Master's thesis

ANOMALY DETECTION OF HTTP SERVER ROLLOUTS

Bc. Filip Čacký

Faculty of Information Technology
Katedra aplikované matematiky
Supervisor: Ing. Daniel Sedlák
January 9, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Anomaly detection of HTTP server rollouts
Student:	Bc. Filip Čacký
Supervisor:	Ing. Daniel Sedlák
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2025/2026

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 7 August 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Instructions

Software deployment is an inseparable part of any internet service,
and it is crucial to ensure the system does not exhibit anomalous behavior after the upgrade is complete. Unexpected CPU utilization, atypical network traffic patterns, and abnormal HTTP status code distributions can lead to overall performance and user experience degradation. Effective monitoring and incident detection are essential for maintaining system stability and preventing unexpected failures.

This thesis aims to design and implement a rollout analysis system using statistical and/or machine learning methods for monitoring HTTP web servers. The expected input of the service is a set of system and service metrics time series, and the expected output is a verdict about the health of the system.

Tasks:

1. Research the current state-of-the-art methods for anomaly detection in time series data.
2. Select an appropriate time series relevant to the web server's and host system's health.
3. Analyze and preprocess the time series data.
4. Choose suitable statistical and/or AI/ML methods for anomaly detection.
5. Develop a rollout anomaly detection/monitoring system.
6. Experimentally evaluate the system's performance, discuss the results, and identify potential improvements.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 7 August 2024 in Prague.

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Čacký Filip. *Anomaly detection of HTTP server rollouts.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I would like to express my deepest gratitude to my supervisor, Daniel Sedlák, for his unending wisdom, guidance and valuable feedback throughout the course of this thesis.

I am also profoundly grateful to my family and friends for their unwavering support and encouragement.

I would like to thank my colleagues at CDN77 for many fruitful and inspiring discussions, code reviews, and for providing me with the necessary resources to complete this thesis. Namely and in no particular order: Vojtěch Štafa and Tomáš Kvasnička.

Lastly, I would also like to thank the wonderful people at Café Jen for providing me with a steady supply of delicious coffee, breakfasts and snacks, without which this thesis would not have been possible, and to Zdeněk Cendra, for generously bankrolling my caffeine addiction.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended

In Prague on January 9, 2025

Abstract

Ensuring the correct and efficient operation of large-scale computer systems is a key challenge, mainly as systems grow in complexity and scale. Effective monitoring and anomaly detection are indispensable for maintaining the system integrity, availability, and performance, as they enable the timely identification and resolution of hardware failures, software regressions, and network issues.

We focus on the problem of anomaly detection in HTTP proxy server metrics and address it by developing a scalable and configurable system for detecting anomalies in time series data. The proposed system comprises a gRPC-based monitoring job orchestration service and a suite of unsupervised anomaly detection models with automatic thresholding. Five custom datasets representing common anomalies in production environments and a publicly available dataset were used to evaluate the system.

Furthermore, this work presents a foundational overview of time series forecasting and anomaly detection methods, as well as a brief introduction to systems performance monitoring methodologies.

Keywords anomaly detection, HTTP server, time series, server monitoring

Abstrakt

Zajištění správného a efektivního provozu rozsáhlých počítačových systémů představuje zásadní výzvu, zejména s jejich rostoucí složitostí. Efektivní monitorování a detekce anomálií jsou nezbytné pro zachování jejich integrity, dostupnosti a výkonu, jelikož umožňují, mimo jiné, včasnou identifikaci a řešení hardwarových poruch, softwarových regresí a síťových problémů.

Zaměřujeme se na problém detekce anomálií v metrikách HTTP proxy serverů a řešíme ho vývojem škálovatelného a konfigurovatelného systému pro detekci anomálií v časových řadách. Navrhovaný systém se skládá z gRPC

servisy sloužící pro orchestraci monitorovacích procesů a sady nesupervizovaných modelů pro detekci anomálií s automatickým prahováním. Pro vyhodnocení navržených modelů bylo použito pět vlastních datových sad reprezentujících běžné anomálie v produkčním prostředí a jedna veřejně dostupná datová sada.

Tato práce navíc poskytuje základní přehled metod predikce časových řad a detekce anomálií, společně se stručným úvodem do standardních metodologií pro monitorování výkonu systémů.

Klíčová slova detekce anomálií, HTTP server, časové řady, monitorování serverů

Contents

List of Figures

List of Tables

List of code listings

List of abbreviations

ACF	Auto-Correlation Function
AE	Auto-Encoder
Conv, CNN	Convolutional Neural Network
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
LSTM	Long Short-Term Memory
NIC	Network Interface Card
PACF	Partial Auto-Correlation Function
RNN	Recurrent Neural Network
RPC	Remote Procedure Call
VAE	Variational Auto-Encoder

Chapter 1

Introduction

One of the key challenges in managing large-scale distributed computer systems is to ensure that the systems operate correctly and efficiently. Monitoring is an essential part of this process, as it allows system administrators and support staff to observe the behavior of the systems and detect both hardware and software issues, performance regressions, network problems, and other issues that may affect the system's integrity and availability as soon as they arise.

Failing to respond quickly and decisively to any infrastructure accidents, be it a network backbone or a distributed application, can lead to a variety of problems, including system downtime, data loss, and poor performance. All of this may lead to vast financial losses and damages to the reputation of the organization and, perhaps most importantly, the annoyance of engineers, who then have to write lengthy root cause analysis reports. With all of this in mind, it is clear that having access to robust observability and anomaly analysis tooling is of utmost importance, as even small issues can quickly escalate into major incidents and outages if not detected and resolved in time.

Considering the ever-growing scale of today's computer systems, it is no longer feasible to rely solely on manual monitoring and analysis if it ever was. Instead, automated monitoring and anomaly detection systems are required to handle the vast amounts of data generated by these systems and to provide timely and accurate insights into their behavior. With this approach arises a new problem, a poorly performing anomaly detection system can lead to false alarms, alert fatigue, and wasted resources.

1.1 Motivation

The motivation for this thesis comes from the author's experience working with large-scale distributed systems and the challenges associated with their monitoring and maintenance. Having led the development of a live video delivery network, currently serving hundreds of millions of users monthly in over

100 points of presence worldwide, the author has experienced first-hand the difficulties of monitoring and troubleshooting network, hardware, and software issues in such a complex and dynamic environment.

Having access to a reliable and robust monitoring system with low false positive rates and high detection rates, which is easily scalable and adaptable to the specific requirements of the system, and most importantly, easy to set up and integrate with existing monitoring tools, would greatly simplify the process of maintaining and troubleshooting such systems.

1.2 Objectives

This thesis aims to address the problem of anomaly detection in large-scale distributed computer systems, by developing a system that can automatically detect anomalies in time series data. The system should be easily scalable to allow monitoring of tens of thousands of servers, while being adaptable to the specific requirements of different systems and applications.

Moreover, the theoretical part of this thesis should serve as a solid foundation for understanding the concepts and methods used in anomaly detection and time series forecasting, and provide an overview of the most commonly used methods and tools in this field for the reader, and for whoever may continue the author's work in the future, the author included.

Chapter 2

Background

Considering the focus of this thesis is on anomaly detection in time series data, It is appropriate to start with a brief introduction to the topic.

This chapter provides a foundational overview of key terms and concepts related to time series data, explores forecasting and anomaly detection methods and reviews commonly used monitoring and infrastructure tools. As well as providing a brief overview of terminology and concepts related to performance engineering.

2.1 Systems performance

> *Systems performance studies the performance of an entire computer system, including all major software and hardware components. Anything in the data path, from storage devices to application software, is included, because it can affect performance. For distributed systems this means multiple servers and applications. - Brendan Gregg, Systems Performance [1]*

The following are key terms related to systems performance, excerpt from the book Systems Performance: Enterprise and the Cloud by Brendan Gregg [1].

1. **IOPS:** Input/output operations per second is a measure of the rate of data transfer operations.
2. **Throughput:** The rate of work performed. Especially in communications, the term is used to refer to the data rate (bytes per second or bits per second). In some contexts (e.g., databases) throughput can refer to the operation rate (operations per second or transactions per second).
3. **Response time:** The time for an operation to complete. This includes any time spent waiting and time spent being serviced (service time), including the time to transfer the result.

4. **Latency:** A measure of time an operation spends waiting to be serviced. In some contexts it can refer to the entire time for an operation, equivalent to response time.

5. **Utilization:** For resources that service requests, utilization is a measure of how busy a resource is, based on how much time in a given interval it was actively performing work. For resources that provide storage, utilization may refer to the capacity that is consumed (e.g., memory utilization).

6. **Saturation:** The degree to which a resource has queued work it cannot service.

7. **Bottleneck:** In systems performance, a bottleneck is a resource that limits the performance of the system. Identifying and removing systemic bottlenecks is a key activity of systems performance.

8. **Workload:** The input to the system or the load applied is the workload. For a database, the workload consists of the database queries and commands sent by the clients.

9. **Cache:** A fast storage area that can duplicate or buffer a limited amount of data, to avoid communicating directly with a slower tier of storage, thereby improving performance. For economic reasons, a cache is often smaller than the slower tier.

2.2 Time series

A time series is a realization of a stochastic process

$$X = \{X_t, t \in T\}, \tag{2.1}$$

where X_t is a random variable observed at time t from a probability space $(\Omega, \mathcal{F}, P)$, T is the index set, Ω is the sample space, $\mathcal{F}$ is the σ-algebra, and P is the probability measure. The index set T can either be *discrete* or *continuous*, and the random variable can be either *univariate* or *multivariate*. We will only consider time series with a discrete index set representing time, where subsequent time points are equally spaced and monotonically increasing, represented as $T = \{1, 2, \ldots, n\}$, where n is the number of observations.

Examples of time series data we will be working with include, for example, the userspace CPU utilization, the number of requests per second received by a server or the throughput of an NIC at a particular time. We call these time series *server metrics* - measurements of various aspects of the system's performance and behavior. The metrics we will be working with can be broadly classified into the categories listed in Table 2.1 and the semantics of these metrics fall into three distinct categories listed in Table 2.2.

Metric	Description
Application metrics	exported by a running application, such as code execution time or event counts.
System metrics	exported by the operating system, such as CPU utilization, memory utilization, or disk IOPS.
Hardware metrics	exported by the hardware drivers, such as the number of packets dropped by an NIC.

Table 2.1 Categories of server metrics

Category	Description
Counters	Monotonically increasing values, such as the number of requests served by a server.
Gauges	Instantaneous values, such as the CPU utilization.
Histograms	Distributions of values, such as the distribution of event loop iteration latency.

Table 2.2 Semantics of server metrics

2.2.1 Seasonality

A time series is said to have a seasonality when it exhibits a repeating pattern at regular intervals. Seasonality can be daily, weekly, monthly, yearly, or any other fixed and *known* period. The seasonalities in the data considered for our purposes are either daily or weekly, mainly caused by varying counts of online users accessing the system at different times of the day or week. This seasonality can be seen in the example of a weekly seasonal pattern in the number of bytes served by a server in Figure 2.1.

Longer seasonal patterns, such as monthly or yearly, are unlikely to impact the data we will be working with significantly. For example, when forecasting the number of requests served by a server in the next few hours based on the data from the past few days, the number of requests served on the same day of the week in the previous week is likely to have a much more significant impact on the forecast, than the number of requests served on the same day of the month in the previous month.

2.2.2 Trend

A time series is said to have a trend when it exhibits a long-term increase or decrease in mean over time. This increase can either be *linear* or *non-linear* and can be caused by various factors. For all of the time series we will be working with, we expect the trend to be linear and mostly *constant*, sometimes with drastic changes caused by external factors, such as a sudden

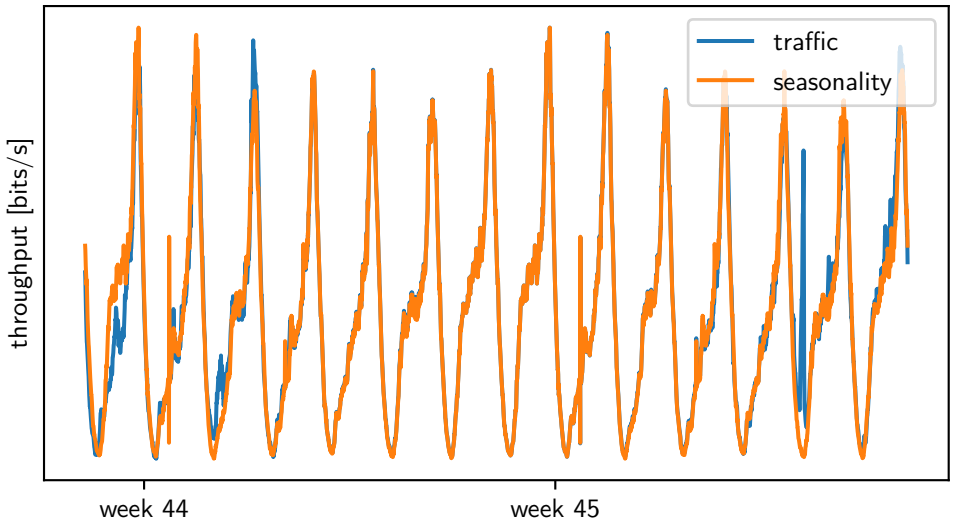

Figure 2.1 Example of a weekly seasonal pattern in the number of bytes transmitted per second

increase in the number of users accessing the server, caused by rerouting traffic from a failing instance to a healthy one.

Most of the time, the series we will be working with will be detrended, as the trend does not provide any interesting information. An example of such a time series is the total number of bytes served by a server. In this case, we are only interested in the immediate changes in the number of bytes served - the current throughput in bytes per second. These series will be exported by *counters*.

In particular time series, we might consider a large positive trend indicative of a potential problem and label it an anomaly. For example, a server's memory utilization exhibiting a large positive trend might be indicative of a memory or space leak, warranting further investigation.

2.2.3 Stationarity

Stationarity is a key assumption in many time series models, such as ARIMA, and it is required for the model to make accurate forecasts. A time series can be considered stationary if the mean, variance and autocorrelation of the series are constant over time. Due to this, time series with a trend or a seasonal pattern are not stationary [2].

Stationary can be tested by various statistical tests, such as the Augmented Dickey-Fuller (ADF) or Kwiatkowski-Phillips-Schmidt-Shin (KPSS) tests.

2.2.4 Autocorrelation

Autocorrelation is a measure of the correlation between a time series and a lagged version of itself, the autocorrelation function (ACF) produces the autocorrelation at different lags and is defined as

$$r_k = \frac{\sum_{t=k+1}^{T}(x_t - \bar{x})(x_{t-k} - \bar{x})}{\sum_{t=1}^{T}(x_t - \bar{x})^2}, \tag{2.2}$$

where r_k is the autocorrelation at lag k, x_t is the value of the time series at time t, $\bar{x}$ is the mean of the time series [3].

Since the values measure the relationship between x_t and x_{t-k}, if x_t is highly correlated with x_{t-1}, it must be highly correlated with x_{t-2}, x_{t-3}, and so on. To overcome this issue, the partial autocorrelation function (PACF) is used, which measures the correlation between x_t and x_{t-k} after removing the effect of the intermediate lags $x_{t-1}, x_{t-2}, \ldots, x_{t-k+1}$ [4].

2.2.5 Structural breaks

Structural breaks are *sudden changes* in the mean or variance of a time series [5]. As mentioned in the subsection 2.2.2, changes in the mean of certain time series can be caused by re-routing traffic between clusters and servers. This causes a significant challenge for the forecasting or anomaly detection models, as they need to be able to adapt to these changes. This may not be possible due to a lack of historical data or the suddenness of the change.

However, we do not expect this to be a significant issue because these events are rare and often done manually by administrators as a last resort in the case of a failing server or an overloaded cluster.

Structural breaks may be detected using a variety of methods, such as the Chow test introduced by the econometrician Gregory Chow in 1960 in his paper Tests of Equality Between Sets of Coefficients in Two Linear Regressions [6], or by using a similarity metric defined over rolling windows of the time series. A nice implementation of the latter can be found in the `ruptures` Python library [7].

2.3 Time series forecasting

Time series forecasting is the process of predicting future values of a time series based on past observations. For the purposes of this thesis, we are not interested in making long-term forecasts, which could be used for server cluster capacity planning or hardware allocation. We will instead be interested in making short-term forecasts and leveraging them as a tool to answer a different question: Is the current value of the time series deviating significantly from the historical values - is the server behaving as expected?

