Predicting resource usage on a Kubernetes platform using Machine Learning Methods

Arvid Gördén <arvid@bag.org>

Faculty of Health, Science and Technology

Master thesis in Computer Science

Second Cycle, 30 hp (ECTS)

Supervisor: Prof. Javid Taheri, University of Karlstad, Karlstad, SWE <javid.taheri@kau.se>

Examiner: Senior Lecturer Mohammad Rajiullah, University of Karlstad, Karlstad, SWE <mohammad.rajiullah@kau.se>

Karlstad, June 15th, 2023

Abstract

Cloud computing and containerization has been on the rise in recent years and have become important areas of research and development in the field of computer science. One of the challenges in distributed and cloud computing is to predict the resource utilization of the nodes that run the applications and services. This is especially relevant for container-based platforms such as Kubernetes. Predicting the resource utilization of a Kubernetes cluster can help optimize the performance, reliability, and cost-effectiveness of the platform.

This thesis focuses on how well different resources in a cluster can be predicted using machine learning techniques. The approach consists of 3 main steps: data collection, data extraction and pre-processing, and data analysis. The data collection step involves stressing the system with a load-generator called Locust and collecting data from Locust and collecting data from Kubernetes with the use of Prometheus. The data pre-processing and extraction step involves extracting relevant data and transforming it into a suitable format for the machine learning models. The final step involves applying different machine learning models to the data and evaluating their accuracy.

The results of this thesis illustrate that machine learning can work well for predicting resources in a cluster based on how stressed the system is and that the best performing machine learning model tested was Support Vector Machine with a polynomial kernel.

Keywords

Master thesis, Kubernetes, scaling, resource management, horizontal pod autoscaler, vertical scaling, machine learning

Sammanfattning

Cloud computing och containerisering har ökat de senaste åren och har blivit viktiga områden för forskning och utveckling inom datavetenskap. En av utmaningarna inom distribuerad och cloud computing är att förutsäga resursutnyttjandet av de noder som kör applikationerna och tjänsterna. Detta är särskilt relevant för containerbaserade plattformar som Kubernetes. Att förutsäga resursutnyttjandet av ett Kubernetes-kluster kan hjälpa med att optimera plattformens prestanda, tillförlitlighet och kostnadseffektivitet.

Denna avhandling fokuserar på hur väl olika resurser i ett kluster kan förutsägas med hjälp av maskininlärningstekniker. Tillvägagångssättet består av 3 huvudsteg: datainsamling, dataextraktion och för-processering, samt dataanalys. Datainsamlingssteget innebär att stressa systemet med en load-generator som heter Locust och samla in data från Locust och även samla in data från Kubernetes med hjälp av Prometheus. Steget för för-processering och extrahering av data innefattar att extrahera relevant data och omvandla den till ett lämpligt format för maskininlärningsmodellerna. Det sista steget innefattar att tillämpa olika maskininlärningsmodeller på data och utvärdera deras noggrannhet.

Resultaten av denna avhandling demonstrerar att maskininlärning kan fungera bra för att förutsäga resurser i ett kluster baserat på hur stressat systemet är och att den bäst presterande maskininlärningsmodellen som testades var Support Vector Machine med en polynom-kernel.

Nyckelord

Masterarbete, Kubernetes, scaling, resurshantering, horisontell pod autoscaler, vertikal scaling, maskininlärning

Acknowledgements

I want to express gratitude to my supervisor Javid Taheri at Karlstad University for his guidance, knowledge, and attitude throughout this whole project. He has been supportive and uplifting and this thesis would not have been possible without his knowledge and advice.

I would also like to thank my friend Adam Rubak for being someone to talk to and bounce ideas with throughout the project.

Finally I would like to thank friends and family who have been supportive and helped me in stressful times.

Contents

Chapter 1

Introduction

This chapter introduces the project and presents some background on the topic while also describing the research problem and presenting a research question.

1.1 Background

Microservices is a software architecture that has risen in popularity over the years. It is a design pattern that splits a complex system into smaller, independent, and loosely coupled services. Each service handles a specific functionality or domain and interacts with other services through well-defined interfaces. This way, microservices enable faster and more reliable delivery of applications, as each service can be developed, tested, deployed, and scaled independently. Microservices also improve fault tolerance and resilience of the system, as failures in one service do not affect the whole system. Containerization is a technology that facilitates the implementation of microservices, as it allows for creating isolated and lightweight environments for running each service [8].

Containerization is a way of virtualizing applications so that they run in their own environments without interfering with each other, but still use the same operating system resources. A container is like a package that has everything an application needs to run, such as libraries, data, configuration files, etc [11] [29]. Containerization differs from virtualization by focusing on the operating system layer instead of the hardware layer. This means that containers are more lightweight and efficient than virtual machines. Some examples of technologies that enable containerization and container management are: Docker, Docker swarm, Kubernetes, Red Hat OpenShift, etc [29].

Kubernetes allows for autoscaling to be done and this is possibly through the Horizontal Pod Autoscaler. Scaling in a cloud system can primarily be done by two methods: 1) Horizontal Scaling. This method involves adding or removing entire containers or virtual machines. The advantage of this method is that it can handle large variations in resource demand. The disadvantages are that it is slower because it

takes time to deploy a new virtual machine and that the time delay is not consistent. 2) Vertical Scaling. This method involves changing the size of the existing resources, such as the CPU or memory allocated to a deployed virtual machine and allows for faster and smoother resource control. The drawback of this method is that it has a limited range of possible control actions [10].

1.2 Problem Description

One of the challenges of developing applications using a microservice architecture is predicting the number of resources needed for each service. Resources include CPU, memory, disk space, network bandwidth, etc. If a service needs more resources, it may become faster, more responsive, and more successful. On the other hand, if a service has too many resources, it may waste money and energy.

To address this challenge, some microservice architectures use autoscaling techniques to adjust the resources allocated to each service based on the demand. Autoscaling can be reactive or proactive. Reactive autoscaling monitors the performance metrics of each service and scales up or down when a threshold is reached. For example, if the CPU utilization of a service exceeds 80%, more instances of that service can be created to handle the load. Reactive autoscaling is simple but may introduce latency or downtime during scaling operations.

Proactive autoscaling tries to predict the future demand of each service and scales up or down in advance. For example, if a service expects a surge of traffic during a particular time of the day, more instances of that service can be created beforehand to avoid congestion. Proactive autoscaling is more complex but may reduce latency or downtime during scaling operations [18] [25].

1.3 Research question

Since little of the related work have studied pro-active scaling for vertical scaling one research question could be:

a) How well can machine learning algorithms be used to accurately predict resource utilization in a microservice application?

This would give insight to how accurate predictions on the vertically-scalable resources is and in turn whether or not these predictions could be used for autoscaling.

Another research question would be to examine the amount of datapoints needed:

b) What fraction of data could be used for training to accurately predict resources?

The answer to this question could give insight to how many training points need to be considered in order to receive accurate predictions on the resources in a Kubernetes

cluster.

1.4 Thesis Goals

This thesis' purpose is to achieve four main objectives. The first objective is to create testing scenarios that are reliable and provide good, validated data. Second is to identify and choose suitable parameters that can deliver the best system performance. Thirdly is to implement a way of collecting the data from the testing scenarios, transforming it to an appropriate form and apply machine learning to it. Finally is to evaluate the training and predictions and compare the machine learning methods to each other.

1.5 Ethics and Sustainability

Microservices and cloud architecture pose some sustainability challenges that need to be addressed. For example, microservices may also increase the number of servers, network traffic, and data transfers required to run an application, which leads to higher energy consumption and carbon footprint and cloud-architecture often involves transferring data across long distances and regions, which adds to the energy cost and environmental impact.

1.6 Methodology

The research methodology of this thesis consisted of conducting literature research to identify different problems and solutions when scaling in a Kubernetes system but also to get a clear overview of the research area.

Furthermore, qualitative and quantitative research was conducted. The qualitative research consisted of learning about the different tools that were used throughout the project such as Kubernetes system and locust load-testing, and more.

The implementation phase included applying the different tools to obtain some results. This phase involved running experiments, collecting data, evaluating the data.

The quantitative research consisted of observing and analyzing the data with help of machine-learning and then evaluating this data.

1.7 Delimitations

The focus of this thesis is to examine how well different machine learning methods perform when predicting the resource utilization of a Kubernetes cluster and it does not compare this to any other method of scaling or predicting resources. The thesis will also not inspect how the predicted results perform in a live system.

1.8 Outline

This thesis is structured in 5 chapters. The first chapter introduces the topic and provides context for the thesis. Chapter 2 provides the necessary background information about tools, environments, methods and concepts for the rest of the thesis.

Chapter 3 describes the choices, methods and implementation details of the work that was done for this project.

In chapter 4 the results from the work performed is presented, evaluated and discussed.

Finally, chapter 5 summarizes the thesis and suggests future work that can be done.

Chapter 2

Background and Related Work

This chapter presents background and concepts that are relevant for the rest of the thesis.

2.1 Microservices

The Microservice architecture has risen in popularity over the years and has become one of the main ways of developing applications. The microservice architecture is a design pattern that divides an application into smaller, independent, and loosely coupled services. Each service is responsible for a specific functionality or domain and communicates with other services through interfaces. The microservice architecture enables faster development, testing, deployment, and scaling of applications, and also has improved fault tolerance and resilience [8].

2.2 Containerization

Containerization is a form of virtualization that run applications in an isolated environment while sharing the hardware resources of the operating system they run on. A container is a runnable instance of an image that contains all of the requirements for it to operate such as libraries, data, configuration data, etc [11] [29].

The focus of containerization is to abstract the operating system level instead of virtualizing the hardware stack by using virtual machines. Some containerization and container orchestration technologies that exist and are used is: Docker, Docker swarm, Kubernetes, Red Hat OpenShift, etc [29]. The differences between virtualization and containerization has been illustrated and can be viewed in figure 2.2.1.

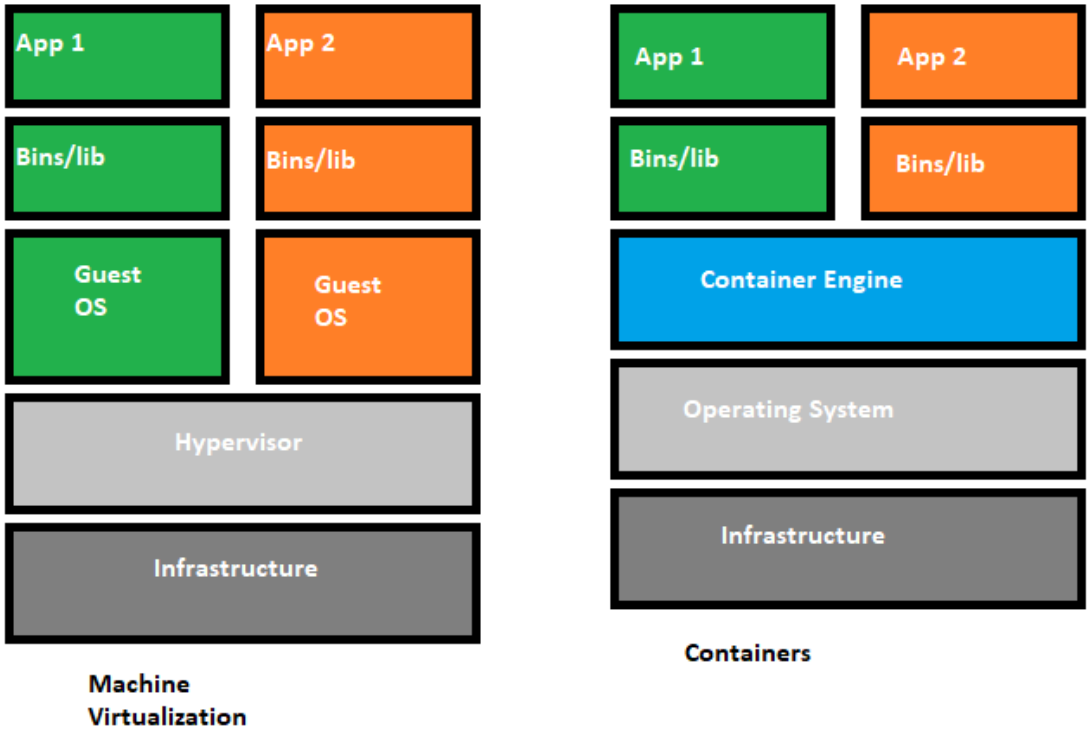

Figure 2.2.1: Virtualization compared to containerization

2.2.1 Kubernetes

Kubernetes is a portable, extensible and open-source platform that orchestrates and manages containers. Some of this orchestration and management involves automating many manual tasks such as: deploying and scaling containerized applications. Kubernetes was originally developed by the company Google but in 2015 the project was donated to the Cloud Native Computing Foundation (CNCF) [9] [26].

Some features and functionality that Kubernetes provides is as follows:

- **Service discovery and load balancing:** A container in Kubernetes can be accessed either by its DNS name or by its unique IP address. Kubernetes provides load balancing and network traffic distribution mechanisms to ensure the stability of the deployment under high demand.
- **Storage orchestration:** Kubernetes enables the automatic integration of various storage solutions, ranging from local devices to public cloud services.
- **Automated rollouts and rollbacks:** Kubernetes enables you to define the desired state of your deployed containers and to coordinate the actual state with the desired state in a controlled manner. For instance, you can instruct Kubernetes to create new containers for your deployment, to terminate existing containers and to transfer their resources to the new containers.
- **Automatic bin packing:** By providing Kubernetes with a cluster of nodes that

it can use to run containerized tasks it can be told how much CPU and memory each container need. Kubernetes then fit the containers to the nodes to make the best use of the resources.

