Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Online Failure Prediction for Microservice Architectures

Tim Zwietasch

Course of Study: Informatik

Examiner: Dr.-Ing. André van Hoorn

Supervisor: Teerat Pitakrat, M.Sc.

Commenced: February 5, 2017

Completed: August 7, 2017

CR-Classification: I.7.2

Abstract

In many modern software architectures, failure avoidance strategies are already an integral part of the system since they provide many ways to contribute to software resilience. Failures are the cause of system downtimes and latencies and often, they can not be completely prevented. In contrast to fully virtualized servers on which applications are run, some microservice architectures allow microservies to operate natively on the underlying OS and they might therefore interact with each other on a much higher level. Microservices that are deployed on the same node may also affect each other much more than VM's, for example by putting a high workload on the underlying host. Traditional VM's run their own Operating system, often in an isolated memory region and a predetiermined, mostly static CPU share whereas microservices are able to cooperate by sharing the same IP-address and other resources.

The goal of this thesis is to show how and to which degree microservices can affect each other when they are being executed on the same host and to discuss the effects that these side effects have on failure predictors. For this, a number of simulations are run on a selected containerized application that demonstrate the container-induced side effects. Certain metrics like the CPU-usage of the containers will be evaluated for each scenario and online failure prediction methods are implemented that try to forecast failures based on these metrics. The results show, that independent microservices can affect each other in various ways, for example, by over-utilizing the CPU resources of the host on which they are deployed on. This effect makes failure prediction with monolithic approaches that do not consider the architecture of the host very difficult. This thesis shows and discusses various scenarios in which hierarchical failure prediction methods show significantly better results than monolithic aproaches when such a side-effect is introduced into the system.

Kurzfassung

In vielen Softwaresystemen sind Fehlervermeidungs-Techniken mittlerweile ein wichtiger Teil der Architektur, da sie viele Möglichkeiten zur Ausfallsicherheit bieten. Oft sind Fehler der Grund für Ausfälle und Störungen und oft können diese nicht komplett verhindert werden. Im Vergleich zu Virtualisierten Betriebssystemen auf denen Anwendungen betrieben werden, operieren manche Microservices nativ auf dem zugrunde liegenden Host-System. Daher können diese Systeme sehr viel enger zusammenarbeiten, können sich gleichzeitig aber auch gegenseitig mehr beeinflussen. Microservices die auf dem selben Host ausgeführt werden, können sich beispielsweise durch eine hohe CPU-Auslastung beeinflussen während dies bei voll-virtualisierten Systemen kaum möglich ist, da diese oft einen abgegrenzten Bereich für CPU-Auslastung und Speichernutzung besitzen.

Das Ziel dieser Arbeit ist es, aufzuzeigen in welcher Weise und bis zu welchem Level sich Microservices gegenseitig beeinflussen können und verschiedene Möglichkeiten zu diskutieren, die dazu verwendet werden können, diese Effekte zu kompensieren. Hierfür werden einige Simulationen, die verschiedene Szenarien wiederspiegeln, erstellt. Anhand diesen werden einige Seiteneffekte von Microservices aufgezeigt und es werden Methoden entwickelt um diese Seiteneffekte zu erkennen und zu beseitigen. Anschließend werden die Ergebnisse verglichen, quantifiziert und dargestellt. Die Ergebnisse zeigen, dass Monolithische Fehlervorhersagen einigen Fällen wesentlich schlechtere Vorhersagen treffen als Algorithmen, die die Architektur des Systems in die Vorhersage miteinbeziehen sobald ein solcher Seiteneffekt in dem System auftritt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software dependability is crucial for most large software systems and is for many companies a key factor for success. Attributes like reliability, availability and survivability [KSS03] are nowadays default requirements that a system has to fulfill. In order to fulfill these attributes, software systems have to be made robust against all kinds of different failures which can arise, may it be network problems, component failures or hardware errors. In a software system, *Faults* can arise from a very wide range of causes and are therefore often hard or even impossible to avoid. Software Faults can occur almost anytime and in every system. They can be induced by hardware problems which may be caused by electrostatic discharge, radiation, dust, high temperatures, user errors, bad hard-drive sectors and other unforeseeable reasons. Software faults can also be caused by third-party libraries, compatibility problems, network limitations, CPU-overload and may even be intentionally induced and exploited by attackers that have access to the system.

However, a system in which a fault occurs does not necessarily suddenly cease to work. In fact, there are generally three phases that a system undergoes before a failure noticeably influences the behavior of it. These phases are ([ALR+01]):

1. *Faults*. As already described, Faults arise from various sources but they do cause any immediate harm to a system.
2. *Errors*. When a system is in an invalid state, the error is the part of functionality that will cause a failure when it is executed. At this point, the fault has established itself in the system and will cause problems when it is executed.
3. *Failures*. The moment when a systems action differ from the specified functionalities, the error has propagated in the system to an extent that a user-recognizable failure has manifested. The failure may be recognized in many ways, e.g. in a slowed-down system or even a complete system crash.

Fortunately, Faults can often be detected before they cause any serious harm by various failure avoidance strategies that nowadays are vital in virtually every computer system available. Most often, a failure is indicated by preceding *symptoms* that can be observed in the application. A symptom may be caused by an undetected error and can manifest as abnormal system behavior [SLM10].

Consider a web service provider which is confronted with a sudden increase of service requests threatening to overload the systems resources. If the provider is not prepared for this (e.g. the increase was too sudden and unexpected), the fault (*'Too Many Requests'*) will eventually result in a noticeable failure (*'Service Unresponsive'*) which the users of the service will experience. If however the service provider can predict the situation beforehand (e.g. they know that every day at a certain time there are going to be more requests from their customers) and distribute the service for this period on more machines to cope with the increased demand, they can avoid any major loss of performance during this time.

As can be seen from this example, a failure can only be avoided if it first can somehow be *predicted*. This however first requires a symptom or symptom trend that is reliable enough to be used as a basis for the prediction. Figure 1.1 shows an example of what the CPU-workload could look like in this case when recorded over several days. In this example, the symptom can easily be determined by analyzing the historical CPU-data. From this data, it would be simple to determine the next CPU-peak and therefore it would be easy to avoid any resource shortcomings at this time.

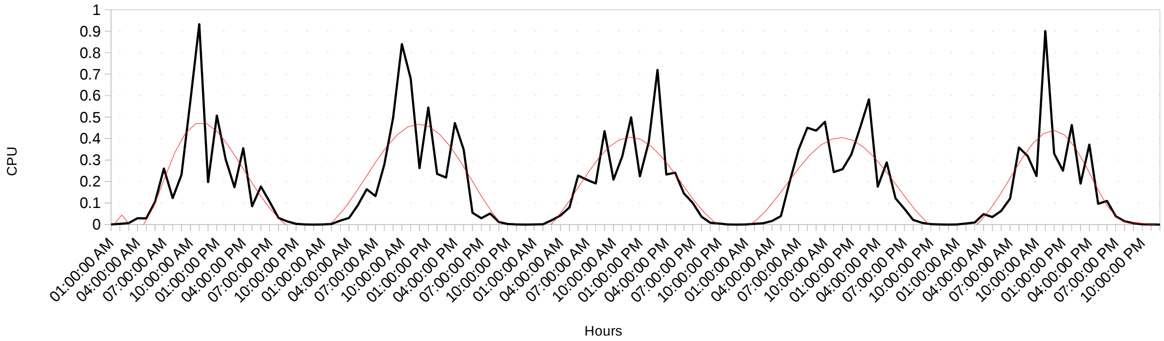


Figure 1.1.: CPU-Workload over several Days.

The application and applicability of failure prediction methods have been controversially discussed ever since the first related articles have been published at around 1970 [OB15]. Over the years, many more articles about this topic were published and lots of failure prediction algorithms and models have been introduced, enhanced and extended. Nowadays, failure prediction methods have found widespread application in one form or another on various system architectures and are crucial for maintaining stability. Failure prediction methods are nowadays used in distributed systems to monitor and maintain the health of individual components and to take precautious steps to prevent system failures, in stock market analysis to predict the future trend of a stock, in astrology to predict the movement of stars and in many more areas [**sasalfner2010survey**].

Naturally, with modern system architectures such as virtualization, theses failure prediction methods have to be re-evaluated, enhanced and re-developed since they have to be adapted for the new environment. This also applies to a very recent concept of virtualization, *Microservices*. Microservices are encapsulated services that, in contrast to virtualization can be run very efficiently in parallel on the same host, therefore allowing for hundreds of Microservices running on a single host. This, in turn, causes several problem that have not been of much importance in systems that only virtualize a constant number of services. Normally, microservices are used to compose a complex, distributed system that consists entirely of independent encapsulated and decoupled components. One of the advantages of microservices is that they can be relatively easily replicated and distributed over a cluster of physical hosts. That way, in case of component failure, resource shortage or similar problems, a microservice application can be scaled to the new needs within seconds manually by system administrators or even automatically by respective load balancers that monitor the health of the system.

The goal of this thesis is to evaluate the applicability of current failure prediction methods in a Microservice environment. The thesis will discuss about how the relevant monitoring data for Microservice can be acquired from the system, how the data can be stored and managed and how failure predictors are affected by the Microservice environment.

Thesis Structure

In Chapter 3 (Related Work), the related work is discussed that might be used to improve and enhance the approach of this thesis. Chapter 4 (Approach) will discuss the general approach of this thesis in detail. Afterwards, Chapter 5 (Implementation) will discuss about the implementation and the setup of the approach and shows the techniques and tools that were used to construct the environment, inject failures, extract the data and analyze the results. The results of the thesis will be presented in chapter Chapter 6 (Evaluation). Finally, Chapter 7 (Conclusion) will summarize the work that is done in this thesis and show potential future work that could be done by a follow-up study.

Chapter 2

Foundations

This chapter introduces the basic concepts of the topics discussed in this thesis. Section 2.1 introduces the concept of containerized architectures and compare them with traditional system architectures. Section 2.2 describes the idea of microservices and discusses about commonly used platforms that provide implementations of this concept. In section 2.3, tools, techniques and properties of failure prediction algorithms are introduced and in section 2.4, the concepts of application monitoring in microservice environments are discussed.