2.3.1 Prediction intervals

A prediction interval with a *coverage probability* of p gives us a range of values within which we expect a random variable to fall. Computing a prediction interval generally requires us to know the distribution of the forecasted values, assuming the distribution is normal, the prediction interval for an n-step-ahead forecast is computed as

$$\hat{y}_{t+n|t} \pm c\hat{\sigma}_n, \tag{2.3}$$

where $\hat{\sigma}_h$ is the estimate of standard deviation of the n-step forecast distribution, the multiplier c is chosen based on the desired coverage probability, for example $c = 1.96$ in the case of a 95% coverage probability [8].

An estimation of the forecast's standard deviation for a one-step-ahead forecast can be obtained from the errors in the following way

$$\hat{\sigma}_1 = \sqrt{\frac{1}{T-K-M}\sum_{t=1}^{T} e_t^2}, \tag{2.4}$$

where T is the number of observations, K is the number of parameters in the model, and M is the number of missing values in the errors (i.e., those unavailable due to a lack of lagged observations) [8].

When the assumption of normality does not hold, or the standard deviation of the n-step-ahead forecast distribution is difficult or impossible to estimate, we can use the bootstrap method introduced by Davidson and Hinckley in their 1997 paper Bootstrap Methods and Their Application [9].

2.3.2 Traditional methods

For the purposes of this thesis, we are considering traditional methods to be those that are not based on (deep) neural networks, but are instead based on statistical models or curve fitting. These methods, including ARIMA, Holt-Winters, or linear regression, have been widely studied and used for time series forecasting for decades in various domains.

ARIMA

First introduced by George Box and Gwilym Jenkins in their 1970s book Time Series Analysis: Forecasting and Control [10], currently published as fifth edition in 2015, the autoregressive integrated moving average (ARIMA) model is a predictive model used for time series forecasting.

The ARIMA model consists of three components [11]. The *autoregressive* component (AR) models the relationship between an observation and a number of lagged observations. It takes the form of

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t, \tag{2.5}$$

where ϕ_i are the parameters of the model, p is the order of the autoregressive component, and ϵ_t is the error term.

The *differencing* component (I) is used to make the time series stationary before it is modeled and is defined as

$$\Delta^d X_t = \begin{cases} X_t - X_{t-1} & \text{if } d = 1, \\ \Delta^{d-1} X_t - \Delta^{d-1} X_{t-1} & \text{if } d > 1, \end{cases} \tag{2.6}$$

where Δ is the difference operator and $d \in \mathbb{N}$ is the order of differencing.

The *moving average* component (MA) models the relationship between an observation and a number of lagged forecast errors. It takes the form of

$$X_t = \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \tag{2.7}$$

Putting it all together, the ARMA model can be written as

$$X_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \tag{2.8}$$

Several approaches can be taken to introduce seasonality into the ARIMA model, such as extending the ARIMA model by adding seasonal components to the autoregressive, differencing, and or moving average components. This model is denoted as ARIMA$(p, d, q) \times (P, D, Q, s)$, where s is the seasonal period and P, D, Q are the seasonal orders of the autoregressive, differencing, and moving average components.

When working with long seasonal periods, such as yearly or monthly, the seasonal ARIMA model can become computationally expensive and challenging to fit. In such cases, it is recommended to provide the model with the seasonal component as an exogenous variable, for example, in the form of a Fourier series fitted to the seasonal pattern. Such a model could take the following form

$$X_t = \sum_{i=1}^{K} \beta_i \sin\left(\frac{2\pi i t}{s}\right) + \gamma_i \cos\left(\frac{2\pi i t}{s}\right) + N_t, \tag{2.9}$$

where β_i and γ_i are the Fourier coefficients, K is the number of harmonics, and N_t is the ARIMA model [12].

Holt-Winters

The Holt-Winters method, also known as triple exponential smoothing, is a forecasting method based on Holt's linear method introduced by Peter Winters in 1960 [13]. It comprises three components - level, trend, and seasonality-

combined in an additive or multiplicative manner. The additive model is defined as

$$\begin{aligned} \hat{y}_{t+h|t} &= l_t + hb_t + s_{t+m-h_{m+1}}, \\ l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1}), \\ b_t &= \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}, \\ s_t &= \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}, \end{aligned} \tag{2.10}$$

where $\hat{y}_{t+h|t}$ is the forecast at time $t+h$ based on the data up to time t, l_t is the level at time t, b_t is the trend at time t, s_t is the seasonal component at time t, α, β, and γ are the smoothing parameters, and m is the seasonal period [14].

Considering different combinations of additive and multiplicative components, various exponential smoothing models can be derived, an incomplete list of which is shown in Table 2.3.

Seasonality	Trend	Model
None	None	Simple exponential smoothing
None	Additive	Holt's linear method
None	Multiplicative	Holt's method
Additive	None	Seasonal naive method
Additive	Additive	Additive Holt-Winters' method

Table 2.3 Exponential smoothing models [15]

Prophet

Open-sourced by the Facebook Core Data Science team in 2017, together with an accompanying paper Forecasting at scale, Prophet is a forecasting model designed to be robust to missing data, outliers, changes in the trend, seasonalities, holidays, while also being easy to use and tune by non-experts [16].

The model is based on a decomposable time series model with three additive components - trend, seasonality, and holidays. The model is defined as

$$X_t = G_t + S_t + H_t + \epsilon_t, \tag{2.11}$$

where G_t is the trend (or growth) component, S_t is the seasonality component, H_t is the holiday component, and ϵ_t is the error term [16].

2.3.3 Deep learning methods

Time series forecasting based on neural networks is most commonly done using recurrent neural networks (RNNs) or their variants, such as long short-term memory (LSTM) networks or gated recurrent units (GRUs). These models have been shown to be effective at capturing complex patterns in the data

and are able to learn *relatively* long-term dependencies. In practice, however, learning truly long-term dependencies is a challenge. Training a deep learning model for time series forecasting is a *supervised* form of machine learning, where the model is trained on historical input-output pairs.

Recurrent Neural Networks

A recurrent neural network (RNN) is a type of neural network designed to handle sequential data. It is characterized by its ability to maintain a state or memory of the previous inputs, which allows it to model temporal dependencies in the data. This behavior is achieved by introducing cycles in the network, which allow the hidden state to be passed from a previous time step to the next.

A single RNN cell's output is calculated as

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h), \tag{2.12}$$

where h_t is the hidden state at time t, W_{hh} and W_{xh} are the weight matrices, b_h is the bias vector, and tanh is the hyperbolic tangent function. This design was first introduced by Elman in 1990 in his paper Finding Structure in Time [17], a similar design was later introduced by Jordan in 1997 in his paper Serial Order: A Parallel Distributed Processing Approach [18]. Today, they are collectively referred to as simple RNNs.

In practice, the RNNs have been shown to be difficult to train due to the vanishing gradient problem, which occurs when the gradients of the loss function with respect to the weights become very small, making it difficult for the model to learn long-term dependencies in the data. Or the exploding gradient problem, which occurs when the gradients become very large, making the model unstable and unable to converge to a solution [19].

Long Short-Term Memory

First introduced by Hochreiter and Schmidhuber in 1997 in their paper Long Short-Term Memory [20], the LSTM network is designed to overcome the vanishing gradient problem that occurs in traditional RNNs.

This is achieved by introducting a gating mechanism which allows the network to learn which information to keep and which to discard. An LSTM cell's state and output is calculated as

$$\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i), \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f), \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c), \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o), \\
h_t &= o_t \odot \tanh(c_t),
\end{aligned} \tag{2.13}$$

where i_t, f_t, o_t are the input, forget, and output gates and c_t is the cell state, with their respective weight matrices and bias vectors $W_{*\{i|f|c\}}$ and $b_{*\{i|f|c|o\}}$. h_t is the hidden state (or output). The weight matrices W and the bias vectors b are learned during training, and σ is the sigmoid function [21].

Due to the network being stateful across time steps, the initial cell state c_0 and hidden state h_0 must be initialized before the first time step. The initial state is usually set to zero or initialized by random noise while training, and is often trained as a parameter of the model.

While training the weights and biases of an LSTM network, an issue with exploding or vanishing gradients may occur in the case of deep networks, just as with traditional RNNs. This can be mitigated by using gradient clipping, which involves scaling the gradients should they exceed a certain threshold, or by normalizing the gradients across training batches.

2.3.4 RNN data preprocessing

Data preparation for a univariate forecasting RNN model with a forecasting horizon of h time steps can be outlined in the following steps. The process assumes the data is preprocessed according to the requirements of the model (e.g., standardized).

Let $X = \{x_t\}_{t=1}^{T}$ be the input time series, where $x_t \in \mathbb{R}^f$ is the feature vector at time t and T is the number of observations, and $y = \{y_t\}_{t=1}^{T}$ be the target time series, where $y_t \in \mathbb{R}$ is the target value at time t.

Split the input time series into overlapping or non-overlapping windows of length w with a stride s. Each window becomes a subsequence

$$X_k = \{x_i\}_{i=ks}^{ks+w-1}, k = 0, 1, \ldots, K-1, \tag{2.14}$$

where $K = \left\lfloor \frac{T-w-h}{s} \right\rfloor$ is the number of subsequences, and $x_k \in \mathbb{R}^{w \times f}$ is the subsequence.

The target vector y_k is defined as

$$y_k = \{y_i\}_{i=ks+w}^{ks+w+h-1}, y_k \in \mathbb{R}^h. \tag{2.15}$$

The subsequences X_k and y_k then form the input-output pairs, which are further split into training and testing sets and finally batched and fed into the model during training.

2.4 Anomaly detection

Anomaly detection is the process of identifying samples that differ significantly from the rest of the data. In the context of time series, anomalies can take many forms, such as sudden changes in the mean, trend, or seasonality, in the form of structural breaks, or as simple outliers. There are three main types of anomalies that can be present in time series data, as defined by [22]:

- *Point anomalies* are individual data points that deviate significantly from the rest of the data.
- *Contextual anomalies* are data points that are anomalous in a specific context but not in others.
- *Collective anomalies* are a group of data points that deviate significantly from the rest of the data.

In the context of server monitoring, we find that point anomalies are often associated only with metrics collected at a very low frequency. Some examples of anomalies we will be primarily concerned with include, but are not limited to, periodic and aperiodic spikes, or peaks, in CPU utilization-related metrics, long-term changes or an unbound growth in memory utilization, or clipping (in the signal processing sense) of various metrics, indicative of a bottleneck present in the system.

2.4.1 Approaches to flagging anomalies

Anomaly detection encompasses a wide range of approaches designed to identify deviations from the typical patterns in data. *Supervised* methods rely on labeled datasets, where instances of typical and anomalous behavior are explicitly provided. These approaches generally involve training a classifier to distinguish between typical and anomalous data based on these labels.

In contrast, *unsupervised* methods operate without any explicitly labeled data, for example, by assigning an *anomaly score* to each data point and comparing it against a threshold, or by using density or distance-based methods such as Isolation Forest or Local Outlier Factor.

Due to the scale of the data we are working with, our focus will be solely on unsupervised anomaly detection methods. This decision is driven by the sheer number of metrics we are collecting, which run into the thousands. Many of these metrics are highly specialized, being relevant only to specific subsystems, and as such, labeling them would be a very time-consuming and error-prone process, which would require a deep understanding of the system and its components.

The servers from which the metrics are scraped are subject to a wide variety of different workloads and software and hardware configurations, making it difficult to label the data in a meaningful way. For example, having a labeled dataset for a server with a specific model of an NIC, which could be replaced due to a hardware failure or a required upgrade, would render the labels for the NIC-related metrics obsolete.

2.4.2 Statistical methods

One commonly used method for anomaly detection in time series data is based on the use of rolling statistics, such as the mean and standard deviation.

An anomaly detection rule based on a rolling standard deviation may be implemented as follows:

$$A_t = \begin{cases} 0 & \text{if } \bar{x}_{t-n:t} \leq \alpha \\ 0 & \text{if } \Delta x_t \leq \beta \\ 1 & \text{if } -\Delta x_t > 3 * s(x_{t-n:t-m}) \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

where $\bar{x}_{t-n:t}$ is the mean of the last n observations, $-\Delta$ is the negative differencing operator with order 1 and $s(x_{t-n:t-m})$ is the standard deviation of the observations falling within the closed interval $[t-n, t-m]$. This rule would ignore any time series whose rolling mean is below a certain threshold, whose rate of change is below a different threshold, and would flag any time points whose negative change is more than three standard deviations from the mean.

With the correct choice of parameters, this rule, in particular, can be used for online detection of sudden drops in the throughput of a server, which could be indicative of a wide variety of problems. The main issue with this method is that it is prone to false positives, as it is based on the assumption that the data is normally distributed and is very sensitive to slow, gradual growths followed by sudden drops - a pattern we see very often in our data.

2.4.3 Methods based on forecasting

Anomaly detection based on forecasting involves training a model on historical data and using it to predict future values. The difference between the predicted and the actual value can then be used as a measure of the anomaly, or by using the prediction interval as a measure of the uncertainty of the prediction. This method is based on the assumption that the model has learned the underlying patterns in the data and can accurately predict future values.

Several issues can arise when using forecasting-based anomaly detection methods, mainly due to unexpected out-of-sample changepoints in the underlying metrics affecting the modelled metric. This can happen, for example, due to reasons outlined in subsection 2.2.5.

The general process of forecasting-based anomaly detection on errors can be outlined as follows:

1. Split the data into a training and a test set.
2. Train the forecasting model on the historical data in the training set.
3. Forecast the out-of-sample values using the trained model on the test set.
4. Compute the standard deviation of the errors from the test set.
5. Compare the standard deviation of the out-of-sample errors to the test set errors and flag any time points whose errors are more than n standard deviations from the mean.

2.4.4 Methods based on reconstruction errors

Anomaly detection based on reconstruction errors involves training a model to reconstruct the input data and using the difference between the input and the reconstructed data as a measure of the anomaly.

A similar approach to labeling data points as anomalies is taken by the reconstruction-based methods, as the approach outlined in subsection 2.4.3, and the issue with unexpected out-of-sample changepoints still applies.

Autoencoders

An autoencoder consists of an encoder, which compresses the input data into a *latent* representation, and a decoder, which then reconstructs the latent representation back into the input data. No further information is provided to the network about the data, and it is expected to learn the underlying patterns during training, and as such is a form of unsupervised learning. The difference between the input and its reconstruction is called the *reconstruction error*, which is then minimized during training as an objective.

Certain constraints may be placed on the latent representation, such as a specific shape or distribution, an example of which is the variational autoencoder explained in more detail in subsection 2.4.4. Without any added constraints, the latent representation corresponds to a one-to-one mapping of the input data.

Denoting the encoder as f and the decoder as f', the output of the autoencoder is calculated as $\hat{x} = f'(f(x))$, where $\hat{x}$ is the reconstructed input data, x is the input data, $f(x)$ is the latent representation, and $x - f'(f(x))$ is the reconstruction error. During anomaly detection, the reconstruction error is used as a measure of the anomaly, with a higher error, indicating a higher likelihood of an anomaly. This approach is based on the assumption that anomalous data points or sequences are rare and therefore, the network will be unable to reconstruct them accurately.

The encoder and decoder can be implemented using a variety of neural network architectures, such as a multi-layer perceptron, a convolutional neural network, or a recurrent neural network.

Variational autoencoders

Variational autoencoders place a constraint on the latent representation to be of a specific shape, such as that of a normal distribution. In contrast to traditional autoencoders, which learn a one-to-one mapping of the input data, variational autoencoders map each input data point to a distribution in the latent space. The encoder in a variational autoencoder outputs two vectors, the mean and the variation of the distribution, which are then used to sample a point used to reconstruct the input data.

During training, the network is trained to minimize both the reconstruction error, using, for example, the mean squared error, as well as the Kullback-Leibler divergence between the learned and the target distributions.

The Kullback-Leibler divergence is a statistical distance measuring how a distribution differs from a target distribution, defined as

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log \left(\frac{P(x)}{Q(x)} \right), \tag{2.17}$$

where Q is the trained distribution and P is the target distribution. The following properties hold for the KL divergence:

- $D_{KL}(P||Q) \geq 0$,
- $D_{KL}(P||Q) = 0$ if and only if $P = Q$,
- $D_{KL}(P||Q) \neq D_{KL}(Q||P)$,

in other words, the KL divergence is not symmetric and is always non-negative. Due to this, the KL divergence is not a true distance measure, as it is not symmetric and does not satisfy the triangle inequality.

In the case of a normal distribution, with a prior of $\mathcal{N}(\mathbb{0}, \mathbb{1})$ and posterior of $\mathcal{N}(\mu, \sigma)$, both k-dimensional, the Kullback-Leibler divergence can be calculated as

$$D_{KL}(\mathcal{N}(\mu, \sigma)||\mathcal{N}(\mathbb{0}, \mathbb{1})) = \frac{1}{2} \left(\mu^2 + \sigma^2 - k - \sigma \right). \tag{2.18}$$

The resulting objective function, the evidence lower bound (ELBO), can be written as

$$\mathcal{L} = \mathcal{L}_{\text{reconstruction}} + \mathcal{L}_{\text{KL}}, \tag{2.19}$$

where $\mathcal{L}_{\text{reconstruction}}$ is the reconstruction loss, for example an L1 or L2 loss, and $\mathcal{L}_{\text{KL}}$ is the loss from the Kullback-Leibler divergence. The KL divergence term acts as a regularizer, forcing the latent representation to be close to the prior distribution.