- **Self-healing:** Kubernetes guarantees the reliability and availability of containers by performing various actions based on their status and health. These actions include restarting failed containers, replacing containers with new ones, terminating unresponsive containers, and delaying their exposure to clients until they are fully functional.
- **Secret and configuration management:** Kubernetes can store and manage information that is sensitive, such as passwords, OAuth tokens, and SSH keys. Secrets and application configuration can be deployed and updated without rebuilding the container images, and without exposing the secrets in the stack configuration [9].

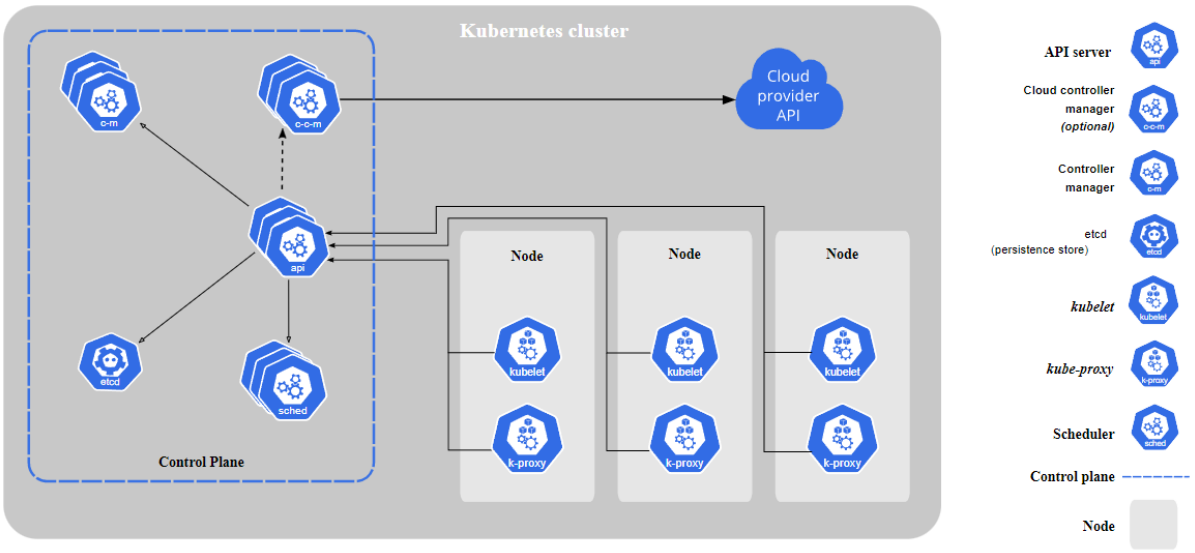

Figure 2.2.2: A diagram showcasing an overview of a Kubernetes cluster

The Kubernetes architecture is made in such a way that when it is deployed, you get a cluster, which is demonstrated in figure 2.2.2. A cluster is made up of two different components, these are nodes and the control plane. The nodes are a set of worker machines that host and run containerized applications. Each Kubernetes cluster has at least one worker node. The purpose of the worker nodes is to host Pods, which is a group of one more containers that have shared storage and network resources. The cluster's worker nodes and pods are managed by the control plane, which guarantees that they properly function and coordinate. When in a production environment, the control plane is usually run across multiple computers and a cluster usually runs multiple nodes which provides fault-tolerance and higher availability [9] [11].

The components that build up the control plane can make global decisions about the cluster and detecting and responding to cluster events. These components can be

run anywhere in the cluster but for the sake of simplicity, the set up scripts usually start all the control plane components on the same machine and they do not run user containers on this machine. The **kube-apiserver** is a component that simply exposes the Kubernetes API and is the main implementation of the API server that serves as the front end for the control plane. It is designed to scale horizontally. **etcd** is another component of the control plane which is a key-value store that is made to be consistent and highly-available. It is used as the backing store for all of the cluster data. The **kube-scheduler** component watches for newly created pods that has not yet been assigned to a node and then assigns it to a node. Several factors are taken into account when a scheduling decision is made, such as: resource requirements, hardware, software, and policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines. **kube-controller-manager** runs controller processes. Controllers in Kubernetes are control loops that watch the state of the cluster and make or request changes when it is needed. The controllers strive to move the clusters state closer to a desired state. Each controller is a separate process but all of them are compiuled into a single binary and are run as a single process, this is to reduce complexity [9] [23]. Some different types of controllers are:

- Node controller: Notices and responds to when a node goes down.
- Job controller: Watches for job objects(one-off tasks) and creates pods that run those tasks to completion.
- EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
- ServiceAccount controller: Create default ServiceAccounts for new namespaces.

The final component of the control plane is the **cloud-controller-manager**. Its purpose is to embed cloud-specific control logic. It lets you link the cluster to a cloud provider's API and then separates the components that interact with the cloud platform from those that only interacts with the cluster.

The components for the nodes run on every node and maintains running pods and provides it with the Kubernetes run-time environment. **kubelet** is a so called "node agent" and it runs on each node in the cluster. Its objective is to make sure that the containers are running in a pod. Kubelet takes a set of the PodSpecs provided to it and makes sure that the containers that are described in the PodSpecs are running and are healthy. It does not manage containers that have not been created by Kubernetes. **kube-proxy** is a network proxy and runs on each node in the cluster and implements a part of the Kubernetes concept that exposes an application running on a set of pods as a network service. It maintains network rules on the nodes and the rules allow communication with the pods from network sessions both inside and outside of the cluster. **Container runtime** is the software which is responsible for running the containers [9].

2.3 Vertical and Horizontal scaling

Scaling resources in a cloud system can primarily be done in two ways: 1) Horizontal Scaling. This is when whole containers or virtual machines are allocated or deallocated and implies that the allocation changes are made in discrete steps. The benefit of using this method is that very big changes can be made and thus works over a large range. The downsides, however, is that it is slower since deploying a new virtual machine takes time and that time delay can be varying. 2) Vertical Scaling. This method is when the size of the already allocated resources are changed, for example like changing how much CPU or memory is allocated to a virtual machine that has already been deployed and allows for quicker and smoother control of the resources. The downside of this method is that the range of possible control actions is limited [10].

2.4 Google Online Boutique

The Google online boutique demo application is a web-based e-commerce platform whose purpose is to showcase the use of the microservice architecture and cloud-native technologies. The application consists of a frontend web server that interacts with multiple back-end services, each of these performs a specific function such as product catalog, checkout, payment, shipping, recommendation, etc. The application is designed to be scalable, resilient, and observable, using various tools and frameworks such as Kubernetes, Istio, Prometheus, Grafana, Jaeger, and OpenTelemetry. The application can be deployed on different cloud platforms such as Google Cloud Platform, Amazon Web Services, and Microsoft Azure, as well as on-premise or hybrid environments. The application serves as a realistic example of how to build and operate a modern e-commerce system using microservices and cloud-native technologies [6]. The different services that make up the Google online boutique demo are:

- **frontend**: Exposes an HTTP server to serve the website. Does not require signup or login and generates session IDs for all of the users automatically.
- **cartservice**: Stores the items in the user's shopping cart in Redis and retrieves it.
- **productcatalogservice**: Provides the list of products from a JSON file and ability to search products and get individual products.
- **currencyservice**: Converts one amount of money to another currency. Uses real values fetched from European Central Bank. It's the highest QPS service.
- **paymentservice**: Performs a mock payment with the credit card info and the amount provided and generates a transaction ID.
- **shippingservice**: Calculates the shipping cost for the items in the cart and sends them to a fake address.
- **emailservice**: Sends a mock email to users confirming their order.

- **checkoutservice**: Retrieves user cart, prepares order and orchestrates the payment, shipping and the email notification.
- **recommendationservice**: Recommends other products based on what's given in the cart.
- **adservice**: Provides text ads based on given context words.
- **loadgenerator**: Continuously sends requests imitating realistic user shopping flows to the frontend [6].

The architecture and relationship between these services are demonstrated in figure 2.4.1

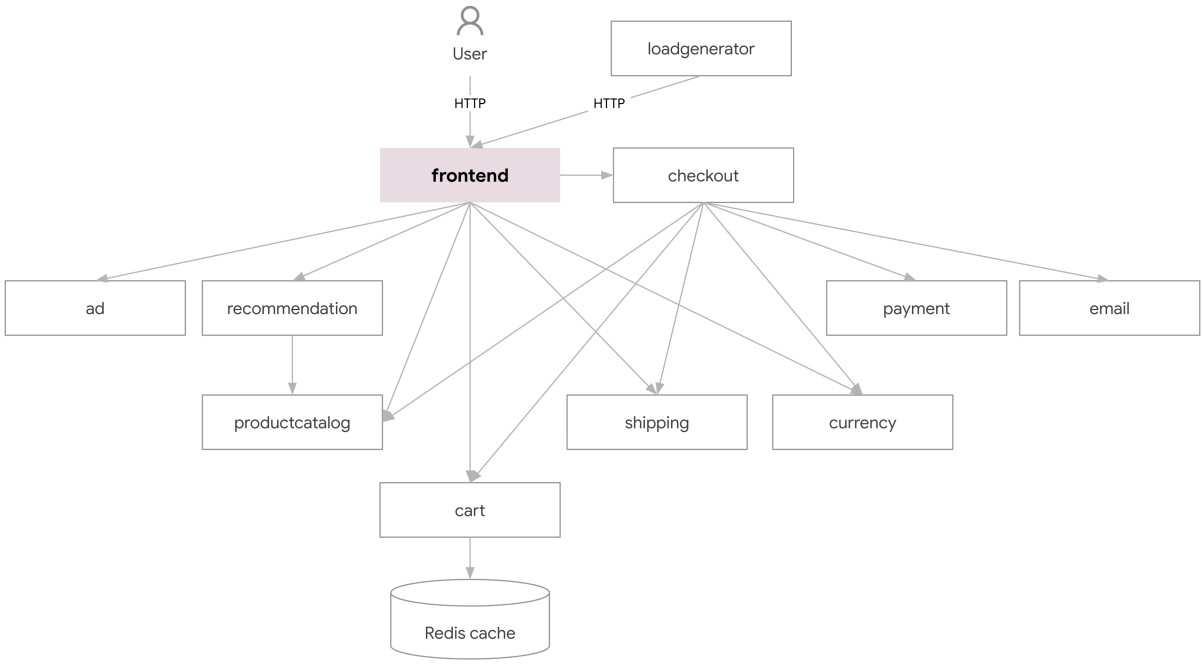

Figure 2.4.1: Architecture of Google Online Boutique

2.4.1 Locust

Locust is an open-source evaluation and load testing tool for websites. It is based on Python and is made to be scriptable, scalable and easy to use. Locust allows the user to simulate a high number of concurrent users and then measures how the application reacts to the generated traffic. It works by having user behavior defined in Python code. The user can write their own custom classes that inherit from the Locust class and can then define tasks that would simulate different actions that can be performed in the application that is being tested. As an example, a task could be to visit a website and then fill out a form on the website and then submit it. The user can specify how often a task should be executed and how many users that should perform each task. A number of instances of the classes are then spawned by Locust and distributes them across different machines that are available. Locust also has a web-based user interface that lets the user monitor and modify the test as it is running. The user can see the

test results and statistics in real-time in the browser. Locust can also be run without the web interface, which makes it convenient for continuous integration and delivery testing [17] [27].

3 different charts are available to the user in the interface when running a test. These are total requests per second that both graphs the rate of requests and the rate of failures with two different curves, Response times (milliseconds) with two different curves for the median response times and the 95th percentile response times, and finally the number of simulated users [27].

In Google Online Boutique, the service called loadgenerator is where Locust is hosted.

2.5 Prometheus

Prometheus is a toolkit for monitoring and alerting systems that is open-source and free to use. It was created by SoundCloud in 2012 and has been adopted by many other companies and organizations. Prometheus is not tied to any specific company and is developed and maintained by a community of developers and users. To reflect this, Prometheus became part of the Cloud Native Computing Foundation in 2016 as the second project to join after Kubernetes [15].

A multi-dimensional data model is provided by Prometheus through collecting data in the form of time series from different data sources at a set interval and then storing it locally on a disk where the data is identified by metric name and key/value pairs. This model is very similar to the way that Kubernetes organizes its metadata. Prometheus uses a pull strategy over HTTP so that it can collect the real-time metrics in a time series database and uses a query language specifically made fro Prometheus called PromQL [15] [22].

The different types of metrics provided by Prometheus can be divided in 4 different types:

- **Counter**: A cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.
- **Gauge**: A metric that represents a single numerical value that can arbitrarily go up and down. Gauges are typically used for measured values like temperatures or current memory usage, but also ”counts” that can go up and down, like the number of concurrent requests.
- **Histogram**: A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

- **Summary**: Is similar to the histogram metric and samples observations. While it provides a total count of observations and a sum of all observed values, it also calculates configurable quantiles over a sliding time window [15].

Prometheus promotes a whitebox monitoring approach which aids in the administration of the internal technicalities about the state of the microservices. The Prometheus ecosystem mainly consists of five components: 1) The Prometheus server which scrapes metrics from jobs and aggregates and records numberic time series. 2) Client libraries that matches the applications language. 3) Alertmanager which manages alert notification, grouping and inhibition. 4) Exporters that distributes existing metric from the third-party systems. 5) Grafana which can pull metrics and display so called Dashboards of the metrics [22].

2.6 Libraries

The **Kubernetes API** is a REST API. All operations and communications between components, and external user commands are REST API calls which the API Server handles. Everything on the Kubernets platform is treated as an API object and thus have an entry in the API. Many client libraries for different languages are available for the API which handles common tasks such as authentication [1].