2.1. Containerized Architectures

Virtualization is often based on system hypervisors that serve as an abstraction layer between the hardware and the virtualized operating system. These hypervisors use a number of methods like *Hardware Virtualization* or *Binary Translation* to transform the instruction set of the virtualized system into the instruction set of the host system [DRK14]. *Containers* are operating systems that are deployed on a host OS and run natively on it. Therefore, the overhead that containers put onto the system is usually lower than a virtualized OS while still providing most of the benefits of a fully virtualized system [DRK14; FFRR15]. Usually, the memory needed to store a container image is significantly lower than that of a fully virtualized OS and the startup-time for containers is relatively low which enables a much easier way of sharing container images over the network [DRK14]. This is due to the fact that virtualized systems need to store and install every required library and dependency directly in the image itself while containers can consist of not much more than a single process.

The overhead in virtual machines can nowadays be very small due to a lot of research and improvement regarding the instruction transformation. The Linux *KVM*-feature

('Kernel Virtual Machine'), for example, enables the host OS to run unmodified images of operating systems by using hardware virtualization [FFRR15]. Strategies like *Kernel Same-page Merging* (*KSM*) enable multiple VM's the use of a single memory page thus optimizing the overhead that comes from memory-page duplication [XXHW13]. Since VM's have a constant, predefined number of virtual CPU's and their own pre-allocated RAM region, the resources are strictly limited and a single VM will therefore affect other VM's that are hosted on the same machine only in a limited way [FFRR15]. There are techniques to dynamically change the resource allocation of VM's (*Self-Balooning*, *Hotplugging*) but these require special hardware support and lead to several other problems which is why they are often not used in practice [FFRR15].

While virtualized operating systems naturally provide a good level of isolation, a containerized architecture has to implement special techniques for providing isolation [FFRR15]. This is usually done by differentiating containers by unique ID's and enhancing the system-call security by new features. The level of isolation depends on the implementation of the containerized architecture which enables the deployment of different types of containers. For example, containers can be designed such that they only run a single process while sharing every other resource with other deployed containers. Containers can also share a single IP-address since natively, they do not have an individual network address for themselves [FFRR15].

2.2. Microservices

Microservices are independently running applications with a single responsibility [Thö15]. They are usually deployed as individual containers within a containerized architecture and managed by a replication controller like *Kubernetes* such that they can easily be scaled up or down by adding or removing instances from the system. Containerization platforms like *Docker* [Doc16a] are currently used by many big companies like Amazon or Netflix [Thö15] and are a powerful tool for developing modularized and scalable systems.

2.2.1. Docker

Docker[1] is a lightweight and open platform that uses containerization to deploy services that run in a completely isolated environment (container) on a host. A host usually allows for many parallel executions of these containers, providing virtualization for

[1] http://docker.com/

applications without actually having to virtualize a complete operating system for each instance. That way, a single host is capable of running hundreds of containers at the same time, providing a very scalable way of load sharing and load balancing for systems and services. One of the main problems docker addresses is the transition of a service in development and a deployed service [And15]. Often, an application is developed on a machine completely different from the machine on which the application is run later on. This often causes several problems which are hard to understand and prevent for the applications developer since they are machine-specific and cannot be created on a developers machine. By using Docker, the developed application can be directly shipped and executed within its required, encapsulated environment. That way, the application can easily be made independent from its real, physical machine and only depend on the file system provided by the docker container.

Another feature of Docker, or containers in general, is that it eases application clustering, which is especially important for systems that provide PaaS ("Platform as a Service") or any other scalable system. In contrast to hardware clustering which requires the exchange of physical hardware upon an extension of a system, application clustering is purely software-based and thus enables a faster and more efficient way to scale a service. By using containers instead of VM's, load balancing within a single node can be performed much more efficient since all containers run on the same host OS. Docker containers can be managed by application clustering systems such as Docker Swarm [Doc16b] or Kubernetes [Kub16] which will be described in more detail in the next section.

Figure 2.1 illustrates the containers created by docker. As can be seen, Docker does not create any guest operating systems for virtualization. Instead, all containers are run on the host OS itself.

Figure 2.2 shows the components of a docker system. Basically, there are three Actors involved:

1. The client represents the user interface to the docker system [Doc16a] which can be used to configure the service. The client enables the user to communicate with the docker daemon.
2. The Host executes the docker daemon which is responsible to create and run the system. The host contains the docker containers and the images which consist of read-only information on describing how a docker container can be created.
3. The registry enables to share the docker images and can be either public or private.

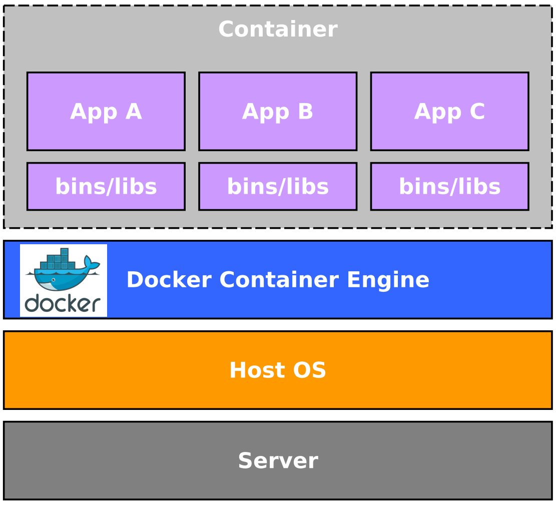


Figure 2.1.: Docker container architecture. The Docker engine separates the applications from their underlying physical host without requiring a guest OS.

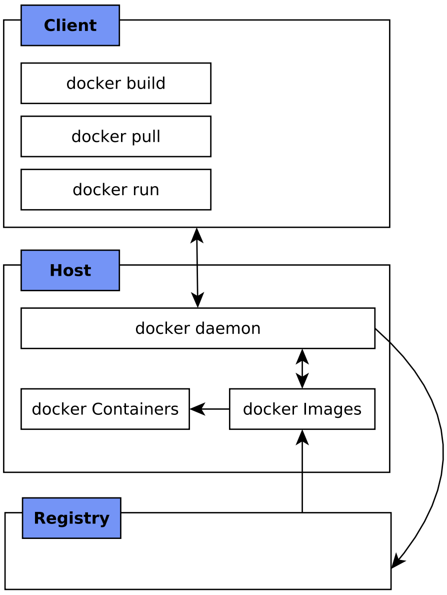


Figure 2.2.: Docker Architecture overview.

2.2.2. Application Clustering

A containerized architecture can be very beneficial for application clustering, which enables the ability to distribute applications equally within several interconnected physical hosts. Application clustering can be realized by installing clustering software on each of these hosts.

For Docker, the currently most popular application clustering managers are Docker Swarm [Doc16b] and Kubernetes [Kub16].

2.2.3. Kubernetes

Kubernetes[2] is an open-source platform for deployment automation and container scaling written by Google. It allows to deploy Docker containers in a computer cluster and provides several methods for container replication and scaling. Kubernetes automatically

[2] http://kubernetes.io/

deploys new containers on physical machines based on some criterion, e.g. the current workload of a specific node. A service deployed in a Kubernetes cluster can be managed by a *replication controller* or a *deployment* which for example enable an automated restart of a service in case of a failure and automate the deployment on the Kubernetes cluster. The deployment of a service on a node is by default managed by Kubernetes and depends on factors like the current workload of a specific physical host.

2.3. Failure Prediction

Now more than ever has failure prediction become a crucial *Quality-Of-Service* keypoint. With the ever growing complexity and increasing challenges of modern software solutions, such as high reliability and availability through distribution and replication of services, the capabilities of current software systems have exploded in recent years. No longer are modern services simply deployed on a static system architecture. Instead, current system architectures are much more dynamic and alive, undergoing constant change and movement in dependencies to other components not only through frequent updates but also by concepts such as deployment of services on a cluster managed by automated replication controllers and monitoring systems that are capable of automatically altering the architectural structure of the system by scaling bottleneck components up, replicating services, adding more physical hosts into the cluster when resources become scarce, regulating the network bandwidth of different components, migrating services between hosts for better efficiency and many other automated system balancing techniques. Not only is failure prediction - the art of detecting problems in a system before they occur - used to create such a self-aware system, it is often also a direct part of the system itself and therefore has to cope with the sometimes intriguing complexity of these systems. Since failure prediction is researched and used for now more than 40 years, it is by no means a new concept and there are many predictors that are simply not designed for this extent of interactivity and interconnection that newer systems implement. Very often, failure prediction is used only on single services or components, considering only measures and metrics that can be observed from within the service itself and no other aspects such as the software system architecture. These kind of services will be referred to as *monolithic services* in the following. The main value of failure prediction consists in using it to alleviate the consequences of a set of different error scenarios, such as extremely high workload on a system, a service or a single component within a service, memory leaks in parts of a service or even deadlocks between different components.

In practice there is a fairly wide range of different failure prediction techniques available, each concerning about different problems, using different input data like for example

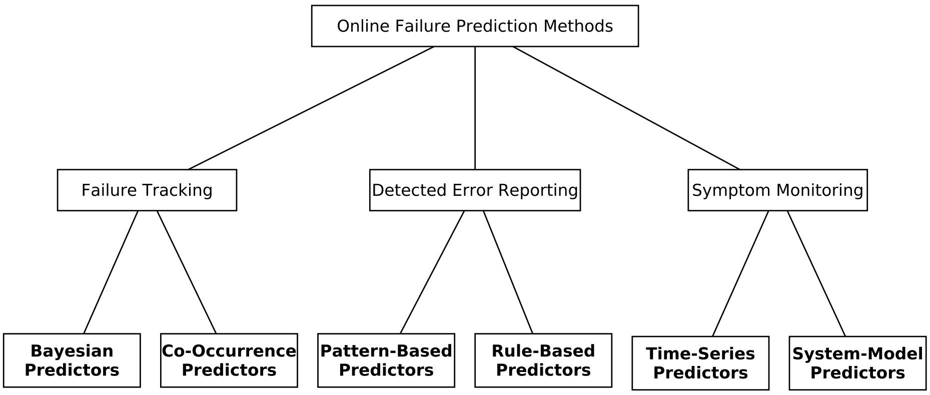


Figure 2.3.: Excerpt of Failure Prediction Methods, based on [SLM10]

the event-log of applications, time-series data like the CPU-utility over time or other, more complex metrics. In [SLM10], online failure prediction methods are classified into

1. Methods that record occurring failures (**Failure Tracking**) and use them to predict failures in the later executions. This type of predictors include, for example, *Bayesian* or *Co-Occurrence* predictors.
2. Event-driven Methods (**Error Reporting**) that try to make sense of the systems event log. Prediction methods in this area are, for example, *Pattern-Recognition* tools that are run over log files to detect faulty behavior based on certain rules, patterns or other descriptions. *Rule-Based* predictors derive failures in log files by gathering rules that indicate failures. For example, a rule may be that if a log entry V occurs close after another entry W, then the system is in a erroneous state.
3. **Symptom Monitoring Methods** which try to detect symptoms of failure in the running applications based on several measurements, metrics or models which are collected during runtime or defined by the user. Predictors that use *time-series* data track, for example, the CPU utilization and the used memory to make assumptions on the current state of the system. *System Model Predictors* create a model of the application (e.g. the different states an application can be in) and make predictions of failures based on, e.g. the state transitions that an executing application makes.