LSTM autoencoders

LSTM autoencoders use LSTM in place of a feedforward neural network in the encoder and decoder. The encoder consists of one or more stacked LSTM layers, which pass their hidden states to the next layer, whose final output is the latent representation. The dimensionality reduction is achieved by reducing the number of LSTM cells in the LSTM layers. Therefore, the latent space is a vector of equal size to the number of LSTM cells in the last LSTM layer. In order to reconstruct the input data, the latent vector is then passed to the decoder for the same number of time steps as the original input sequence. The

encoder state is responsible for reconstructing the input data from a repeated vector from the latent space.

In the case of variational LSTM autoencoders, the hidden state of the LSTM cells is fed into two separate dense layers, representing the mean and the variance of the distribution, which take shape of num_time_steps×latent_dim, and are then used for sampling from the normal distribution.

Convolutional autoencoders

A convolutional layer is a layer consisting of several learnable *kernels*, which are *convolved* with the input data by "sliding" over it, computing a dot product between the kernel weights and the input data, and passing the result through an activation function, the result of which is called a *feature map*, which contains the learned features of the input data.

A discrete 1D convolutional operation is defined as

$$(x \star w)_n = \sum_{m=0}^{M-1} x_{n-m} w_m, \tag{2.20}$$

where x is the input data, w is the kernel, and n is the index of the output. In the context of CNNs, the operation is often performed using the *cross-correlation* operation instead. The difference between the two is that in the cross-correlation operation, the kernel is *not flipped*, so the operation can be written as

$$(x \star w)_n = \sum_{m=0}^{M-1} x_{n+m} w_m. \tag{2.21}$$

This does not matter, as the weights of the kernel are learned during training, but it is an interesting detail to note. The resulting output of a simple 1D convolutional layer is

$$y_{i,j} = \text{bias}(C_{\text{out}j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}j}, k) \star x_{i,k}, \tag{2.22}$$

where C_{in} and C_{out} are the number of input and output features, N is the batch size and $\star$ is the cross-correlation operator [23].

Since convolving a kernel over the input data results in a smaller output, padding is often added to the input data to preserve its size, such as zero padding or same-value padding [24].

In the case of a Conv-AE, the encoder consists of one or more stacked convolutional layers, which reduce the input data to a latent representation, which is then passed to the decoder, often implemented as a mirrored version of the encoder, consisting of one or more stacked deconvolutional layers.

The dimensionality reduction can be achieved by either using a stride greater than one in the convolutional layers, by using pooling layers, which

reduce the size of the input data by aggregating the values in a window, or by reducing the number of kernels in the convolutional layers.

2.4.5 Autoencoder data preprocessing

Similar to the process outlined in subsection 2.3.4, data preparation for an autoencoder model can be outlined in the following steps. Again, the process assumes the data is already preprocessed according to the requirements of the model.

Let $X = \{x_t\}_{t=1}^{T}$ be the input time series, where $x_t \in \mathbb{R}^f$ is the feature vector at time t and T is the number of observations.

Split the input time series into overlapping or non-overlapping windows of length w with a stride s. Each window becomes a subsequence

$$X_k = \{x_i\}_{i=ks}^{ks+w-1}, k = 0, 1, \ldots, K-1, \tag{2.23}$$

where $K = \left\lfloor \frac{T-w}{s} \right\rfloor$ is the number of subsequences, and $x_k \in \mathbb{R}^{w \times f}$ is the subsequence.

The subsequences X_k then form the input-output pairs, which are further split into training and testing sets and finally batched and fed into the model during training.

2.5 Monitoring tools

This section provides a concise overview of widely deployed and used monitoring tools and their features. Ranging from metric collecting daemons, such as vmagent, to time series databases, such as Prometheus and VictoriaMetrics, along with their respective query languages.

2.5.1 VictoriaMetrics

VictoriaMetrics, licensed under the Apache License 2.0, is a high-performance time series database designed to store and query large amounts of time series data. It builds on the ideas of Prometheus, a popular monitoring system and time series database. Moreover, it is designed to act either as a drop-in replacement for it or as a long-term storage solution [25].

Metric ingestion

VictoriaMetrics can ingest data in the Prometheus exposition format, depicted in Code listing 2.1, which is a simple text-based format for representing time series data. The first column contains the metric name, along with optional key-value pairs called labels, and the second column contains the value of the metric.

The data is ingested using the Prometheus remote write protocol, which allows for the data to be sent in batches, reducing the overhead of sending individual data points. This is most often done using a Prometheus remote write adapter, such as vmagent, which scrapes the data from the monitored services in predefined intervals and sends it to a VictoriaMetrics instance [26].

Code listing 2.1 Prometheus exposition format

```
# HELP tx_quic The total number of bytes sent over HTTP3/QUIC
# TYPE tx_quic counter
tx_quic{listener="443",method="get",code="200"}  123
tx_quic{listener="443",method="get",code="404"}  456
```

MetricsQL

VictoriaMetrics uses a query language called MetricsQL, a superset of PromQL, the Prometheus query language. The full list of supported functions and operators, as well as syntax references, can be found in the official documentation [27].

An example of a MetricsQL query is shown in Code listing 2.2, this command is a translation of Equation 2.16 into MetricsQL.

Code listing 2.2 MetricsQL query

```
group(average_over_time(tx_quic[5m] > 10G)) by (listener)
* on (listener) (delta(tx_quic[5m:30s]) < 1G)
< on (listener) (3 * stddev_over_time(delta(tx_quic[6h:30s])))
```

Other features

VictoriaMetrics offers a wide range of utilities and tools to work with the data stored in it, such as vmalert - a tool for threshold-based alerting for data stored in VictoriaMetrics [28].

A paid, separately licensed enterprise version is also offered, which, among other utilities, includes a closed-source anomaly-detection solution called vmanomaly, which offers a respectable range of anomaly detection methods based on well-known models such as Holt-Winters, Facebook Prophet, STL decomposition, or Isolation Forest [29].

2.5.2 Node Exporter

Node Exporter, licensed under the Apache License 2.0, is a Prometheus exporter for hardware and OS metrics exposed by *NIX kernels, which scrapes the host system's metrics and exports them in the Prometheus exposition format. Each set of metrics is collected by a separately configurable *collector*, an example of which is the *cpu collector*, which collects various CPU-related

metrics, or the *ethtool collector*, which collects NIC metrics available from the `ethtool` command. The full list of available collectors can be found in project README [30].

Chapter 3

System Requirements

As mentioned in the introduction, the goal of this thesis is to design and implement a reliable and scalable monitoring and health-checking system capable of detecting anomalies in various host system and application metrics.

We place a special emphasis on the behavior of the monitored system after a change to its configuration or software, further referred to as *post-period*, in contrast to the behavior during the *pre-period*, in order to aid with performance analysis of in-development products and features during *canary testing* and *blue-green deployments*, by automatically analyzing key performance metrics and detecting performance regressions.

Canary testing constitutes deploying a new version of software to either a small subset of users, or, in our case, a small subset of server instances. Whereas blue-green deployments, most suitable for containerized applications, involve running two distinct production environments, between which traffic is slowly shifted in order to test the new environment's stability and performance, while keeping the old environment running in case of a failure [1].

3.1 Monitoring methodologies

There are two common methodologies used to monitor system performance, the USE method introduced by Brendan Gregg [1], and the RED method, introduced by Tom Wilkie [31].

We choose to employ these methodologies in our monitoring system, as they are well-established and widely used in the industry, as well as easy to understand, implement, and automate.

The USE method

The Utilization Saturation and Errors (USE) methodology is used to monitor the system's resource usage under load, and should be used early on in the

performance investigation process to identify bottlenecks. It can be summarized as: "For every resource, check utilization, saturation, and errors. If any show anomalous readings, investigate further" [1], with the definitions of these terms provided below:

- **Resources:** The physical server functional components.
- **Utilization:** The percentage of time a resource is busy servicing work.
- **Saturation:** The degree to which the resource has extra work that it cannot service.
- **Errors:** The count of error events.

An example of CPU utilization and saturation metrics written in MetricsQL on NodeExporter metrics is shown in Code listing 3.1.

Code listing 3.1 CPU utilization and saturation in MetricsQL

```
# CPU utilization in the last minute
1 - avg(rate(node_cpu_seconds_total{mode="idle"}[1m]))

# CPU saturation in the last minute
sum(node_load1) / sum(node:node_num_cpu:sum)
```

The RED method

The RED method, standing for Rate, Errors, and Duration, is used to monitor the system's performance from the perspective of the end-user. It can be summarized as: "For every service, check the request rate, errors, and duration" [31], with the definition of these terms provided below:

- **Rate:** The number of requests per second.
- **Errors:** The number of failed requests per second.
- **Duration:** The time taken to complete a request.

Monitoring request rate, error rate, and request duration metrics can help determine if the issue lies with the system's load or architecture. If the request rate is steady while the request duration is increasing, it points to an architectural problem. In comparison, if both the request rate and request duration are increasing, it points to a problem with load [1], and workload analysis should be performed.

An example of request rate, error rate, and request latency metrics written in MetricsQL is shown in 3.2.

Code listing 3.2 Request rate, error rate, and request duration in MetricsQL

```
# Request rate in the last minute
sum(rate(remove_resets(http_requests_total)[1m]))

# Error rate in the last minute
sum(rate(
  remove_resets(http_requests_total{status=~"5.."}[1m])
))

# Percentage of requests with first byte latency under 50ms
# calculated from a cumulative histogram
sum(rate(remove_resets(response_latency_bucket{le="50"})))
/
sum(rate(remove_resets(response_latency_bucket{le="+Inf"})))
```

3.2 What constitutes an anomaly?

The set of anomalies and deviations from the expected behavior is far too broad for it to be possible to provide an exhaustive list.

In [32], the authors are concerned with multivariate time-series point anomaly detection in server metrics, such as CPU utilization, memory utilization, disk IOPS, and network traffic. The dataset used in the study was collected from a real-world production environment, the metrics themselves, however, were anonymized.

In [33], the authors are concerned with detecting performance anomalies in API gateways, and define such anomalies as high CPU utilization, high memory utilization, high disk IOPS, all due to request mediation or a long response time (latency) of backend services. The dataset used in the study was artificially generated, for example, by running code that allocates 500MB of memory on each request. To our best knowledge, this is the only publicly available non-anonymized metrics dataset that can be used for anomaly detection in the context of proxy servers.

We extend this definition by considering the following additional anomalies, as well as some of the aforementioned ones, albeit in an interpretation that is more suitable for caching proxy servers.

1. long-term increase in CPU utilization
2. sudden, seemingly irregular short-term increase in CPU utilization
3. increased memory utilization
4. unbound growth of memory utilization
5. decrease in cache hit ratio

6. transfer throughput bottleneck

Long-term increases in CPU utilization or performance regressions may cause various issues, such as increased response latency or decreased throughput, and may be caused by factors like lock contention or atomic false sharing, as well as ”simple” algorithmic inefficiencies. Two common approaches to monitoring performance regressions are discussed in 3.3.

Sudden and seemingly irregular short-term increases in CPU utilization, or spikes, may be indicative of, counterintuitively, serious issues. One of the more common causes for such behavior is a misconfigured garbage collector or a misconfigured log managing utility such as `logrotate` [1]. Being configured to run too infrequently or without a nice value [2], can cause either to monopolize CPU resources for a long time, which is particularly problematic when the main workload is CPU-bound, pinned to a specific core [3], and low response latency is crucial.

Cache hit ratio and eviction rate are important metrics to monitor when dealing with caching proxies, as their anomalous values may be indicative of misconfigured caching policies or incorrect handling of Cache-Control [4] headers. Misconfigured caching policies may lead to a vast increase in cache fill traffic, which may lead to an overload of the origin server and, in turn, increased response latency and decreased throughput.

Bottlenecks in transfer throughput may be caused by various factors, some of which are beyond our scope - such as network congestion. The factors we will be concerned with the most are ones such as a misconfigured Linux networking stack [5], or programming errors in the application's networking code, for example, bugs in userspace protocol implementations, such as HTTP3/QUIC [6].

3.3 Monitoring performance regressions

There are two common perspectives to consider when monitoring performance - *resource analysis* and *workload analysis*. The tasks of workload analysis include identifying and confirming issues. For example, by looking for latency beyond an acceptable threshold, then finding the source of the latency and confirming that the latency is improved after applying a fix. Or by applying a well-characterized workload, such as a synthetic benchmark, and measuring the system's performance [1].

In comparison, resource analysis involves monitoring the system's resource usage, such as CPU, memory, and disk IO, when under load. This perspective

[1] https://linux.die.net/man/8/logrotate
[2] https://www.man7.org/linux/man-pages/man2/nice.2.html
[3] https://www.man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html
[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control
[5] https://www.man7.org/linux/man-pages/man8/tc.8.html
[6] https://www.rfc-editor.org/rfc/rfc9000.html

focuses on monitoring utilization to identify when resources are at or near their limits, or whether they are being used inefficiently [1].

The latter approach is particularly suitable to be done preemptively, as it can help identify potential bottlenecks before they become a problem. It is most commonly performed by system administrators during capacity planning in order to gather information to help make informed decisions about the system's future growth [1].

One common approach to workload analysis is the use of *micro-benchmarks*. These are small, isolated tests, run in a controlled environment as part of a CI pipeline, that measure the performance of specific components within a program or system by collecting various performance counters. The target of the micro-benchmark is typically a small piece of code, such as a function or data structure, which is common in the code's critical path. As such, they do not represent the resulting system's overall performance characteristics. Instead, they serve as a preventative measure to catch performance regressions early and before they are deployed to production. The approach is based on the observation that the performance of a system is often dominated by a small number of critical components [34]. Micro-benchmarks can be implemented using various tools, including Google's benchmarking framework aptly named *Google Benchmark* [7]

Another approach involves using synthetic workloads that simulate the expected load on the system, using benchmarking tools such as *wrk* [35]. This approach would be considered a *macro-benchmark*. For example, we can generate a stream of HTTP requests and measure the request latency and other metrics of interest. However, in the context of multi-layered caching proxies, this approach can be particularly challenging to implement, as the entire system's load is highly variable and influenced by users' browsing habits and factors such as local cache state, which varies over time and between different proxy instances. Therefore, generating a realistic workload is non-trivial, so a supplementary check using different methodologies is needed to ensure the system behaves as expected.

In order to ensure that the system behaves as expected under real-world conditions, and to reduce the time running canary tests in production, after the system is rigorously tested by both micro and macro benchmarks, we choose to mirror [8] real-world workloads from a production environment to a staging environment, in which can then perform an automated performance analysis. Using real-world workloads mirrored from a production environment to a staging environment, is more realistic and can provide well-representative results, albeit at the cost of scale and repeatability, as well as a higher load on the production environment.

[7] https://github.com/google/benchmark

[8] https://gateway.envoyproxy.io/latest/tasks/traffic/http-request-mirroring/

3.4 HTTP proxy server monitoring

The primary focus of this thesis is on monitoring HTTP proxy servers. HTTP proxies can be deployed in a variety of configurations and cluster architectures, for example, as a sidecar proxy, providing advanced networking features such as load balancing, rate limiting, and circuit-breaking to a microservices application or as a content caching proxy in a single or multilayered cache.

This drives many distinct design decisions for a monitoring process. When monitoring a canary deployment within a multilayered caching proxy cluster, it seems fitting to use other servers within the same cluster as a reference since traffic between them is usually distributed equally. In practice, however, a non-uniform distribution of traffic between servers is common, either as a design decision due to hardware non-homogeneity or due to reasons such as third-party DNS caching.

The hardware configuration of servers within the same cluster should be consistent. However, in practice, the clusters may have non-homogeneous hardware configurations due to maintenance and component replacement. Different clusters are designed to accommodate varying traffic patterns, which leads to differences in their hardware configuration.

Given these factors, its historical performance data is the most reliable reference for identifying anomalous behavior on a specific server. Accordingly, we will employ a synthetic control method to detect performance regressions and anomalies in the monitored system.

3.5 Input data

The monitoring system will be utilizing data stored in the VictoriaMetrics time-series database, imported from various sources, such as Node exporter for system-level metrics and custom application-built-in exporter for application-level metrics. The scraping agents responsible for collecting the data, described in subsection 2.5.1, can be configured to scrape the data at different intervals.