Matplotlib is a plotting package for Python that has the capability of generating quality graphs. The design is made in such a way that both simple and complex plots can be made with the use of a few commands. It also has capabilities to integrate with both the Pandas and NumPy libraries [2].

NumPy is a Python package that provides tools for scientific computing and data analysis and is a fundamental package for working with multidimensional arrays and matrices. It also offers a wide number of choices of mathematical functions, random number generators, linear algebra routines, etc. and can be used for image and signal processing, machine learning, statistics, physics and more [13].

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. It provides data structures designed to make working with relational or labeled data easy and intuitive. These data structures are called DataFrame, which is used for 2-dimensional data, and Series, which is used for 1-dimensional data [3] [16].

Scikit-learn is an open-source machine learning library for Python that includes algorithms that are based on classification, regression, and clustering and has support for supervised and unsupervised learning. In addition to its core functionality, it offers a range of features that support different aspects of the machine learning workflow, such as data pre-processing, model selection, model evaluation, and more [19] [20].

2.7 Machine Learning

Machine Learning is a branch of artificial intelligence where computers learn from data without being explicitly programmed. This is done by utilizing algorithms that iteratively improve the predictions that are made by incorporating feedback from the data [4]. By extending on classical techniques from statistical modeling, modern machine learning has become a powerful tool because of the increased volume of data, exponential growth in computational power and advances in the design of the algorithms. These algorithms are typically referred to as ”models” and the choice of model for a problem is based on the characteristics of the dataset and the output that is desired. Machine learning can usually be divided into supervised learning and unsupervised learning. Supervised learning is when the model is being given a collection of input data and the corresponding correct output data. Unsupervised learning is when the model trains itself on the data provided and tries to find patterns or structure without any guidance [12].

2.7.1 Support Vector Machine

Support Vector Machine(SVM) is a supervised machine learning model that can be used for both classification and regression problems. The main idea of SVM for classification problems is to create a hyperplane between categories that maximizes the margin between them [5] For regression the idea is similar, however, now what is being maximized is the amount of points that fit within the margin [28]. SVM can make use of different so called kernels to fit the data, like linear or different orders of polynomials [5].

2.7.2 Multilayer Perceptron

Multilayer Perceptron(MLP) is a type of feedforward artificial neural network that can learn complex functions from input-output pairs. It has multiple layers of nodes, each with a nonlinear activation function, that connect in a directed graph. The first layer is the input layer, which receives the input features. The last layer is the output layer, which produces the output values. Between them, there can be one or more hidden layers that perform the different computations. To train an MLP, a supervised learning technique called backpropagation is used. It adjusts the weights and biases of the nodes by minimizing the error between the predicted and actual outputs. The error can be measured by different metrics, such as RMSE. An MLP can handle both linear and nonlinear data [14].

2.8 Related Work

Previous research done on this topic proposes new AI based models for autoscaling.

One study proposed a proactive framework based on machine learning. The proposed solution is an autoscaler that scales horizontally. 4 different machine learning models were compared: ARIMA, LSTM, Bi-LSTM and transformer based models. The model that performed the best out of all of them was the Informer model which is a transformer based model [21].

Another paper presents an system that centralizes cluster autoscaling and resource management. It offers a low-latency automated system for managing containers and assessing resiliency for dynamic systems. The system predicts the load using a Bi-LSTM and periodically updates the autoscaling policy for cluster performance. The system is proactive and performs both horizontal and vertical scaling. The study also compared 3 different algorithms: ARIMA, Holt Winters and Bi-LSTM where Bi-LSTM performed the best [7].

A third study proposes a proactive horizontal scaling solution that makes use of different machine learning methods to optimize resources. This is done by calculating the accuracy of each model and choosing the one with the highest accuracy. 3 models were used, namely: HTM, LSTM and AR where AR was used the most and LSTM short thereafter [24].

Many of these studies propose similar solutions and are usually only for horizontal scaling. Many machine learning methods are also the same, leaving a gap for examining other models. Usually, only one resource metric is also used for predictions which is CPU usage. For these studies, the system that is being tested is primarily small and does not contain several different deployments.

Chapter 3

Approach and methodology

In this chapter, the different methodologies and environments that have been used are described. Furthermore, the implementation of the methods are described in this chapter.

3.1 Environment

3.1.1 Kubernetes cluster topology

The Kubernetes cluster used during this project was hosted on vSphere that was running Ubuntu. It consisted of 4 worker nodes and 1 master node. The specifications for the nodes are presented in Table 3.1.1. The virtual machine running Kubernetes was available on Karlstad University and could be accessed through the use of VPN and SSH.

Node	CPU	Memory	SSD
1	8 vCPU	8GB	20GB
2	8 vCPU	8GB	20GB
3	8 vCPU	8GB	20GB
4	8 vCPU	8GB	20GB

Table 3.1.1: Table of the nodes used

3.1.2 Demo application

The demo application that was used for testing and measuring is called Google Online Boutique and is described more in depth in Chapter 2.4. The services are written in different programming languages like Python, Go, Java, and Node.js. The primary objective of the demo application is to showcase how to deploy, test, and monitor a microservice application with Kubernetes and Google Cloud.

3.1.3 Programming environment

Python was the programming language that was chosen for the work of this thesis. The reason for this choice is that Python is a language that is quite easy to work with and writing and prototyping code is relatively fast. Furthermore Python has a plethora of libraries and packages available that can ease and add to development. Many machine learning libraries are also available for Python which was a contributing factor in the decision.

The libraries used during the work were:

- **requests**: a library for sending HTTP requests and handling responses
- **numpy**: a library for numerical computing and manipulating arrays
- **pandas**: a library for data analysis and manipulation
- **matplotlib**: a library for creating plots and visualizations
- **kubernetes API**: a library for interacting with the Kubernetes cluster management system
- **scikit-learn**: a library for machine learning and data mining

A more in-depth description of these libraries can be found in chapter 2.6 For writing the code, Visual Studio Code (VScode) was used as the integrated development environment. It is fast and light-weight and has support for many different programming languages and can run code directly from the program itself. It also has support for different features like syntax highlighting, debugging, testing version control, etc. The features of VScode can also be extended through the use of its extension library.

Jupyter notebook was also used to do calculations and better display the graphs and tables from packages like pandas and matplotlib as Jupyter has support for outputting this directly.

3.2 Method and approach

3.2.1 Data collection process

The collection of the data can be separated into different parts.

Firstly, before swarming the cluster, the Kubernetes API needed to be invoked in order to make sure that everything is running correctly and/or to make changes to the cluster.

For the API library to work on a different system than that on which the cluster is hosted on the kube_config must be loaded into the library. The kube_config file was obtained by running the command `kubectl config view --raw` on the system where the cluster is hosted. This command outputs the kube_config file with the raw authentication

tokens. The Kubernetes API, and all other python packages, were installed through the use of Pythons package manager called PIP.

A python file was created for handling the Kubernetes API communication. Within this file, two classes were created. One called ”Patcher” whose main objective and functionality was to perform so called patch operations on Kubernetes objects such as setting the amount of replicas or setting resources like CPU or Memory limits. The second class was called ”Reader” whose functionality was to perform ”get” operations like getting the amount of replicas for one pod or to watch the cluster for changes and tell the program to halt until a change was made or a change was registered. Both of these classes needed to load in the kube_config file to be functional.

Secondly, the Locust swarm needed to be started. The Locust web UI has an internal API that was visible in the network tab when using developer tools in a browser. This was utilized when making the management of locust swarms available programmatically. In order for this to work the python requests library was used which simplifies the usage of HTTP POST and HTTP GET. The class that handled Locust swarms was called ”LocustManager” and it contained 3 methods, including its constructor. These were: start_swarm which composed a dictionary containing the user count, spawn rate, host and total run time for the swarm which it then sent to Locust with an HTTP POST, download_report which downloaded the results from the swarm and saved it as an HTML file. This class together with the Patcher and Reader class was then used in a python file whose purpose was to start new tests and download the data. Since many tests would be run after each other, blueprints for different collections of tests were made. These were defined in a YAML-file where the user can define specifications for patching the Kubernetes cluster and define the specifics for the locust swarm such as run time, user count and spawn rate. An example:

```
test_cases:
- k8: {}
  locust:
    host: http://172.16.19.100:32000
    run_time: 181
    spawn_rate: 100
    user_count: 100
  test_number: 0
- k8: {}
  locust:
    host: http://172.16.19.100:32000
    run_time: 181
    spawn_rate: 100
    user_count: 100
  test_number: 1
```

A python module for quickly creating blueprints was also created. Which would iterate over a number of defined user counts and create a blueprint. The run time was

calculated by using the formula $round(users/locust\_spawn\_rate) + 180$. The 180 is added to allow the Kubernetes cluster to stabilize.

When a swarming round was done running and it had been downloaded the HTML-file needed to be converted into JSON in order to make the extraction and parsing of the data easier. The code for this was provided by the project supervisor and it uses etree from the lxml library for parsing the HTML.

After each swarm, data from Prometheus needed to be collected as well in order to obtain information on CPU usage, memory usage, network usage, etc. The code for this was also provided by the supervisor and uses a Python library for Prometheus where data can be collected by the use of queries with the query language PromQL. The query results are returned as a dictionary and the results were put in a JSON-file.

3.2.2 Data pre-processing

When the gathering of the data was done the correct data had to be chosen and extracted, and put into a form that would be of benefit for the machine learning models.

This was done by loading the JSON-files for each test into a dictionary and placing the desired points of data into a Pandas dataframe together with the name of the pod and the user count for that particular swarm. The data points that were extracted:

- CPU usage
- Memory
- Memory Irate
- Network receive
- Network transmit
- Replicas

First, the data from the Locust and Prometheus JSON-files was retrieved. This was done by loading the JSON into a python dictionary and then searching through the dictionary for each desired metric and creating a new dictionary with all the datapoints for each metric. The new dictionaries were then converted into Pandas dataframes by utilizing the from_dict method that is present in the Pandas library. After this, all of the dataframes with the different metrics were merged into one and user count from the Locust dictionary was inserted into the dataframe as well.

When this was done a second dataframe was made that would contain the normed data points. The normalization method that was used was min-max normalization which uses the formula: $x' = x - x_{min}/x_{max} - x_{min}$, where x' is the normed value, x is the actual value, x_{min} is the minimum value of the data and x_{max} is the maximum value of the data. This method of normalization turns the data with the minimum value into 0 and the maximum value into 1 and the rest in between.

These were then saved into a CSV-file for quick and easy access at any time.

Before applying machine learning to the data, the pod names needed to be encoded to a non-string value since the machine learning methods used from scikit-learn cannot use string values. This was achieved by using the class LabelEncoder which is build in in scikit-learn and transforms a string into an integer which it can later decode as well.

3.2.3 Machine Learning

Three different machine learning methods were used and compared to predict resources. These were:

- Multilayer perceptron
- Support Vector Machine with linear kernel
- Support Vector Machine with polynomial kernel

The rationale behind choosing these is because when using scikit-learn these are quite easy to train and not that many parameters need to be considered but also because there is still a gap in the related studies when using machine learning methods.

When using machine learning in this project, the input variable chosen was amount of users and the features were: CPU, Memory, Memory irate, Network transmit and Network receive. The reason these were chosen is because they are relevant resources that could be bottlenecks for different pods. In retrospect, the memory irate feature could have been ignored as this is a result from a Prometheus query that calculates the per second rate of change and is not an actual resource in the system.

When training was being done, a unique model was trained for each combination of feature, pod name, user step and machine learning algorithm. What this means is that the input would always be user count and the output would be one of the features mentioned earlier. This was done to achieve a higher accuracy for the models as this have previously been tested by using all features as output and it had resulted in abnormally low accuracy. What user step means in this context is that every datapoints where every 200, 300, 400.. etc. users coincided with the user count would be chosen as training data. This was done to evaluate how much of the training data would be needed to be used and stilled achieve good accuracy on the predictions. The different users steps that were used for training was: 200, 300, 400, 500, 600, 700, 800 ,900, 1000, 2000, and lastly 3000. To make this clearer as an example a unique model could be:

```
feature: CPU
pod name: frontend
user step: 300
ML algorithm: SVM polynomial
```

while another could be:

```
feature: Memory
pod name: currencyservice
user step: 200
ML algorithm: SVM linear
```

The training was done on the normed data and the predicted data was then un-normalized by using the formula $x = x'(x_{max} - x_{min}) + x_{min}$.

Hyperparameter tuning

Hyperparameter tuning was performed on each model to find the best training parameters for that particular data set. The parameter grid used for multilayer perceptron was:

```
{
    'hidden_layer_sizes': [(50,), (100,), (150,)],
    'activation': ['relu','tanh','logistic'],
    'alpha': [0.0001, 0.001 ,0.05],
}
```

and for both kernels for support vector machine it was:

```
{
    "C": [0.01 ,0.1, 0.5, 1,],
    "degree": [3, 4, 5]
}
```

The reason for choosing these hyperparameters for optimization is because these are the ones that would have the most frequent changes when training different models. C and alpha are regularization parameters and help with over-and underfitting. The motive behind choosing a small search space for the different machine learning algorithms was because of time constraint and resource constraint based on my personal computer as hyperparameter tuning became a bottleneck during training.

The method chosen for hyperparameter tuning was GridSearchCV which chooses a set of the cartesian product of the hyperparameters and trains, validates and calculates the accuracy of the chosen set until it finds the most accurate set of parameters.

Error

When measuring the prediction error, 20 random samples from the whole data set (which includes the training data) was used. The error was simply calculated by subtracting the predicted value from the actual value.

From these error values, two measures of error were obtained: mean squared error(MSE) and the 95th percentile. MSE was calculated by taking squaring all of the errors and getting the average of that.