Figure 2.3 illustrates the most important failure prediction types for this area of application and shows their relations between each other.

This thesis will focus mostly on failure prediction that uses **Time-Series Data** like the memory and CPU usage of applications or the response time of key-functions within a component. These measures can easily be observed from an external monitoring component with the advantage of treating the applications more or less like a black-box.

Other prediction techniques like **System-Model Predictors** are designed to understand the architecture of individual components by computing a model that represents non-anomalous execution states, for example by creating a graph that contains nodes and edges representing valid and invalid transitions of system states which can be obtained by a preceding training phase of the predictor [SLM10].

2.3.1. Time-Series Failure Prediction

Especially for service providers like data-centers or systems where the workload can change at any point in time, time-series failure predictors and monitoring techniques are vital for the stability and quality of the provided service. The term *time-series* refers to a set of data points that have been stored successively over a certain amount of time [Cha00], like the CPU-utilization of a host which is stored every second into a logfile or a collection of temperature measurements obtained by a temperature-sensor. A more detailed definition and discussion about time-series can be found in [Cha00], on which the following is based on.

Consider a set of data points $X_{1:n} = \{(t_1, x_1), (t_2, x_2), ..., (t_n, x_n)\}$. If $\forall(t, x) \in X : t = c$ with $c \in \mathbb{R}$ constant. In this thesis, this kind of data is referred to as a *time-series*. The task of time-series forecasting methods is to determine the set of $\hat{X}_{n+1:n+h} = \{(c, x_{n+1}), (c, x_{n+2}), (c, x_{n+h})\}$ with h being defined as the *forecasting horizon* out of the set X with the goal of forecasting $\hat{X}$ such that the continuation of X, $X_{1:n+h}$ and the set $X_{1:n} \cup \hat{X}_{n+1:n+h}$ is as *similar* as possible. The term similar can be interpreted in many different ways and often depends on the use-case. The *Mean Squared Error* is a metric that can be used to measure similarity. There are many different ways of designing such a forecasting method and there are many factors such as time-efficiency and resource utilization which makes it impossible to create a predictor that fits the need of every use-case. Therefore, time-series predictors range from simple linear regression algorithms to more complex techniques, for instance predictors that combine standard time-series prediction with the use of neural-networks [Zha03].

Depending on the use-case, failure prediction methods are used over different inputs and they are often specially designed for a certain pattern of time-series data. Loosely, they may be classified into **supervised**, **semi-supervised** and **unsupervised** methods whereas a supervised method would represent a monitoring tool that displays gathered metrics as a graph and the administrator has to decide which actions to take. An unsupervised method on the other side is an automated process that is able to self-determine actions that need to be undertaken in certain situations. Unsupervised algorithms are for example able to derive classification rules out of labeled training data and use these rules to classify new data points into two or more categories. Aside from that, [Cha00] classifies time-series predictors into

1. *Subjective forecasts.* These forecasts are made from inside-knowledge that cannot easily be put into mathematical terms and are often subjective. Example: Heuristics, supervised techniques.
2. *Univariate forecasts.* These forecasting methods use only one single metric to determine failures in a system. Example: Regression, Detection of memory leaks by monitoring memory-usage-time-series.
3. *Multivariate forecasts.* In cases where failures only arise when multiple criterion are met, multivariate forecasting methods combine different time-series into one single forecast. This may be done by using simple mathematical transformations or more advanced techniques like neural networks. Example: Feature Analysis.

Often, the parameters of forecasting methods are, at least in parts, adjusted subjectively to individual components since it may very difficult or even impossible to determine the perfect settings for some cases.

Most of the time, time-series follow a certain repetitive pattern which can be hidden by random errors and noise within the measurements of the series. There are many different forecasting algorithms that specialize in making predictions for a certain pattern type. Some characteristic time-series patterns will be described in the following.

Seasonality Of Time-Series

The term *seasonality* describes the occurrence of repeating patterns within a time-series whereas the dataset consists of one or several *seasons* that are repeated by a period L [Kal04]. Examples: A web-service which is used only at a certain time every day, a temperature sensor measuring similar temperatures each day, the network bandwidth of a host that performs repetitive tasks, etc. *Additive Seasonality* describes the behavior of a time-series where the seasonal shift is about constant for every season. Likewise, a *Multiplicative Seasonality* represents an increase by a constant factor.

Stationarity Of Time-Series

For time-series prediction it is often required that the dataset is stationary or can be transformed into a stationary representation. First-order stationarity is true for time-series where $\forall t : E(t) = \gamma$, with γ being constant holds [Kal04]. Second-Order or covariance-stationarity holds if the data is First-Order Stationary and the covariance $C(t_{a+i}, t_{a+j})$ is constant for all variations of a, i.e. given i, j, $\exists c \forall a : C(t_{a+i}, t_{a+j}) = c$ and the variance $V(t_i)$ is finite for all i. Often, but depending on the prediction algorithm, first-order stationarity is sufficient to achieve the desired results. A non-stationary time-series can be transformed into at least a First-order stationary series by applying a differencing transformation onto the data-points.

2.3.2. Time-Series Failure Prediction Techniques

As could be seen in the previous section, time-series prediction techniques are based on system or application measurements like the CPU-utilization of a service. Two very commonly used prediction techniques for time-series are *ARIMA* and *Holt-Winters* which will be discussed in detail in the following.

ARIMA

The *Autoregressive Integrated Moving Average* forecasting method is one of the standard tools when it comes to time-series forecasting. The ARIMA Model can be adjusted with three parameters $(p, q, d) \in \mathbb{N}_0+$ where p determines the number of data points $\bigcup_{i=0}^{p-1} x_{t-i}$ on which the regression is applied on, q represents the *Moving-Average* degree and d is the number of differencing operations that is applied onto the data set beforehand.

The ARIMA method first converts the input time-series data into a non-stationary sequence by applying a differencing algorithm to it (this step is referred to as the *Integrated* part in the ARIMA acronym). This can be done by applying the function 2.1 to all datapoints in the series X. In case of $d > 1$, the function is iteratively applied, i.e. $x_d = Diff^d(x) = (Diff_1 \circ Diff_2 \circ ... \circ Diff_d)(x)$

$$Diff(x_t) = x_t - x_{t-1} \tag{2.1}$$

Basically, the method predicts future data points by computing a linear function out of preceding data points which are assumed to each have an independent random error ϵ_t with a variance of σ^2 where $\sum_t \frac{\epsilon_t}{T} = 0$ holds (refer to [Zha03] for a more detailed definition). At any point in time t, the actual input data can therefore be represented as

$$\tilde{X}_t = \sum_{i=1}^{p} \phi_i \tilde{X}_{t-i} + \epsilon_t - \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{2.2}$$

where ϕ and θ are parameters determined by the model. As can be seen by equation 2.2, if p is set to 0, the prediction method becomes a pure Moving-Average model and, respectively, setting q to 0 results in the method becoming purely autoregressive. In contrast to AR and MA Models, the difficult part of adjusting $ARIMA(d, p, q)$ for individual scenarios is to decide on the values of d, p and q instead of determining the coefficients ϕ and θ for $\tilde{X}$. There are however different ways to automatically calculate the optimal parameters for a given time-series. This can be done for example by using *Akaike's Information Criterion* [HK+07].

Holt-Winters

Especially for seasonal data, the *Holt-Winters* algorithm is commonly used to apply exponential smoothing and calculate forecasts for datasets. The forecast values that are produced by this algorithm are estimated by first smoothing the trend values of the already known data and then iteratively inferring the future trend continuation of the time-series (compare to [HK+07]).

The most simple way to smooth a time-series (*Single Exponential Smoothing*) is by applying Equation 2.3 to any given set $X = x_1, x_2, ..., x_n$.

$$\hat{x}_t = \alpha x_t + (1-\alpha)\hat{x}_{t-1}, 0 \leq \alpha \leq 1, t > 1 \tag{2.3}$$

by choosing the weight α, it is possible to determine the degree to which the previous data point influences the smoothed value at the point t. Notice that the initial value of the smoothing algorithm becomes more important the smaller the parameter α is chosen and should therefore be selected with care.

The next level of exponential smoothing (*Double Exponential Smoothing*) is applied to data sets of which are to be expected to have a certain pattern or *trend* that they follow.

$$\hat{x}_t = \alpha x_t + (1-\alpha)(\hat{x}_{t-1} + b_{t-1}), 0 \leq \alpha \leq 1 \tag{2.4}$$

$$b_t = \gamma(x_t - x_{t-1}) + (1-\gamma)b_{t-1}, 0 \leq \gamma \leq 1 \tag{2.5}$$

Finally, the **Holt-Winters** exponential smoothing (*Triple Exponential Smoothing*) considers additionally to the trend the seasonality of the data set. It is different for an additive or Multiplicative seasonality.