The input data consists of a set of n time-series metrics represented as:

$$\{m_{iT_i} \mid i = 1, \ldots, n\}, \tag{3.1}$$

where m_{iT_i} denotes the i-th metric with an associated index set T_i. Each index set T_i consists of a series of timestamps defined as:

$$T_i = \{t + m\Delta_i t \mid m = 0, \ldots, k\}. \tag{3.2}$$

In this equation, t represents the timestamp of the first data point, k indicates the total number of data points, and Δ_i refers to the scraping interval of the i-th metric. It is important to note that these sets may vary in length and sampling intervals.

The metrics can categorized into two types:

- **Causal** metrics independent, or explanatory, of other metrics.
- **Response** metrics dependent on other metrics.

An example of a causal metric is the number of requests per second, while an example of a response metric is the CPU utilization of the server.

Chapter 4

System Architecture

The anomaly detection system is implemented as a Google Remote Procedure Call (gRPC) service with unary RPCs [1], composed of multiple components, each of which is responsible for either workflow orchestration, data and artifact storage, or model training and inference. The chosen technologies and tools are listed in Table 4.1.

Component	Responsibility
gRPC	Entry point for the system
Argo Workflows	Workflow orchestration
Metaflow	Workflow definition and observability
VictoriaMetrics	Time series storage
PostgreSQL	Metadata storage
Ceph with S3 interface	Workflow artifact and model storage

Table 4.1 Technology stack

The training and inference workflows are orchestrated using Argo Workflows via the Metaflow interface, in order to offload computational heavy operations to dedicated hardware, providing scalability, fault tolerance, and observability via the Argo UI and Metaflow Cards to the entire workflow execution.

In the case of unreachable services, such as VictoriaMetrics or PostgreSQL databases, we retry all operations with an exponential backoff before failing the entire RPC and returning an internal error.

4.1 gRPC interface

We chose gRPC as the communication protocol mainly due to its ease of use, and ability to generate client code in a variety of languages via `protoc`, allow-

[1] https://grpc.io/docs/what-is-grpc/core-concepts/#unary-rpc

ing for easy integration into whichever system requires the anomaly detection service. The interface is shown in Code listing 4.1.

Code listing 4.1 CanaryMonitor service definition

```
service CanaryMonitor {
  rpc BeginCanaryMonitor(BeginCanaryMonitorRequest)
      returns (BeginCanaryMonitorResponse) {}
  rpc GetCanaryStatus(GetCanaryStatusRequest)
      returns (GetCanaryStatusResponse) {}
  rpc RetireCanaryMonitor(RetireCanaryMonitorRequest)
      returns (RetireCanaryMonitorResponse) {}

  rpc CheckCanary(CheckCanaryRequest)
      returns (CheckCanaryResponse) {}
}
```

The interface defines a simple workflow for monitoring a given server for anomalies, the first three RPCs are used to start, poll, and stop the monitoring process respectively, while the last RPC is equivalent to executing `BeginCanaryMonitor` followed by `CheckCanary` RPCs.

BeginCanaryMonitor RPC

The `BeginCanaryMonitor` RPC is responsible for starting the canary monitoring process on a given server, its arguments and return value are displayed in Code listing 4.2. The input arguments are sanitized and validated before being processed by the service.

After sanitizing the input, the RPC creates a new monitoring job in the database and orchestrates a parametrized workflow to acquire, preprocess, and filter the server's `metrics`. All the steps taken during the data fetching workflow are outlined in section 4.2.

After obtaining the necessary data, the service spins up a workflow for each trainable `series_monitor` defined in the request, training the defined models and storing the resulting artifacts in the artifact storage.

Each `series_monitor` is responsible for detecting anomalies in a specific time series and may be executed periodically or on a call to the `GetCanaryStatus` RPC.

Code listing 4.2 BeginCanaryMonitor argument and return value definition

```
message MonitorOneOf {
  oneof monitor {
    canary.<...>.SeriesMonitor monitor_simple = 1;
    canary.<...>.PeriodicSeriesMonitor monitor_periodic = 2;
    canary.<...>.EnsembleSeriesMonitor monitor_ensamble = 3;
  }
}
```

```
message BeginCanaryMonitorRequest {
  canary.monitor.common.ServerInfo server_info = 1;
  canary.monitor.common.PrometheusInfo prometheus_info = 2;
  repeated canary.<...>.PrometheusQuery metrics = 3;
  repeated MonitorOneOf series_monitors = 4;
  google.protobuf.Timestamp pre_period_begin = 5;
  google.protobuf.Timestamp pre_period_end = 6;
}

message BeginCanaryMonitorResponse {
  string message = 1;
  optional MonitorInfo monitor_info = 2;
}
```

GetCanaryStatus RPC

The `GetCanaryStatus` RPC is responsible for assessing the current health status of the monitored server. Its arguments and return value are displayed in Code listing 4.3. The call may be polled periodically to check the server's health status at different points in time by setting the `post_period_end` field to the desired timestamp. If the field is unset, the RPC will return the health status at the current time.

The RPC fetches the latest metrics past the `pre_period_end` time point, runs the configured `series_monitors` and provides a verdict on whether the server is performing as expected.

Setting `verbose` flag populates the `anomaly_reports` field with detailed information about any detected anomalies, including the faulty metrics, the anomaly score, and the target value. It is undesirable for the client application, such as an automatic rollout system, to take action based on the scores contained in anomaly reports, as it could lead to the application of meaningless heuristics. Therefore, it is advised to only take action based on the value of the `canary_status` field, leaving the `anomaly_reports` field purely for debugging and development purposes.

Code listing 4.3 GetCanaryStatus argument and return value definition

```
message GetCanaryStatusRequest {
  string monitor_id = 1;
  optional google.protobuf.Timestamp post_period_end = 2;
  bool verbose = 3;
}

message AnomalyReport {
  string model_name = 1;
  repeated canary.monitor.common.Feature faulty_metrics = 2;
  google.protobuf.Timestamp begin = 3;
  google.protobuf.Timestamp end = 4;
```

```
  repeated float predicted_value = 5;
  repeated float target_value = 6;
  repeated float anomaly_score = 7;
}

message GetCanaryStatusResponse {
  CanaryStatus canary_status = 1;
  repeated AnomalyReport anomaly_reports = 2;
}
```

RetireCanaryMonitor RPC

Calling the `RetireCanaryMonitor` RPC cancels and stops the scheduling of any periodic inference jobs associated with a given monitoring job.

CheckCanary RPC

`CheckCanary` is a convenience RPC that combines the `BeginCanaryMonitor` and `GetCanaryStatus` RPCs, without the ability to register a periodic monitoring job or manually poll the server's health status via `GetCanaryStatus`.

4.2 Metrics acquisition

The metrics acquisition flow is orchestrated before the training and inference workflows during the `BeginCanaryMonitor`, `GetCanaryStatus`, or `CheckCanary` RPCs. The flow consists of the following steps:

1. Fetch all metrics from the defined Prometheus `api/v1/query_range` endpoint.
2. Infill any missing points in the time series using previous/next values or by interpolation.
3. Resample the metrics to the user-defined sampling frequency.
4. Erase any anomalous subsequences based on incidents in third party monitoring systems.
5. Store preprocessed metrics in Ceph storage for later use.

Infilling data points is generally not required. However, it may occur, for example, due to resource constraints on the scraping agent forcing it to drops metrics. Additionally, misalignment between the scraping and querying intervals can also lead to missing data points in the query result.

While these situations should be rare or non-existent in a properly configured VictoriaMetrics setup, we must still account for them. We prefer using

linear interpolation to fill in the missing values, but we also provide the option to use the previous or next available value if needed.

Different metrics are scraped at varying intervals: system metrics are generally collected every n seconds to capture short spikes in data, some application specific business metrics related to server and cluster capacity may be collected at 1-second intervals, and some highly specialized metrics related to a specific subsystem or component, such as feature-flag counters, may be collected at 5-minute intervals to reduce the load on the VictoriaMetrics cluster. We resample all metrics based on a user-defined sampling frequency. This method allows for both up-sampling and down-sampling, depending on the specific use case. Gauges should be resampled by taking the mean value for down-sampling or by interpolating the data points for up-sampling. Meanwhile, counter metrics should be resampled using linear interpolation or extrapolation.

The erasure step is used to remove any subsequences deemed anomalous by incident reports from third-party monitoring systems responsible for handling incidents such as failing hardware, network outages, or power outages. This causes a problem for the ARIMA and Holt-Winters models, which require a continuous time series to function correctly. This issue is mitigated by truncating the historical data to the last non-anomalous point. However, seasonal patterns may prevent the model from fitting correctly or being able to be trained at all. Prophet, forecasting LSTM, and the autoencoder models are unaffected by this issue.

4.3 Series monitors

The anomaly detection system supports multiple algorithms and models for detecting anomalies, each referred to as a `SeriesMonitor`. The configuration for each series monitor is illustrated in Code listing 4.4.

When defining a `SeriesMonitor` by invoking the `BeginCanaryMonitor` or `CheckCanary` RPCs, users must specify the following:

- The time series to monitor.
- The context metrics (or regressors) to consider, if any.
- The anomaly detection algorithm or model to use.

The anomaly detection algorithm is defined using the `monitor` field. Users have the option to provide their own configuration via the `monitor_config` field or use a predefined built-in configuration by specifying a type URL.

Defining a `PeriodicSeriesMonitor` allows the execution of a `SeriesMonitor` at a specified frequency, reporting any detected anomalies to Prometheus Alertmanager [2]. The periodic monitor will consider each anoma-

[2]https://prometheus.io/docs/alerting/latest/alertmanager/

lous subsequence only once in the first period in which it occurs. Alerts will continue to fire for the monitored time series until one of the following occurs:

- The `RetireCanaryMonitor` RPC is called.
- The maximum duration, set by `retire_after`, elapses.
- The `max_alerts` number is reached.

The purpose of limiting the duration of a periodic series monitor is to reduce alert fatigue. Other than the per-monitor limit, a global limit specified in the service configuration is enforced to prevent the system from running indefinitely.

Code listing 4.4 Series monitor definition

```
message SeriesMonitor {
  oneof monitor {
    string monitor_type_url = 1;
    google.protobuf.Any monitor_config = 2;
  }
  optional canary.monitor.common.Feature target_metric = 3;
  repeated canary.monitor.common.Feature context_metrics = 4;
  optional float threshold_multiplier = 5;
  optional PruningConfig pruning_config = 6;
}

message PeriodicSeriesMonitor {
  oneof monitor {
    SeriesMonitor monitor_simple = 1;
    EnsambleMonitor monitor_ensamble = 2;
  }
  google.protobuf.Duration trigger_frequency = 3;
  optional google.protobuf.Duration retire_after = 4;
  optional uint32 max_alerts = 5;
}
```

4.3.1 Training and inference

Each `SeriesMonitor` is implemented as an optional training and an inference flow, where the training flow runs on pre-period data in order to train the model, and the inference flow runs on post-period data to detect any anomalies.

The training flow is responsible for hyperparameter tuning and model training/selection for models that require extensive training time, such as LSTMs or autoencoders. The training flow produces a trained model artifact, reconstruction, or prediction errors for the training and validation data and various statistics and charts for debugging and observability purposes. All this information is persisted in the Ceph object storage. First, the pre-period data is

scaled and reshaped according to the model requirements, mentioned in subsection 2.3.4 and subsection 2.4.5, and then split into training and validation sets. Each supplied configuration is then trained on the training data, and the best model is selected based on either the validation loss, in the case of neural networks, or AIC/BIC scores in the case of statistical models.

The inference flow is responsible for running the trained reconstruction or prediction models on the post-period data, generating anomaly scores and labels for each data point. The anomaly labeling approaches are described in subsection 4.3.7. First, we preprocess the post-period data-standardize or normalize it according to the training data, and reshape it to the model's input shape. Then, we run the model on the data and calculate the reconstruction or prediction errors and anomaly labels.

4.3.2 Exponential smoothing

The first statistical model we support for time series forecasting is the Holt-Winters exponential smoothing. We rely on the `statsmodels` library for the implementation. Due to no support of exogenous regressors in the statsmodels' exponential smoothing and all state space models, it is only suitable for less complex time series and anomaly detection tasks. The `context_metrics` field is ignored. The best configuration supplied to `BeginCanaryMonitor` is selected based on the lowest AIC score.

The hyperparameters for the Holt-Winters model are listed in Table 4.2, trend and seasonal components correspond to the taxonomy mentioned in Table 2.3.

Parameter	Type	Default	Description
trend	str	additive	Type of trend component, None, "additive" or "multiplicative"
seasonal	str	additive	Type of seasonal component, None, "additive" or "multiplicative"
seasonal_periods	uint	None	Number of periods in a season, depends on input frequency

Table 4.2 Holt-Winters model parameters

4.3.3 ARIMA

The second statistical model we support for time series forecasting is the ARIMA model from the `statsmodels` library. The training flow for ARIMA, SARIMA, and SARIMAX models searches over a range of hyperparameters to

find the best configuration based on the lowest AIC score, if none or multiple configurations are supplied. The hyperparameters for the ARIMA model are listed in Table 4.3. The `trend` parameter is unsuitable for fitting and should be determined based on the data. The `seasonal_periods` parameter should be set to a value corresponding to a weekly or daily seasonality for most time series we consider.

Parameter	Type	Default	Description
order	tuple3[uint]	None	ARIMA model order
trend	str	c	Type of trend component, "c" for constant, "l" for linear
seasonal_order	tuple3[uint]	None	ARIMA order of the seasonal component
seasonal_periods	list[uint]	None	Number of periods in a season, depends on input frequency
fourier_terms	uint	20	Number of Fourier terms to generate for each seasonality

Table 4.3 ARIMA model parameters

The hyperparameters `order` and `seasonal_order` correspond to the order of the ARIMA model and the order of the seasonal component, respectively. If none are specified, the training flow will search for the best configuration in the range of $[0, 3]$ for each parameter, the seasonal components are fit only if the `seasonal_periods` parameter is set to a non-null value.

When dealing with long seasonal periods, seasonal Fourier terms should be preferred to reduce the training time and memory footprint [12]. Setting the `fourier_periods` parameter to a non-null value will provide the model with `fourier_terms` number of Fourier terms for each seasonal component. It is important to note that this configuration option cannot be used simultaneously with the `seasonal_order` and `seasonal_periods` parameters. Another option is to use an ARIMAX model with a colinear, or highly correlated, exogenous regressor, which directly affects the target metric, such as the number of requests per second and CPU utilization.

4.3.4 Prophet

Prophet, available from the `fbprophet` package, is a highly configurable model with a support for multiple seasonalities, changepoint detection, and exogenous regressors. The hyperparameters for our Metaflow wrapper around the Prophet model are listed in Table 4.4. Not all available hyperparameters are configurable for a monitor, since majority of them are not applicable to our use

case, such as yearly seasonality. The only hyperparameters that should be optimized are the `seasonality_prior_scale` and `changepoint_prior_scale` parameters [36], other hyperparameters should be set to reasonable values based on the monitored time series.

Parameter	Type	Default	Description
growth	str	flat	Growth model, ”linear” or ”logistic”
weekly_seasonality	any	auto	Enable weekly seasonality
daily_seasonality	any	auto	Enable weekly seasonality
seasonality_mode	str	additive	Seasonality mode, additive or multiplicative
seasonality_prior_scale	float	10.0	Seasonality prior scale
changepoint_prior_scale	float	0.05	Changepoint prior scale
mcmc_samples	uint	0	Number of MCMC samples
uncertainty_samples	uint	1000	Number of uncertainty samples

Table 4.4 Prophet model parameters [37]

4.3.5 LSTM

We support a simple stacked LSTM model for time series forecasting. The architecture consists of several LSTM layers which pass their hidden state to the next, except for the last layer, which feeds into a dense layer to produce the output. The reference source code for the LSTM model in Keras is available in Appendix A.

The LSTM model is used to predict short-term future values of the univariate `target_metric` time series, generally up to a few data points, or minutes, ahead. The model is trained on overlapping subsequences of the multivariate `context_metrics` time series with stride of one, as described in subsection 2.3.4. During inference, we reshape the post-period data in the same manner and aggregate the overlapping predicted windows to obtain the final forecast as shown in Figure 4.1, where the blue window represents the input data and the yellow windows represent the forecasted data at various time points.

The fittable hyperparameters for the LSTM model are listed in Table 4.5. The `lstm_activation` and `output_activation` parameters are unsuitable for fitting and should be determined beforehand based on the data or left at the default values.

