UNIVERSITY OF OSLO

MASTER'S THESIS

Black-box performance modelling and analysis of microservice systems

Author:
Magnus Nordbø

Supervisor:
Hui Song

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

in the

Faculty of Mathematics and Natural Sciences
Department Of Informatics

July 3, 2023

Declaration of Authorship

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

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

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

Dave Barry

UNIVERSITY OF OSLO

Abstract

Faculty of Mathematics and Natural Sciences
Department Of Informatics

Master of Informatics

Black-box performance modelling and analysis of microservice systems

by Magnus NORDBØ

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

Acknowledgements

Contents

List of Figures

List of Tables

AI	Artificial Intelligence
ML	Machine Learning

TABLE 1: Abbreviations

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Keywords

Observability
Microservices
Signal-to-noise ration

1.2 Abstract

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

1.3 Research area and questions

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

1.4 Approach

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

then analyzed both manually and mathematically, and preprocessed to reduce noise and extract important features. Several methods of preprocessing and classifying were quantitatively compared. Finally, the results of the analysis were discussed and conclusions drawn.

1.5 List of important terms

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

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

Chapter 2

Background

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

2.1 Microservices

2.1.1 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 arises 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.

2.1.2 The problems with monolithic software

The core issue that permeates all the main problems 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.
Scalability refers to how easy it is to increase the workload capacity of the software. 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.

Chapter 3

Methodology

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

3.1 Background

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

3.2 System selection and creation

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

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

3.2.1 Cloud solutions

Cloud computing is in several ways a natural fit for this project. Microservices as a concept is formulated with cloud computing in mind. Cloud service providers tend to provide some microservice-specific services. The three biggest players in cloud services are Amazon, Microsoft and Google [4]. 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

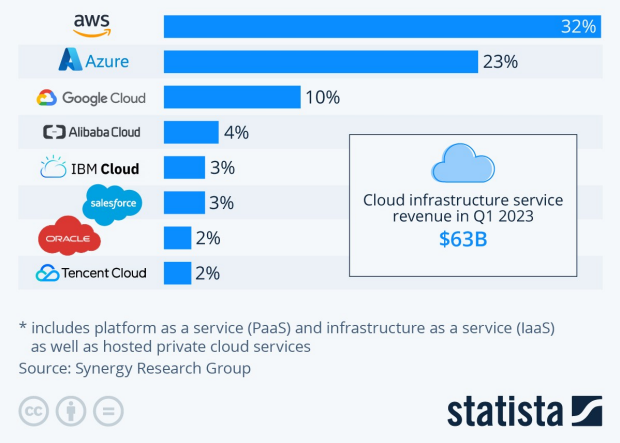

FIGURE 3.1: Worldwide market share of leading cloud infrastructure providers in Q1 2023. Source: Statista

that I would be granted the necessary amount of credits.

3.2.2 Local machine

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

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

3.2.3 Candidates

DeathStarBench

DeathStarBench is an open-source benchmark suite developed by Cornell University for research purposes. It features a total of five complete services: A social network, a media service, hotel reservation, an e-commerce site, a banking system, and a drone coordination system for drone swarms. If chosen as the system for this project, we would have focused on one of the first three services. They are in a more complete state than the rest according to the project's GitHub page. DeathStarBench makes a compelling choice as it seems to be quite complete and sophisticated, as well as being made specifically for testing. [5]
However, there were two main issues with it.

