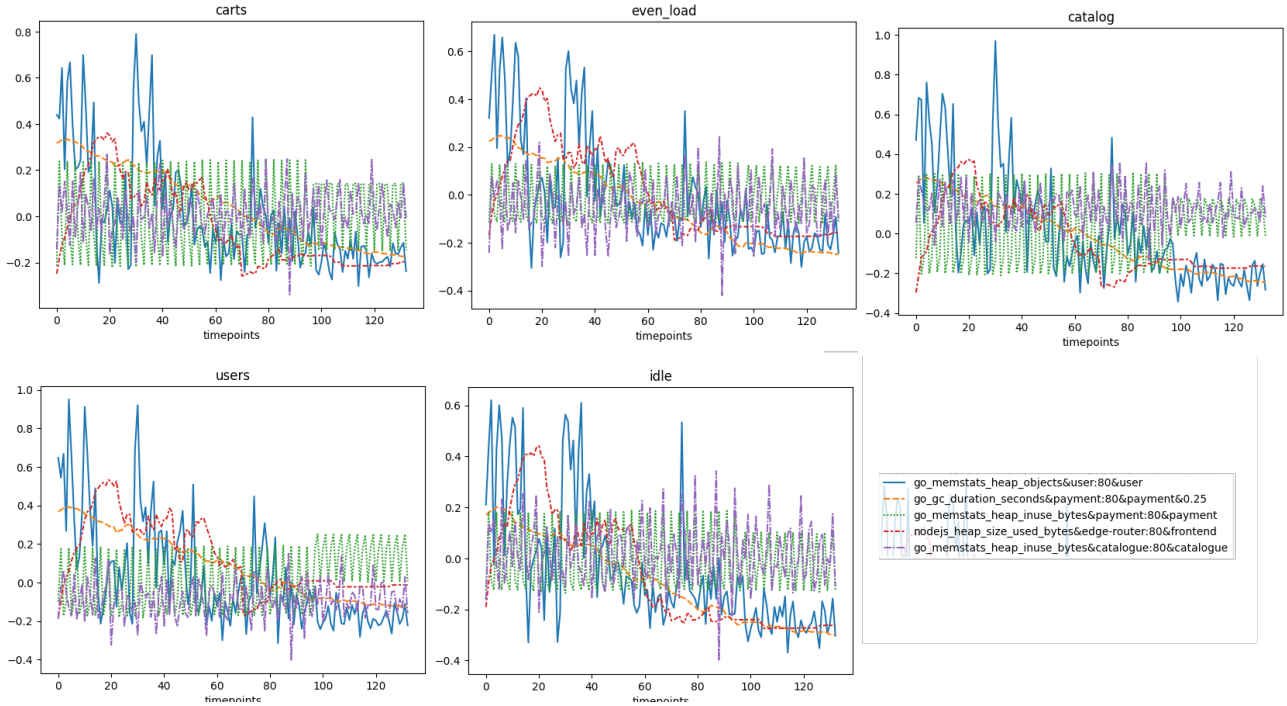

FIGURE 4.9: Metric 15-20 by variance

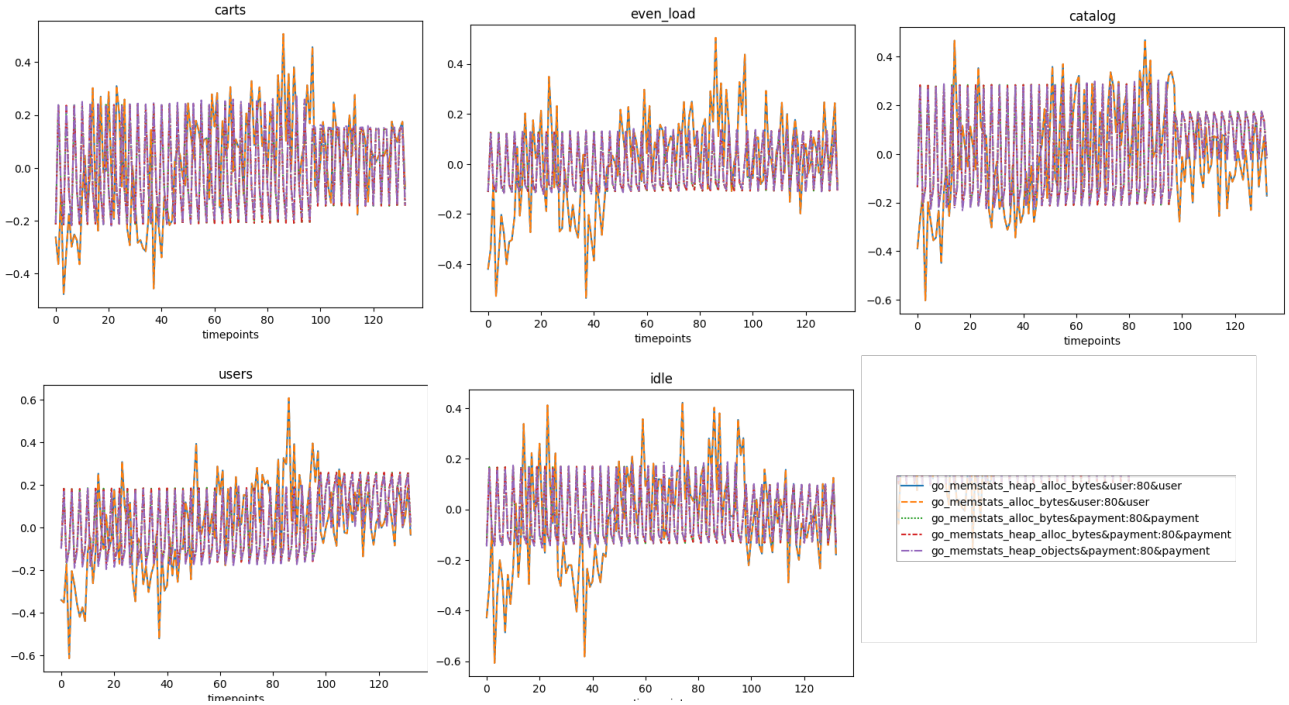

FIGURE 4.10: Metric 25-20 by variance

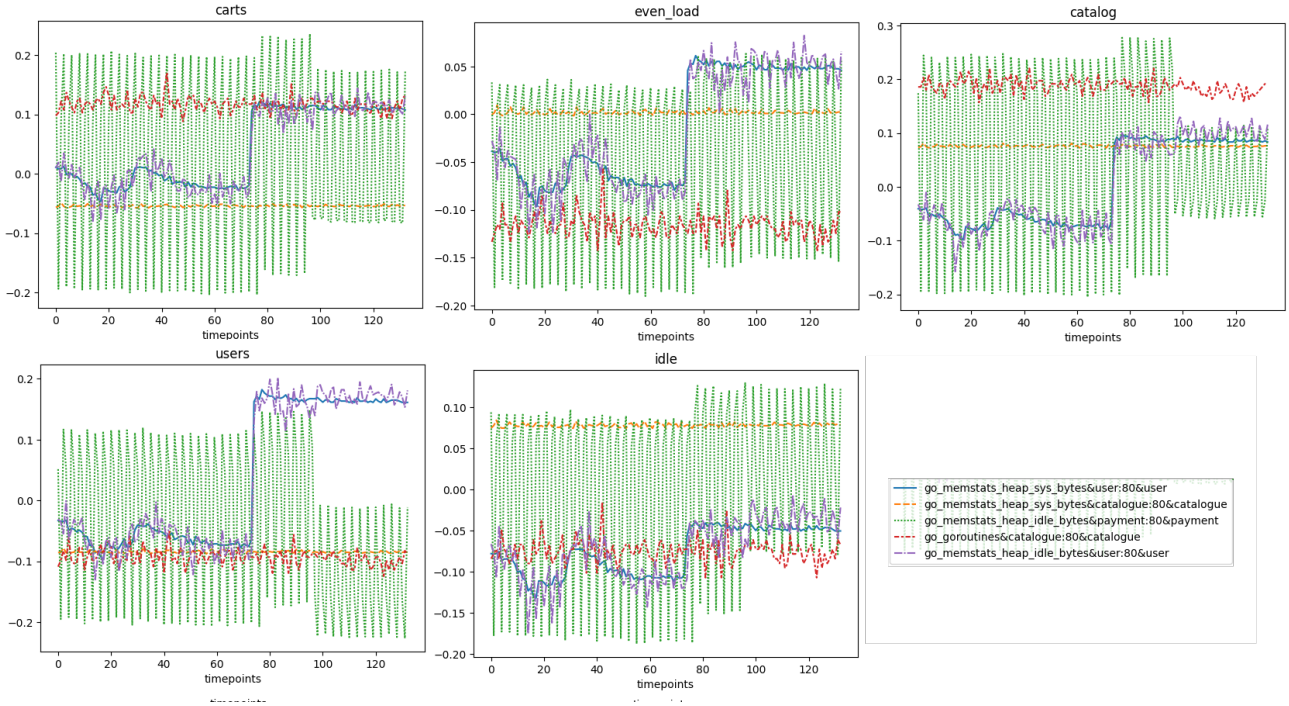

FIGURE 4.11: Metric 25-30 by variance

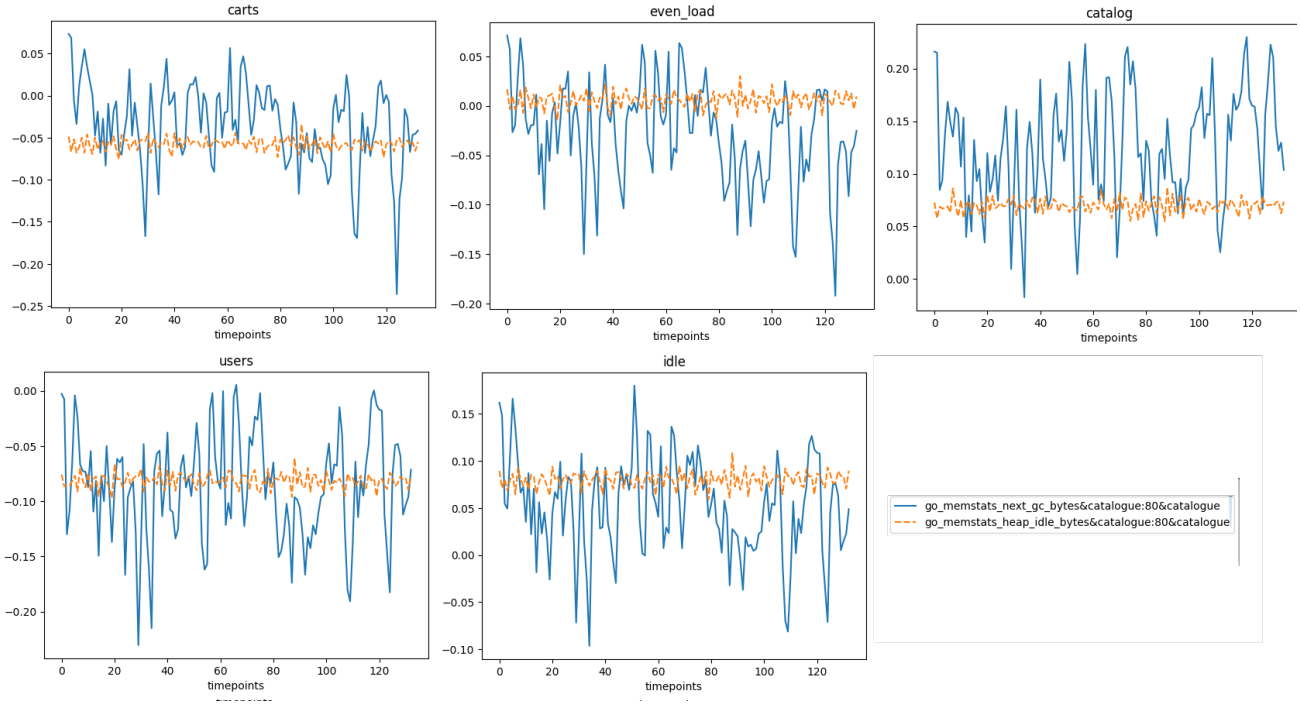

FIGURE 4.12: Metric 30-32 by variance

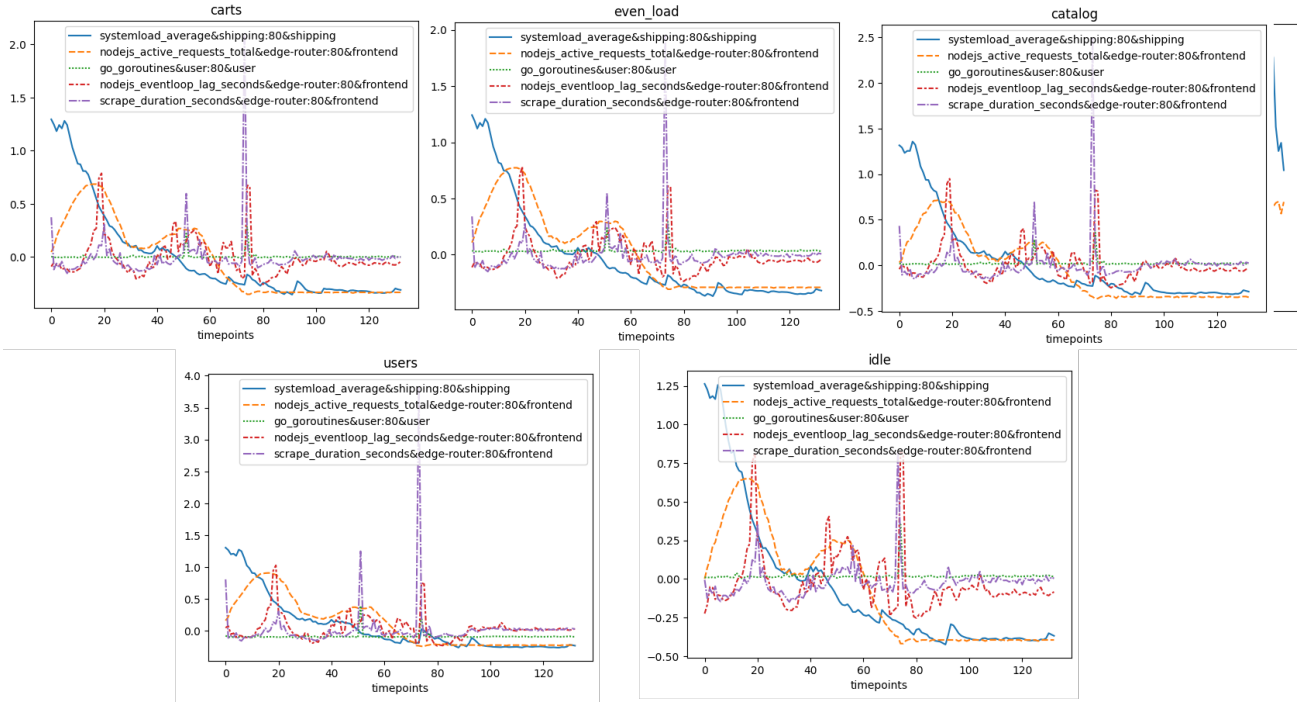

FIGURE 4.13: Top five metrics with non-ubiquitous features

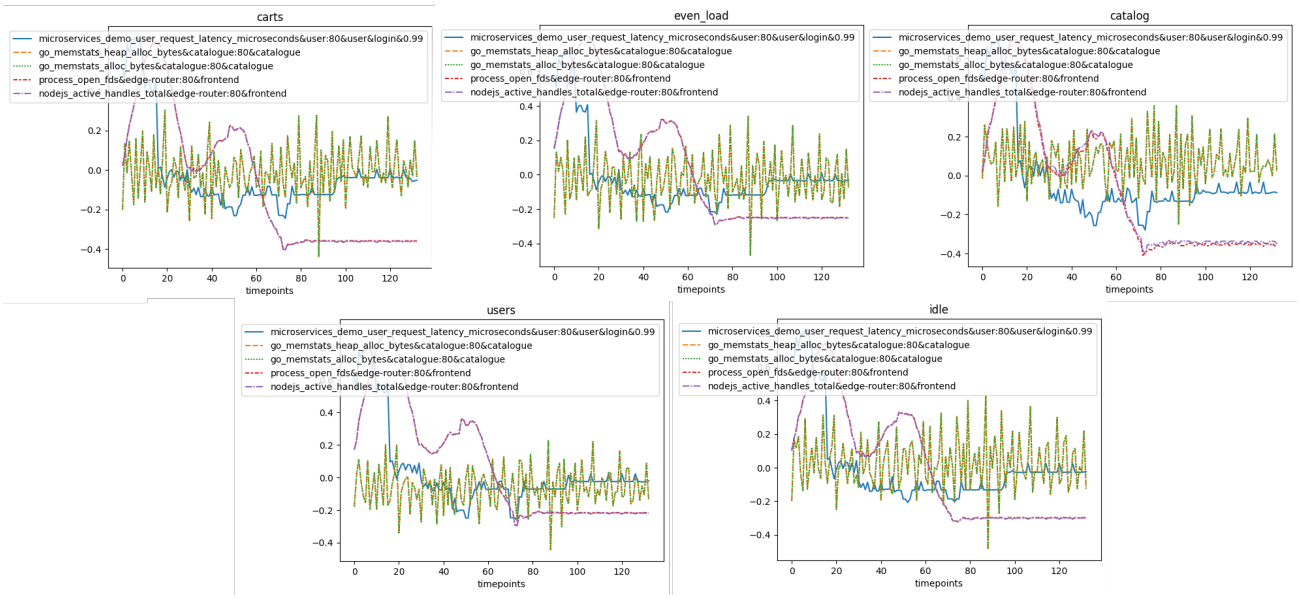

FIGURE 4.14: Non-ubiquitous features, 5-10

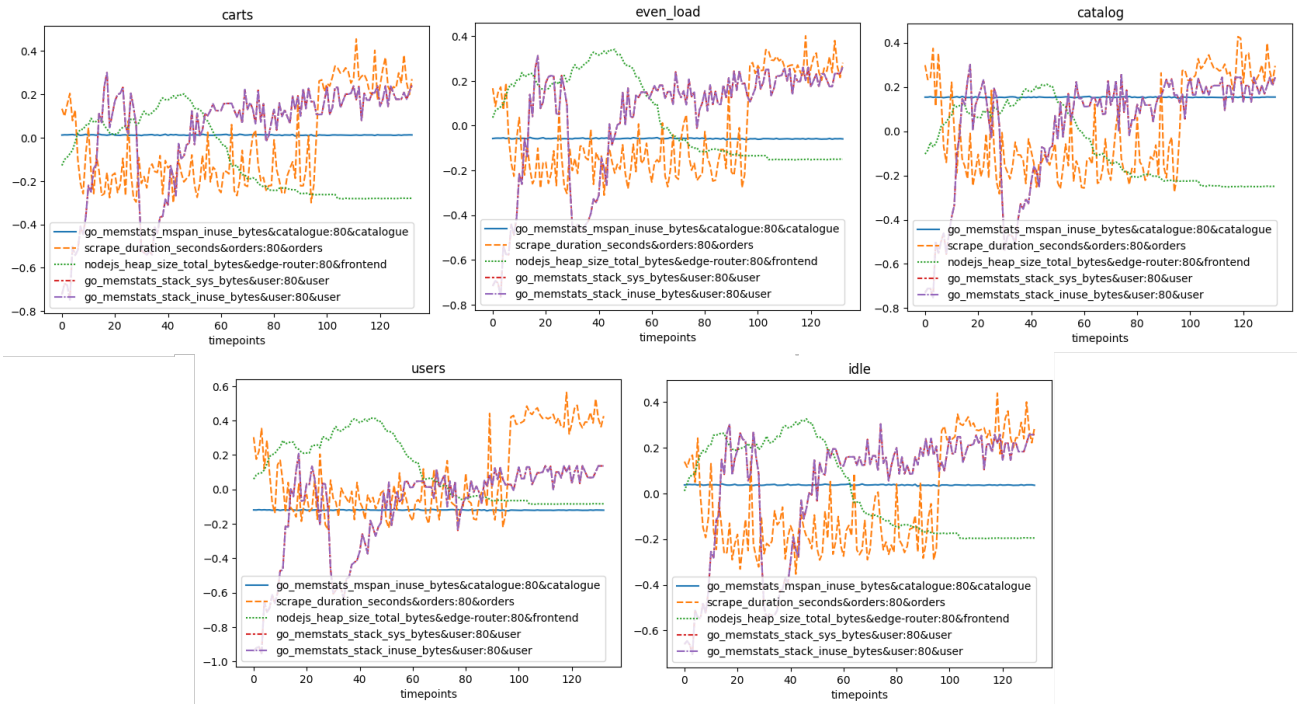

FIGURE 4.15: Non-ubiquitous features, 10-15

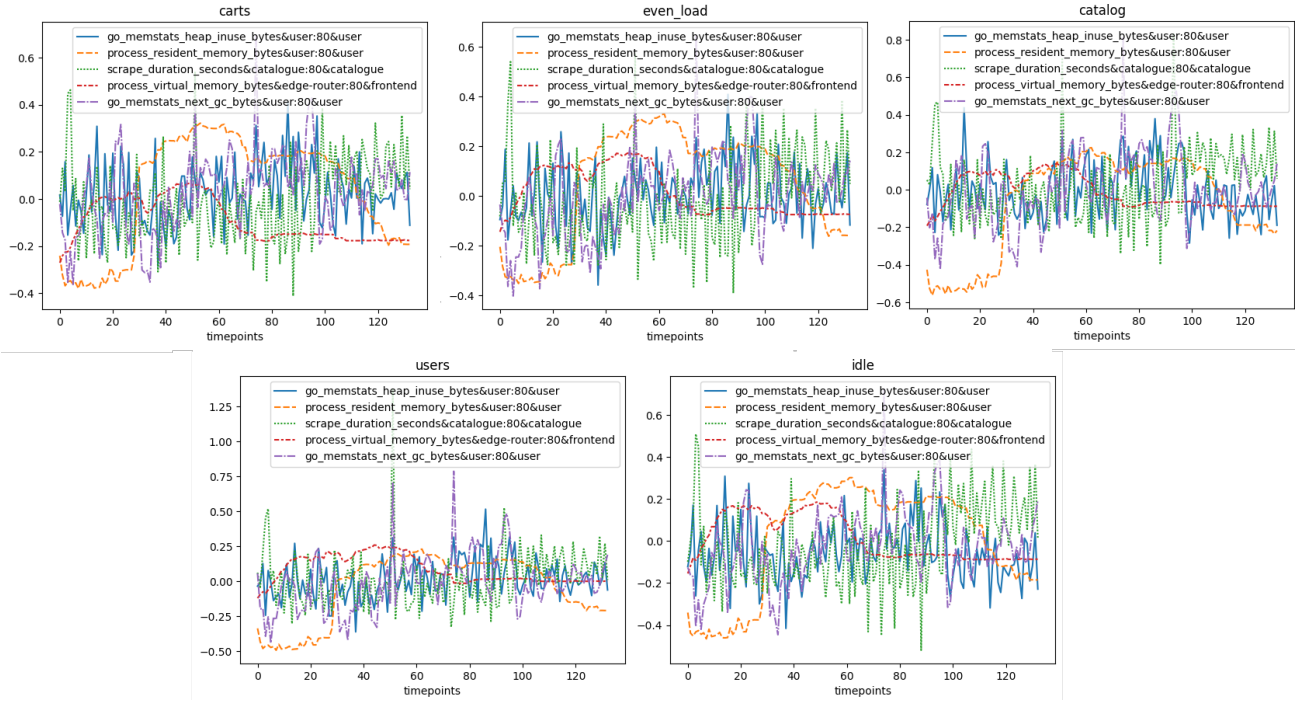

FIGURE 4.16: Non-ubiquitous features, 15-20

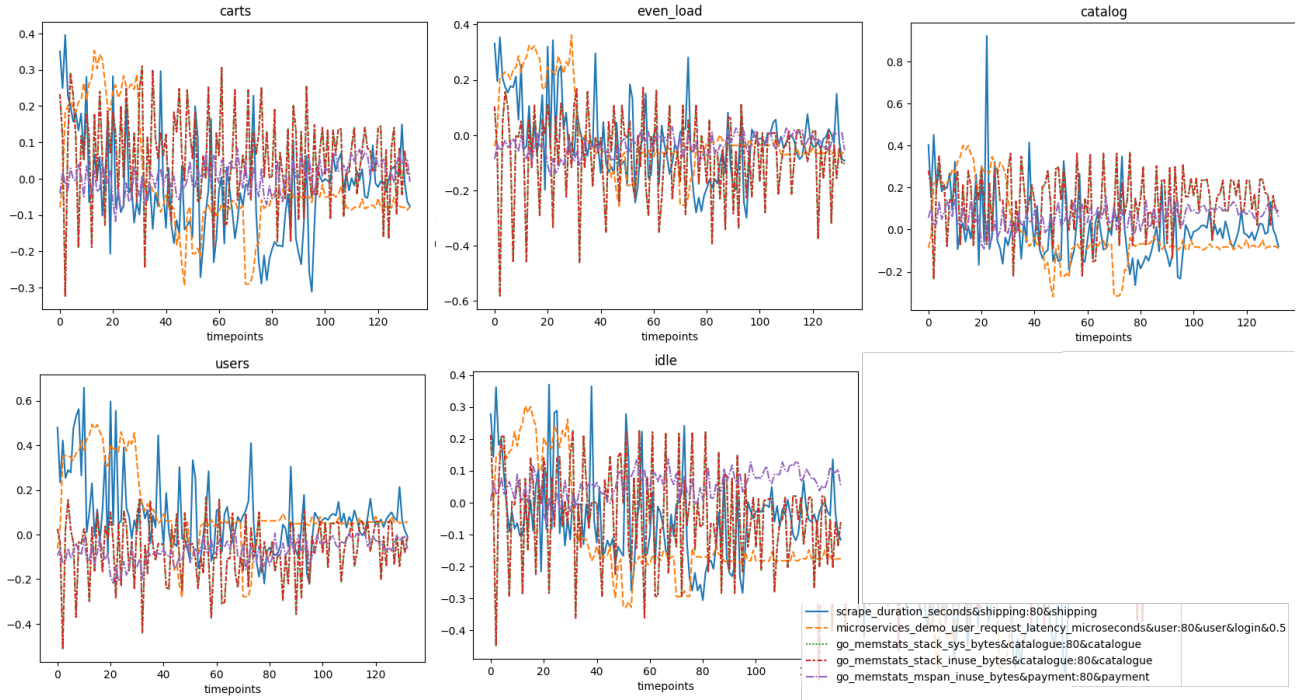

FIGURE 4.17: Non-ubiquitous features, 20-25

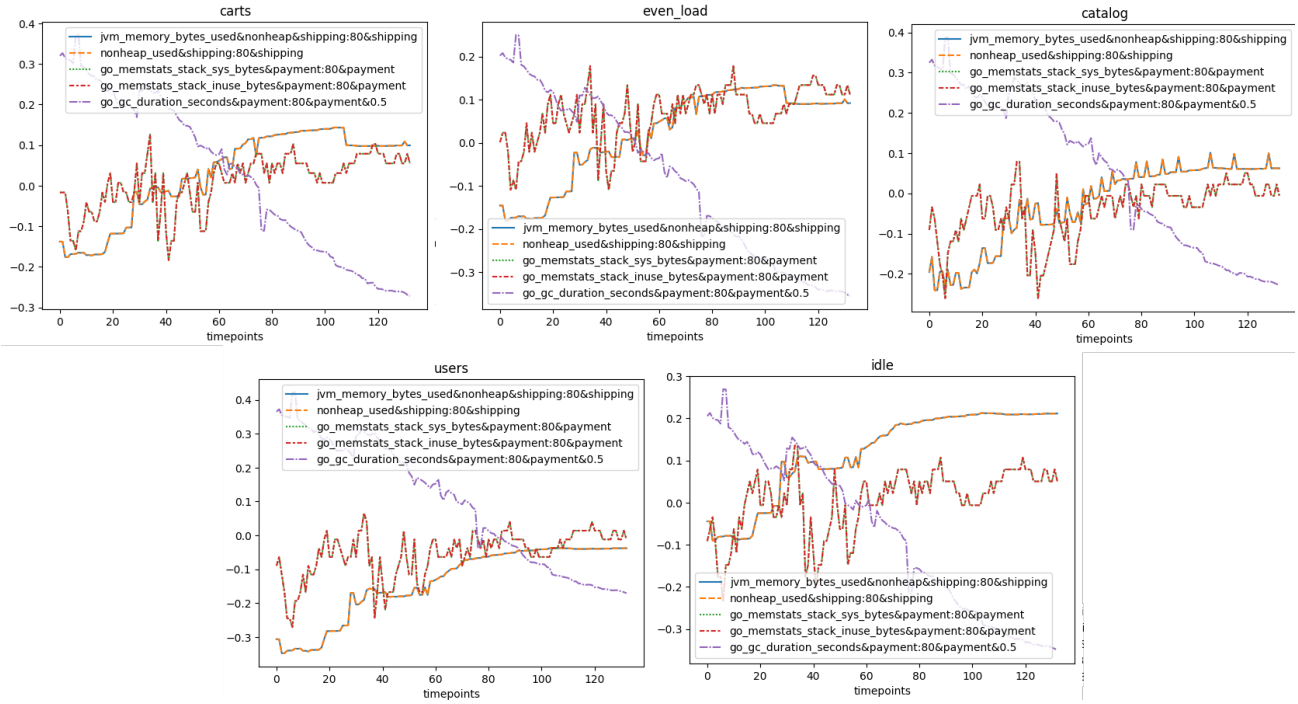

FIGURE 4.18: Non-ubiquitous features, 25-30

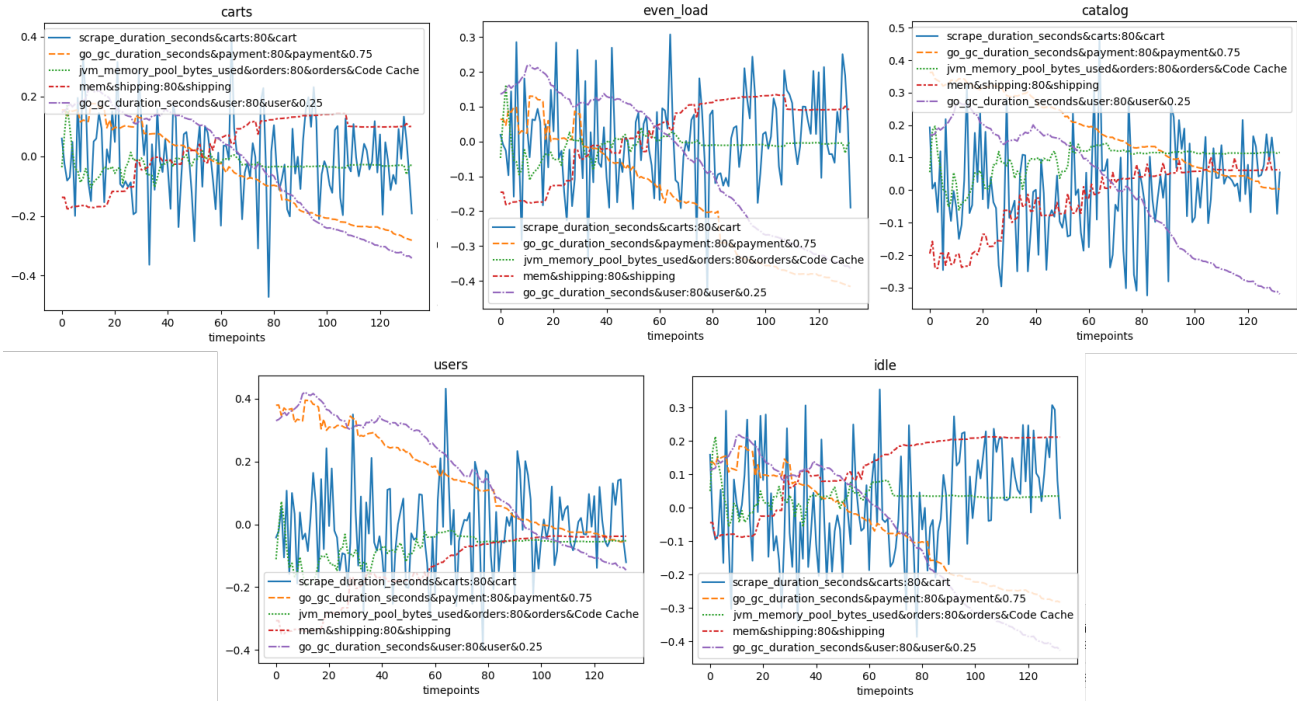

FIGURE 4.19: Non-ubiquitous features, 30-35

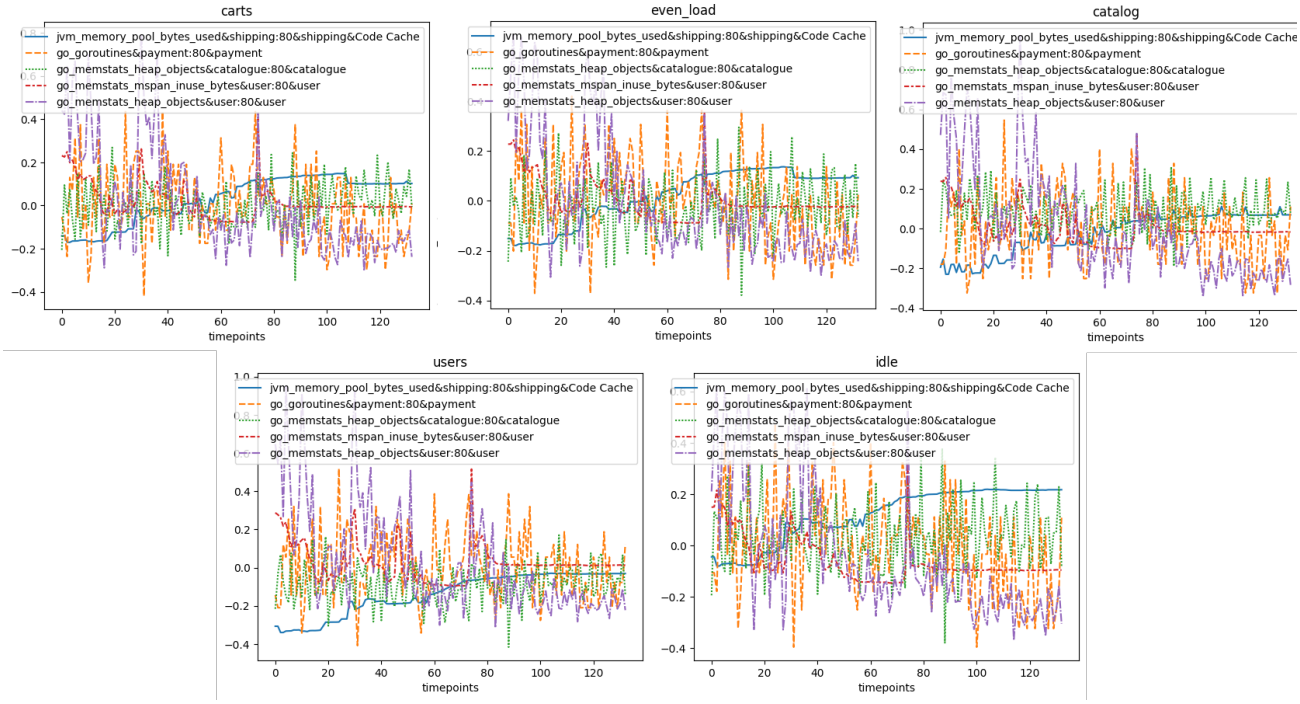

FIGURE 4.20: Non-ubiquitous features, 35-40

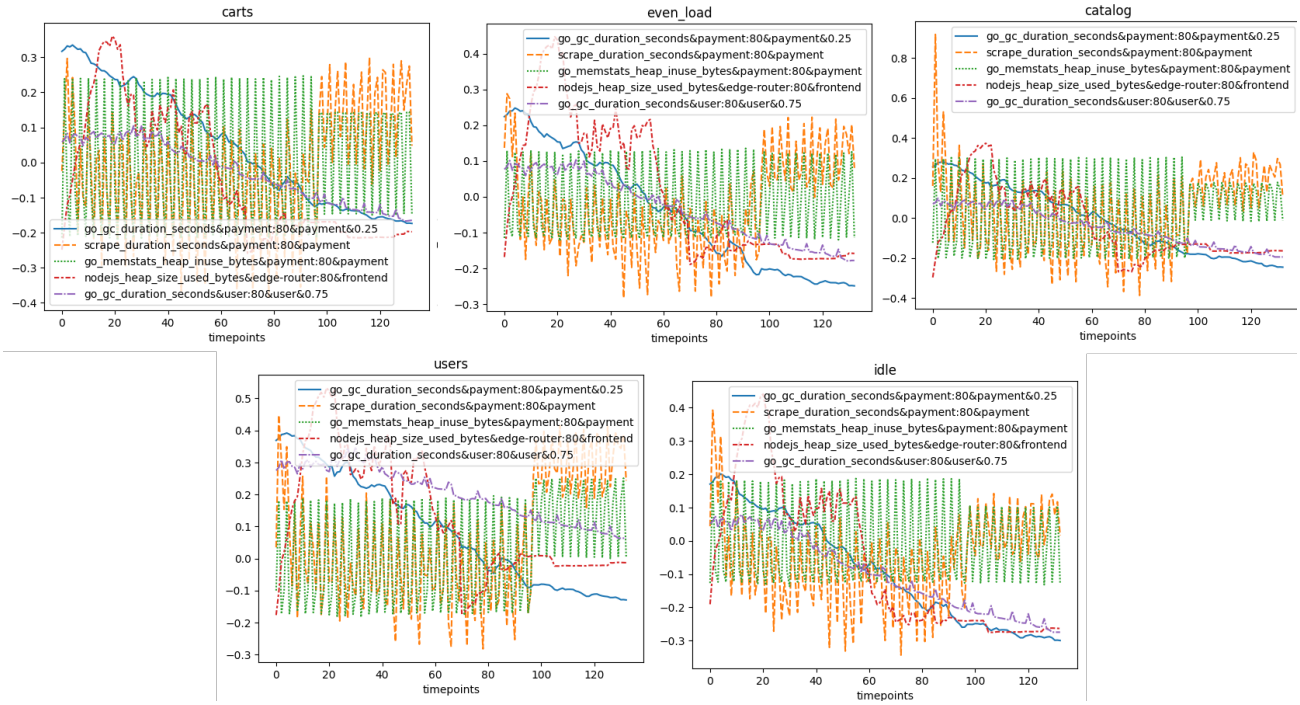

FIGURE 4.21: Non-ubiquitous features, 40-45

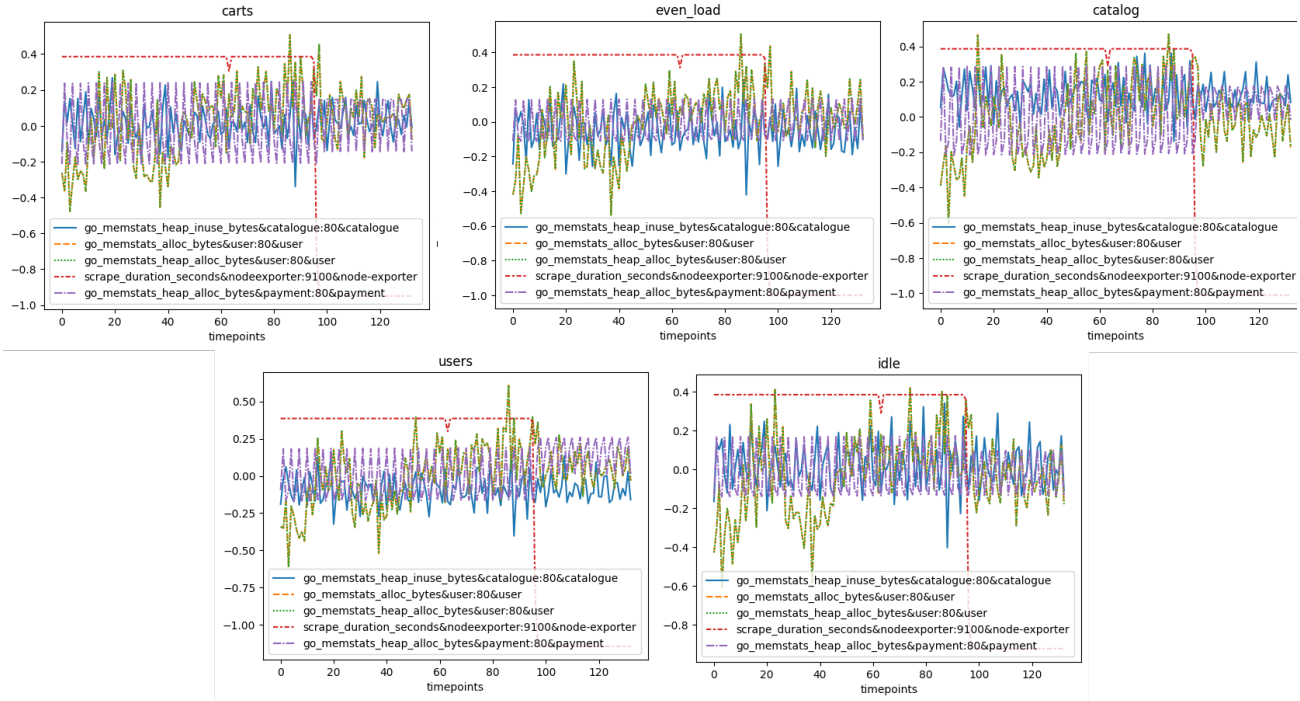

FIGURE 4.22: Non-ubiquitous features, 45-50

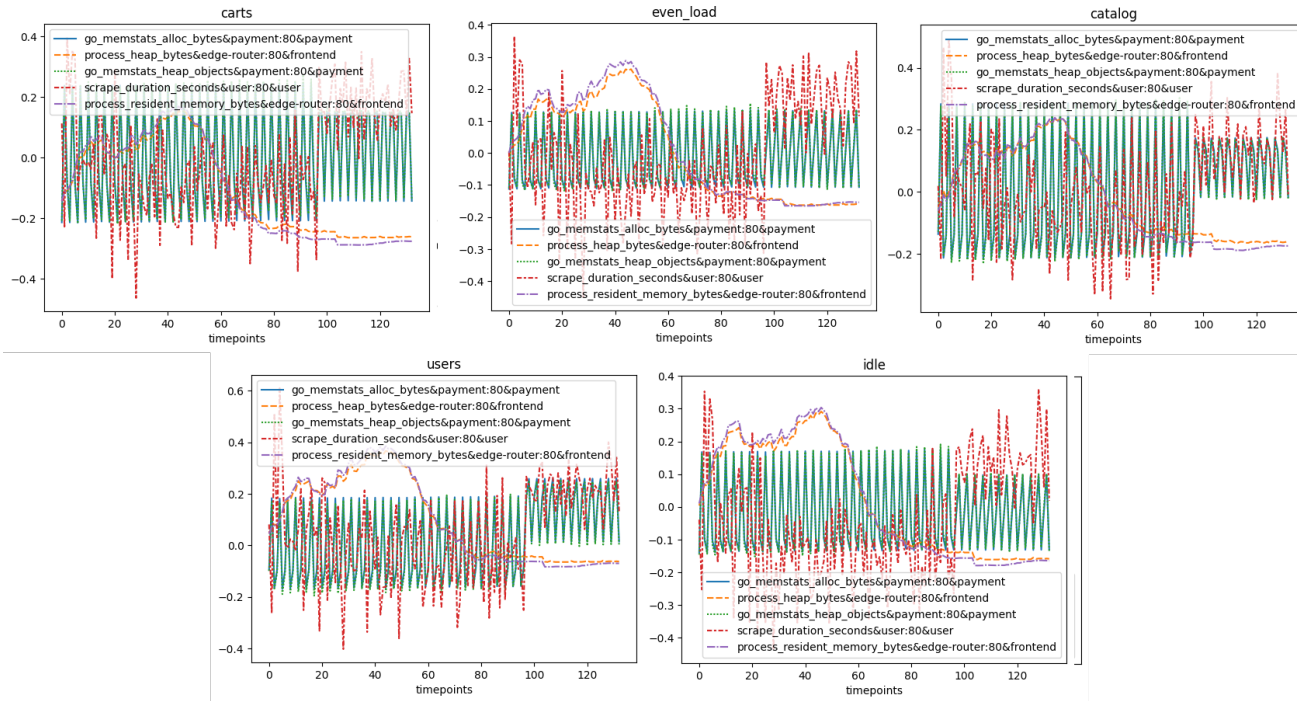

FIGURE 4.23: Non-ubiquitous features, 50-55