For the **Multiplicative Holt-Winters** which can be applied to multiplicative seasonal data, Equation 2.6 shows the general smoothing function. Equation 2.8 calculates the the seasonal component of the smoothing. Equation 2.7 represents the trend of the series and is determined by the difference of the last two data points. The parameters

$0 < \alpha, \beta, \gamma < 1$ may be chosen appropriately. p determines the period length of the seasonal part in the respective time-series.

$$A_t = \alpha(\frac{x_t}{S_{t-p}}) + (1-\alpha)(A_{t-1} + B_{t-1}) \tag{2.6}$$

$$B_t = \beta(S_t - S_{t-1}) + (1-\beta)G_{t-1} \tag{2.7}$$

$$S_t = \gamma(\frac{x_t}{A_t}) + (1-\gamma)S_{t-p} \tag{2.8}$$

After computing the exponential smoothing components A_t, B_t and S_t, forecasts can be iteratively made by determining $\hat{x}_{t+T}$ for $T > 0$.

$$\hat{x}_{t+T} = (A_{t-1} + T \cdot B_{t-1})S_{t+T-p} \tag{2.9}$$

The **Additive Holt-Winters** is almost identical to the multiplicative Holt-Winters and can be determined in almost the same way. Equations 2.10, 2.11 and 2.12 show the computation of the smoothing components for additive seasonal effects.

$$A_t = \alpha(x_t - S_{t-p}) + (1-\alpha)(A_{t-1} + B_{t-1}) \tag{2.10}$$

$$B_t = \beta(S_t - S_{t-1}) + (1-\beta)G_{t-1} \tag{2.11}$$

$$S_t = \gamma(x_t - A_t) + (1-\gamma)S_{t-p} \tag{2.12}$$

The forecasts for the additive model can be computed by using Equation 2.13.

$$\hat{x}_t = A_{t-1} + B_{t-1} + S_{t-p} \tag{2.13}$$

Note that determining the adjustable parameters of the triple exponential smoothing is, except from determining whether the dataset is additively or multiplicatively seasonal, another part of the Holt-Winters method. For more information refer to [HK+07].

C2 fails	C3 fails	C1 fails = *true*	C1 fails = *false*
false	*false*	P(C1)	1-P(C1)
false	*true*	0.4	0.6
true	*false*	0.6	0.4
true	*true*	1.0	0.0

Table 2.1.: Conditional probability Table of the Failure Propagation Model.

2.3.3. Hierarchical Failure Prediction

Time-Series prediction methods like *ARIMA* can be used to forecast the the future trend of a a measurement like the CPU-utilization of a service or component. However, if the system is modularized or distributed over several physical nodes, the detection of failures becomes harder since it will not suffice to only analyze the failure probability of a single component. Instead, it becomes necessary to consider every other component on which an observed component depends on first since failures may propagate along dependency paths.

Pitakrat et al. [POVG16] proposed a Hierarchical Online Failure Prediction approach called *HORA* that combines prediction results of single components by considering the architecture of the whole system.

First, the Architectural Dependency Model (*ADM*) is created that describes the dependencies between any observable subcomponents $\mathcal{C}$ of the system. The ADM is a set ADM where for each Entry $ADM_{C_i} = (C_i, \{(C_j \in \mathcal{C}, w_{i,j}) : C_i$ *depends on* $C_j\})$. The transitive relation *depends on* can be defined in various ways, e.g. according to the architectural dependencies of the system. The weights $w_{i,j} \in [0, 1]$ can be automatically computed or manually defined if required. The sum of weights $\sum_j w_{i,j}$ for every component C_i is 1.

Next, the Failure Propagation Model (*FPM*) is constructed. Basically, the FPM is a Bayesian Network that is derived from the ADM. For every combination of components, the failure probability is calculated by analyzing the ADM. Let $ADM := \{(C1, \{(C2, 0.4), (C3, 0.6)\}), (C2, \emptyset), (C3, \emptyset)\}$. The Conditional probability Table (*CPT*) for component $C1$ can be seen in Table 2.1. The FPM is the collection of all CPT's in the system.

The failure prediction in the *HORA* approach is done in two steps. The first step is the failure prediction for every individual subcomponent $C_i \in \mathcal{C}$. This can be done by using single-component failure predictors like an *ARIMA* forecast based on the CPU-utilization. In such a case, the failure probability $P(X > \alpha)$ where α is the CPU-threshold can be calculated by using a simple probability density function. The failure probability is constantly updated in the *FPM* during execution.

The second step combines the failure probabilities by considering the architectural dependencies defined in the *ADM*. This is done by using *Bayesian Inference*. This way, the computed failure probability for any component $C_i \in \mathcal{C}$ considers the failure probability of every ancestor component $C_j \in \mathcal{C} : C_i \textit{depends on} C_j$.

2.4. Microservice Monitoring

There are many ways to monitor a software system. Depending on the microservice platform, there are also several different tools that can be used for monitoring. Generally, there are two different ways of monitoring a VM or a container:

1. *System-level Monitoring*. These tools observe the container without making any assumptions about the internal structure or dependencies of it. This means, that those tools can be applied to almost any container but at the same time, they can only observe measurements that every container has, like the CPU- and memory utilization of a container.
2. *Application-level Monitoring*. When it becomes important to consider the internal structure of a container, application-level monitoring tools can be used in order to observe specific metrics that may be unique to a single component. An example for this would be the response-time of a single function in a component.

Often, it is beneficial to only consider system-level monitors as a source of information since especially for microservices it can soon become very complex the more structure is included in the monitoring step. Also, since the structure in microservice systems can change dynamically, often automatically during runtime, application-level monitoring becomes even harder.

2.4.1. cAdvisor & Heapster

A system-level Monitoring tool for Docker is cAdvisor [cAd17]. cAdvisor can be used to collect information like the CPU-usage of every deployed container. The monitoring tool Heapster [Hea17] can be deployed on Kubernetes to fetch data for all deployed containers in the cluster from cAdvisor.

2.4.2. Kieker Framework

The Open-Source Kieker Framework [HWH12] is a framework that offers techniques for performance monitoring and software analysis of applications at runtime. The Kieker Framework is designed to be extensible, flexible and modular and it is based on a *Pipes-and-Filter* Framework, meaning that customized components can simply be plugged into the Kieker Workflow [HWH12]. It can be used as a performance evaluation framework by for example using performance anomaly detection and other monitoring techniques.

2.5. Fault Injection

Fault injection methods are often used in scenarios like *robustness* analysis. A system is called *robust* when it can cope with runtime-induced errors and faults to some degree which may in practice be generated by defect components or intentional attacks on the system [Sve11]. Fault injection methods can then be used to simulate a certain kind of malicious behavior by for example injecting defect code into a certain part of the system.

In [Sve11], a failure category system is defined which evaluates the different fault injection strategies that can be applied to a software system. The thesis categorizes injected faults into either *Model-Implemented* (called *MIFI*, Model Induced Fault Injection), *Software-Implemented* (*SWIFI*, Software Induced Fault Injection) or *hardware Implemented* (*HIFI*, Hardware Induced Fault injection). Software-based faults can be implemented for example by directly changing the source code of a software system such that a failure is triggered while or after the program executes the changed code fragments. Model-based fault injection can be done by providing special interfaces in the software project through which a failure can externally be produced. Hardware-based fault injectors use e.g. radiation bombardment methods to induce hardware faults.

SWIFI is divided into faults injected during or before runtime. Runtime Fault-Injection tools are used to invoke faults in a system that is being executed. The simplest way to inject software faults is to change the program during compile time. For this method, there are no special tools required and it is easy to implement permanent software faults into the application. The injection at compile-time is however not as flexible as the injection at runtime.

Runtime failures can be injected through special tools that for example inject the following types of faults ([HTI97]):

- Timeouts. This simple type of fault may cause timeout and synchronization failures in the respective component.
- Exceptions. Throwing exceptions or deliberately perform actions that cause them.
- Foreign Code. It may also be possible to directly inject external code into parts of the components. This method provides the highest flexibility in regard to injecting faults but may be very complex.

MIFI techniques on the other hand require the component to implement certain interfaces through which an external tool can invoke several faults into the execution. In [Sve11], Svenningsson developed an injector called *MODIFI* that injects blocks of faulty code into a Simulink model.

2.6. Evaluation Metrics

Sometimes, it can be quite hard to compare different failure prediction results with each other. Error metrics can therefore be a useful approach to show the differences of several different predictions and to make a conclusion about what strategy yields the best results.

2.6.1. Failure Prediction Evaluation

The degree to which a failure predictor is able to predict a failure can be quantified by the *True Positive Rate* (‘*Sensitivity*’) and *True Negative Rate* (‘*Specificity*’) values. Those metrics can be calculated by using amount of True Positives (TP’s), True Negatives (TN’s), False Positives (FP’s) and False Negatives (FN’s) that are calculated by comparing the prediction of the failure predictor with the actual value.

These values can only be calculated upon *binary classification functions* which either output *TRUE* or *FALSE*. This means, that the result of any non-binary failure predictor first has to be transformed into a binary prediction which can be done for instance by using a threshold value.

The True Positives represent the amount of data points that were correctly classified as being a failure. Respectively, the True Negatives are points that were correctly classified as non-failure, the False Positives are points that were incorrectly classified as failures and False Negatives are points that were incorrectly classified as being non-failures. Figure 2.4 shows the confusion matrix for this relation.

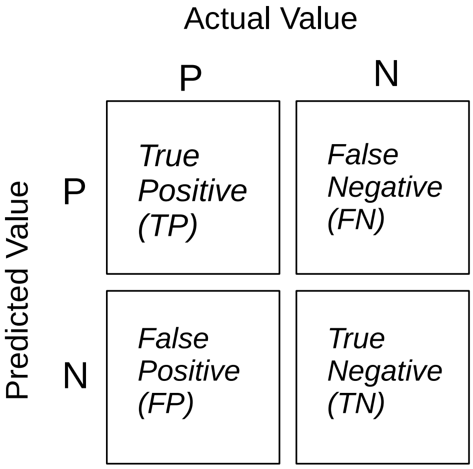


Figure 2.4.: Confusion Matrix - actual and predicted results in relation.