Presenting and visualizing the data

Matplotlib was used for visualizing and plotting the data. Figures for every resource were created that contained an axes for each pod. Each axes contained different graphs for every trained model and also the 20 values that had been randomly sampled from the original data set. The same was done for the graphs that display the error.

3.2.4 Experiment setup

The experiments and collected data went through several iterations before landing on using 10000 users as maximum and HPA(Horizontal Pod Autoscaler) at 60%.

At first, a lower user count was used as the cluster needed to be provided with more resources to handle a bigger load. At the start the Kubernetes API was used to scale CPU and memory resources of the pods, but this was later abandoned as it gave bad response times and sometimes resulted in pod failures. When HPA was tested, different thresholds were tested. These were: 20, 30, 40, 50, 60, 70, and 80 percent, and it was determined that 60% performed the best out of these.

3.2.5 Limitations

The largest limitations on the methodology used was the hyperparameter tuning as well as training a unique model for every combination as described earlier since this resulted in training all the models and tuning their hyperparameters took a long time. The choice of hyperparameters could also be a limitation as choosing more hyperparameters might have produced better results but at the same time this would have increased time to train even more.

Chapter 4

Result, evaluation and discussion

In this chapter, the various predictions made by the machine learning methods are presented, analyzed and discussed.

4.1 Results

4.1.1 Machine learning predictions

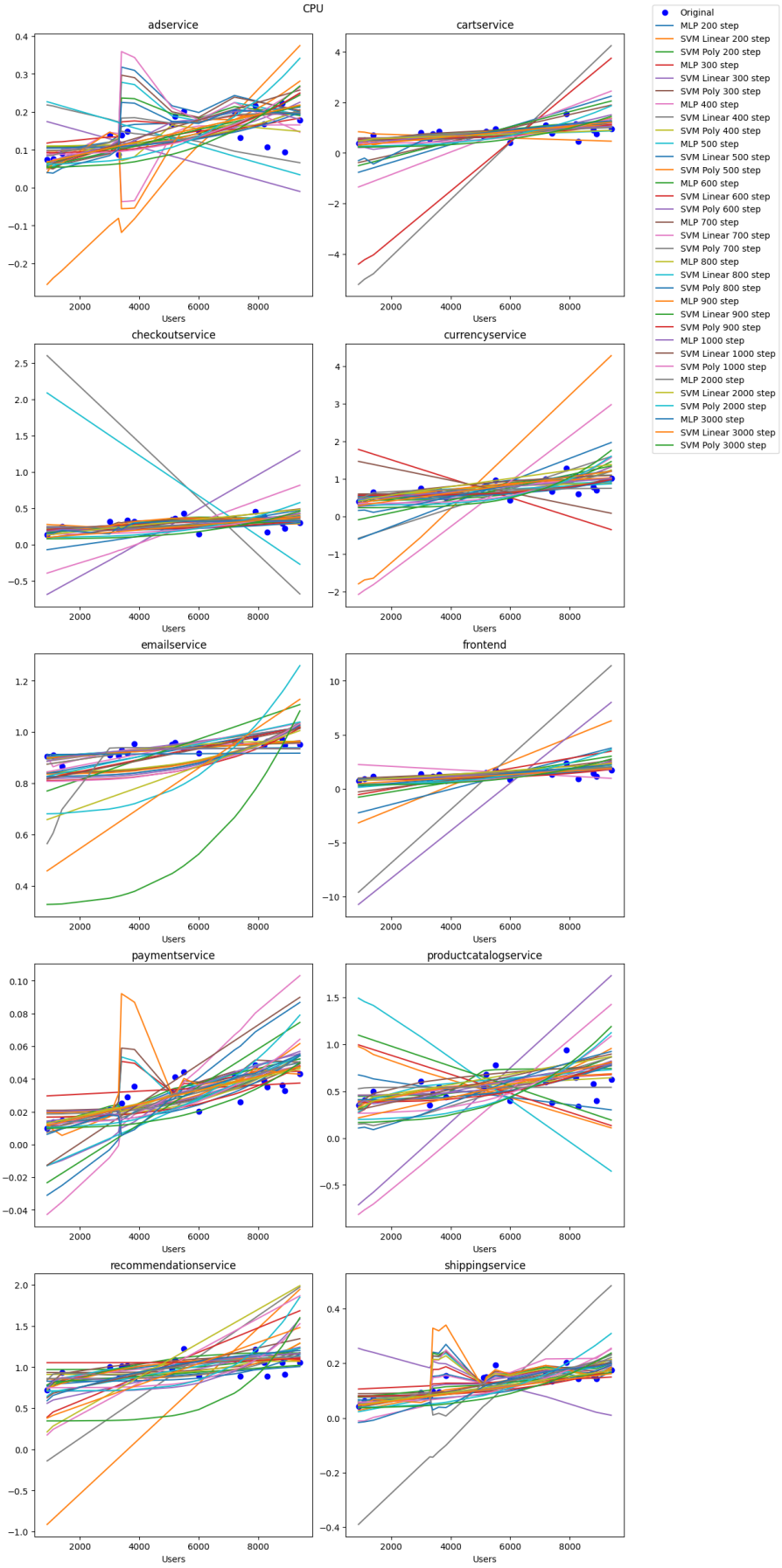

Figure 4.1.1: CPU prediction

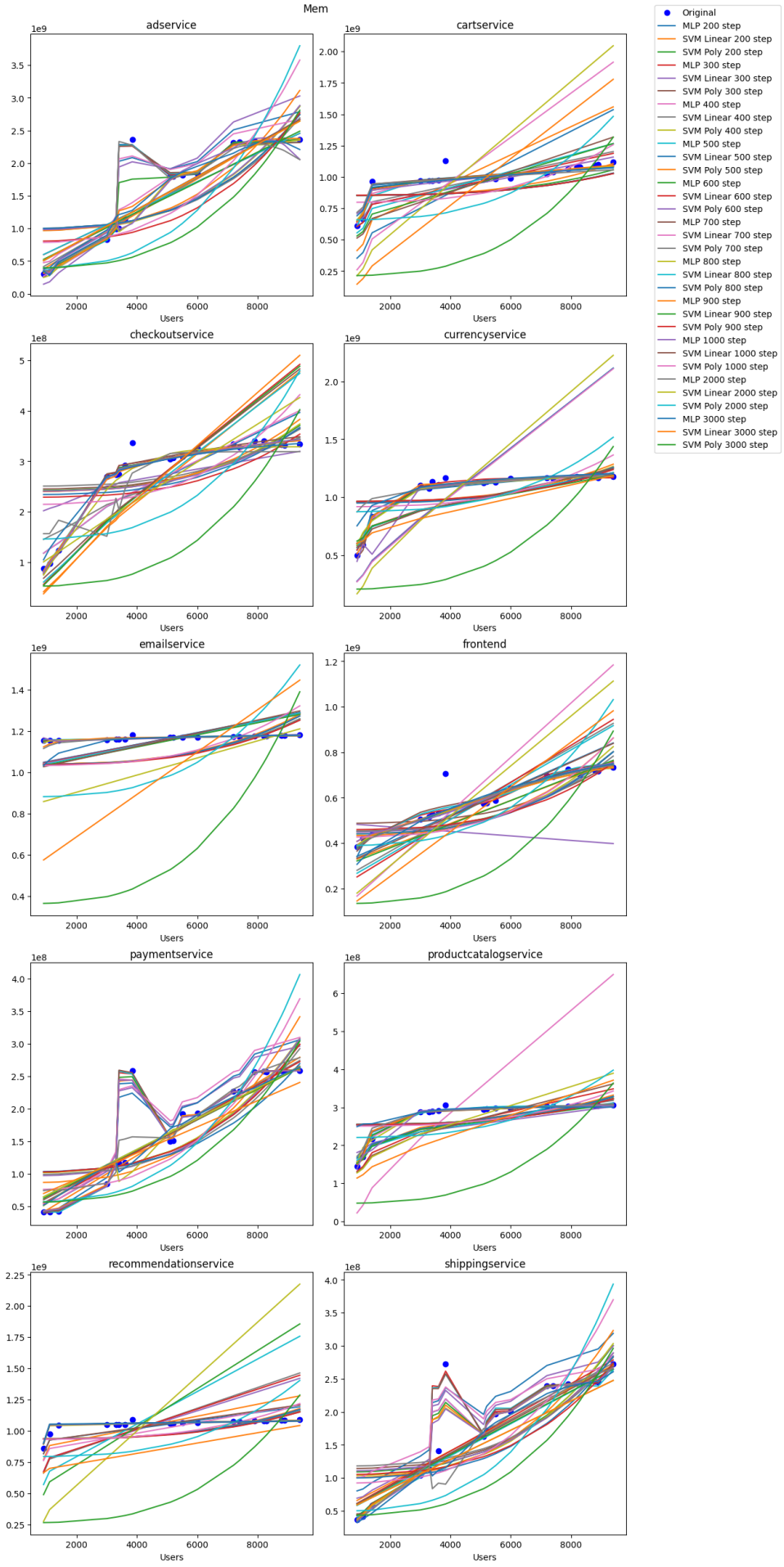

Figure 4.1.2: Memory prediction

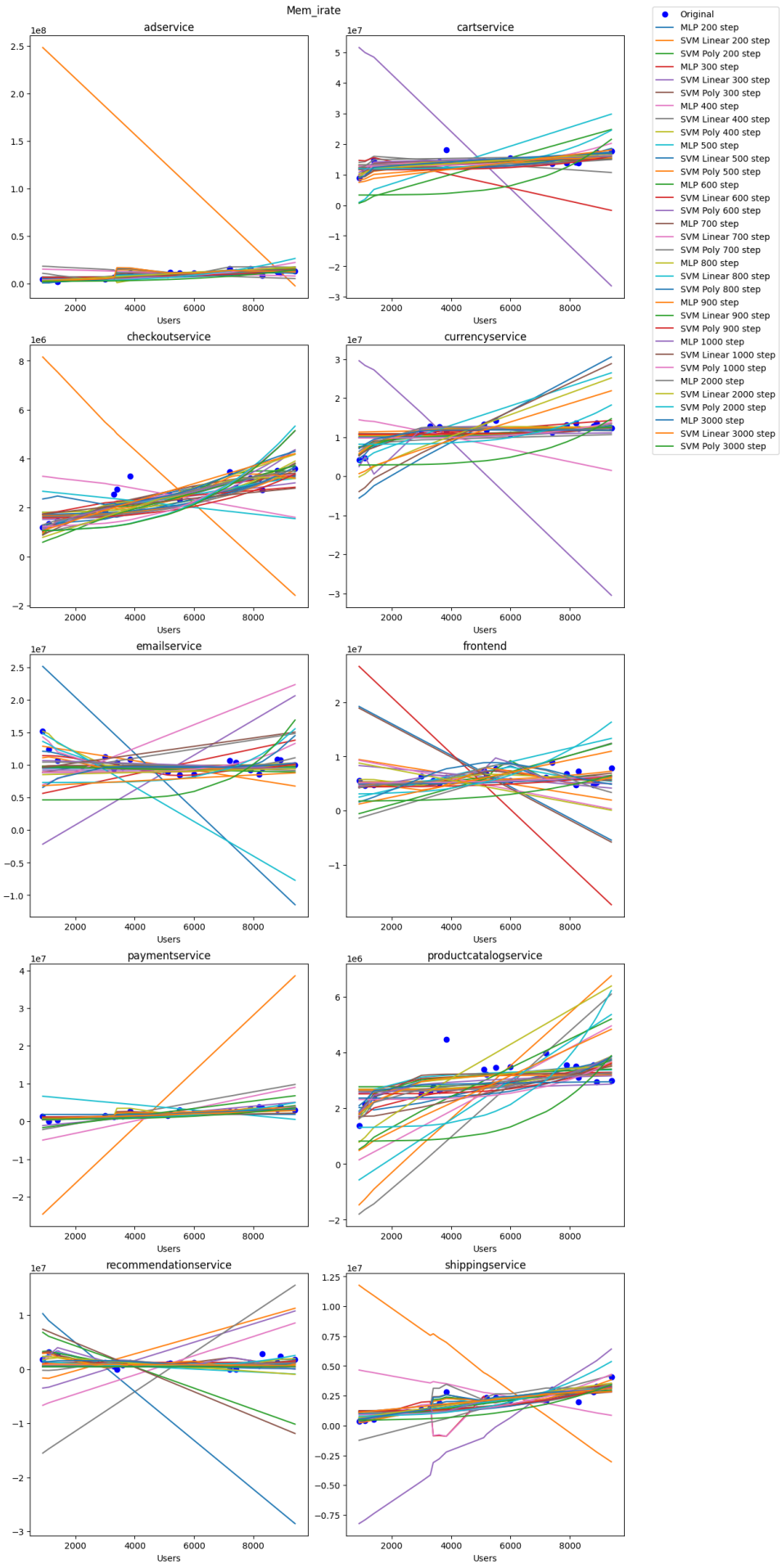

Figure 4.1.3: Mem_irate prediction

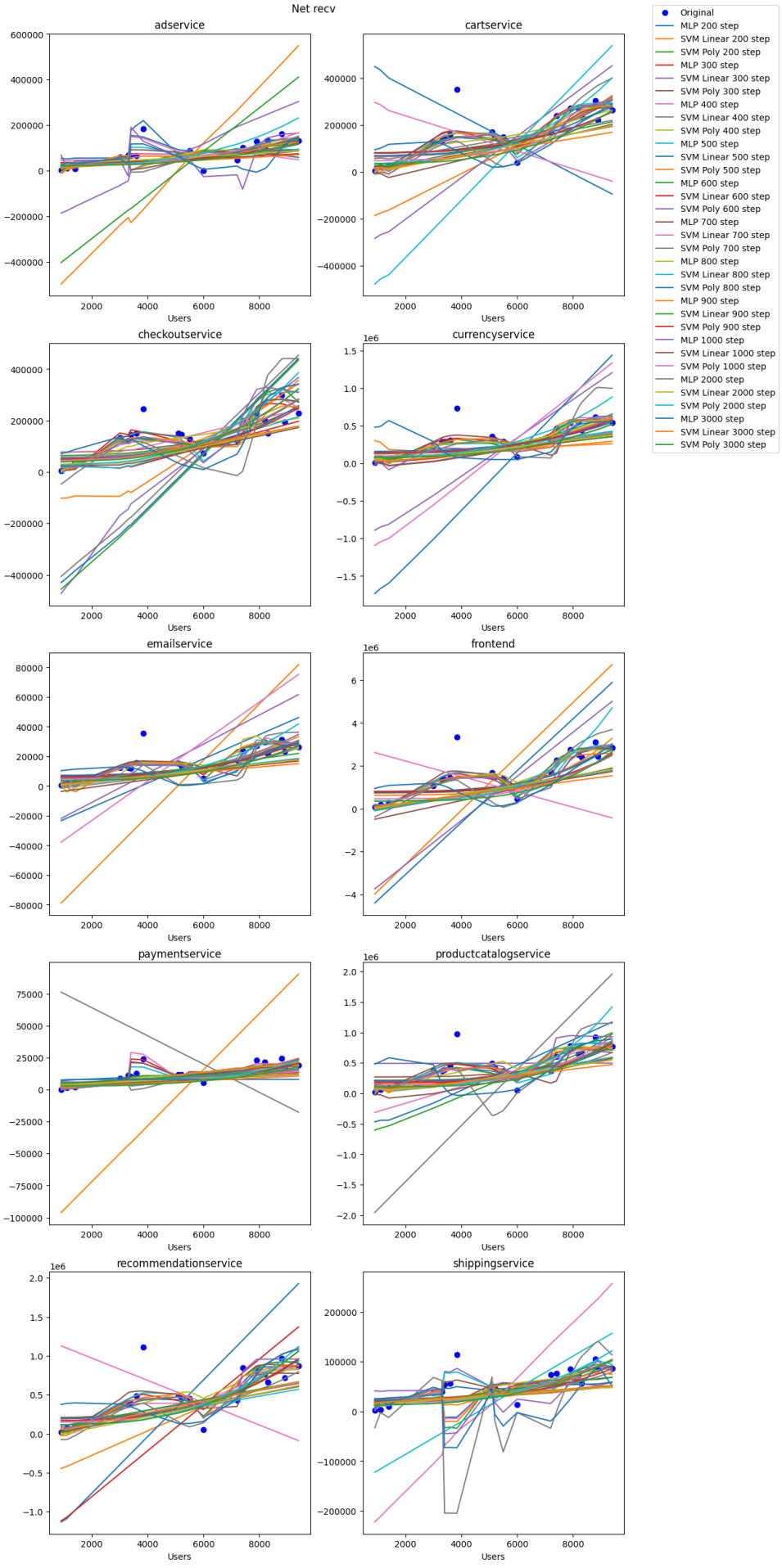

Figure 4.1.4: Network receive prediction

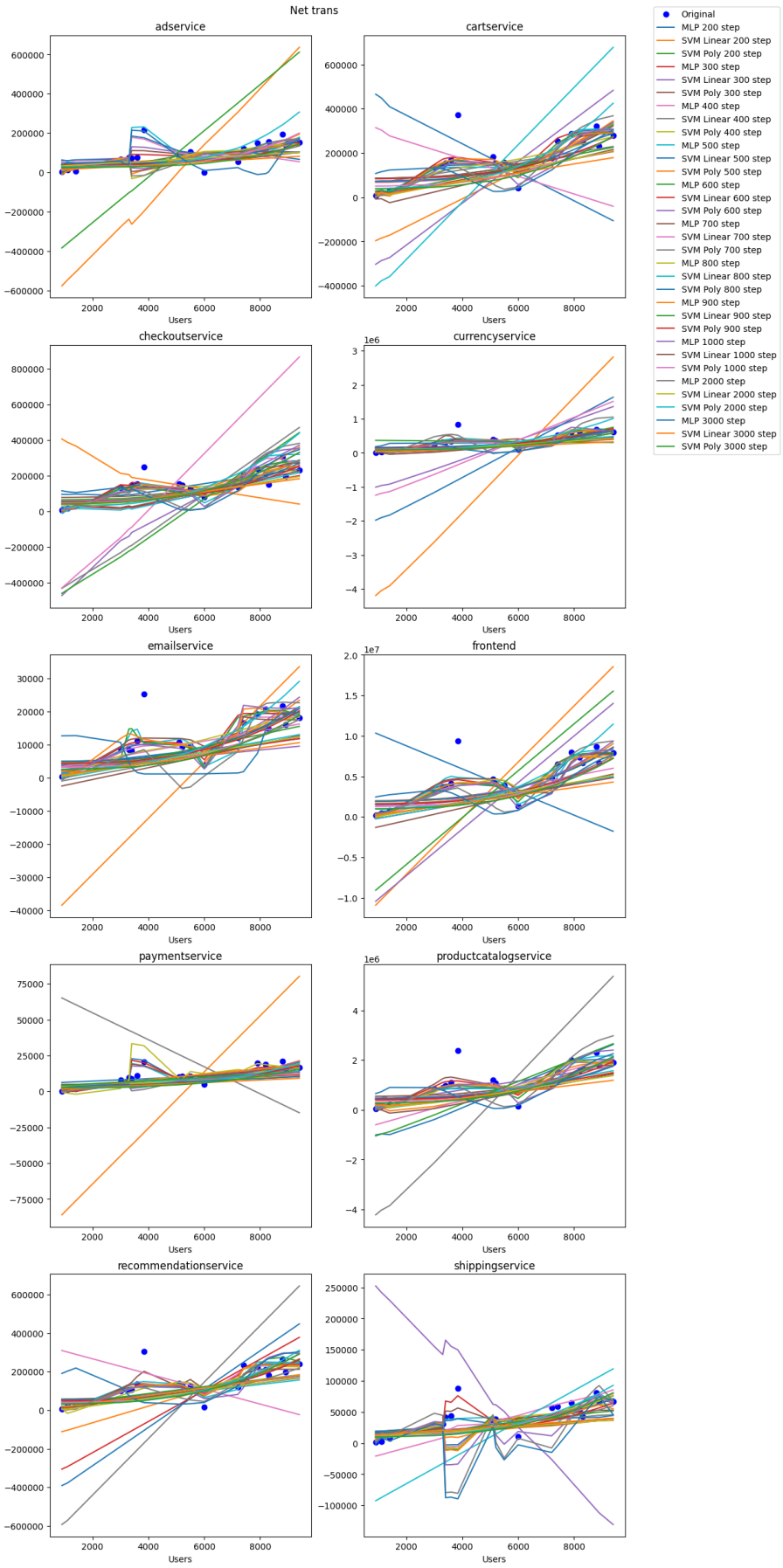

Figure 4.1.5: Network transmit prediction

Figures 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.1.5 present the predictions from all the different models as graphs and the 20 random original points as blue dots. Each models graph can be identified by looking at the figures' legends. The title of the figures describe the resource that was predicted and the y-axis is that resources' value. The x-axis represent the user count and the title for each subfigure represent a Kubernetes pod.