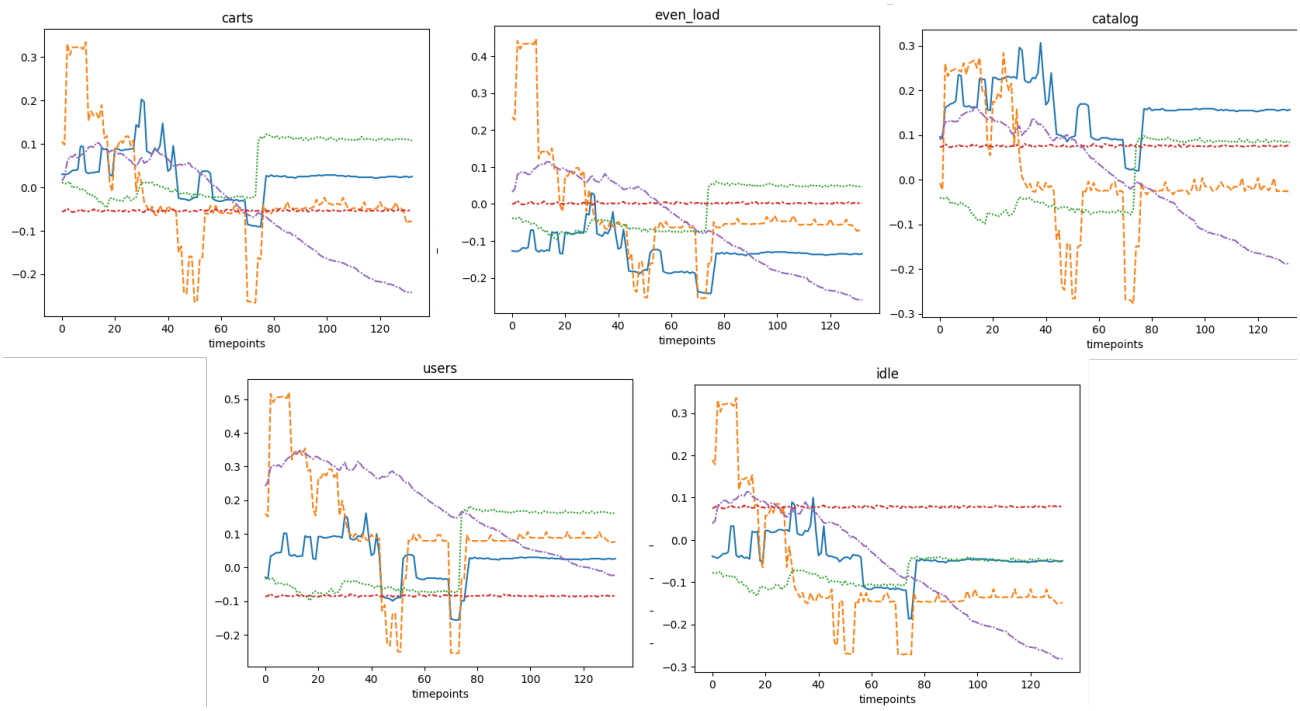

FIGURE 4.24: Non-ubiquitous features, 55-60

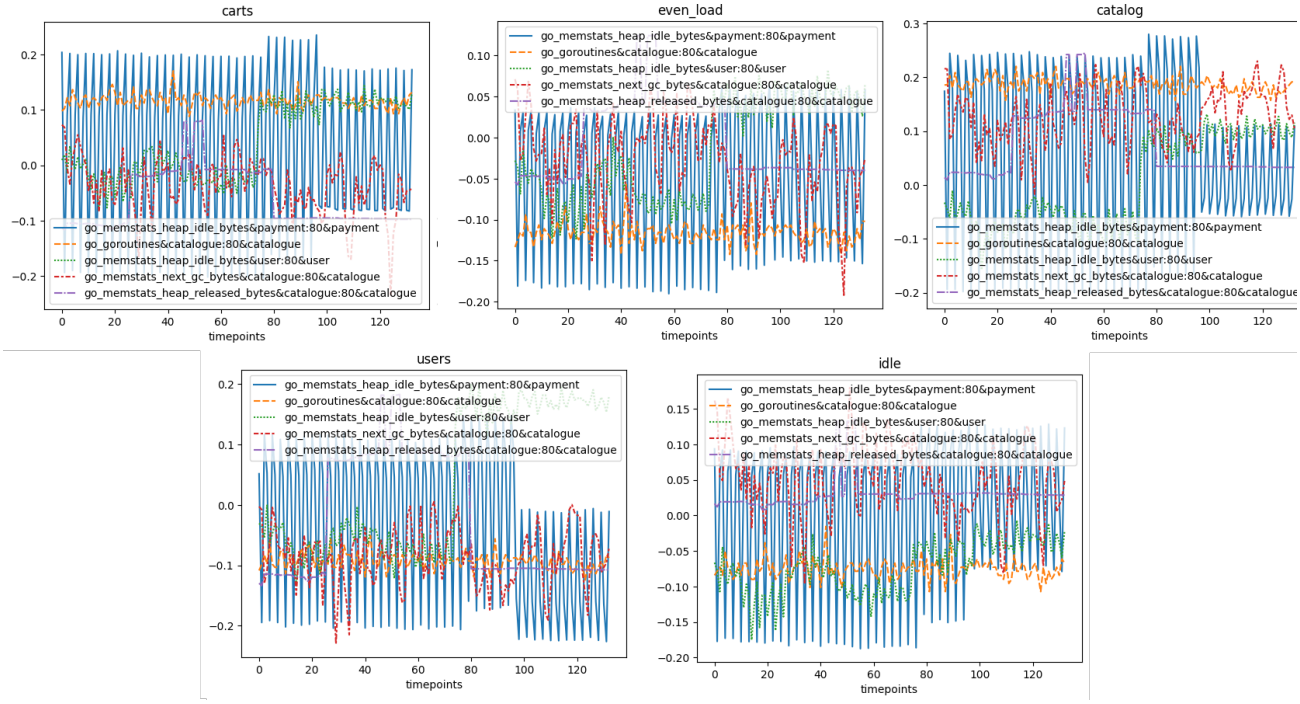

FIGURE 4.25: Non-ubiquitous features, 60-65

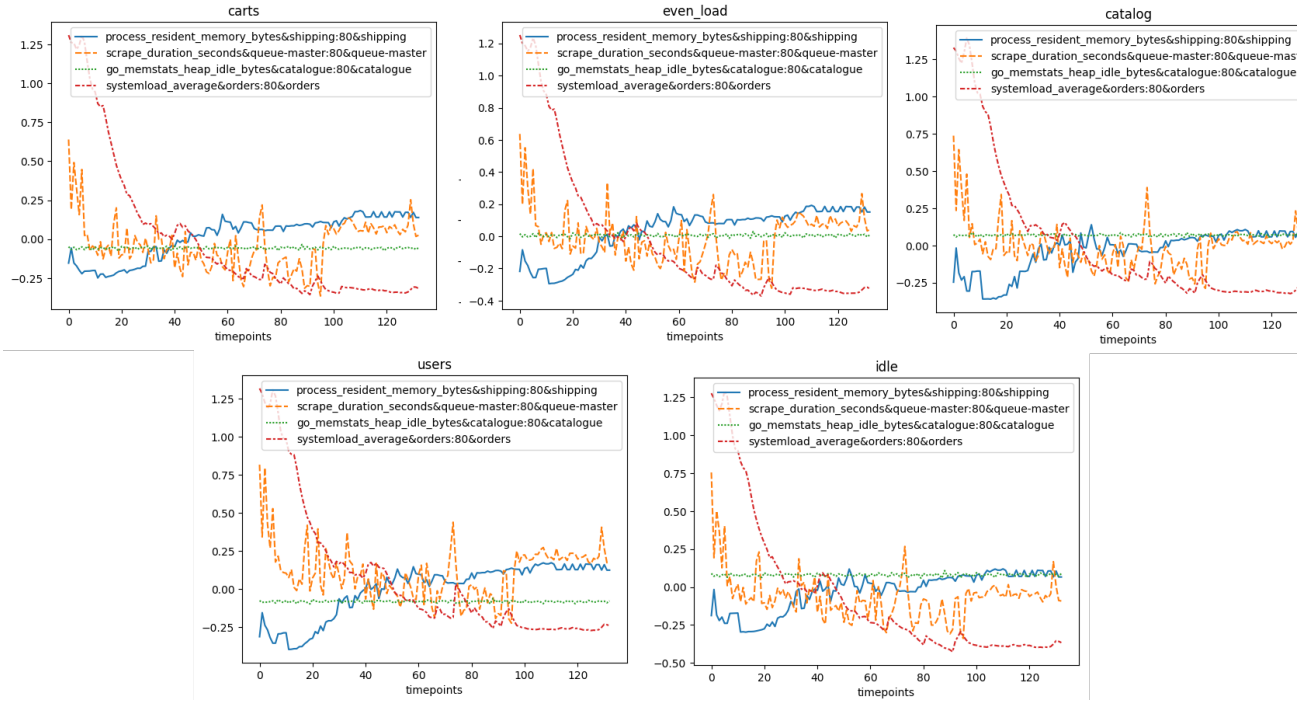

FIGURE 4.26: Non-ubiquitous features, 65-70

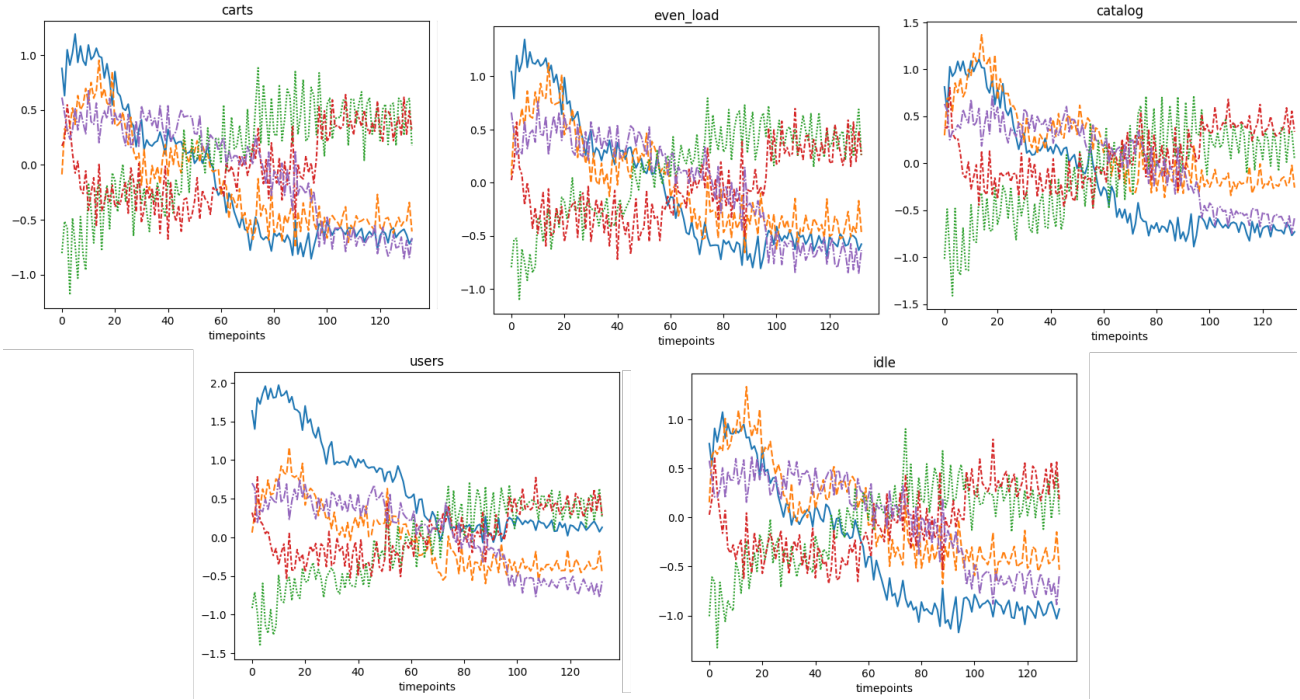

FIGURE 4.27: Top 5 PCA(10)-reduced features by variance.

Discussion

The initial goal of the project was to properly identify a clear path to modelling and diagnosing a microservices system using stress testing and an automated pipeline. This can only be counted as partially successful: While the both the collection and the pipeline performs its duties, the data collected carries little in the way of very useful or signifying information. To specify, there is little correlation with the stressing of the system and the metrics in the collected data. The discussion will therefore largely focus on potential reasons this could have occurred, and ways to improve data collection and testing in the future. It will go over each part of the process and discuss potential issues and improvements, and what went well.

The microservices demo