The true positive rate can then be calculated in the following way:

$$TPR = \frac{TP}{TP + FP} \tag{2.14}$$

and the True Negative Rate:

$$TNR = \frac{TN}{TN + FN} \tag{2.15}$$

2.6.2. Error Metrics

The result of a time-series prediction can be quantified by comparing the prediction to the original data set. This can be done for example by calculating the median of the absolute difference between any two data points (*MAD*). X hereby represents the forecast and Y the original data.

$$MAD(X, Y) = Median(|X - Y|), |X| = |Y| \tag{2.16}$$

Another failure Metric is the Medium Absolute Percentage Error (*MAPE*) which calculates the mean of the absolute percentage error.

$$MAPE(X, Y) = \frac{100}{|X|} \sum_{i=1}^{|X|} \left| \frac{X_i - Y_i}{Y_i} \right|, |X| = |Y| \tag{2.17}$$

The Mean Square Error (*MSE*) shows the average squared error of two datasets.

$$MSE(X,Y) = \frac{\sum_i^{|X|}(X_i - Y_i)^2}{|X|}, |X| = |Y| \tag{2.18}$$

The Mean Squared Deviation (MSD) sums the overall squared deviations of the datasets.

$$MSD(X,Y) = \sum_i^{|X|}(X_i - Y_i)^2, |X| = |Y| \tag{2.19}$$

Chapter 3

Related Work

This chapter discusses topics that are related to this thesis but are not directly considered in the approach. In chapter 3.1, alternative metrics are discussed that can be used for failure prediction. Next, chapter 3.2 describes several failure prediction algorithms that can also be used to predict time series data. In chapter 3.3, studies about *performance isolation* are presented and lastly, in chapter 3.4, a few studies about resilience in microservice architectures are discussed. Note, that the presented articles in this section do not represent the entire related work for this thesis. Instead, some articles were picked that are supposed to represent each research area and give insight to the big variety of research that exists in the context of this thesis.

3.1. Failure Correlation

Most failure prediction methods require some metrics that correlate with a failure that is to be predicted. A failure metric is supposed to indicate the failure in some way (i.e. show a failure trend or *symptom*) such that the predictor only needs to forecast this metric and evaluate the failure probability based on that forecasted value. A multivariate failure predictor may use several failure metrics that are then combined in some way to calculate a failure percentage.

However, what is it actually that causes a failure in a system? This thesis uses mostly system-level failure metrics like the CPU-utilization of a node to compute the failure probability. In a different scenario, this metric may not correspond to the number of failures and other ways to estimate failures need to be developed.

A study by Nagappan et al. [NBZ06] researched two sources of software failure symptoms in applications - Historical data and the actual program code the software was written in. For this, they looked at some deployed Microsoft software projects and linked the failures that were reported after their release to the respective component of the system. They then tried to create failure metrics that best correlate with these reported failures in newly added components by using a combination of metrics like the Lines Of Code (LOC) and combined them using *Principal Component Analysis*.

The study concluded, that there is no generally best combination of complexity metrics for all software systems. They have however found evidence, that for instance the number of classes in a project correlates with the number of failures post-release in some of the analyzed projects. They concluded that the way the software is designed may significantly determine which metrics correlate with the failures since often, developers already use these complexity metrics during development to reduce the amount of failures in the application. In one of the analyzed projects, almost no metric correlated with the failure count because the developers considered those metrics throughout the creation of the project. They state, that, in terms of post-release failures, it might not be a good idea to build failure predictors that solely consider a static set of complexity measurements for each application. They were however able to find correlating metrics for every single analyzed project.

This study focuses mostly on system-level measurements like the CPU- and memory usage and does not evaluate the results for different metrics. The results and findings may however be applied to other metrics as well.

3.2. Failure Prediction Techniques

Today, there is a broad variety of failure prediction methods available that cover a wide area of use. In this thesis, only basic failure prediction techniques have been used for online time-series forecasting. Higher quality results may be achieved by selecting a better forecasting method. The results in this study were made by using very common failure predictors that already exist for many years and are researched quite well. The prediction quality of the approach could however be improved by using more sophisticated failure predictors like those that are described in the following.

3.2.1. Neural Networks and ARMA

A study by Rojas et al. [RVR+08] discusses about a time-series predictor that uses a combination of *ARMA* (Autoregressive Moving Average) and *ANN*’s (Artificial Neural Networks). Their goal was to develop a prediction method which determines the prediction model without the help of expert knowledge for each scenario by using artificial intelligence strategies. The study found, that their model is able to reliably predict time-series data and is at the same time easy to use and potentially faster than other Neural Network predictors for larger data sets.

Another study by Kasabov and Song [KS02] proposed a *Neural Fuzzy Interference System* and showed how it can be applied to time-series forecasting. Their algorithm (called *DENFIS*) can be used for adaptive online learning and operates as a hybrid supervised and unsupervised algorithm. Sotani [Sol02] implemented a neural network predictor for time-series that uses *multiscale-filtering* or *wavelet-decomposition* to create a hierarchy of time-series datapoints which make classification and prediction easier and remove noise from the dataset.

3.2.2. Grey System Time Series Prediction

Grey time-series prediction Models, which were introduced by Deng [Jul89] predict time-series by only considering a subset of recent history data points [KUK10]. Grey models can help to analyze data sets with only incomplete information and are usually denoted as $GM(n, m)$ where n is the order of the differential equation and m the variable count [KUK10]. The study by Kayacan, Ulutas and Kaynak show the applicability of grey prediction models for time series and compare different Grey predictors with each other.

3.2.3. Support Vector Machine Time Series Forecasting

Support Vector Machines (SVM's) are another non-linear prediction technique that can be used for time-series prediction. Basically, SVM's are trained with sample data that enables the classification of newly inserted points by calculating a hyperplane that separates the training data into two categories. The goal of SVM's is to construct this *maximum margin hyperplane* which is the hyperplane that separates the two classes maximally [Kim03]. A study by Kim [Kim03] shows that SVM's can be used to predict financial time-series and is able to outperform neural network back-propagation models (BPN) and Case-Based Reasoning Models (CBR).

3.2.4. Bayesian Time-Series Prediction

Especially for noisy, nonstationary data, bayesian neural networks show promising forecasting abilities [BB04]. Brahim-Belhouari and Bermak [BB04] developed a bayesian time-series predictor based on a gaussian process that uses different covariance functions for temporal analyses. They show, that the forecasting results of the predictor are well suited for non-stationary datasets. The complexity of the prediction algorithm is however $\mathbb{O}(n^2)$ where n represents the number of datapoints in the set which might be a problem in many systems where fast forecasts are required.

3.2.5. Hierarchical Prediction Models

Berliner [Ber96] analyzed a hierarchical time-series predictor that uses bayesian methods to combine the prediction results of different components. In an article by Hyndman, Ahmed and Athanasopoulos [**Athanasopoulos**], a hierarchical forecasting technique is proposed that combines multiple time-series in a bottom-up or top-down fashion. They conclude, that the implemented algorithm performs well in forecasts that considers multiple levels of a system to draw conclusions. Rodrigues, Gama and Pedroso [RGP08] discusses about a time-series prediction method called *ODAC - Online Divisive-Agglomerative Clustering* that stores multiple evolving clusters in a binary tree that represents a hierarchical dependency tree of different components. ODAC is designed to cope with large amounts of data and categorizes data-points into highly-correlating sets.

3.3. Performance Isolation

A big advantage of virtual machines in contrast to monolithic systems is, that a single physical host is capable of running several isolated operating systems at the same time. Often, data centers and PaaS systems that provide VM's to customers are required to grant performance guarantees for every VM they deploy. An important task of *performance isolation* is to make sure that these contracts can be guaranteed at any time, for every virtual machine [GCGV06]. There are many studies that research on how performance isolation can be achieved. A study by Gupta et. al [GCGV06] proposed two new techniques to ensure performance isolation in the hypervisor Xen [BDF+03]. They concluded, that their algorithms can enable performance isolation in various situations. Shue, Freedman and Shaikh [SFS12] proposed a tool called *Pisces* that is able to ensure performance isolation in a data-center environment. Fedorova, Selzer and Smith [FSS07] discuss about ways to ensure performance isolation on chip multiprocessors.

In microservice environments, the problem of performance isolation also exists and the effects that result from non-isolated services will be researched in detail in this thesis.

3.4. Microservice Resilience

Microservices have many benefits over monolithic architectures that only deploy applications on one single unit. The use of microservices enables easy and fast deployment without the need of hypervisors [Ber14], offers high availability, for example, by checkpoint/restore mechanisms [Yan15] and provide a high potential for application scaling and distribution due to the independence of individual microservices [BHJ16; Has16].

A study by Hasselbring and Steinacker [HS17] evaluated the reliability of a large microservice system by considering the degree to which the system can be monitored, scaled, integrated and deployed. They argue that microservices enable a separation-of-concern policy that can help to define responsibilities amongst team members in the development phase and conclude, that microservices can in fact achieve reliability and agility if used correctly.

Another study on *learning-based testing (LBT)* by Meinke and Nycander [MN15] analyzed how learning-based tests perform on distributed microservices. They injected faults into a commercial product and evaluated how LBT tools perform. They showed, that LBT performs sufficiently well in microservices and that correctness and robustness properties can be modeled for this type of system architecture.

In this thesis, the resilience of the researched application was established by service replication, several monitoring tools and Kubernetes, which, for example, automatically redeploys services that failed.

Chapter 4

Approach

In this chapter, the overall approach of the thesis is described in detail. As already mentioned, the goal of the thesis is to find and evaluate side effects that arise in failure predictors due to the use of microservice architectures. First of all, this includes creating a containerized system environment which is capable of logging and monitoring executed applications in such a way that the extracted information can be used to apply online failure prediction. Next, an application has to be chosen on which the failure prediction can be used upon. The last step is then to transform the results into the evaluation analysis which will be presented at the end of the thesis.

This chapter contains the overall approach of the thesis without too many implementation details (for implementation details, see chapter 5). Where possible, third-party open-source software was used over reimplementations in order to demonstrate to which degree failure prediction on microservices can already be realized with available tools and, of course, to save time and efforts. This chapter will discuss about the following points:

- The general microservice architecture used in the implementation. This section describes the overall system architecture, how different nodes are interconnected to each other and which advantages and disadvantages this specific architecture holds.
- The application that is run on the microservice architecture. This part explains the details of the applications that are deployed and evaluated on the implemented architecture. It will describe the different parts needed in order to provide the system, explain the functionality of the applications and show how the components interact with each other.
- The fault injection methods used to produce errors in the system. In here, it will be discussed which specific tools are suited for fault injection in the built

up environment and how they are used to produce predictable problems in the components of the application.

- The failure prediction approach. This will explain which specific tools are being used for online failure prediction and how the prediction tools are run with the extracted system information.