4.1.2 Prediction error

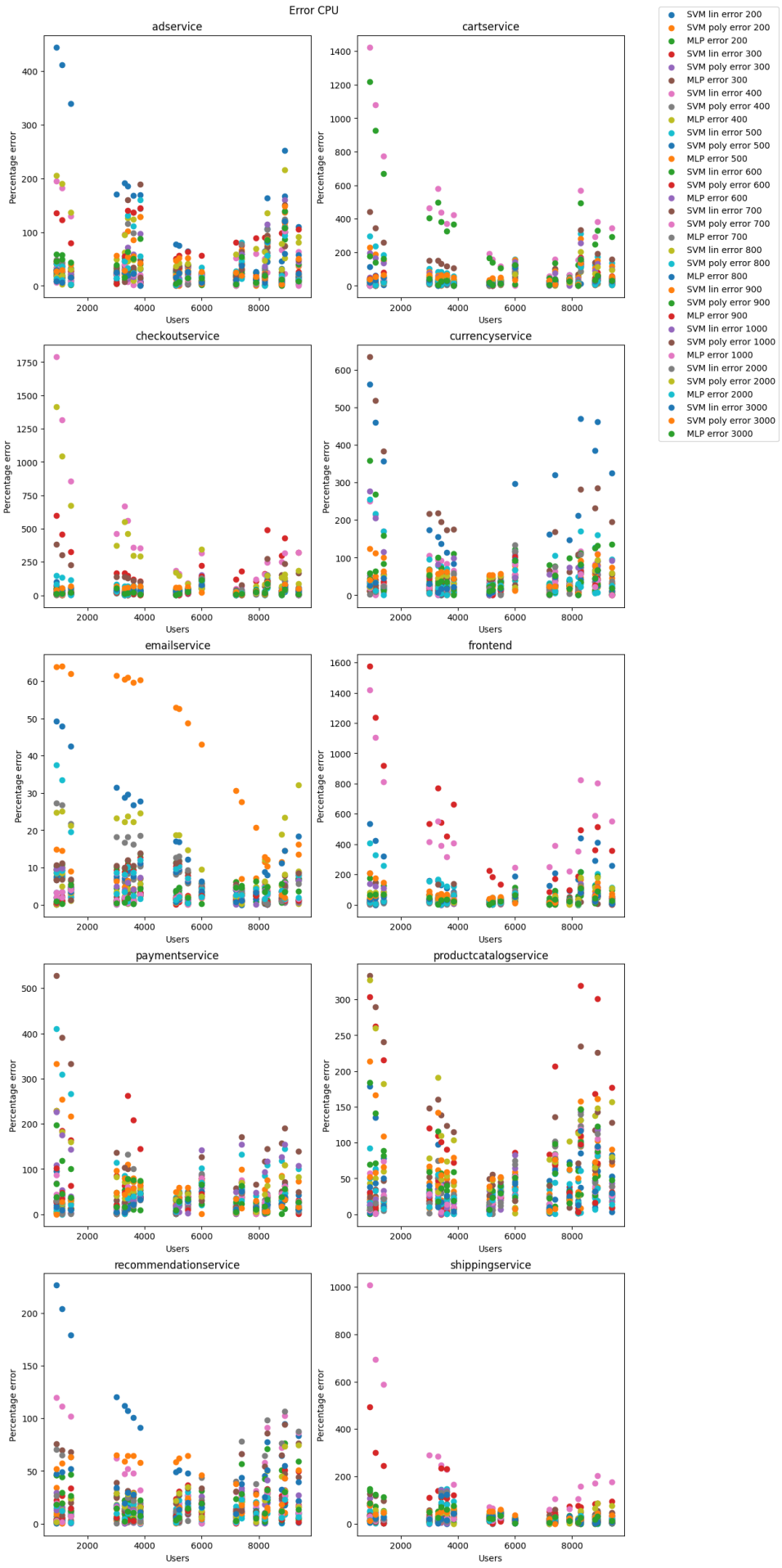

Figure 4.1.6: CPU error

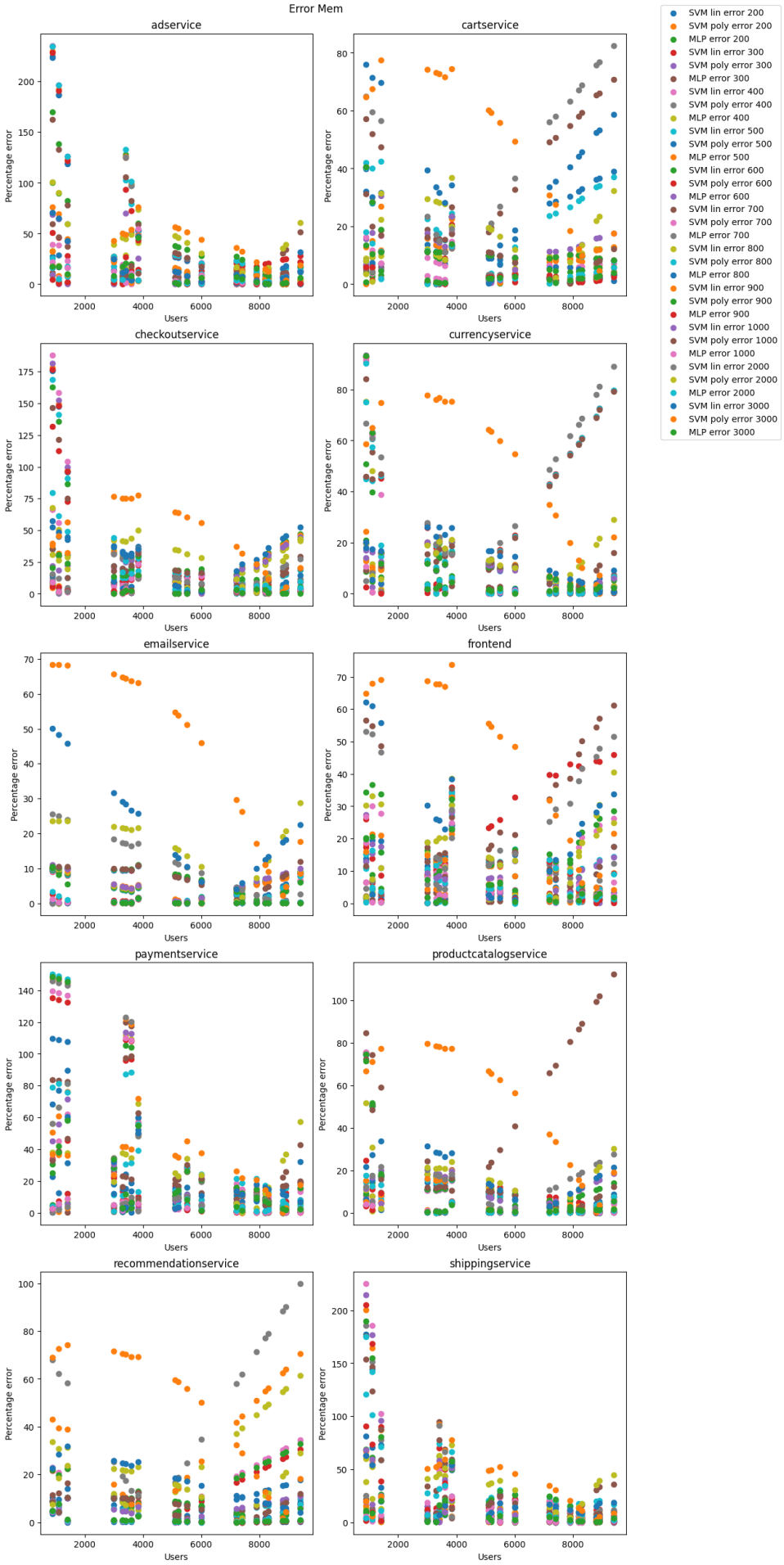

Figure 4.1.7: Memory error

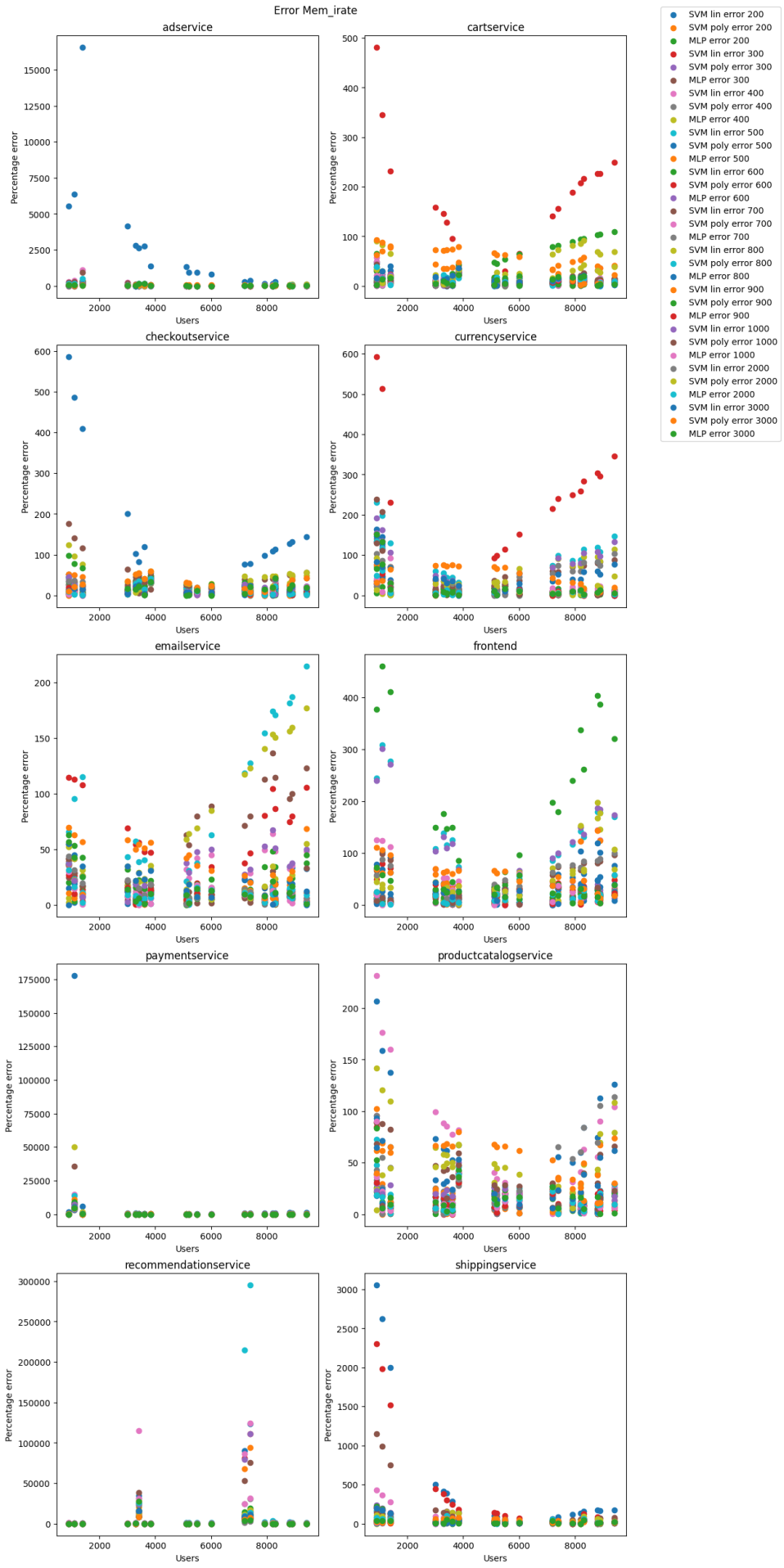

Figure 4.1.8: Mem_irate error

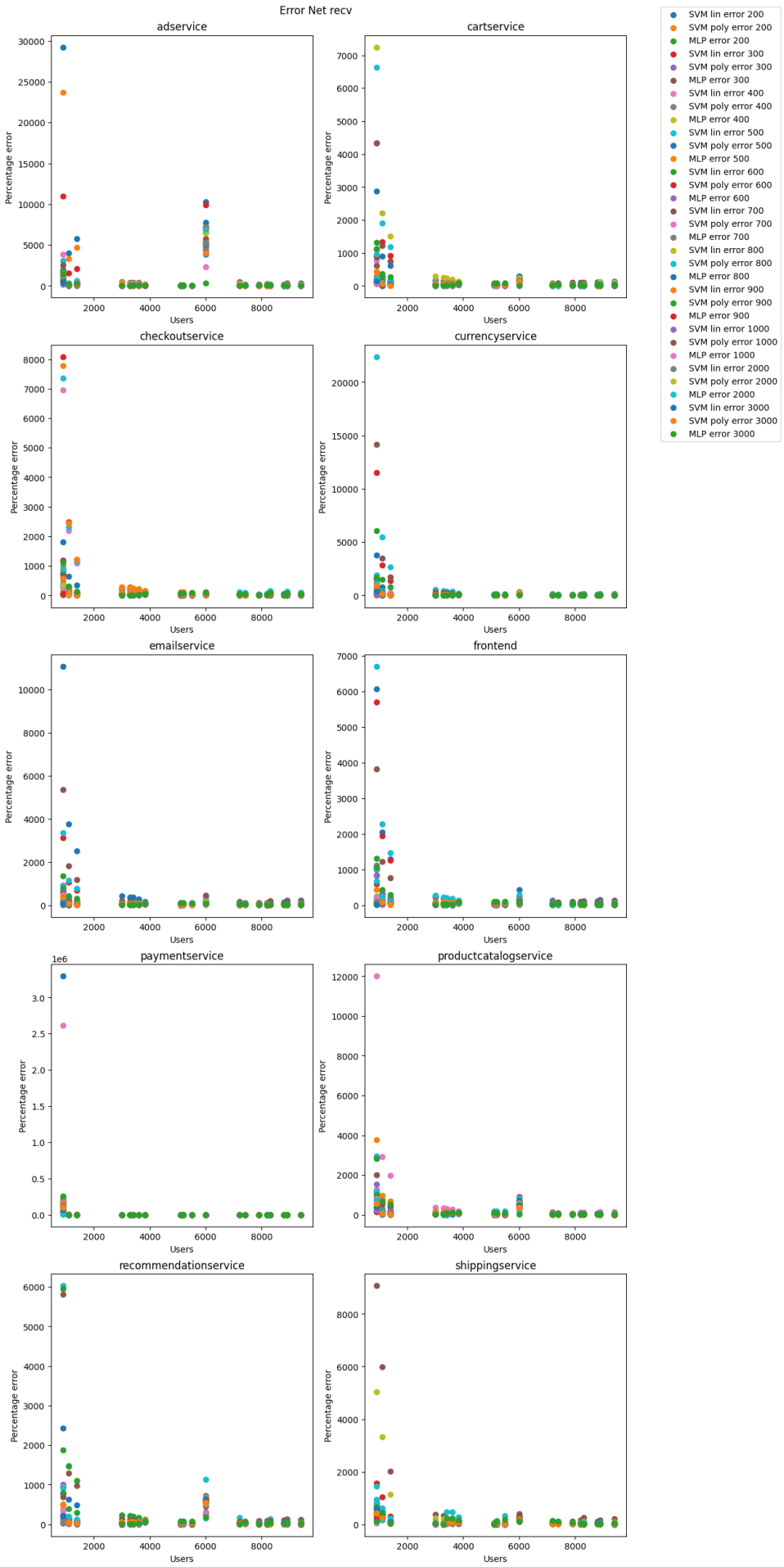

Figure 4.1.9: Network receive error

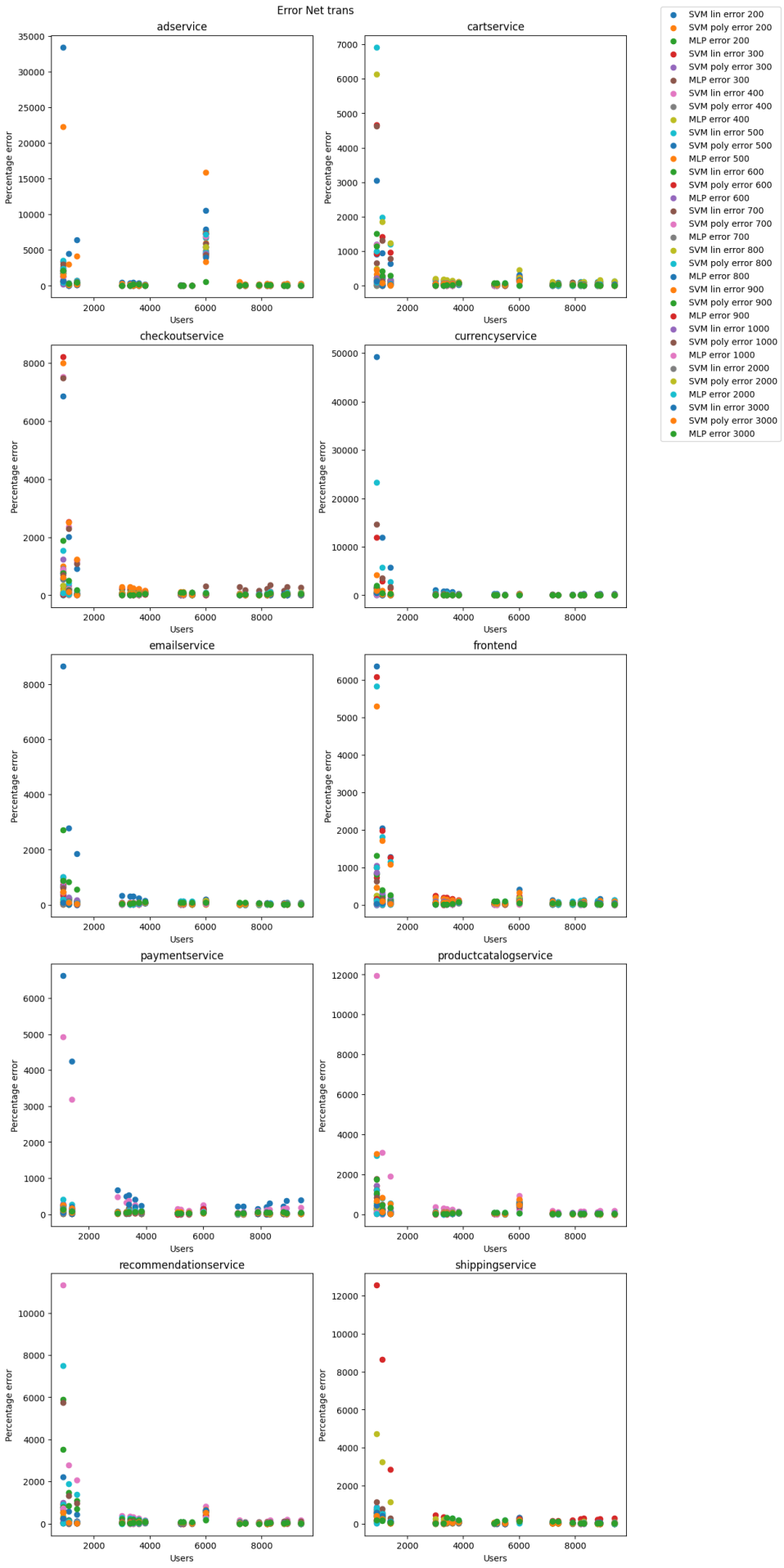

Figure 4.1.10: Network transmit error

The figures 4.1.6, 4.1.7, 4.1.8, 4.1.9, 4.1.10 demonstrate the error of each models prediction. The title of each figure describes what resource prediction the error belongs to and the legend identifies each models error as a colored dot.