Parameter	Type	Default	Description
lstm_num_units	list[uint]	None	List of number of LSTM cells for each layer
lstm_activation	str	tanh	LSTM layers activation function
dropout	float	0.05	LSTM layer dropout
output_activation	str	linear	Output dense layer activation function
prediction_length	uint	2	The prediction length in number of data points

Table 4.5 LSTM model parameters

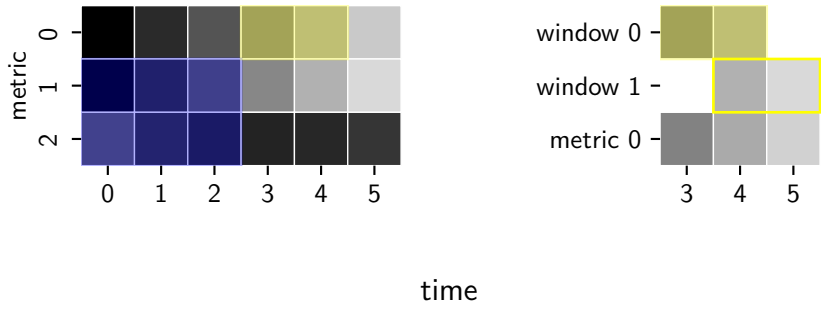

Figure 4.1 LSTM inference process

4.3.6 Autoencoders

Four different types of autoencoders are supported: an LSTM-AE, a Conv-AE, and their respective variational counterparts. All autoencoders are symmetric, with the encoder and decoder having the same architecture. The reference source code for the autoencoders in Keras is available in Appendix A.

The LSTM encoder is implemented with several configurable stacked LSTM layers, which pass their hidden state to the next layer. The final LSTM layer outputs the encoded latent representation of the input sequence, which is then time-distributed to the decoder LSTM layers in order to recreate the shape of the input sequence. The final output sequence is then passed through a dense layer to match the input shape. The compression, or bottleneck, is achieved by reducing the number of LSTM units in the encoder layers.

The variational LSTM autoencoder is implemented similarly to the LSTM autoencoder, but instead of using the final output of the encoder as the latent representation, the hidden state of the final LSTM layer is passed through two dense layers to produce the mean and log-variance vectors. The latent representation is then sampled from the mean and log-variance vectors. The decoder part of the variational autoencoder is the same as the LSTM autoencoder.

The fittable hyperparameters for the LSTM and LSTM-VAE models are listed in Table 4.6. The `lstm_activation` and `output_activation` parameters are unsuitable for fitting and should be determined beforehand based on the data or left at the default values.

Parameter	Type	Default	Description
lstm_num_units	list[uint]	None	List of number of LSTM cells for each layer
lstm_activation	str	tanh	LSTM layers activation function
dropout	float	0.05	LSTM layer dropout
output_activation	str	linear	Output dense layer activation function
latent_dim	uint	None	LSTM-VAE mean and log-variance layer dimensionality

Table 4.6 LSTM-AE and LSTM-VAE model parameters

The Conv-AE is implemented using several stacked one-dimensional convolutional layers that compress the input sequence by applying filters with a stride greater than one. The final convolutional layer generates the encoded latent representation of the input sequence, which is then upsampled by the decoder's transposed convolutional layers to recreate the input sequence's original shape.

In the Conv-VAE, the latent representation is sampled from the mean and log-variance vectors. Instead of using the output directly, these vectors are produced by processing the output of the final convolutional layer through two dense layers.

The fittable hyperparameters for the Conv-AE and Conv-VAE models are listed in Table 4.7. Parameters not suitable for fitting are `conv_activation` and `output_activation` and should be determined beforehand, or set to the default values.

We train the autoencoders on overlapping subseries of the multivariate `context_metrics` time series with a stride of one. The approach is further detailed in subsection 2.4.5. During inference, we reshape the post-period data in the same manner. In order to reconstruct the input time series, we aggregate the corresponding time points within the overlapping windows similarly to the process illustrated in Figure 4.1.

In contrast to the univariate models, the autoencoder anomaly score is obtained by taking the L_2 norm of the reconstruction errors, as opposed to the reconstruction error itself.

Parameter	Type	Default	Description
conv_filters	list[uint]	None	List of number of filters for each convolutional layer
conv_activation	str	tanh	Activation function of the CONV layers, "tanh" by default
stride	int	2	Stride of the convolution
padding	str	same	Convolution padding
kernel_size	list[uint]	5	Size of the 1D convolution filters
output_activation	str	linear	Output dense layer activation function
dropout	float	0.05	CONV layer dropout
latent_dim	uint	None	CONV-VAE mean and log-variance layer dimensionality

Table 4.7 Conv-AE and Conv-VAE model parameters

4.3.7 Anomaly detection algorithms

Different approaches may be used to detect anomalies, such as training a supervised or unsupervised classifier on errors or the series itself, or thresholding the metrics or errors directly. Both anomaly detection approaches we use are unsupervised and based on thresholding of prediction or reconstruction errors. Training a supervised classifier on the errors would require labeled data per anomaly type and server host, which is poorly scalable and infeasible in a large-scale production environment.

The first approach we use is based on thresholding of forecasting or reconstruction errors. We calculate the errors for the post-period data and compare them to the threshold of $\mu + z\sigma$, where μ is the mean, σ is the standard deviation of the pre-period errors.

The other approach we use to label data points as anomalies was proposed by NASA in their work on the detection of anomalies in spacecraft telemetry data [38]. The method consists of computing such threshold that if all values above were removed, it would result in the most significant percent decrease in the mean and standard deviation of the errors, while penalizing the number of anomalous points and subseries. Given a series of errors $e = \{e_1, e_2, \ldots, e_n\}$, the algorithm finds a threshold ε such that the following is maximized:

$$\frac{(\Delta\mu - \mu) + (\Delta\sigma - \sigma)}{|e_{\text{anom}}| + |e_{\text{seq}}|^2}, \tag{4.1}$$

where μ and σ are the mean and standard deviation of the post-period errors. $\Delta\mu$ and $\Delta\sigma$ are the changes in the mean and standard deviation after removing the anomalous points $|e_{\text{anom}}|$ refers to the number of anomalous points, and

$|e_{\text{seq}}|$ refers to the number of consecutive anomalous subseries. Points are classified as anomalies based on the condition $e_i > \varepsilon$, where $\varepsilon = \mu_{\text{period}} + z\sigma_{\text{period}}$, z is the value that maximizes this expression, and μ_{period} and σ_{period} are the mean and standard deviation of either the pre or post-period errors.

The anomaly detection approach is chosen based on the settings of the optional `threshold_multiplier` field of the `SeriesMonitor` shown in Code listing 4.4. Setting it to a non-null value will result in thresholding based on training errors, while setting it to `null` will use the dynamic thresholding approach.

The `pruning_config` field in the `SeriesMonitor` message configures the anomaly pruning algorithm described in [38].

4.3.8 Chow test

Other than as a test for a structural change, mentioned in subsection 2.2.5, the Chow test can detect changes in the relationship between causal and response variables. For instance, the test can be applied to determine whether the relationship between a server's CPU utilization and its throughput has shifted. By regressing CPU utilization against throughput using a linear model for pre-period, post-period, and combined data, we can then compare the coefficients with Chow's test. If the test statistic is significant, we can conclude that the relationship between the time series has changed. This is illustrated in Figure 4.2, where in the right subplot, the relationship between CPU utilization and throughput has significantly changed after the pre-period.

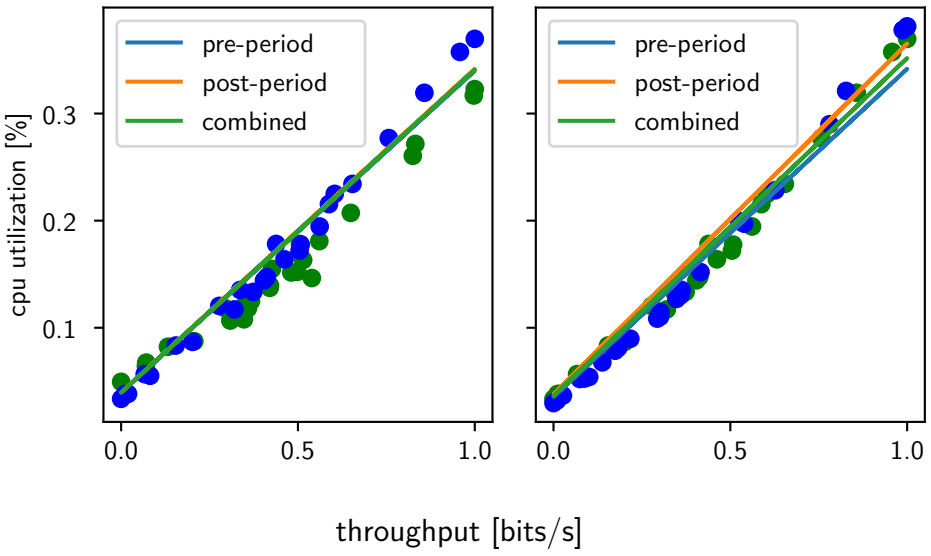

Figure 4.2 Server's CPU utilization per throughput

4.3.9 Monitor ensemble

When defining an ensemble series monitor using the configuration in Code listing 4.5, the service will combine the anomaly labels from all individual series monitors through a majority vote to determine the final verdict on the server's health status, rather than relying solely on the verdicts from the individual monitors.

Code listing 4.5 An ensemble series monitor definition

```
enum EnsembleMethod {
  ENSEMBLE_METHOD_VOTING = 0;
}

message EnsembleSeriesMonitor {
  repeated SeriesMonitor series_monitors = 1;
  EnsembleMethod method = 2;
}
```

The ensemble method works with series of anomaly flags, where each flag is a binary value indicating the presence of an anomaly. The majority vote is then performed on anomalous subseries from the defined series monitors.

Given m time series a_i, where each a_i is a binary time series indicating anomalies detected by its corresponding monitor, the next step is to extract all consecutive anomalous subseries for each monitor i.

An index set intersection is then performed across all these sets of anomalous subseries. Specifically, for each possible subsequence, we compute the intersection of the index sets of the anomalous subseries across all monitors. If the majority of the monitors agree on a subsequence (i.e., the intersection contains anomalous subsequences from a majority of the monitors), the entire index set intersection is marked as anomalous.

4.4 Example configuration

Code listing 4.6 shows an example configuration for the `BeginCanaryMonitor` RPC written in the protobuf text format. The request defines two series monitors with built-in configurations, one for the CPU user time and the other for the cache hit rate. All time series are fetched from the Prometheus server at the specified URL, with a step of one second and linear infilling. The pre-period is defined by the `pre_period_begin` and `pre_period_end` fields. The `pre_period_end` field defines the beginning of the post-period, which is terminated by the `post_period_end` field in the `GetCanaryStatusRequest` message.

The first monitor uses an LSTM forecasting model in default configuration to monitor the CPU user time metric, regressed on the number of bytes sent over the HTTP2 and HTTP3 protocols per second. All metrics are standard-scaled by the mean and standard deviation of the pre-period data. Since no

threshold multiplier is specified, the monitor will use the dynamic thresholding approach. The pruning configuration specifies a window size of 10 minutes and a threshold of 0.05, discarding any anomalous subsequences with an anomaly score within 5% of the next closest anomaly score in the window. The monitor is triggered every 10 minutes, sending detected anomalies to the Prometheus Alertmanager.

The second monitor uses Chow's test for the cache hit rate metric. Since no context metrics are specified, Chow's test will be used to test whether any event that happened on `pre_period_end` caused a structural break in the time series.

Code listing 4.6 Example gRPC request

```
{ server_info { suid: ... }
  prometheus_info { url: ... }
  pre_period_begin: ...
  pre_period_end: ...
  metrics: [
    { query_built_in: cpu_user
      step { seconds: 1 }
      infill_method: INFILL_LINEAR
    }, {
      query_built_in: hit_rate
      step { seconds: 1 }
      infill_method: INFILL_LINEAR
    }, {
      query_external: {
        series_name_template: "tx_{protocol}_bps"
        query: "rate(tx_bytes_total{protocol=~'h2|h3'})"
      }
      step { seconds: 1 }
      infill_method: INFILL_LINEAR
    }
  ]
  series_monitors: [
    { monitor_periodic: {
        monitor_simple: {
          monitor_type_url: "url/canary.monitor.LSTM"
          target_metric: {
            name: "cpu_user",
            scaling: SCALING_STANDARD
          }
          context_metrics: [
            { name: "tx_h2_bps", scaling: SCALING_STANDARD },
            { name: "tx_h3_bps", scaling: SCALING_STANDARD }
          ]
          pruning_config {
            threshold: 0.05
            window_size { seconds: 600 }
```

```
          }
        }
        trigger_frequency { seconds: 600 }
      }
    }, { monitor_simple: {
        monitor_type_url: "url/canary.monitor.Chow"
        target_metric: { query_built_in: hit_rate }
      }
    }
  ]
}
```

Executing the `GetCanaryStatusRequest` RPC with the returned `monitor_id` will execute the defined series monitors and return the server's health status at the specified `post_period_end` timestamp.

Chapter 5

Monitoring scenarios and datasets

Due to the lack of publicly available high-quality performance anomaly detection datasets, briefly mentioned in section 3.2, we have chosen to evaluate our approaches mainly using purposefully created datasets.

We selected and reverted several system and program configuration improvements and bug fixes to achieve this. These changes were intentionally chosen to create relatively difficult-to-detect anomalies with micro-benchmarks or threshold-based monitoring. Such issues are common in high-performance, highly concurrent systems, particularly in caching proxies. The datasets are detailed below, and their characteristics are summarized in Table 5.1.

- `clipping`: Throughput clipping anomaly due to a bottleneck - high mutex saturation.
- `regression`: Higher long-term CPU utilization on a workload with the same characteristics than previously.
- `spikes`: Short bursts of high CPU utilization due to misconfigured utilities.
- `trend`: Unbound memory utilization growth due to a space leak.
- `chaos`: Increased response latency.

The datasets `clipping`, `regression`, `spikes`, and `trend` correspond to the USE methodology, while the `chaos` dataset relates to the RED methodology. All anomalies in the dataset are contextual or collective as per the definition in section 2.4. The collection methodology is described in section 5.1. The collected metrics and their labels and units are described in Table 5.2. Examples of the anomalies can be found in Figures 5.1, 5.2, 5.3, 5.4, and 5.5.

The anomaly exhibited in the `clipping` dataset is a bottleneck in the system and manifests itself with an increased number of context switches, decreased throughput, as well as increased system CPU utilization and decreased userland CPU utilization.

The `regression` dataset represents a performance regression in the system, the userland CPU utilization is increased for a prolonged period of time, and unexplained by the workload characteristics.

The `spikes` dataset represents a unexplained short burst of high CPU utilization caused by misconfigured utilities which lack a nice level, but similar issues could be caused by other factors.

The `trend` dataset represents a space leak or an increased memory utilization over time which is unexpected and unexplained by the workload characteristics.

The `chaos` dataset represents a sidecar proxy service latency increase caused by chaos testing (shutting down services, introducing latency). The increased latency is accompanied by an increased number of failed requests.

Dataset	**Testing set size**	**Anomaly %**
clipping	14400	10.03
regression	14400	30.04
spikes	14400	5.72
trend	14400	100
chaos	14400	11.93

Table 5.1 Dataset characteristics

5.1 Data collection methodology

We used a virtual server designated for debugging and quality assurance to create these datasets. The server was configured with the latest kernel and network stack settings, reflecting the configuration used in most of our networks. We then ran the latest production version of our caching proxy and supporting utilities and mirrored a percentage of requests from our production environment to this server, while collecting all relevant system and application-level metrics at 30-second intervals for three weeks. We believe this dataset accurately represents a performance anomaly-free behavior of the system. Other anomalies may still be present in both the training and testing data, such as a higher-than-expected load due to client behavior, or anomalies caused by network congestion or other external factors. As a result, this may lead to a higher than expected number of false positives.

In order to generate the anomalous datasets, we deployed the modified binaries and configurations to additional virtual servers and mirrored the same

Metric name	Labels	Source	Unit	Description
cpu	user|system	Node	%	CPU utilization
context_switches		Node	counts/s	Number of context switches per second
memory		Node	%	Memory utilization
tx	h2|h3	proxy	bit/s	Number of sent bits per second
rx	h2|h3	proxy	bit/s	Number of received bits per second
cx	h2|h3	proxy	count	Current number of active connections
drps	h2|h3	proxy	count/s	Number of new downstream requests per second
urps	h2|h3	proxy	count/s	Number of sent upstream requests per second
u_status	2xx|4xx|5xx	proxy	count/s	Number of received status codes per second
u_latency_50		proxy	%	Percentage of requests with first byte latency under 50ms

Table 5.2 Dataset metrics

production traffic to them. Yielding four different datasets, each exhibiting a different type of a performance anomaly in the caching proxy. One more dataset was created by running a proxy server in a sidecar configuration while we ran chaos tests on the application server.