4.1. The Workflow

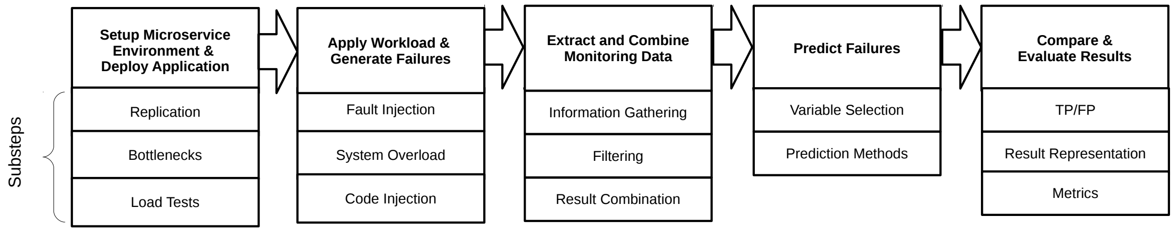


Figure 4.1.: Project Workflow

Figure 4.1 shows the overall workflow that is used to create the evaluation results of this thesis. The microservice architecture is the foundation on which the actual evaluation can be executed on. As such, it is important, that this system is able to deploy and replicate multiple containerized applications, enable application clustering and lets each component interact with other components as needed. As stated in [KJP15], a microservice should be **composable** from multiple small services. The first step of the approach is therefore to set up a test-environment for microservices which includes an appropriate test-application on which all further experiments can be executed on.

The fault generation step will then evaluate strategies for producing faults in the system. The main point of this is to apply workload onto the system and evaluate the effects of different bottlenecks that are induced within the application. For example, it will be researched what effect a CPU-over-utilization on the physical host has on the deployed microservices or whether microservices that are deployed on the same host can influence the behavior of other microservices in any way, and if they can, to what degree.

After this, the data has to be extracted from the microservice environment. Since the data comes from many different sources (monitoring tools, Kieker Framework, Workload Generator), a way has to be found to collect all data in a structured manner.

The collected data then serves as an input to failure prediction methods. In this step, it will be analyzed whether current failure prediction techniques are capable of predicting failures in a containerized system architecture. Any problematic scenarios will be looked at and possible solutions to them will be discussed.

Based on the failure prediction results, multiple different metrics will be created in order to compare and discuss about the results.

4.2. Basic Architecture

The hardware system on which the tests will be executed on comprises of four virtual machines that are managed by an OpenStack service and can be accessed via a proxy server. The VM's are hosted on a 64-Bit system with 4 2500mhz CPU's. The virtual machines are connected with each other and build the basis of a Kubernetes cluster on which the microservices are executed on.

Within the Kubernetes cluster, there are several replication controllers and deployments that are responsible for load balancing and scaling of the Pods that each contain a Docker image which represents a single microservice.

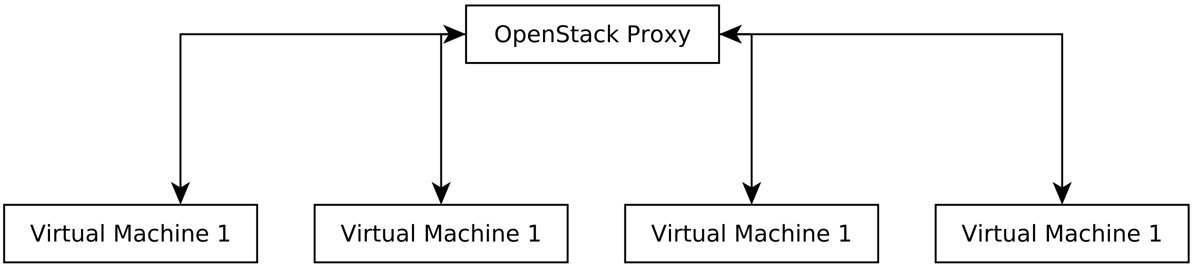


Figure 4.2.: Hardware Setup.

The microservices that are deployed on the Kubernetes Cluster are

- Monitoring containers (Heapster) that log the current state of the other containers. Heapster is the default monitoring tool for Kubernetes and logs data like the CPU-utilization of every running container into an instance of InfluxDB.
- Data storage containers (InfluxDB) that are used to gather and store all relevant information abut the state of the deployed services.
- The RSS-Reader application which consists of individual containers for the RSS-Edge service, The middletier service, The RSS-server and Cassandra. In order to obtain the response-times for the RSS-Reader service, each container is instrumented with a Kieker monitoring instance that writes all internally monitored

data to a separate Kieker-logging container by using ActiveMQ. Also, the fault injection tools that are used to manipulate the internal behavior of the containers are directly installed within each of these containers as well. The functionality of the RSS-Reader is described in more detail in section 4.3.1.

- An instance of ActiveMQ which is a Message Broker that uses JMS (Java Messaging Service).
- A Kieker Logging Server (KLS) that receives the logged data from the Kieker monitoring instances deployed on the RSS-Reader containers through ActiveMQ.
- Locust, which comprises of a Locust-Master container and multiple slave-containers. Locust is used to deploy workload onto the RSS-Reader.

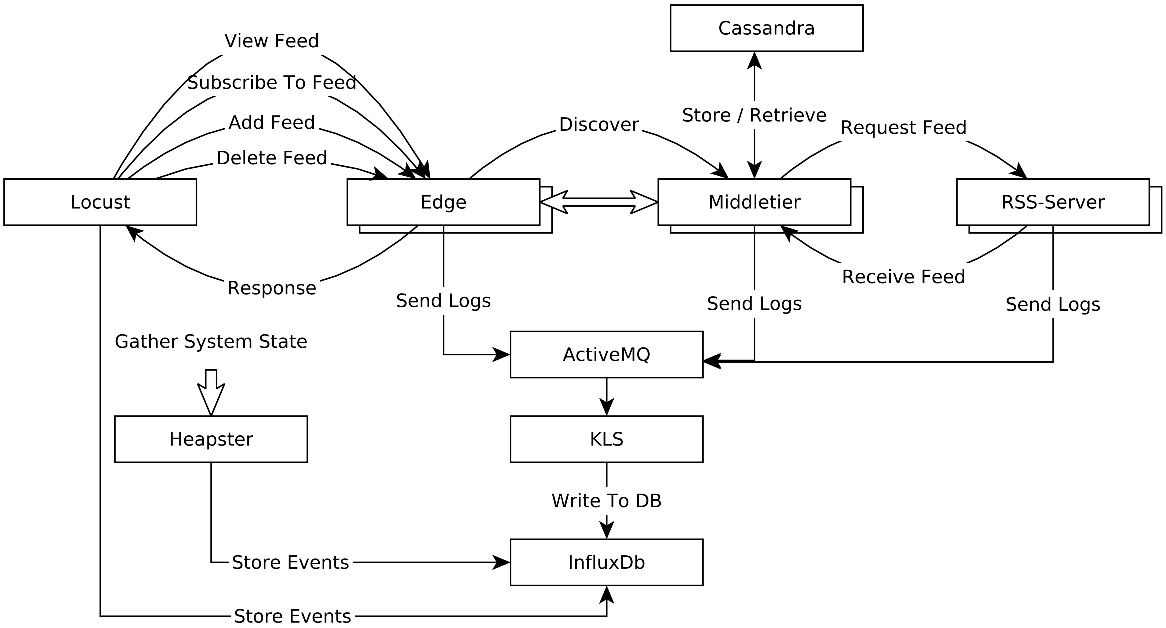


Figure 4.3.: Interaction between the different containers that are deployed on the Kubernetes cluster.

Figure 4.3 shows the main containers that are deployed on the cluster. First of all, the cluster consists of the RSS-Reader components which are designed such that they can be individually and at runtime scaled up and down by using a discovery mechanism. Next, The Locust containers are deployed which can be used to create workload on the system by simulating a certain amount of users that periodically send *GET*, and *POST* commands to the respective edge service. The Heapster container then logs the generated workload and writes it into InfluxDB where it will later be extracted from and used as evaluation data. As already mentioned, the *Kieker Logging Server* is responsible for writing the internal metrics of the containers like the response times of certain functions into InfluxDB.

On the basis of this system, several tests will be performed which will be discussed in more detail in the evaluation chapter. After each test run, the data is gathered and combined into a CSV-file on which the further evaluation is performed on. In the later phase of failure prediction, the gathered data is then replayed by using a sliding window that iterates over the data set. That way, it is ensured that the online failure prediction algorithms perform identical as if they were running on live data. By storing and replaying the data, it can be guaranteed that each run of a failure prediction algorithm is run on the exact same input data.

One problem that was encountered with the use of InfluxDB was, that it is currently impossible to join different measurements like the CPU and memory of a container into one dataset. Hence, in order to create a CSV that contains all measurements at once, a separate tool had to be developed within the thesis that is able to automatically join these measurements.

4.3. System Under Test

This section describes the application on which the failure predictors were executed on. The application is chosen on certain key factors. First, it is important, that it is easy to deploy the Application as a container and that it is possible to scale the individual components up and down. The application should also not be too complex to change parts of the program code if required. Also, the deployed containers should be highly coupled with each other and it should be possible to analyze various different failure scenarios as well. In order to emulate a system under stress, the application should have one or more bottleneck components that can easily be stressed to a high degree.

The *Netflix RSS-Reader* application which is presented in the following meets all of these requirements and was used as the test subject in this thesis.

4.3.1. Netflix RSS-Reader

The Netflix RSS-Reader application [Net] was developed to show how multiple Netflix components can interact with each other. The application implements a scalable 3-tier system for fetching and displaying RSS feeds. The components are written in Java and completely open source.

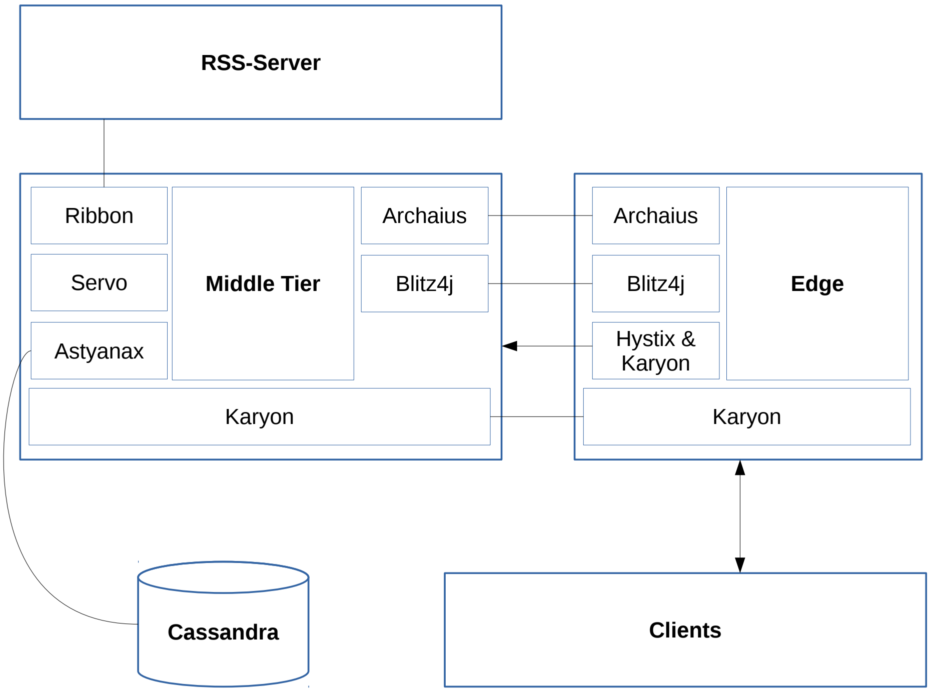


Figure 4.4.: Architecture of the RSS Recipes Application.