4.1.3 Error MSE and 95th percentile

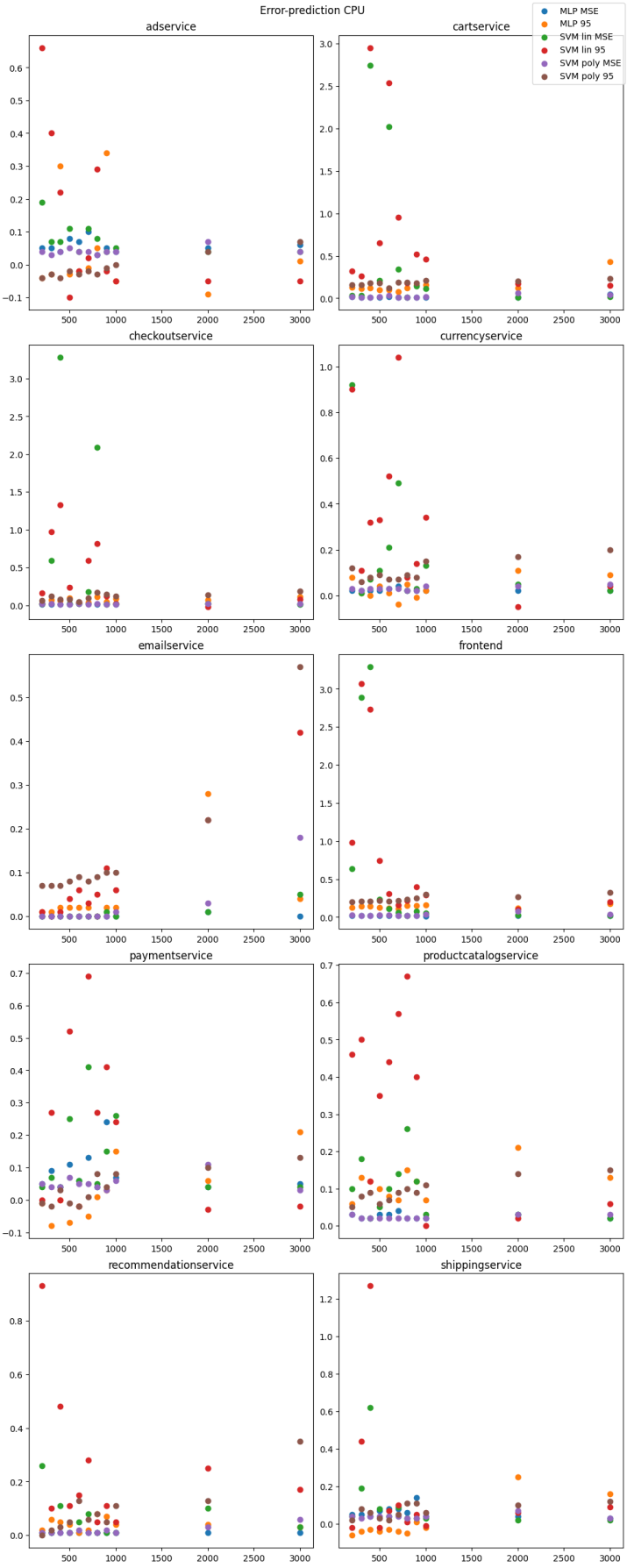

Figure 4.1.11: CPU error metrics

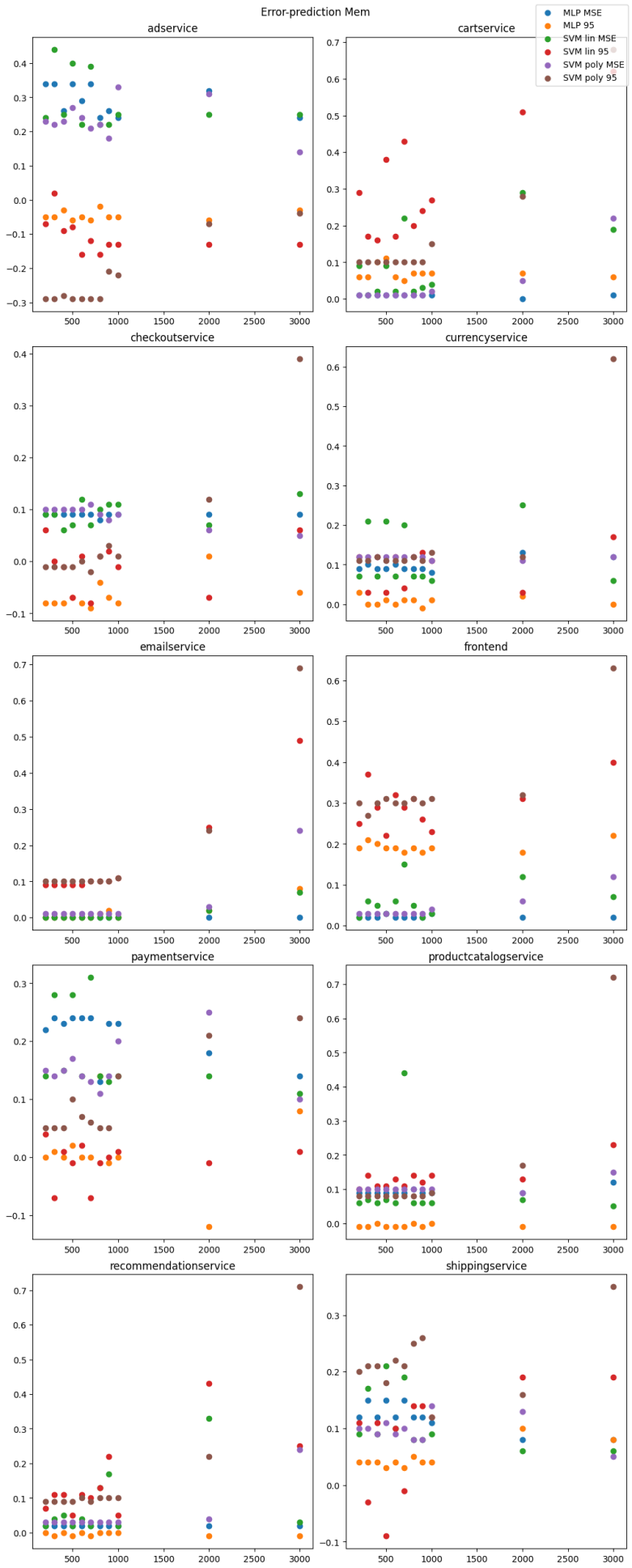

Figure 4.1.12: Memory error metrics

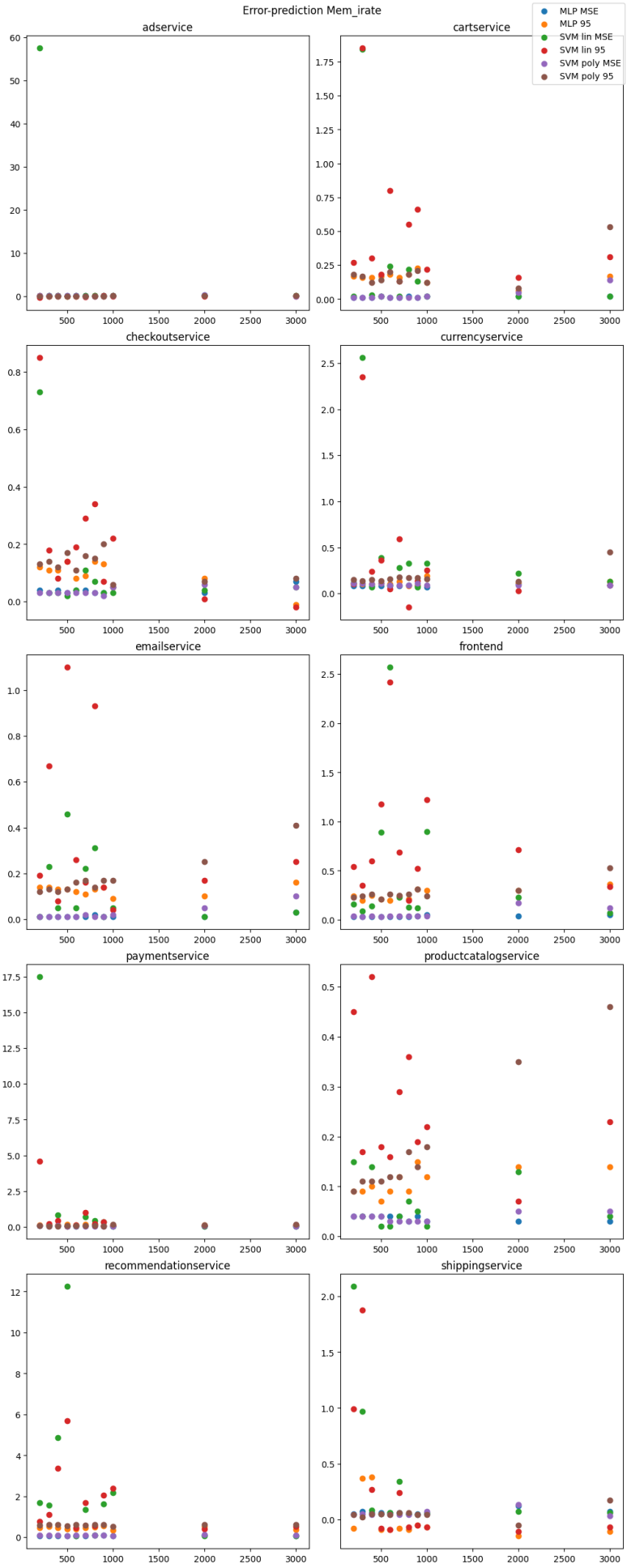

Figure 4.1.13: Mem_irate error metrics

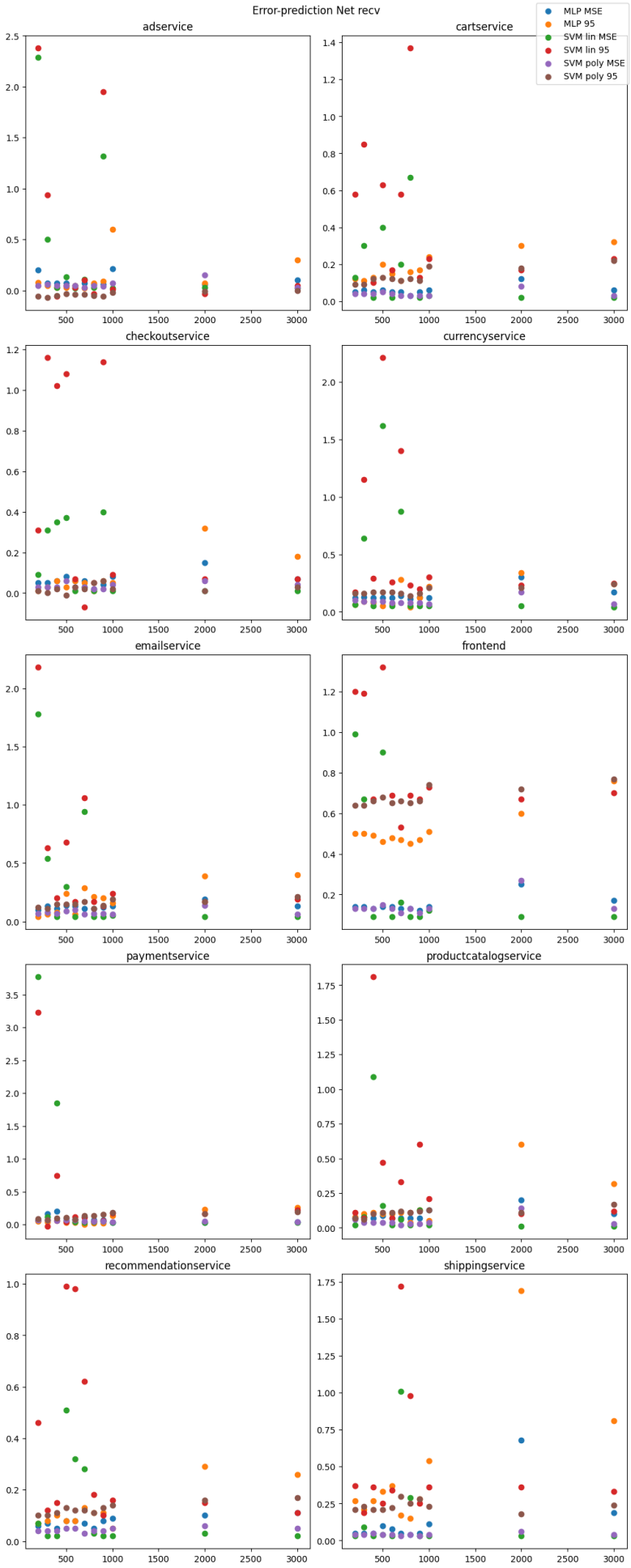

Figure 4.1.14: Network receive error metrics

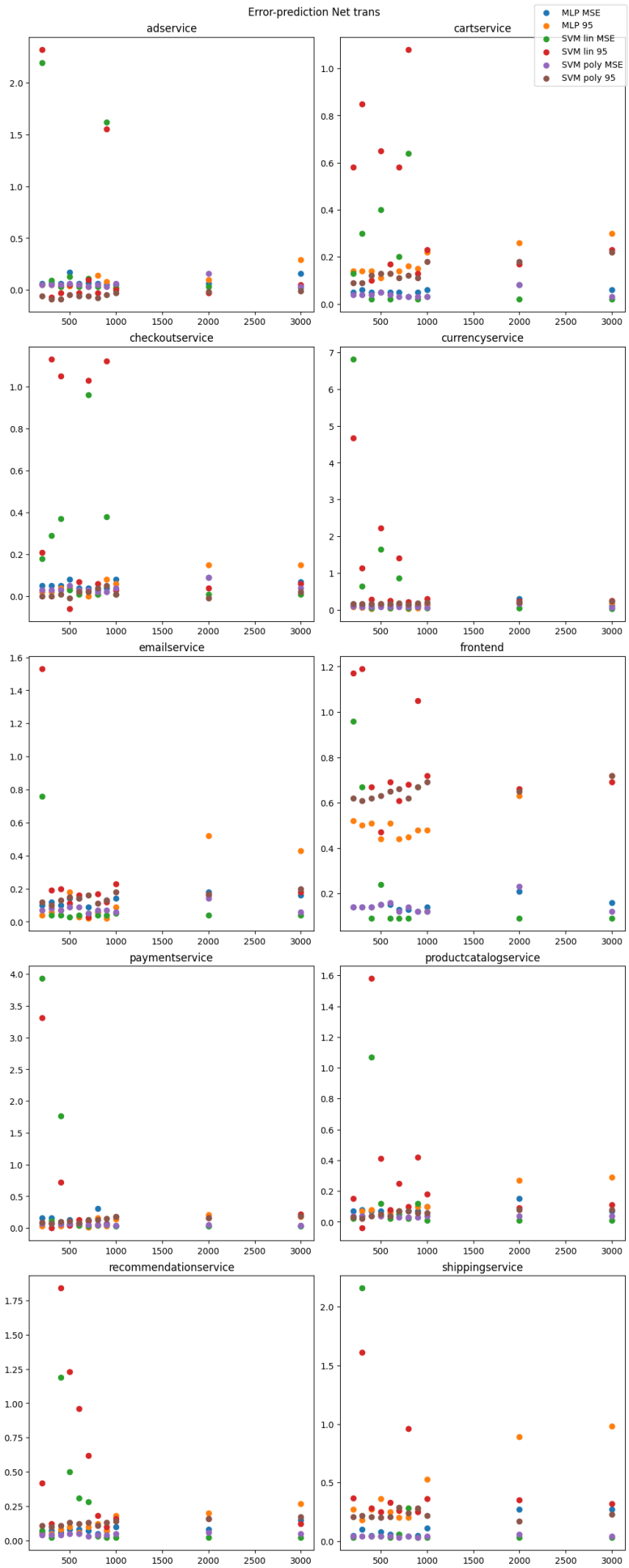

Figure 4.1.15: Network transmit error metrics

The figures 4.1.11, 4.1.12, 4.1.13, 4.1.14, 4.1.15 contain the mean squared error and 95th percentile error for all of the predictions made. Each models mean squared error and 95th percentile error are represented as colored dots and can be identified in the legend. The title of the figures represent the resource that was predicted. The x-axis represent the user steps that was used to train the models.

4.2 Evaluation and discussion

The results in section 4.1.1 show that SVM linear usually performs quite severely and the model is also underfitting. This is made clear in a figure like 4.1.1 where SVM linear is far off several of the original data points for every pod. There are, however, figures where it performs better such as 4.1.2 for the frontend and productcatalogservice pod. The case of the model performing badly is also supported by the figures in section 4.1.3 where SVM linear has a generally high mean squared error and 95th percentile error. This is reinforced in a figure like 4.1.11 where, for most pods, SVM linears MSE and 95th percentile error is higher than the other models.

MLP performed relatively well in regard to mean squared error and 95th percentile error. However, the figures in section 4.1.1 suggest that the model is overfitting. This is made clear in figure 4.1.1 but even more in figure 4.1.7 where MLP takes statistical outliers and noise into heavy consideration in pods paymentservice, adservice and shippingservice. This would explain why it also sometimes outperforms SVM poly in section 4.1.3.

SVM poly performs the best overall when considering error measurements and whether or not it is overfitting or underfitting. The figures in section 4.1.1 indicate that SVM poly has achieved a balance between underfitting and overfitting when compared to SVM linear and MLP. Although figures such as figure 4.1.1 show that some models for SVM poly does not fit the original datapoints particularly well. The results in section 4.1.3 also show that the model's error is relatively low.

Combining the results from figures in sections 4.1.2 and 4.1.3 demonstrates that a lower user step has a smaller error and that going above 1000 user steps gives a higher error.

The results from the machine learning models suggest that when predicting resource utilization for Kubernetes, using SVM with a polynomial kernel is a good middle ground as this provides a lower error while at the same time finding a balance when fitting the model. MLP could also be used, but as it is overfitted, it may produce misleading predictions. SVM linear should not be used as the model is underfitted and also has a high error.

When looking at what user step performed best the discrepancy in lower x-axis in figures in section 4.1.2 should be addressed. These high errors are most likely a result of SVM linear underfitting, which can be confirmed by looking at figures in section 4.1.1, where SVM linear usually begins very far down away from the actual values of the

data. One can also see a jump in errors at 6000 users. This could result from higher user steps like 1000, 2000, and 3000 having higher errors. From this, it seems that the lower user steps are better for training; however, there is no significant change in mean squared error between 200 and 1000 user steps. This means that any user step could probably be used between these values and still produce good predictions but taking more datapoints into consideration is probably better and which is also quite intuitive.

Chapter 5

Conclusions and Future Work

This chapter summarizes the whole of the thesis and presents some suggestions that could be done for future work on the topic.

5.1 Conclusion

This thesis aimed to evaluate different machine learning methods used on data for resource utilization in a Kubernetes cluster when putting the cluster under different loads. This project was done by doing a literature review, setting up the testing environment in Kubernetes, running experiments, collecting data, extracting and analyzing data, and learning about different techniques in machine learning.

Data was collected by running different tests with the goal of analyzing the data from these tests and using it with different machine learning models to predict resource utilization and then compare these different models to each other. The test results and the comparison suggest that the best machine learning model out of the 3 was Support Vector Machine with a polynomial kernel as this provided an excellent middle ground.

5.2 Future Work

For future work, there are many alternatives that could be explored. One such direction would be to conduct more experiments with a larger number of users to evaluate the usability and effectiveness of the machine learning methods. This would in turn give more data for training and more different user steps could be examined.

Another direction would be to extend the machine learning, such as having more data, more computational power, and more time, to perform more hyperparameter tuning and optimization of the models so that the under- and overfitting might be fixed.

A third alternative would be to compare more machine learning methods, such as SVM with a custom kernel or any other type of regression model, to each other and examine

how those perform.

Bibliography

[1] *API Overview | Kubernetes*. https://kubernetes.io/docs/reference/using-api/. Accessed: 2023-05-16.

[2] Ari, Niyazi and Ustazhanov, Makhamadsulton. "Matplotlib in python". In: (2014). DOI: 10.1109/ICECCO.2014.6997585.

[3] Ashish, Kumar. *Mastering Pandas : A Complete Guide to Pandas, From Installation to Advanced Data Analysis Techniques, 2nd Edition*. Packt Publishing, 2019. ISBN: 9781789343236.

[4] Ayache, Eliot H. and Omand, Conor M. B. *Generating Scientific Articles with Machine Learning*. 2022. arXiv: 2203.16569 [cs.LG].