- System requirements and complexity

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

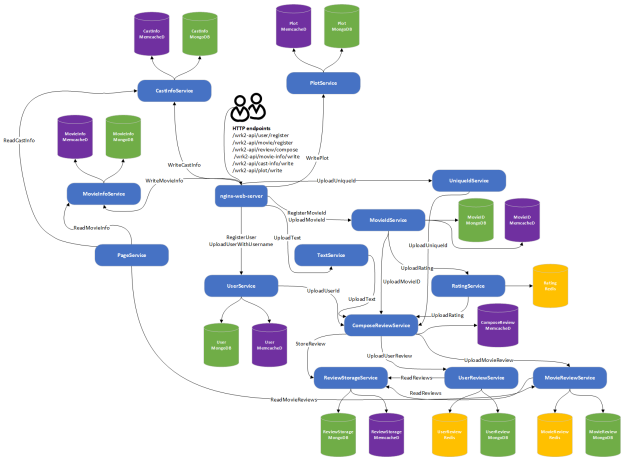

FIGURE 3.2: The workload architecture for DSB's media service

- 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. [6], [7], [8]

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". [9] 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. [5] The smaller, less complex nature of this project means that it could feasibly run on our available home machines. It is composed of 14 services running in individual containers. The messaging between them is handled by RabbitMQ, which runs as its own container as one of the 14.

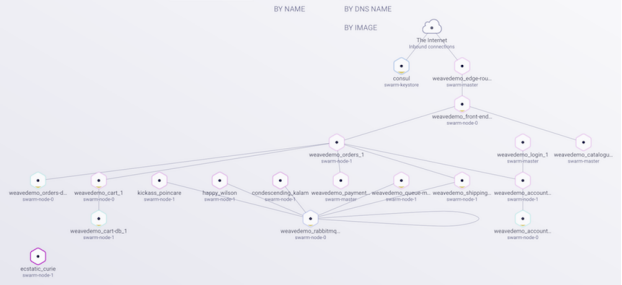

FIGURE 3.3: The architecture of the sock shop shown using WeaveScope, Weaveworks' own mapping tool.

Sockshop comes with some strong benefits:

- Forgiving system requirements

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

- Centralized logging

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

There are also some major downsides to the Sockshop.

- Poor documentation

The website for the project [9] 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.

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 [10].
- Implement imputation of missing values, which entails using an algorithm to make an informed guess about a plausible value based on existing values in similar positions.
- Perform dimensionality reduction on the data. Reduction algorithms like Singular Value Decomposition (SVD) are good for removing dummy data like NaNs.

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

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

4.2 Series length

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

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

There are two main ways to deal with this problem:

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

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

4.3 Feature selection

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

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

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

4.3.1 Low variance or monotonic features

The variance numbers shown in figure 4.1 are scaled. This means the variance value for each variable is proportional to the mean value of that variable. Of the 579 variables in the dataset, 110 of them have a variance of exactly 0. This means they are entirely unaffected by both time and stress testing. Such variables are entirely useless and simply dilute the information in the data. Another issue is monotonically increasing variables. 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. **Monotonically increasing variables can still be very useful in machine learning if treated properly, but it is out of the scope of this project. This is because there are two main ways of fitting such variables into machine learning model like the ones used in this project: One is differencing, where the changes in value instead of the absolute number is taken into account.** The more interesting problems arise from variables with low variance.

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

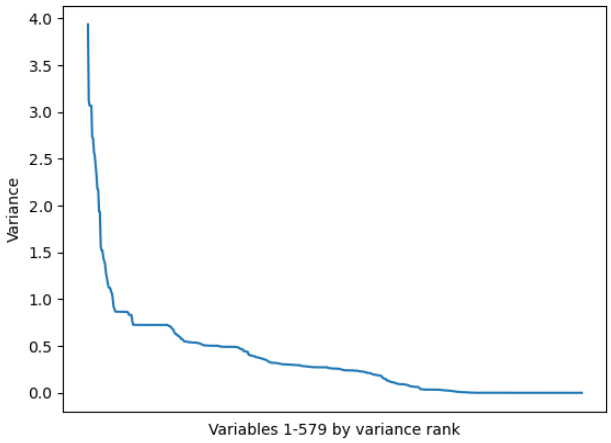

FIGURE 4.1: Scaled variance of all the variables

4.3.2 Variable vs feature

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

Bibliography

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

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

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

[4] 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).

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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).