In order to deploy the application on microservices, a few actions had to be taken such as converting the components into microservices, which will be explained after in this chapter in more detail.

Overview

Figure 4.4 gives an overview of the applications architecture. As can be seen, the system is separated into mainly three parts: The edge server provides the UI of the service and accepts REST calls from any client. A client can subscribe to a new RSS server by providing the URL of the RSS feed to the edge server. The address is then transmitted to the middletier which stores the URL of the RSS-server in a Cassandra database. The middletier then subscribes to the RSS-feed by using the Netflix Ribbon component. On a new request from a client, the middletier collects all available feeds from the RSS-server and transmits them to the edge server where the data is transformed to HTML.

RSS-Server

The RSS-server is not part of the original Netflix Application. This component simply represents a RSS provider like one that can be found everywhere in the web. The reason why no external feed publisher was chosen is to have a consistent access pattern that is not distorted by any unknown factors like the workload or delay of the external server or the differing amount of feeds and feed size produced by it. The RSS-server consists of only a constant amount of feeds that will never be changed throughout the regular simulations that are performed in the evaluation chapter.

Apache Cassandra

Cassandra is an Open-Source Database that is developed to handle a high number of requests and operations performed on the data. It is a NoSQL system that is capable of storing all kinds of data. The data-layout is designed to be column-oriented and of course the database provides indexing capabilities like Primary Keys to efficiently handle the stored data. Aside from this, Cassandra offers many techniques that aim to optimize the stability and performance of the system like a replication mechanism without any bottleneck or single-point-of-failure components.

In the context of the RSS-Reader, Cassandra is used to store data like the URL's of the requested feeds and which users have subscribed to them.

RSS-Middletier

The middletier is responsible for gathering the feeds from the available RSS-servers and distributing them to the edge instances. The used implementation is however based on polling which means that the edge instances have to actively request a feed update in order to receive any data. Moreover, the middletier component does not cache any feeds whatsoever, which means that for every request that is queried by one of the edge instances, the middletier first loads the data from the RSS-server, then processes it and eventually answers the request with a set of feeds for one particular user. Obviously, this implementation is not very efficient but for the purpose of creating workload on a system it is sufficient.

RSS-Edge

The edge services represent the endpoint of the RSS-reader to which users can connect to in order to acquire their feeds. The edge provides a REST-interface that is capable of allowing users to subscribe and unsubscribe to feeds. Again, the feeds are only received by polling, which means that the user has to send view requests every time he wants to get updated. Also, every view-request results in the edge returning every subscribed feed, regardless of whether it has been modified or sent before already.

4.4. Failure Prediction Approach

As already mentioned, the failure prediction is based on the extracted data that has been obtained from several different scenarios and stored in a file. The CSV file contains different combined measures that were gathered and joined from InfluxDB. A sample may look like the set from Table 4.1 which in this case consists of the recorded measurements for the CPU-usage of a particular container, the CPU-usage of the host system, the number of simulated users that generate workload by frequently sending requests, the number of successful requests and the number of unsuccessful requests. Also some other metrics like the response-time in milliseconds of a particular method of one or multiple containers is contained in the dataset.

This data can then be directly used for failure prediction since all measurements were taken once every minute and therefore the data-points are equally spaced. Once this file is extracted, the next step is to create a sliding window that iterates over the time-series in order to simulate a live execution of the system. Algorithm 4.1 shows the function $SlidingWindow(X, size, length, slide, f)$ that creates a sliding-window over a dataset X that starts at the index $start$, holds at most $size$ data-points in the window and slides the window by $slide$ data-points. Each time the window is iterated, the function $f(X_t)$ is called which performs an action on the current window X_t. The function f is, for instance, the implementation of a failure prediction algorithm like $ARIMA(d, p, q)$.

Algorithmus 4.1 Sliding-Window Iteration

```
function SLIDINGWINDOW(X : Set, start, size, slide : Number, f : Function)
    $R \leftarrow \emptyset$
    $i \leftarrow start$
    while $i \leq |X|$ do
        $X_t \leftarrow X[\max(i - size, 0) \ldots i]$
        $R \leftarrow R \cup f(X_t)$
        $i \leftarrow i + slide$
    end while
end function
```

Minutes	cpu	cpuNode	users	successes	fails	pView	pView
...	...	...	...	...	...	...	...
24	0.419	0.245	250	11876	3	40.011	39.171
25	0.48	0.247	250	13608	11	38.469	34.23
26	0.52	0.264	250	15593	11	39.984	38.497
27	0.513	0.277	300	17431	155	47.213	43.622
28	0.619	0.279	300	19566	161	48.478	46.412
29	0.647	0.329	300	21935	161	47.364	47.04
30	0.667	0.328	350	24303	169	70.199	64.747
31	0.723	0.347	350	26839	199	68.195	67.276
32	0.752	0.356	350	29599	201	83.675	80.584
33	0.747	0.38	400	32372	201	79.451	79.268
34	0.81	0.377	400	35280	206	93.823	92.129
35	0.865	0.399	400	38446	216	111.275	108.527
36	0.864	0.424	500	41583	234	115.902	109.128
37	0.949	0.42	500	44928	328	169.266	169.185
38	0.967	0.458	500	48615	544	201.121	199
39	0.986	0.469	600	52217	837	194.294	191.301
40	0.991	0.475	600	55843	1125	203.475	202.649
41	0.989	0.479	600	59435	1499	212.767	211.891

Table 4.1.: Sample CSV-File (with a Reduced number of Columns).

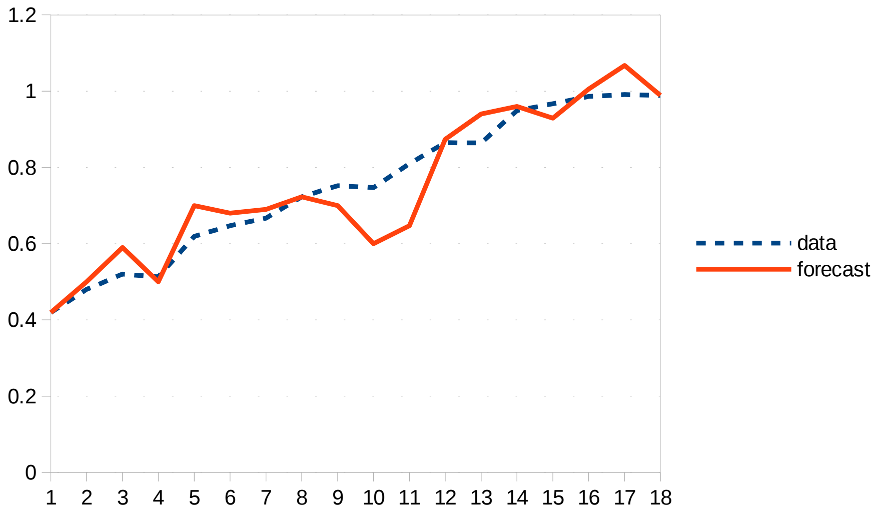


Figure 4.5.: Sample CPU forecast.

The forecast of this data then may look like the graph in Figure 4.5. The graph shows at each point in time t, the predicted value for time $t + 5$.

Now there are several ways to evaluate the *quality* of the prediction. It would, for example, be possible to quantify the prediction quality by taking a similarity measure like the *T-Test* or a similar technique. Another important aspect is to prove or disprove a correlation with the actual number of failure in the application. By taking the number of actual failures as a point of reference, it can directly be shown, that the respective metric correlates with the number of failures in the system.

Since failures only start to arise when the CPU-usage is at a higher level, we can quantify the failure correlation by defining a new metric (see for example equation 6.1) that considers this threshold on top of the predicted trend-line. Experimental results for this strategy are discussed in chapter 6.

Chapter 5

Implementation

This chapter describes the implementation and the setup of the approach. It describes each component in detail and discusses about the implementation of the overall system to give the reader insight to the capabilities and possible shortcomings of this particular setup. This chapter is supposed to provide sufficient information to reproduce most of the system such that the evaluation results can easily be proven and understood in more depth.

Overall, this chapter discusses about the configuration and setup of Kubernetes, the creation of the containerized services, the setup of the Netflix RSS-Reader application, the embedding of the fault injection tools into the system and the configuration and implementation of the used fault injector.

5.1. Kubernetes Setup

The Kubernetes cluster is installed on 4 VM's that are managed by OpenStack. The Kubernetes server and client version that was used is 1.5.2. The machines have 4 CPU's and 8 GB RAM. The OS installed on the nodes is Fedora Cloud 25 and the architecture is AMD64 (Kernel Version 4.8).

5.2. Netflix RSS-Reader Setup

The RSS-Reader first had to be transformed into a microservice application. For this, we containerized the middletier and edge applications by creating a Kubernetes deployment File. The deployment file for the Edge-container can be seen in the Appendix A.

5.3. Fault Injection Tools

For this thesis, fault injection is used in order to generate workload onto a certain aspect of the system. In our case, there are two classes of fault injection methods that that each affect the system in a different way.

1. *Workload Generators.* A fault injector that applies stress onto the system like *Locust* does by simulating a number of users which each query multiple requests over respective REST-calls can be used to increase the CPU, memory or network of the bottleneck components to a very high level such that eventually the components are unable to process any further requests in an adequate time.
2. *Internal Fault Injection.* Fault injection tools can be used to manipulate the behavior of a specific part of a targeted component. Such a tool can be used to simulate several different internal failures that are usually not part of the application itself. For research purposes it can be convenient to use a fault injector which is able to dynamically activate and deactivate a specific failure such that the outcome can be analyzed without having to restart or recompile the application.