Each of the anomalous datasets was recorded at five consecutive days in order to account for daily and weekly variations in the workload. In total, the common training dataset contains 72,000 samples (25 days) and each of the testing datasets contains 14,400 samples (5 days) worth of data.

It is important to note that all virtual servers were run on the same hypervisor. This approach does not precisely reflect a real-world scenario where the caching proxies will likely run on bare-metal servers or at least on separate hypervisors. In the case of the sidecar proxy, this is not as relevant. However, this approach is sufficient for this evaluation, as hypervisor overhead is negligible in this case and unlikely to affect the result. At the same time, the effects of noisy neighbors should be minimal due to the relatively low load.

The dataset `spikes` was labeled automatically by scraping system logs for timestamps of when a misbehaving utility was scheduled to run and cross-referencing them with the collected metrics. The datasets `clipping` and

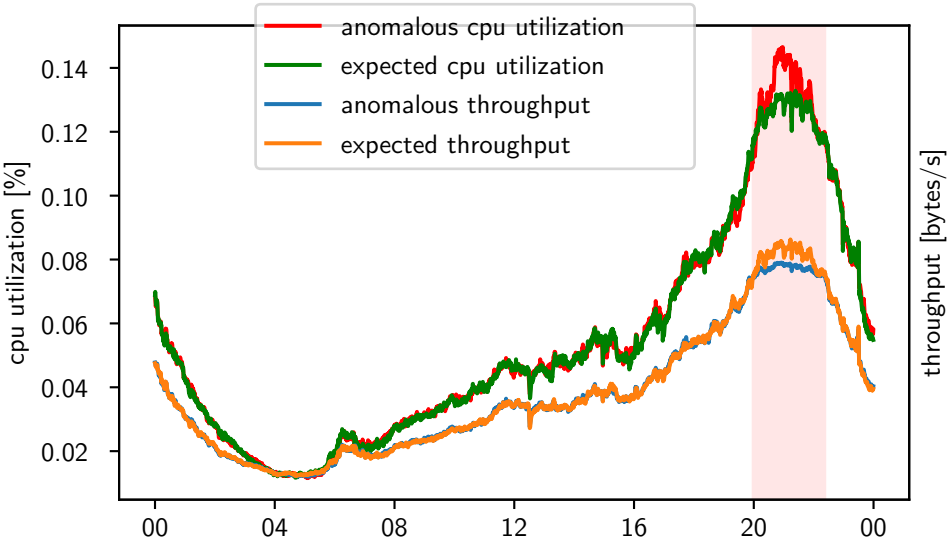

Figure 5.1 Throughput clipping

`regression` were labeled manually by inspecting the collected metrics, which means the labels may not be entirely accurate. Dataset `trend` was labeled entirely as anomalous due to prior knowledge of the space leak. Lastly, the `chaos` dataset was also labeled manually, using prior knowledge of when performance degradation was scheduled to occur.

5.1.1 Third-party datasets

In the study by [33], the authors present several artificial datasets generated by running an industry-standard API gateway under normal and anomalous conditions. The anomalies were caused, for example, by executing dead code that consumed CPU or memory resources. The final dataset used for evaluation was created by merging segments of the normal and anomalous data. However, the authors did not provide any anomaly labels for this interleaved dataset, making it infeasible for comparative analysis. Consequently, the interleaving process resulted in a dataset that does not accurately represent real-world scenarios, due to the lack of temporal continuity and unrealistic transitions between normal and anomalous states.

In [38], the authors focused on detecting anomalies in spacecraft telemetry using univariate LSTM prediction. The dataset comprises real telemetry data and anomalies from two sources, the Soil Moisture Active Passive (SMAP) satellite and the Curiosity Rover on Mars (MSL). The response variable is the telemetry value, while the regressor variables represent the commands sent to and received from various spacecraft subsystems. Because of the characteristics of the response variable and the anomalies present in many telemetry channels,

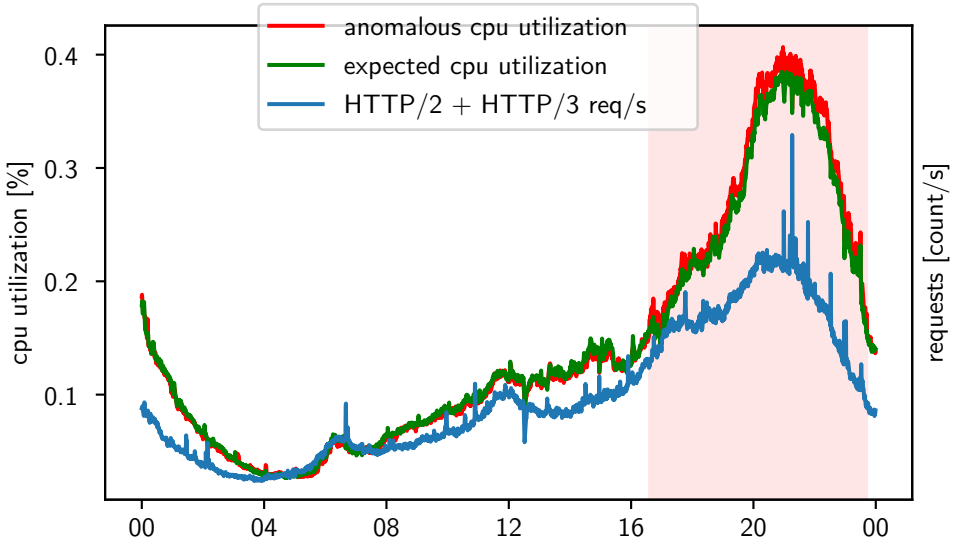

Figure 5.2 Performance regression

such as mean shifts that are only considered anomalous when contextualized with regressor variables, models that are unable to effectively regress or reconstruct the target variable from sparse binary series are likely to perform poorly. We tried.

In [32], the authors present an anonymized Server Metrics Dataset (SMD) from an "internet company". Their study concerns server metrics interpretation and anomaly detection without domain-specific knowledge. This is the only dataset applicable to our work, even if only for multivariate reconstruction-based anomaly detection. The dataset consists of 28 multivariate time series, each with 38 metrics. The training and test sets each contain 708,405 and 708,200 samples, respectively, and the overall percentage of anomalies is 4.16%.

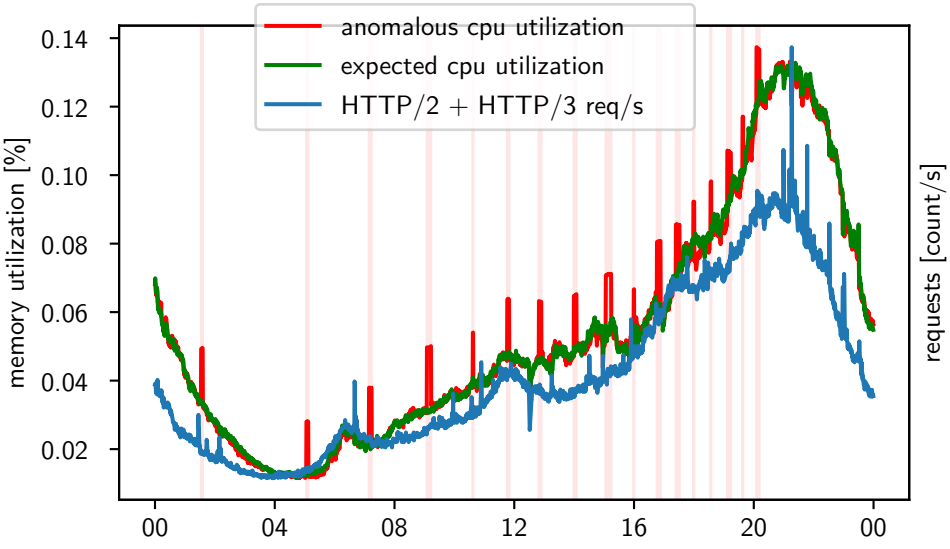

Figure 5.3 CPU utilization spikes

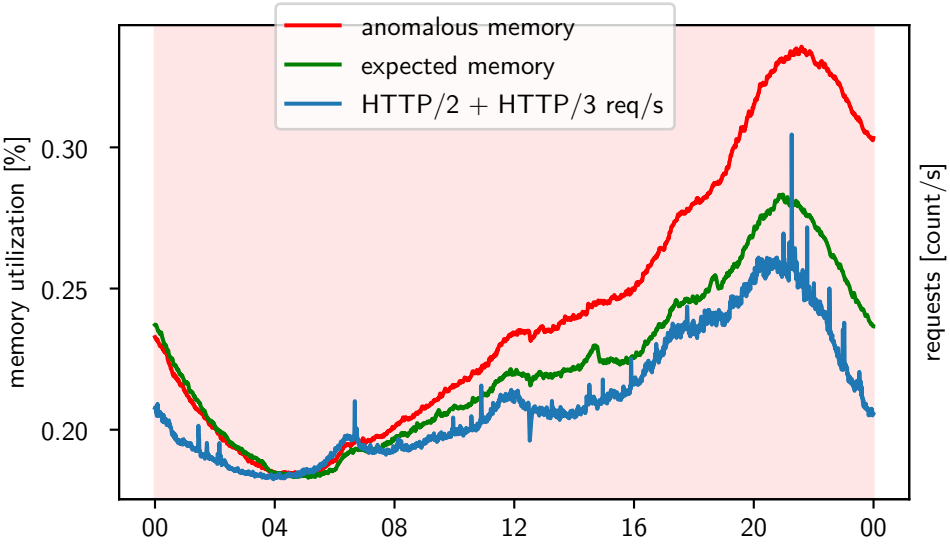

Figure 5.4 Space leak

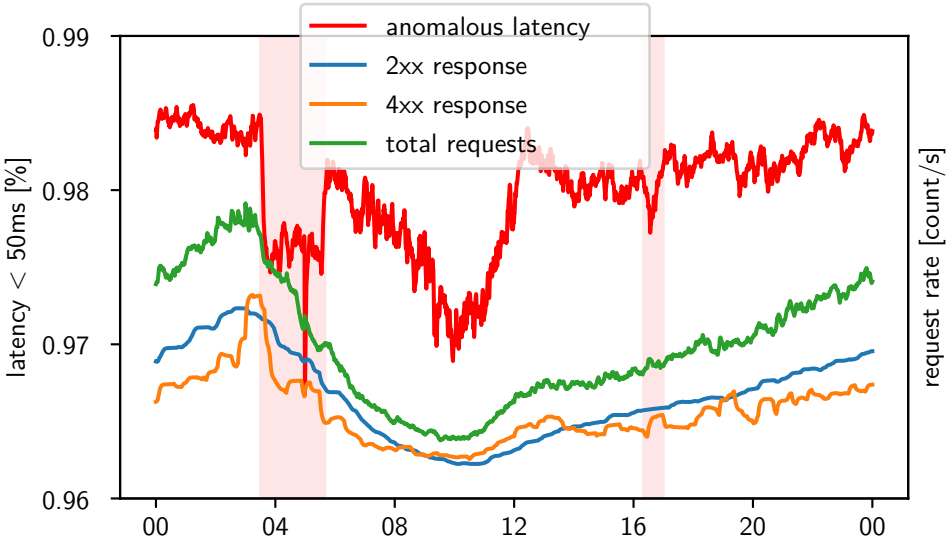

Figure 5.5 Sidecar service latency increase

Chapter 6

Experimental evaluation

We evaluate the performance of the predictive and reconstructive models by measuring the average ROC-AUC (auc) across all datasets, precision (p), recall (r), and $F_{0.5}$ score regarding both point-wise and sequence-wise anomaly detection approaches, how early the models can detect anomalies (tta), and their training and inference duration. An $F_{0.5}$ score is used instead of an F_1 score because our main priorities are minimizing false positives and alert fatigue. The time to alert (tta) is defined as the time difference between a predicted anomaly corresponding to a true anomaly. A negative value indicates that the model predicted (or "hallucinated") the anomaly before it occurred, and a positive value indicates that the model detected the anomaly with a delay.

In point-wise anomaly detection, a true positive is recorded when the model correctly predicts an anomaly at a given time. In sequence-wise anomaly detection, the metrics are calculated according to the following rules:

- A true positive is recorded when an anomalous sequence is overlapped by one or more predicted anomalous sequences.
- All predicted anomalous sequences that do not overlap with any anomalous sequences are counted as false positives.

In [32], the authors measure the performance of their models using a point-adjusted recall, taken from [39]. The adjustment is made such that if a chosen threshold can detect any point in an anomaly segment in the ground truth, the entire segment is detected correctly, and all points in this segment are treated as if this threshold can detect them. Meanwhile, the points outside the anomaly segments are treated as usual.

All model evaluations on our datasets were done with anomaly pruning and automatic thresholding set up to maximize the $F_{0.5}$ score. The parameters are bound to $z \in [3, 10]$, $p \in 0.01, 0.05$, $w \in 2\text{h}, 8\text{h}$, where z is the number of standard deviations from the mean, p is the anomaly pruning percentage threshold, and w is the window size.

6.1 Statistical methods evaluation setup

The models were fitted to the entire training set (three weeks) and then forecasted for the entire testing set (one day). The datasets were resampled to 5-minute intervals for the Holt-Winters model due to the high computational cost of fitting it with a weekly seasonality and 30-second frequency.

The Prophet and exponential smoothing model hyperparameters were either manually configured or left at their default values. The ARIMA models were fitted with a daily and weekly seasonal components represented by Fourier harmonics. The order of the ARIMA model was selected by conducting a time-series cross-validation on a range of potential hyperparameter values and selecting the best-performing set based on the AIC score. This process was repeated for each dataset, however ended up being the same across all datasets. All chosen hyperpameter values are listed in Table 6.1 and the response and causal variables are listed in Table 6.2.

Chow test was evaluated by concatenating the training and testing sets for the `regression` and `trend` datasets, resampling them to 1-hour intervals, and then running the test on every 48-hour interval with 24 hours for the pre and post periods, regressing the response variable of the dataset on the causal variable `tx`.

Model	Parameter	Value

Prophet	`seasonality_mode`
	`weekly_seasonality`
	`daily_seasonality`
		additive
		True
		True

Holt-Winters	`seasonal` `trend`	additive additive

ARIMA	`order`
	`fourier_seasonalities`
	`fourier_terms`
		(3, 0, 2)
		[one_week, one_day]
		5

Table 6.1 Statistical model hyperparameter values

Dataset	Response	Causal
`clipping`	`cpu{type=system}`	`tx`, `cx`, `drps`
`regression`	`cpu{type=user}`	`tx`, `cx`, `drps`
`spikes`	`cpu{type=user}`	`tx`, `cx`, `drps`
`trend`	`memory`	`tx`, `cx`, `drps`
`chaos`	`u_latency_50`	`rx`, `urps`, `u_status`

Table 6.2 Statistical model forecasting setup

6.2 Neural network evaluation setup

The architecture and hyperparameter values of the neural networks were initially selected by a Hyperband search and then manually manually adjusted to be consistent across all datasets. The autoencoders were set up such that they have comparable compression rates from the input to the latent space. These parameters are listed in Table 6.3. The response and causal variables, as well as window sizes and forecasting horizons, for each model and dataset are listed in Tables 6.4, 6.2, and 6.5. If no label is specified, the metric name refers to the, possibly aggregated, corresponding set of metrics listed in Table 5.2. All models were trained for up to 25 epochs with a batch size of 512 and an early stopping patience set to 5 epochs.

The autoencoders were set up such that the space savings from the input to the latent space, calculated as $1 - \frac{s_{\text{latent}} * f_{\text{latent}}}{s_{\text{input}} * f_{\text{input}}}$, where s is the sequence length and f is the number of features, are the same across all models. The space savings are 60% for dataset `clipping` and 50% for the other datasets.

The SMD dataset was used to evaluate the performance of the multivariate reconstructive models, as the univariate predictive models are not applicable. The models were trained for 50 epochs with a batch size of 512 and an early stopping patience set to 5 epochs. The models were configured similarly to those used in our datasets, aiming for a space-saving of 50% to 60%. The measured metrics are the average ROC-AUC across all datasets, precision, recall, and $F_{0.5}$ score for both point-wise and sequence-wise anomaly detection approaches.

6.3 Performance metrics

Table 6.6 shows the mean and standard deviation of the training and inference durations for each model over all datasets, based on which we have set up remote execution for Metaflow steps. All models except for Prophet are offloaded to either a GPU or a CPU instance with a higher core count in Kubernetes for training. All of the statistical models are ran on host machine during inference.

Unsurprisingly, the LSTM-based models have the longest training and inference durations. This is due to the recurrent nature of the LSTM cells, which are inherently slower to train and infer than feed-forward models.

Holt-Winters is currently infeasible to run on time series with long seasonal periods and high frequency, both due to the high computational cost of fitting the model and the lack of support for exogenous regressors.