During the early phases of the project, the sockshop demo by WeaveWorks was selected as the system to be tested on. It was largely selected out of pure pragmatism: Its small scope meant it could be run and tested in its entirety on a personal computer, without having to rent or requisition a server for testing. The fact that it came already supplied with instrumentation, a stressing framework in Locust and locally published API ports made it seem like a no-brainer selection. However, it proved more challenging that initially assumed to work with. The biggest caveat was that it was not nearly as open source as initially assumed. While at first glance the entire source code is available on GitHub for inspection and comprehension, it turns out that the inner workings of the microservices themselves are in large part hosted on the Docker Hub, closed source. This made it much harder than anticipated to inspect and understand the workings of the system, like how exactly several of the Prometheus instrumentations are implemented. Also, as the data shows, it seems to handle the stress testing in a remarkable way, often seemingly reducing the amount of resources needed at times during runtime when stressed. There are several potential reasons for this:

The Traefik edge router

The exact implementation of the Traefik edge router is not known at the time of writing: It is one of the elements that turned to be less open source than initially assumed. However, one can surmise from the Traefik documentation that it is completely capable of causing such a result given the appropriate implementation. It sports two main features to potentially cause this: Rate limiting and the circuit breaker.
The rate limiter is dependent on the implementation done on the back end of the sockshop application, but it provides functionality for limiting the amount of incoming requests to any service to a maximum amount within a given timeframe: Any extra requests will get blocked early on in the process, and not get considered by the proxy at all. While this is a possible reason for the behavior of some measured metrics, it seems unlikely to be the main culprit. This is because the rate limiter is off by default, so it would have to be configured by the WeaveWorks to be on. Considering the same system comes with the stress testing tool Locust, it seems counterintuitive to kneecap their own functionality like that. The other potential cause from Traefik's side is the circuit breaker functionality. It tries to prevent large amounts of requests to an unhealthy service, which could cause cascading failures. This is also a functionality that has to be specified to be turned on. It is, however, more likely to be active than the rate limiter. This is because it works more as a fallback to ensure system