[5] Cervantes, Jair, García-Lamont, Farid, Rodríguez-Mazahua, Lisbeth, and López, Asdrúbal. "A comprehensive survey on support vector machine classification: Applications, challenges and trends". In: *Neurocomputing* (Sept. 2020). DOI: 10.1016/j.neucom.2019.10.118.

[6] GoogleCloudPlatform. *microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC*. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed: 2023-05-14. 2023.

[7] Hettiarachchi, Lasal Sandeepa, Jayadeva, Senura Vihan, Bandara, Rusiru Abhisheak Vikum, Palliyaguruge, Dilmi, Arachchillage, Udara Srimath S.Samaratunge, and Kasthurirathna, Dharshana. "Expert System for Kubernetes Cluster Autoscaling and Resource Management." In: *2022 4th International Conference on Advancements in Computing (ICAC), Advancements in Computing (ICAC), 2022 4th International Conference on* (2022). ISSN: 979-8-3503-9809-0.

[8] Khazaei, Hamzeh, Ravichandiran, Rajsimman, Park, Byungchul, Bannazadeh, Hadi, Tizghadam, Ali, and Leon-Garcia, Alberto. "Elascale: Autoscaling and Monitoring as a Service." In: (2017). URL: https://login.bibproxy.kau.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsarx&AN=edsarx.1711.03204&lang=sv&site=eds-live.

[9] *Kubernetes Overview*. https://kubernetes.io/docs/concepts/overview/. Accessed: 2023-03-22.

[10] Millnert, Victor and Eker, Johan. “HoloScale: horizontal and vertical scaling of cloud resources”. In: (2020), pp. 196–205. DOI: 10.1109/UCC48980.2020.00038.

[11] Moravcik, Marek, Kontsek, Martin, Segec, Pavel, and Cymbalak, David. “Kubernetes - evolution of virtualization.” In: *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA), Emerging eLearning Technologies and Applications (ICETA), 2022 20th International Conference on* (2022), pp. 454–459. ISSN: 979-8-3503-2033-6. URL: https://login.bibproxy.kau.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.9974681&lang=sv&site=eds-live.

[12] Nichols, James A., Chan, Hsien W. Herbert, Chan, Hsien W. Herbert, and Baker, Matthew A. B. “Machine learning: applications of artificial intelligence to imaging and diagnosis”. In: *Biophysical Reviews* (Feb. 2019). DOI: 10.1007/s12551-018-0449-9.

[13] *NumPy*. https://numpy.org/. Accessed: 2023-05-17.

[14] Obiora, Chibuzor N, Ali, Ahmed, and Hasan, Ali N. “Using the Multilayer Perceptron (MLP) Model in Predicting the Patterns of Solar Irradiance at Several Time Intervals”. In: (Jan. 2023). DOI: 10.1109/saupec57889.2023.10057744.

[15] *Overview | Prometheus*. https://prometheus.io/docs/introduction/overview/. Accessed: 2023-05-16.

[16] *pandas - Python Data Analysis Library*. https://pandas.pydata.org/. Accessed: 2023-05-17.

[17] Pradeep, S. and Sharma, Yogesh Kumar. “A Pragmatic Evaluation of Stress and Performance Testing Technologies for Web Based Applications”. In: *International Conference on Artificial Intelligence* (Feb. 2019). DOI: 10.1109/aicai.2019.8701327.

[18] Rampérez, Víctor, Soriano, Javier, Lizcano, David, and Lara, Juan A. “FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services.” In: *Future Generation Computer Systems* 118 (2021), pp. 56–72. ISSN: 0167-739X. URL: https://login.bibproxy.kau.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S0167739X20330879&lang=sv&site=eds-live.

[19] Rondon, Carlos Vicente Nino, Delgado, Byron Medina, Casadiego, Sergio Castro, and Rojas, Jorge Gomez. “Computational Learning Models of Scikit-Learn for Automatic People Identification Integrated in a GUI.” In: *2022 IEEE XXIX International Conference on Electronics, Electrical Engineering and Computing (INTERCON), Electronics, Electrical Engineering and Computing (INTERCON), 2022 IEEE XXIX International Conference on* (2022), pp. 1–4. ISSN: 978-1-6654-8636-1.

[20] *scikit-learn: machine learning in Python*. https://scikit-learn.org/. Accessed: 2023-05-17.

[21] Shim, Simon, Dhokariya, Ankit, Doshi, Devangi, Upadhye, Sarvesh, Patwari, Varun, and Park, Ji-Yong. "Predictive Auto-scaler for Kubernetes Cloud." In: *2023 IEEE International Systems Conference (SysCon), Systems Conference (SysCon), 2023 IEEE International* (2023). ISSN: 978-1-6654-3994-7.

[22] Sukhija, Nitin and Bautista, Elizabeth. "Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus". In: *Ubiquitous Intelligence and Computing* (Aug. 2019). DOI: 10.1109/smartworld-uic-atc-scalcom-iop-sci.2019.00087.

[23] Todorov, Milen Hrabarov. "Deploying Different Lightweight Kubernetes on Raspberry Pi Cluster." In: *2022 30th National Conference with International Participation (TELECOM), International Participation (TELECOM), 2022 30th National Conference with* (2022), pp. 1–4. ISSN: 978-1-6654-8212-7.

[24] Toka, Laszlo, Dobreff, Gergely, Fodor, Balazs, and Sonkoly, Balazs. "Adaptive AI-based auto-scaling for Kubernetes." In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Cluster, Cloud and Internet Computing (CCGRID), 2020 20th IEEE/ACM International Symposium on* (2020). ISSN: 978-1-7281-6095-5.

[25] Toka, Laszlo, Dobreff, Gergely, Fodor, Balázs, and Sonkoly, Balazs. "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters". In: *IEEE Transactions on Network and Service Management* (Jan. 2021). DOI: 10.1109/tnsm.2021.3052837.

[26] *What is Kubernetes?* https://www.redhat.com/en/topics/containers/what-is-kubernetes. Accessed: 2023-03-22.

[27] *What is Locust? — Locust 2.15.1 documentation*. https://docs.locust.io/en/stable/what-is-locust.html. Accessed: 2023-05-15.

[28] Yaqin, Ainul, Rahardi, Majid, Abdulloh, Ferian Fauzi, Kusnawi, Budiprayitno, Slamet, and Fatonah, Siti. "The Prediction of COVID-19 Pandemic Situation in Indonesia Using SVR and SIR Algorithm." In: *2022 6th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE), Information Technology, Information Systems and Electrical Engineering (ICITISEE), 2022 6th International Conference on* (2022). ISSN: 979-8-3503-9961-5.

[29] Zhou, N., Zhou, H., and Hoppe, D. "Containerization for High Performance Computing Systems: Survey and Prospects." In: *IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng* 49.4 (2023), pp. 2722–2740. ISSN: 0098-5589. URL: https://login.bibproxy.kau.se/login?url=https://search.ebscohost.

com/login.aspx?direct=true&db=edseee&AN=edseee.9985426&lang=sv&site=eds-live.