Tables 6.7 and 6.8 show the performance metrics for each model and dataset, where the best-performing model is highlighted in bold. The results are further discussed per dataset in the following sections.

Model	Parameter	Value
Forecasting LSTM	`num_units`	[20, 10]
	`inner_activation`	tanh
	`dropout`	0.1
	`output_activation`	linear
LSTM-AE	`num_units`	[16, 8, 2]
	`inner_activation`	tanh
	`dropout`	0.1
	`output_activation`	linear
LSTM-VAE	`num_units`	[16, 8]
	`inner_activation`	tanh
	`dropout`	0.1
	`output_activation`	linear
	`latent_dim`	2
CONV-AE/VAE	`filters`	[16, 8]
	`kernel_size`	7
	`stride`	2
	`inner_activation`	tanh
	`dropout`	0.1
	`output_activation`	linear
	`latent_dim`	8

Table 6.3 Neural network hyperparameter values

clipping

The automatic thresholding and anomaly pruning parameters were set to $z \in [3, 10]$, $p = 0.05$ and $w = 8h$. A window of 8 hours was chosen in order to split the testing set into three parts, with the latter two containing the mid-day and prime-time hours. The traffic patterns are generally more stable during the prime-time hours due to the increased number of users, and more volatile during the mid-day.

All models have a satisfactory AUC-ROC score, indicating that all models can effectively separate normal and anomalous classes. However, this does not ensure that they have a sufficiently high level of precision. An excessive number of non-consecutive false positives can result in a high false alarm rate.

Considering the forecasting models, ARIMA, Prophet, and LSTM outperform the Holt-Winters model regarding all metrics. Due to the lack of exogenous regressors, the Holt-Winters model cannot detect anomalies with high precision. All LSTM-based models predict false positive anomalous sequences, which aren't filtered out by the anomaly pruning step. This is likely due to the general noisiness of LSTM models, mentioned in [38].

An interesting thing to note is that ARIMA and Prophet, as well as the convolutional autoencoders, can detect the anomalous sequences with a win-

Dataset	Response	Causal	Window	Horizon
clipping	cpu{type=system}	tx, cx, drps	120	5
regression	cpu{type=user}	tx, cx, drps	120	5
spikes	cpu{type=user}	tx, cx, drps	120	5
trend	memory	tx, cx, drps	120	5
chaos	u_latency_50	rx, urps, u_status	120	5

Table 6.4 Forecasting LSTM setup

Dataset	Metrics	Window
clipping	cpu{type=system}, cpu{type=user}, tx, cx, drps	120
regression	cpu{type=user}, tx, cx, drps	120
spikes	cpu{type=user}, tx, cx, drps	120
trend	memory, tx, cx, drps	120
chaos	u_latency_50, rx, urps, u_status	120

Table 6.5 Autoencoder setup

dow precision of 1.00, indicating that the anomalies are correctly detected, albeit too soon or are labeled as anomalous for too long. This is expected, as the anomalies in the clipping dataset were labeled manually.

Given that the sequence-wise metrics are nearly identical between the convolutional autoencoders, ARIMA, and Prophet, and the multivariate nature of the anomaly, as well as the early detection of the anomaly by the convolutional autoencoders, we hypothesize that the convolutional autoencoders are the best-performing models for this dataset.

regression

The performance regression dataset contains an anomaly in the CPU utilization metric manifesting as an increase in the userspace utilization of around 5% during an increased load, which, due to the seasonality of the data, may happen during mid-day or prime-time hours, therefore the window size w was set to 8 hours. The other parameters were set to $z \in [3, 10]$ and $p = 0.05$.

The best performing model in terms of both point-wise and sequence-wise precision is ARIMA, with the other models reporting a high number of false

Model	Training time [s]	Inference time [s]
Holt-Winters	113.9±5.6	**0.0±0.0**
ARIMA	70.7±5.7	**0.0±0.0**
Prophet	**1.7±0.0**	4.8±0.0
LSTM	128.7±9.6	2.8±0.2
LSTM-AE	256.3±6.5	4.9±0.4
LSTM-VAE	226.4±5.0	5.1±0.4
Conv-AE	30.3±4.1	0.6±0.0
Conv-VAE	21.6±6.2	0.7±0.0

Table 6.6 Training and inference duration per model

positives in the sequence-wise metrics.

The autoencoders in general have an extremely low sequence-wise precision. We hypothesize that this is due to the large training set size, exposing the model to a broad variety of normal patterns, causing it to become insensitive to small deviations.

The sequence-wise recall is not very significant as anomalous sequences happen generally only once or twice per day during the mid or prime-time hours and last for at least a few hours, allowing for a high recall across all models.

We also evaluated the Chow test on the regression dataset. The experiment was set up to regress the response variable on the aggregated `tx`. The dataset was resampled to 1-hour intervals. The test was then run every two consecutive 24-hour intervals for the pre and post-periods, with a significance level of 0.05. The test detects the change in relationship in all of the test datasets with only one false positive. While this is a rather small sample size, it indicates that the relationship between the response and causal variables is stable under normal conditions and the test may be useful as a preliminary check.

spikes

The automatic thresholding and anomaly pruning parameters were set to $z \in [3, 10]$, $p = 0.05$ and $w = 2h$. A window of 8 hours was chosen in order to split the testing set into three parts for similar reasons as the `clipping` and `regression` datasets.

ARIMA and Prophet models demonstrate high precision and recall for detecting anomalies on a sequence-wise basis. The Forecasting LSTM performs similarly to the Holt-Winters model regarding point-wise precision, with both models producing a significant number of false positives. For the LSTM, this is likely caused by the forecast noise being attributed to short-term utilization spikes. However, when considering sequence-wise metrics, the LSTM model shows much better performance.

The multivariate autoencoders perform relatively poorly, with an average point-wise and sequence-wise precision of 0.14 and 0.7, respectively. Indicating they are not well suitable for univariate anomaly detection via a multivariate reconstruction, as is also apparent from the `regression` dataset results. This may be due to the distance metric used in the reconstruction loss (L_2), however, we've attempted anomaly detection via Isolation Forests and Local Outlier Factor on the residuals with no significant improvement in performance.

trend

As the entire post-period is labeled as anomalous with prior knowledge, we are primarily focused on measuring the model's capability to detect the anomaly as early as possible.

Holt-Winters model was able to predict the anomaly with a tta of 25 minutes and a recall of 0.96. This is likely due to the model's ability to capture the seasonality of the response variable, as the model has no exogenous regressors with explanatory power and would not be applicable if the memory utilization was more volatile.

The autoencoders struggle to detect the anomaly early, indicating insensitivity to gradual changes in the response variable. Interestingly, the same issue occurs with the forecasting LSTM.

This leaves the ARIMA model as the best performing model, with a tta of 144 minutes and a recall of 0.86.

chaos

The `chaos` dataset contains anomalies with similar characteristics to the `spikes` and `regression` datasets, exhibiting sudden spikes and mean shifts in the response variable. The automatic thresholding and anomaly pruning parameters were set to $z \in [3, 10]$, $p = 0.05$ and $w = 2h$.

The best performing model in terms of both point-wise and sequence-wise precision is the forecasting LSTM, with the other models reporting a high number of false positives in the sequence-wise metrics. Compared to its performance on the `spikes` dataset, it is apparent the model can detect longer-lasting anomalies more effectively.

The autoencoders in general have an extremely low point-wise precision and recall, however, when aggregated into sequences, the precision and recall of the LSTM-based autoencoder are significantly higher.

SMD

The anomalies were labeled and filtered with automatic thresholding and anomaly pruning, the parameters were to $z \in [3, 10]$, $p = 0.01$ and $w = 2400$.

The results in Table 6.8 show that all models achieve comparable performance with an average ROC-AUC of 0.85 across all datasets, indicating that the models can distinguish reasonably well between normal and anomalous cases. However, when considering the overall precision, recall, and $F_{0.5}$ score for point-wise anomaly detection, the results vary significantly, with the LSTM-based autoencoders vastly outperforming the convolutional autoencoders. The relatively high auc and low recall across all models suggest that either the thresholding approach, the anomaly pruning, or the metric used to obtain anomaly scores is unsuitable for high-dimensional data or this dataset in particular.

Table 6.9 presents metrics adjusted in the same way as [32]. This shows that the approach of automatic threshold selection followed by anomaly pruning vastly outperforms the approach taken in [32]. We believe that the score adjustment overinflates the performance of the models. As a result, filtering of anomalous sequences based on duration is likely to be necessary to achieve satisfactory performance and limit alert fatigue. Furthermore, measuring the precision and recall of the models in terms of anomalous sequences seems more appropriate.

Dataset	Model	auc	tta [m]	p	r	$F_{0.5}$	p_w	r_w	$F_{0.5w}$
clipping	Holt-Winters	0.79	-140±14	0.37	0.93	0.42	0.50	**1.00**	0.55
	ARIMA	0.87	9±4	0.83	0.79	0.82	**1.00**	**1.00**	**1.00**
	Prophet	0.90	36±3	0.86	0.48	0.74	**1.00**	**1.00**	**1.00**
	LSTM	**0.93**	**7±5**	**0.91**	0.82	**0.89**	0.83	**1.00**	0.85
	LSTM-AE	0.84	142±10	0.76	**0.89**	0.78	0.33	**1.00**	0.38
	LSTM-VAE	0.91	35±6	0.80	0.71	0.78	0.33	**1.00**	0.38
	CONV-AE	0.81	-42±7	0.63	0.86	0.66	**1.00**	**1.00**	**1.00**
	CONV-VAE	0.88	-43±8	0.74	0.86	0.76	**1.00**	**1.00**	**1.00**
regression	Holt-Winters	0.80	5±34	0.83	0.81	0.82	0.38	**1.00**	0.43
	ARIMA	0.90	98±18	**0.91**	0.77	**0.87**	**0.87**	**1.00**	**0.89**
	Prophet	0.91	107±21	0.57	**0.93**	0.61	0.77	**1.00**	0.80
	LSTM	**0.93**	**28±8**	0.80	**0.93**	0.82	0.25	**1.00**	0.29
	LSTM-AE	0.82	244±55	0.79	0.76	0.78	0.23	**1.00**	0.27
	LSTM-VAE	0.91	348±41	0.86	0.77	0.84	0.32	**1.00**	0.37
	CONV-AE	0.85	156±30	0.74	0.81	0.75	0.25	**1.00**	0.29
	CONV-VAE	0.88	352±44	0.82	0.71	0.79	0.53	**1.00**	0.58
spikes	Holt-Winters	0.65	-67±72	0.20	0.66	0.23	0.84	0.60	0.77
	ARIMA	0.94	3±1	0.62	**0.83**	0.65	**1.00**	**0.94**	**0.98**
	Prophet	**0.96**	**1±1**	**0.85**	0.72	**0.82**	**1.00**	0.71	0.92
	LSTM	0.82	3±2	0.35	0.25	0.32	0.89	0.47	0.75
	LSTM-AE	0.61	-8±14	0.14	0.42	0.16	0.59	0.41	0.54
	LSTM-VAE	0.57	-12±17	0.14	0.39	0.16	0.66	0.35	0.56
	CONV-AE	0.62	-20±20	0.11	0.45	0.12	0.80	0.47	0.70
	CONV-VAE	0.67	-21±28	0.25	0.39	0.26	0.81	0.35	0.64
trend	Holt-Winters		**25±14**		**0.96**				
	ARIMA		144±26		0.86				
	Prophet		861±29		0.75				
	LSTM		973±31		0.54				
	LSTM-AE		1001±22		0.36				
	LSTM-VAE		669±25		0.53				
	CONV-AE		746±24		0.55				
	CONV-VAE		667±24		0.57				
chaos	Holt-Winters	0.66	-98±92	0.25	0.68	0.31	0.33	0.27	0.31
	ARIMA	0.70	9±6	0.68	**0.80**	0.71	0.64	0.82	0.66
	Prophet	0.73	9±4	0.74	0.57	**0.70**	0.76	0.91	0.78
	LSTM	**0.91**	**1±2**	**0.88**	0.35	0.50	**0.84**	**1.00**	**0.86**
	LSTM-AE	0.61	-13±19	0.18	0.46	0.26	0.66	0.91	0.69
	LSTM-VAE	0.63	-17±20	0.21	0.47	0.30	0.79	**1.00**	0.82
	CONV-AE	0.66	-14±25	0.16	0.63	0.25	0.58	0.91	0.62
	CONV-VAE	0.67	-15±21	0.29	0.38	0.36	0.62	0.91	0.66

Table 6.7 Model metrics per dataset and model

Model	auc	p	r	$F_{0.5}$	p_{w}	r_{w}	$F_{0.5\mathrm{w}}$
CONV-AE	**0.88**	0.65	0.03	0.13	0.34	0.67	0.38
CONV-VAE	0.85	0.77	**0.05**	**0.21**	0.37	0.67	0.40
LSTM-AE	0.85	**0.85**	**0.05**	**0.21**	**0.39**	**0.68**	**0.43**
LSTM-VAE	0.85	0.77	**0.05**	0.20	0.37	0.67	0.40

Table 6.8 SMD metrics

Model	p	r	F_1
CONV-AE	0.97	0.76	0.92
CONV-VAE	**0.98**	0.77	**0.93**
LSTM-AE	**0.98**	0.78	**0.93**
LSTM-VAE	**0.98**	0.77	**0.93**
OmniAnomaly	0.83	**0.94**	0.88

Table 6.9 Adjusted SMD metrics

Chapter 7

Summary

The main goal of this work was to develop a monitoring and health-checking system for anomaly detection in the context of HTTP proxy servers. We have achieved this goal by developing a gRPC service responsible for the orchestration of the monitoring processes, and a set of unsupervised anomaly detection models with automatic thresholding.

In order to benchmark the performance of the proposed models, we have created five different datasets, each exhibiting a different type of anomaly encountered in a production environment. The models were then benchmarked against these datasets, along with a dataset published by [32], showing satisfactory results.

The system is currently running in production and is being used to monitor the health of proxy servers running within a staging environment with mirrored production traffic. Furthermore, work is being done on integrating it with an automated rollouts system as a part of a canary deployment strategy within a major CDN provider.

In conclusion, we can confidently state that all the objectives of the thesis have been successfully achieved.

7.1 Future Work

There are several improvements that can be made to the system mainly regarding the precision and recall of the anomaly detection models. In order to accomplish this, we are currently in the process of gathering data from the production environment and creating a more extensive anomaly dataset encompassing a wider range of anomaly types, including network-related ones. Furthermore, we would like to explore the use of ensemble methods, which currently have basic support in the system, but are not yet fully tested.

Another improvement would be to integrate a root-cause analysis system that would help admins, operators, as well as developers to better understand

the reasons behind the anomalies detected by the system, as it currently requires a somewhat deep understanding of the monitored services and their exported metrics, depending on the configuration.

Finally, we would like to extend the system to support real-time streaming of metrics. Near-real-time batch processing with a latency of a few minutes can be currently achieved by one of the gRPC calls that the system provides, but in reality, it only serves as a proof of concept and is unsuitable for production use.

7.2 Contributions to the Open-Source Community

As a bonus, we have contributed to the open-source community by both reporting and fixing bugs in the StatsForecast and Metaflow projects.

The contribution to the StatsForecast project was a bug report filed under issue #937 and fixed by the author in a subsequent pull request #939. The issue was related to invalid memory accesses during ARIMA model training.

The first contribution to the Metaflow project was a bug report filed under issue #2034, it was promptly resolved by the maintainers in a pull request #2053 based on the author's suggestions and code.

The second contribution was related to inter-process communication and subprocess management in the Runner and Deployer components. It was developed in pull request #2056 and merged upstream. As is tradition, we introduced a bug in the process of fixing another one, causing confusion to at least one Netflix engineer. Curiously, pipes in MacOS start blocking after buffering around 8 kB of data, which was causing a deadlock. It was promply fixed by the maintainers in pull request #2169.