stability than a hard rule for how much work one can give the system. So it is possible that the circuit breaker is configured to ensure that this demo system has as good stability as possible, not considering the potential hidden implications on stress testing. One important detail here is that the stress testing not only stressed the locally running sockshop application, but also the entire host PC. During the process of configuring the number of stressors for the Locust testing, the number was incrementally tweaked to be close to system failure to ensure proper metric generation. It is possible that this backfired, causing the system to run out of resources occasionally and triggering the circuit breaker. This could explain why a large amount of features that should be increasing during heavy load, end up decreasing during stressing instead: The load balancer starts correcting for the high load and lack of hardware resources.

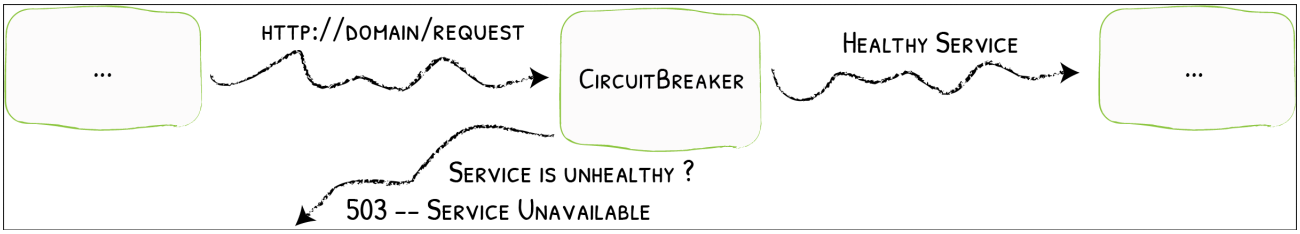

FIGURE 4.28: Illustration of the circuit breaker functionality. Credit: Traefik.io

Microservices caching and automated response