Appendix A

Keras models

Code listing A.1 Keras model for the forecasting LSTM

```
def lstm(
    shape: tuple,
    num_units: list[int],
    prediction_length: int,
    inner_activation: str = "tanh",
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    model = keras.models.Sequential()

    model.add(
        keras.layers.InputLayer(
            shape=shape,
            name="input",
        )
    )

    for i, units in enumerate(num_units):
        return_sequences = i < len(num_units) - 1
        name = (
            f"lstm_sequences_{units}"
            if return_sequences
            else f"lstm_output_{units}"
        )

        model.add(
            keras.layers.LSTM(
                units,
                activation=inner_activation,
                dropout=dropout,
                return_sequences=return_sequences,
```

```
                name=name,
            )
        )

    model.add(
        keras.layers.Dense(
            units=prediction_length,
            activation=output_activation,
            name="output",
        )
    )

    return model
```

Code listing A.2 Keras model for the convolutional autoencoder

```
def conv_ae(
    shape: tuple,
    conv_filters: list[int],
    conv_activation: str = "relu",
    stride: int = 2,
    padding: str = "same",
    kernel_size: int = 3,
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    model = keras.models.Sequential()

    model.add(
        keras.layers.InputLayer(
            shape=shape,
            name="input",
        )
    )

    for i, filters in enumerate(conv_filters):
        model.add(
            keras.layers.Conv1D(
                filters,
                kernel_size=kernel_size,
                padding=padding,
                strides=stride,
                activation=conv_activation,
                name=f"encoder_{i}",
            )
        )
        model.add(keras.layers.Dropout(dropout))

    for i, filters in enumerate(reversed(conv_filters)):
```

```
        model.add(
            keras.layers.Conv1DTranspose(
                filters,
                kernel_size=kernel_size,
                padding=padding,
                strides=stride,
                activation=conv_activation,
                name=f"decoder_{i}",
            )
        )
        model.add(keras.layers.Dropout(dropout))

    model.add(
        keras.layers.Conv1DTranspose(
            shape[-1],
            kernel_size=kernel_size,
            padding=padding,
            strides=1,
            activation=output_activation,
            name="output",
        )
    )

    return model
```

Code listing A.3 Keras model for the LSTM autoencoder

```
def lstm_ae(
    shape: tuple,
    num_units: list[int],
    inner_activation: str = "tanh",
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    if len(num_units) < 2:
        raise ValueError(
            "Number␣of␣LSTM␣units␣must␣be␣at␣least␣2"
        )
    if len(shape) != 2:
        raise ValueError("Input␣shape␣must␣be␣2D")

    model = keras.models.Sequential()

    model.add(
        keras.layers.InputLayer(
            shape=shape,
            name="input",
        )
    )
```

```
    for i, units in enumerate(num_units):
        model.add(
            keras.layers.LSTM(
                units,
                activation=inner_activation,
                dropout=dropout,
                return_sequences=i < len(num_units) - 1,
                name=f"encoder_{i}",
            )
        )

    model.add(keras.layers.RepeatVector(shape[0],
                                        name="repeat_vector"))

    reversed_units = list(reversed(num_units))[1:]

    for i, units in enumerate(reversed_units):
        model.add(
            keras.layers.LSTM(
                units,
                activation=inner_activation,
                dropout=dropout,
                return_sequences=True,
                name=f"decoder_{i}",
            )
        )

    model.add(
        keras.layers.Dense(
            shape[-1],
            activation=output_activation,
            name="output",
        )
    )

    return model
```

Code listing A.4 Keras model for the variational autoencoder

```
class Sampling(keras.layers.Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def call(self, inputs):
        z_mean, z_log_var = inputs
        epsilon = keras.random.normal(
            shape=keras.ops.shape(z_mean),
            seed=self.seed_generator,
        )
```

```
        return (
            z_mean
            + keras.ops.exp(0.5 * z_log_var) * epsilon
        )

    def compute_outer_shape(self, inputs):
        z_mean, z_log_var = inputs
        return keras.ops.shape(z_mean)


class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.loss = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.loss_tracker,
            self.reconstruction_loss,
            self.kl_loss,
        ]

    def train_step(self, data):
        x, y = data
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(x)
            reconstruction = self.decoder(z)
            reconstruction_loss = keras.ops.mean(
                keras.ops.square(y - reconstruction)
            )
            kl_loss = (
                - 0.5
                * (1 + z_log_var - keras.ops.square(z_mean)
                - keras.ops.exp(z_log_var)
            )
            total_loss = reconstruction_loss + kl_loss
            grads = tape.gradient(total_loss,
                                  self.trainable_weights)
            self.optimizer.apply_gradients(
                zip(grads, self.trainable_weights)
            )

        self.optimizer.apply_gradients(
```

```
            zip(grads, self.trainable_weights)
        )
        self.loss.update_state(total_loss)
        self.reconstruction_loss.update_state(
            reconstruction_loss
        )
        self.kl_loss.update_state(kl_loss)

        return {
            "loss": self.loss.result(),
            "reconstruction_loss":
                self.reconstruction_loss.result(),
            "kl_loss": self.kl_loss.result(),
        }

    def call(self, inputs):
        _, _, z = self.encoder(inputs)
        return self.decoder(z)
```

Code listing A.5 Keras model for the convolutional variational autoencoder

```
def encoder(
    shape: tuple,
    conv_filters: list[int],
    latent_dim: int,
    conv_activation: str = "relu",
    stride: int = 2,
    padding: str = "same",
    kernel_size: int = 3,
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    encoder_inputs = keras.Input(shape=shape)

    x = encoder_inputs
    for i, filters in enumerate(conv_filters):
        x = keras.layers.Conv1D(
            filters,
            kernel_size=kernel_size,
            padding=padding,
            strides=stride,
            activation=conv_activation,
            name=f"encoder_{i}",
        )(x)
        x = keras.layers.Dropout(rate=dropout)(x)

    z_mean = keras.layers.Dense(latent_dim,
                                name="z_mean")(x)
    z_log_var = keras.layers.Dense(latent_dim,
                                   name="z_log_var")(x)
```

```
    z = Sampling()([z_mean, z_log_var])

    return keras.Model(
        encoder_inputs,
        [z_mean, z_log_var, z],
        name="encoder"
    )


def decoder(
    shape: tuple,
    conv_filters: list[int],
    latent_dim: int,
    conv_activation: str = "relu",
    stride: int = 2,
    padding: str = "same",
    kernel_size: int = 3,
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    latent_inputs = keras.Input(shape=(None, latent_dim))

    x = latent_inputs
    for i, filters in enumerate(reversed(conv_filters)):
        x = keras.layers.Conv1DTranspose(
            filters,
            kernel_size=kernel_size,
            padding=padding,
            strides=stride,
            activation=conv_activation,
            name=f"decoder_{i}",
        )(x)
        x = keras.layers.Dropout(rate=dropout)(x)

    decoder_output = keras.layers.Conv1DTranspose(
        shape[-1],
        kernel_size=kernel_size,
        activation=output_activation,
        strides=1,
        padding=padding,
    )(x)

    return keras.Model(latent_inputs,
                       decoder_output,
                       name="decoder")


def conv_vae(
    **kwargs,
```

```
) -> keras.Model:
    encoder_model = encoder(**kwargs)
    decoder_model = decoder(**kwargs)
    return VAE(encoder_model, decoder_model)
```

Code listing A.6 Keras model for the LSTM variational autoencoder

```
def encoder(
    shape: tuple,
    num_units: list[int],
    latent_dim: int,
    inner_activation: str = "tanh",
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    encoder_inputs = keras.Input(shape=shape)

    x = encoder_inputs
    for i, layer in enumerate(num_units):
        x = keras.layers.LSTM(
            layer,
            activation=inner_activation,
            dropout=dropout,
            return_sequences=True,
            name=f"encoder_{i}",
        )(x)

    z_mean = keras.layers.Dense(latent_dim,
                                name="z_mean")(x)
    z_log_var = keras.layers.Dense(latent_dim,
                                   name="z_log_var")(x)
    z = Sampling()([z_mean, z_log_var])

    return keras.Model(encoder_inputs,
                       [z_mean, z_log_var, z],
                       name="encoder")


def decoder(
    shape: tuple,
    num_units: list[int],
    latent_dim: int,
    inner_activation: str = "tanh",
    output_activation: str = "linear",
    dropout: float = 0.0,
    **kwargs,
) -> keras.Model:
    latent_inputs = keras.Input(shape=(None, latent_dim))
```

```
    x = latent_inputs
    for i, layer in enumerate(reversed(num_units)):
        x = keras.layers.LSTM(
            layer,
            activation=inner_activation,
            dropout=dropout,
            return_sequences=True,
            name=f"decoder_{i}",
        )(x)

    decoder_output = keras.layers.Dense(
        shape[1],
        activation=output_activation,
        name="output",
    )(x)

    return keras.Model(latent_inputs,
                       decoder_output,
                       name="decoder")


def lstm_vae(
    **kwargs,
) -> keras.Model:
    encoder_model = encoder(**kwargs)
    decoder_model = decoder(**kwargs)
    return VAE(encoder_model, decoder_model)
```

Bibliography

1. GREGG, Brendan. *Systems performance*. 2nd ed. Upper Saddle River, NJ: Pearson, 2021. Addison-Wesley Professional Computing Series.
2. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 9.1 Stationarity and Differencing. 3rd. OTexts, 2021. Available also from: `https://otexts.com/fpp3/stationarity.html`.
3. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 2.8 Autocorrelation. 3rd. OTexts, 2021. Available also from: `https://otexts.com/fpp3/acf.html`.
4. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 9.5 Non-Seasonal ARIMA models. 3rd. OTexts, 2021. Available also from: `https://otexts.com/fpp3/acf.html`.
5. TRUONG, Charles; OUDRE, Laurent; VAYATIS, Nicolas. Selective review of offline change point detection methods. *Signal Processing*. 2020, vol. 167, p. 107299. ISSN 0165-1684. Available from DOI: `10.1016/j.sigpro.2019.107299`.
6. CHOW, Gregory C. Tests of Equality Between Sets of Coefficients in Two Linear Regressions. *Econometrica*. 1960, vol. 28, no. 3, p. 591. ISSN 0012-9682. Available from DOI: `10.2307/1910133`.
7. AUTHORS, ruptures. *ruptures: A Python Library for Change Point Detection*. Centre Borelli, 2020. Available also from: `https://centre-borelli.github.io/ruptures-docs/`.
8. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 5.5 Distributional forecasts and prediction intervals. 3rd. OTexts, 2021. Available also from: `https://otexts.com/fpp3/prediction-intervals.html`.
9. DAVISON, Anthony; HINKLEY, D. Bootstrap Methods and Their Application. *Journal of the American Statistical Association*. 1997, vol. 94. Available from DOI: `10.2307/1271471`.

10. BOX, George E P; JENKINS, Gwilym M; REINSEL, Gregory C; LJUNG, Greta M. *Time series analysis*. 5th ed. Nashville, TN: John Wiley & Sons, 2015. Wiley Series in Probability and Statistics.

11. HYNDMAN, Rob J; ATHANASOPOULOS, George. 9 ARIMA Models. In: *Forecasting: Principles and Practice*. 3rd. OTexts, 2021. Available also from: https://otexts.com/fpp3/arima.html.

12. HYNDMAN, Robert J. 2010. Available also from: https://robjhyndman.com/hyndsight/longseasonality/.

13. WINTERS, Peter R. Forecasting Sales by Exponentially Weighted Moving Averages. *Management Science*. 1960, vol. 6, no. 3, pp. 324–342. ISSN 1526-5501. Available from DOI: 10.1287/mnsc.6.3.324.

14. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 8.3 Methods with seasonality. 3rd. OTexts, 2021. Available also from: https://otexts.com/fpp3/holt-winters.html.

15. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Forecasting: Principles and Practice*. 8.4 A taxonomy of exponential smoothing methods. 3rd. OTexts, 2021. Available also from: https://otexts.com/fpp3/taxonomy.html.

16. TAYLOR, Sean J; LETHAM, Benjamin. Forecasting at scale. 2017. Available from DOI: 10.7287/peerj.preprints.3190v2.

17. ELMAN, Jeffrey L. Finding Structure in Time. *Cognitive Science*. 1990, vol. 14, no. 2, pp. 179–211. ISSN 1551-6709. Available from DOI: 10.1207/s15516709cog1402_1.

18. JORDAN, Michael I. Serial Order: A Parallel Distributed Processing Approach. In: *Neural-Network Models of Cognition - Biobehavioral Foundations*. Elsevier, 1997, pp. 471–495. ISSN 0166-4115. Available from DOI: 10.1016/s0166-4115(97)80111-2.

19. BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*. 1994, vol. 5, no. 2, pp. 157–166. ISSN 1941-0093. Available from DOI: 10.1109/72.279181.

20. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. *Neural Computation*. 1997, vol. 9, no. 8, pp. 1735–1780. ISSN 1530-888X. Available from DOI: 10.1162/neco.1997.9.8.1735.

21. CONTRIBUTORS, PyTorch. *PyTorch Documentation: LSTM (v2.5)*. 2024. Available also from: https://pytorch.org/docs/2.5/generated/torch.nn.LSTM.html. Accessed: 2024-12-03.

22. NASSIF, Ali Bou; TALIB, Manar Abu; NASIR, Qassim; DAKALBAB, Fatima Mohamad. Machine Learning for Anomaly Detection: A Systematic Review. *IEEE Access*. 2021, vol. 9, pp. 78658–78700. ISSN 2169-3536. Available from DOI: 10.1109/access.2021.3083060.

23. CONTRIBUTORS, PyTorch. *PyTorch Documentation: torch.nn.Conv1d (v2.5)*. 2024. Available also from: https://pytorch.org/docs/2.5/generated/torch.nn.Conv1d.html#torch.nn.Conv1d. Accessed: 2024-12-03.

24. CONTRIBUTORS, PyTorch. *PyTorch Documentation: Pooling Layers (v2.5)*. 2024. Available also from: https://pytorch.org/docs/2.5/nn.html#pooling-layers. Accessed: 2024-12-03.

25. VICTORIAMETRICS. *VictoriaMetrics Documentation*. 2024. Available also from: https://docs.victoriametrics.com/. Accessed: 2024-12-03.

26. VICTORIAMETRICS. *vmagent Documentation*. 2024. Available also from: https://docs.victoriametrics.com/vmagent/. Accessed: 2024-12-03.

27. VICTORIAMETRICS. *MetricQL Documentation*. 2024. Available also from: https://docs.victoriametrics.com/metricsql/. Accessed: 2024-12-03.

28. VICTORIAMETRICS. *vmalert Documentation*. 2024. Available also from: https://docs.victoriametrics.com/vmalert/. Accessed: 2024-12-03.

29. VICTORIAMETRICS. *Anomaly Detection Documentation*. 2024. Available also from: https://docs.victoriametrics.com/anomaly-detection/. Accessed: 2024-12-03.

30. PROMETHEUS AUTHORS. *Node Exporter*. GitHub, [n.d.]. Available also from: https://github.com/prometheus/node_exporter.

31. WILKIE, Tom. *The RED Method: Patterns for instrumentation & monitoring*. 2018. Available also from: https://www.slideshare.net/slideshow/the-red-method-how-to-monitoring-your-microservices/120321563. Accessed: 2024-12-03.

32. SU, Ya; ZHAO, Youjian; NIU, Chenhao; LIU, Rong; SUN, Wei; PEI, Dan. Robust Anomaly Detection for Multivariate Time Series through Stochastic Recurrent Neural Network. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019. KDD '19. Available from DOI: 10.1145/3292500.3330672.

33. GEETHIKA, Deshani; JAYASINGHE, Malith; GUNARATHNE, Yasas; GAMAGE, Thilina Ashen; JAYATHILAKA, Sudaraka; RANATHUNGA, Surangika; PERERA, Srinath. Anomaly Detection in High-Performance API Gateways. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 995–1001. Available from DOI: 10.1109/hpcs48598.2019.9188100.

34. AMDAHL, Gene M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. ACM Press, 1967. AFIPS '67 (Spring). Available from DOI: 10.1145/1465482.1465560.

35. GLOZER, Will. *wrk: A HTTP benchmarking tool* [https://github.com/wg/wrk]. 2012. Accessed: 2024-12-05.

36. FACEBOOK. *Prophet Documentation: Hyperparameter tuning*. [N.d.]. Available also from: https://facebook.github.io/prophet/docs/diagnostics.html#hyperparameter-tuning. Accessed: 2024-12-25.

37. TAYLOR, Sean J.; LETHAM, Ben. *Prophet: Forecasting at Scale*. 2024. Available also from: https://github.com/facebook/prophet/blob/main/python/prophet/forecaster.py. GitHub repository, Accessed: 2024-12-25.

38. HUNDMAN, Kyle; CONSTANTINOU, Valentino; LAPORTE, Christopher; COLWELL, Ian; SODERSTROM, Tom. Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. London, United Kingdom: Association for Computing Machinery, 2018, pp. 387–395. KDD '18. ISBN 9781450355520. Available from DOI: 10.1145/3219819.3219845.

39. XU, Haowen; CHEN, Wenxiao; ZHAO, Nengwen; LI, Zeyan; BU, Jiahao; LI, Zhihan; LIU, Ying; ZHAO, Youjian; PEI, Dan; FENG, Yang; CHEN, Jie; WANG, Zhaogang; QIAO, Honglin. Unsupervised anomaly detection via variational auto-encoder for seasonal KPIs in Web applications. 2018. Available from arXiv: 1802.03903 [cs.LG].