The core tool of dynamic programming is to store the results of previous function calls: If the function is called once again in the same way, it can use the stored value instantly instead of running potentially expensive calculations inside the function. The microservices could potentially implement a similar resource saving algorithm. The locust scripts do contain some degree of randomness to try to keep the host guessing and for variation's sake. But the degree or randomness is not that large, and it's possible that the system over time realizes that it can simply send cached responses to every request, evening out over time. It seems a bit unlikely, however, since the Locust script that has been developed during this project to maximize system stress while avoiding full on crashes spawns between 30 and 60 new user instances per second, each one continuously making new requests. The full range of possible request types should be reached within the first couple of seconds.

The stress testing environment

The stress testing environment was developed incrementally to work within the hardware constraints of the project, to maximize effective stress on the system. The final values of randomly generated stressors in the range of 10000 and 12000 user instances was selected because it could semi-consistently complete a full run without Docker or some other core component crashing due to lack of resources. There is also the physical stress on the host machine to consider. The Docker environment hosting the sockshop service was configured to run on a limited amount of system resources. Docker does allow for a limit on hardware memory and CPU usage in the dockerfile for a given system. Following the instructions on Docker's page on the subject [29], memory, memory-swap and CPU was limited to well below the host machine's capabilities. However, these limits did not work for whatever reason: The running containers consistently disregarded the set limit under stress testing. I was unable to solve this issue during the course of the project. This has severe implications on the ease of reproducing the data, as well as achieving consistent constraints during testing. Because of this, the Locust parameters were forced to be tuned to such a level that the host machine itself was struggling. This can cause unforeseen crashes. One outcome of this is that Docker had to be restarted

in between each full run of a stress test. Otherwise, the host PC would eventually fully crash and data would be lost. This has a high chance of impacting the value of the data.

The preprocessing pipeline

Making use of multivariate time series data for analysis of microservice systems is a problem that still has a long way to go. As noted in [30], a major problem is the sheer volume of information from microservice systems. The received data effectively had 3 dimensions: The class label, the time of collection, and the collected metric. The initial assumption at the beginning of the project was that sktime, the library built on top of Scikit-Learn specifically for analysis of time series data, would have the appropriate tools to quite easily perform preprocessing and machine learning on the supplied data. This turned out to be far from the case. There are inconsistencies in the format the various algorithms want to receive the data in, ranging from simple python lists of data frames to multi-indexed single dataframes to 3d arrays. Some supplied works could only work with 2-dimensional data. It resulted in a lot of juggling and various implementations for converting the data between the different formats without losing information or consistency. Implementing the bare minimum turned out to be such a time sink that there is much left that could be done with it.

For one, there are multiple more possible ways of performing each step of preprocessing. A complete pipeline would have more options for each step. One big aspect is the treatment of monotonically increasing features. The pipeline only has the option of removing them. This means that potentially valuable features that could give an indication of stress constantly increasing in parts of the system are lost. The counterargument is that such features tend to simply count things like threads started and bytes allocated, which already get recorded in other metrics that measure their current values. Nevertheless, properly implementing differencing would make more metrics viable for further consideration.

When imputing missing values, the kNN imputer was implemented as the only option, with the n_neighbors argument set to 1. This implementation of kNN was the result of the following experimentation:

- Create a complete dataset from the stress testing by removing all features with missing values.
- Store that as a target dataset, then programmatically create NaNs in random spaces in the data at a frequency emulating that found in the real data set.
- Try various settings for kNN and select the one that best emulates the target dataset.

This landed on n_neighbors=1 as the most accurate option. This makes good sense, since it kNN imputation works columnwise. With n_neighbors=1 it simply calculates the mean of the nearest non-NaN before and after. This should usually fit well with time series data. However, kNN is just one of many ways to perform imputation, and it has not been measured against other imputation methods. For example, one could use sklearn's experimental IterativeImputer class, which is meant to handle multivariate data. It trains a regression classifier on the data that models features as a function of other features in the dataset, and performs several rounds of imputation before selecting a winner among the various classifiers it created. However, for this time series data the kNN classifier proved to be quite robust and gave good results in the testing. Sadly the exact testing data showing the percentage success was lost.

The removal of non-ubiquitous features in the dataset is added as an option in the pipeline. Choosing to apply that operation cuts the available data roughly in half. In the results section, there are both the result when removing and the result when not removing. Removing them means that

one does not have to rely on potentially faulty imputation when reconstructing missing features in some instances. However, it remains a large loss of data. Whether to remove non-ubiquitous or not will depend on the needs of the end user: Do they need more data, at risk of some of it being diluted or wrongfully imputed, or do they have enough data already, and can prioritize sticking to the facts? But not all non-ubiquitous features are created equal. Non-ubiquitous just means that is not present in every single instance. But there is vast difference between a feature that only appears a couple of places and a feature that appears in all but a few. If a feature tends to be present in most datasets, the risk of accuracy loss upon imputation is minimal. Therefore, a more elegant solution than the one in this project would be a function that measures how many instances a feature is missing from. Instead of a simple yes or no for removal, one could supply a threshold for the percentage of the instances a feature can be missing from before it is removed.

Inside the various functions in collection, sorting and storage are various long-winded workarounds for problems that seemed simple at a glance. For example, most of the internal data structuring has been done in a dataframe with multiindex. It seemed tidiest and most sensible way of organizing the data, because the hierarchical structure let one visualize and display the three levels a bit more easily. However, during the project's course it turned out to be poorly supported both in data and in the implementations for various algorithms that were being tested. It came to a head when an internal function used for conversion between different data representations stopped working, and all the code came to a standstill. It was months of painstakingly implementing manual workarounds until the developers of sktime published a new version of the package, that could run on a newer version of Python. At the start of the project, all code had to be in Python 3.9.12, as it was the most recent supported version. Only the switch to supporting 3.10 and the subsequent Python environment upgrade was able to fix the internal conversion issues. The theme of the project has been that treading relatively new ground and trying to work in a still developing field with narrow support is full of setbacks that I was not ready for. In hindsight, it would be less immediately visible and comprehensible but much more efficient time-wise to simply store all the raw number data in a 3d numpy array, and keep all secondary information like timestamps and column names in separate files. Countless hours could have been saved from manually implementing specific reindexing logic for thousands of rows and dozens of dataframes.

Learning and contributions

The data ultimately proved to not be up to the standard that was envisioned at the start of the project. The generated test data exhibits little variation between instances and has not responded well to the focused stress testing. The project has not been without its upsides, however. It proved a valuable learning experience about the intricacies of multivariate time series, and the delicate process of preprocessing. While the generated data is not of very high quality, it is well sorted and visualized. The preprocessing process itself also functions well, and the PCA dimensionality reduction successfully divides the dataset into highly verbose features that do exhibit noticeable differences between the different instances.
It is my strong suspicion and working hypothesis that the initial data from the stress testing is of low quality and carries little identifying information. This project has provided experience in Docker management, stress testing, data manipulation with pandas, Prometheus, and general data treatment and Preprocessing.
In terms of contributions, the most direct and already impacting contribution is a contribution to the documentation of the Locust tool. It also uncovered the issue of Locust stressing ignoring hardware limits on Docker containers, meaning one should use some other hosting method to generate reliable

stress test data. All the generated stress data has been uploaded to the public research hosting library Zenodo.

Future work

In the stressing and collection process, the program runs a stressing cycle on each tag in order, before closing and restarting Docker. This can possibly have a polluting effect on the generated data, as the order of the tags is the same in each run. This means that the data could reflect the potential buildup of memory leaks, unclosed connections or other consequences of stressing that bleeds over into the next tag. Because of time constraints, this was not accounted for in the project.
Also, this whole project ended up working on being the prelude to potentially doing machine learning on the data that has been created and processed through it. It serves as a natural springboard for a future project to build upon and try to make a classifier based on either the data or new data generated similarly and put through the same or a modified pipeline. The low information density of the data also leaves several possible explanations for why. It could be an interesting project to do more research and pinpoint the exact reason for the data's ambiguity.

One could perform further testing on the following questions:

1. Does built up stress bleed over into other rounds of testing?
2. Would alternating the order of the stress testing tags solve this problem, if it exists?
3. What metrics, if any, get affected by this conservation of stress?
4. Would other types of dimensionality reduction produce more concise and informative results?
5.

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] Springboot, *spring-petclinic/spring-petclinic-microservices: Distributed version of Spring Petclinic built with Spring Cloud*, 2013. [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-microservices (visited on 10/24/2023).

[21] A. Pasos Ruiz, M. Flynn, J. Large, ·. M. Middlehurst, and ·. A. Bagnall, "The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 35, pp. 401–449, 2021. DOI: 10.1007/s10618-020-00727-3. [Online]. Available: https://doi.org/10.1007/s10618-020-00727-3.

[22] F. Kiraly, *Sktime*. [Online]. Available: https://www.sktime.net/en/stable/ (visited on 11/15/2023).

[23] M. L. Waskom, "Seaborn: Statistical data visualization," *Journal of Open Source Software*, vol. 6, 2021. DOI: 10.21105/joss.03021. [Online]. Available: https://doi.org/10.21105/joss.03021.

[24] 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).

[25] M. Nordbø, *Locust pull request*, 2023. [Online]. Available: https://github.com/locustio/locust/pull/2248 (visited on 10/31/2023).

[26] M. Nordboe, *Prometheus stress testing data from the microservices-demo "sockshop" application*, 2023. [Online]. Available: https://zenodo.org/records/10107954?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6IjEyZTJhZDI3LTk5ZTMtNDkyNy1iMWQyLThiNWJiN2RjOWQ1NyIsImRhdGEiOnt9LCJyYW5kb20iOiI1NGUxIdwfvH0D7pc9QZnVk0t8cz7CK49XJV75hW8OFUfFQ3dBdfyvGfT2dV0TEDr (visited on 11/12/2023).

[27] S. Nyberg, *siimon/prom-client: Prometheus client for node.js*. [Online]. Available: https://github.com/siimon/prom-client (visited on 11/14/2023).

[28] Traefik, *traefik/traefik: The Cloud Native Application Proxy*. [Online]. Available: https://github.com/traefik/traefik{\#}supported-backends (visited on 11/14/2023).

[29] Docker, *Runtime options with Memory, CPUs, and GPUs | Docker Docs*. [Online]. Available: https://docs.docker.com/config/containers/resource{_}constraints/ (visited on 11/15/2023).

[30] R. Heinrich, A. Van Hoorn, H. Knoche, *et al.*, "Performance engineering for microservices: Research challenges & directions," *ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering*, pp. 223–226, 2017. DOI: 10.1145/3053600.3053653. [Online]. Available: http://dx.doi.org/10.1145/3053600.3053653.