As already mentioned previously, this thesis mostly discusses about failures that arise due to a high CPU- or memory utilization and does not focus on all types of failures that can occur in the created system.

5.3.1. Locust

Locust can be configured by creating a python file in which it is possible to exactly specify which requests are generated, how much delay there is between each request and a few more options. In our case, we specify functions that each request a subscription or a unsubscription from a feed that is provided by our RSS-server. Overall, there are 8 feeds stored on the server that each are sent to the client as soon as the subscription command has been issued. Listing 5.1 shows the code that creates the REST-Call to view the the ABC-News feed stored on the server.

```
@task(1)
def addAbc(self):
   with self.client.post("/jsp/rss.jsp", {"url":"http://rssserver/abc.xml",
       "username":self.user_id}, catch_response=True) as response:
      self.log_response(response)
```

Listing 5.1: REST-call generation by Locust

The results of this simulation is stored into InfluxDB by using the python Library that InfluxDB provides. Every minute, the respective variables are stored into the database (*'test_stats'*) by calling the function shown in Listing 5.2.

```
def writeToInfluxDB():
   json_body = [
      {
         "measurement": "test_stats",
         "tags": {
            "hostname": hostname
            },
         "fields": {
            "status_200_count": status_200_count,
            "status_500_count": status_500_count,
            "status_0_count": status_0_count,
            "status_other_count": status_other_count,
            "hostname": hostname
         }
      }
      ]
   InfluxDBWriter.write(json_body)
```

Listing 5.2: REST-call generation by Locust

The function $writeToInfluxDB()$ simply creates a JSON that contains all the status-information variables and stores them into the database by passing the file to the $InfluxDBWriter$ class which in turn passes the file to the InfluxDB-Library.

After Locust has been deployed on Kubernetes, the actual workload can then be adjusted via several different REST-calls.

Locust logs the number of different HTTP-status codes to determine the current successful and unsuccessful requests. As could be seen in listing 5.2, the status codes that are produced by the deployed RSS-Reader application are:

1. Status *200*, *OK*. Indicates that the request was successful.
2. Status *500*, *Internal Server Error*. This indicates an unknown server error which may be caused for example by a system over-utilization.
3. Status *0*, *Service Unreachable*. Server cannot be reached.

In the following, a *failure* in this context is defined as a request that resulted in a status code that was unequal to 200.

5.3.2. Byteman

Byteman is used to inject foreign code into the application by manipulating the underlying bytecode of the respective module. Listing 5.3 shows the simple commands that are needed to inject a foreign function ($allocateMemory()$) into the beginning of another function ($subscribe()$).

```
RULE allocateMemory
CLASS javax.ws.rs.core.Response
    com.netflix.recipes.rss.jersey.resources.MiddleTierResource
METHOD subscribe
AT ENTRY
IF true
DO org.k8sfp.bytemanfi.common.CommonFiUtils.allocateMemory()
ENDRULE
```

Listing 5.3: *memleak.btm*. Byteman rule for injecting a foreign function.

This rule can then be injected into any Java package by including

```
java -javaagent:script:/memLeak.btm
```

when executing the jar-file. Optionally, it is also possible to inject the failure at runtime by using the following option.

```
java -javaagent:byteman.jar=listener:true
```

The foreign function can then be defined in a separate library (here *BytemanFIUtils.jar*) which can be made available by including *boot:/BytemanFIUtils.jar* into the command line string.

5.4. Data Extraction

The data extraction and failure prediction part was designed such that the workflow can be automated easily which might be beneficial when the online failure prediction method would be used as a part of an application.

Figure 5.1 shows the projects and their dependencies between each other. The Project *K8sfpServiceBridge* contains the interfaces that are used for inter-project communication. As Figure 5.2 shows, the interfaces are mostly different representations of data items or representations of a failure predictor instance. The Interface *IK8sDataElement* represents any data-point and is therefore just an empty interface with no special requirements. The interface *IK8sTimeseriesDataElement* is a representation of a distinct time-series

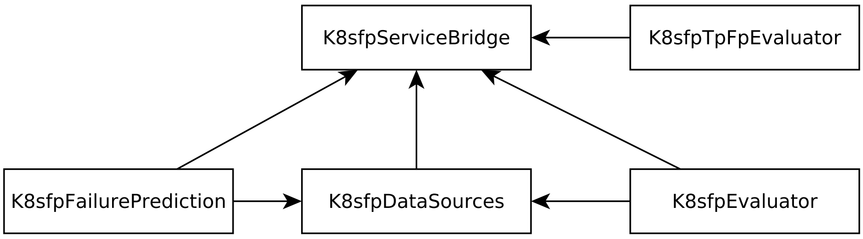


Figure 5.1.: Dependencies of the Maven-Projects.

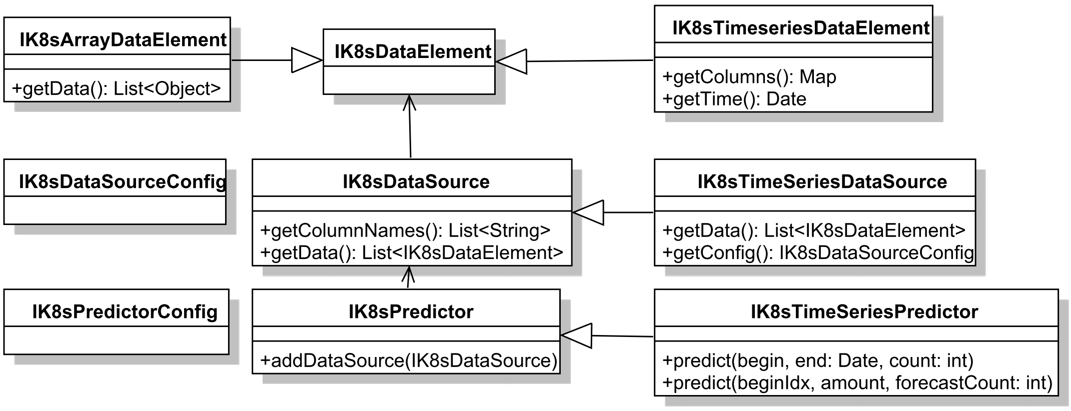

IK8sArrayDataElement
+getData(): List<Object>
IK8sDataElement
IK8sTimeseriesDataElement
+getColumns(): Map
+getTime(): Date
IK8sDataSourceConfig
IK8sDataSource
+getColumnNames(): List<String>
+getData(): List<IK8sDataElement>
IK8sTimeSeriesDataSource
+getData(): List<IK8sDataElement>
+getConfig(): IK8sDataSourceConfig
IK8sPredictorConfig
IK8sPredictor
+addDataSource(IK8sDataSource)
IK8sTimeSeriesPredictor
+predict(begin, end: Date, count: int)
+predict(beginIdx, amount, forecastCount: int)

Figure 5.2.: Interfaces between the projects.

data-point that each stores a Dictionary of Name-Value Pairs and a timestamp that indicates when the data point was created. The *IK8sArrayDataElement* interface is another representation that only requires the storage of an object array and is therefore more flexible.

The first step in the workflow is to extract the logged data from InfluxDb. For Java, there is a special library provided by InfluxDb that can be used for the communication with the database. However, since there is no option of joining two or more measurements into a single data-point, this part was implemented within the project as well.

Figures 5.3 and 5.4 show the most important classes of the project *K8sfpDataSources*. The class *DbFetcher* is responsible for gathering and query-joining the requested measurements which were defined within the *InfluxDbDataSourceConfig* class. Overall, there are three join-Operations needed:

1. *Inter-Query Join*.

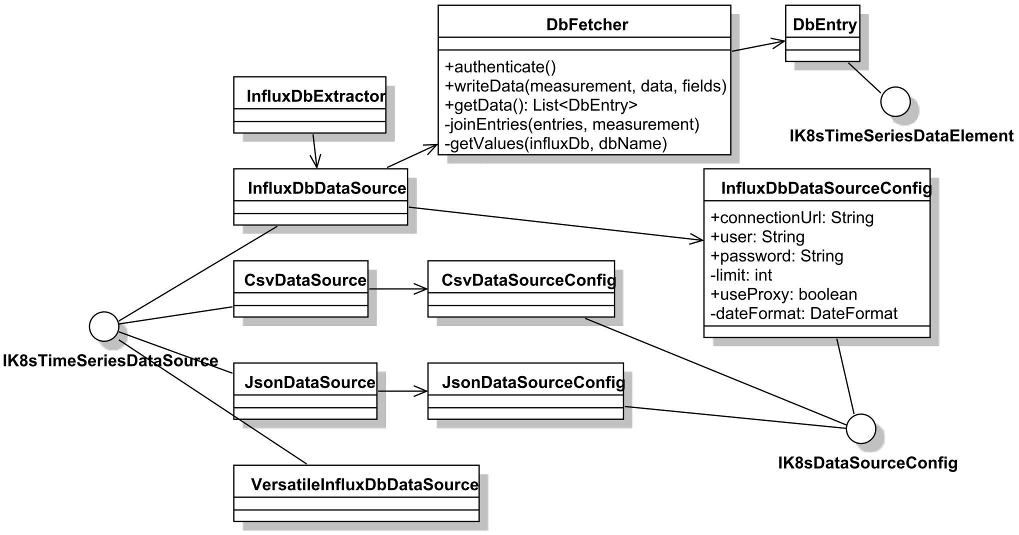


Figure 5.3.: Main classes of the Project *K8sfpDataSources*.

2. *Measurement Join*.
3. *Database Join*.

The **Inter-Query Join** is needed to join the data that result from Instructions like the *GROUP-BY* clause, which returns multiple datasets for each group. A simple example would look like this:

```
SELECT value as network_edge FROM \"network/rx_rate\" WHERE pod_name =~ /edge/ GROUP BY
    pod_name ORDER BY DESC
```

This instruction would return a dataset for each distinct service-name pattern that was matched by the regular expression .**edge.**. When there are for example two replicated services that would contain the name ‘edge‘, the instruction would return a results for both of these services.

A response in InfluxDb is constructed in a similar way - a sequential collection in which each item contains another collection that contains the relevant measurement results. The task is now to transform this into a dictionary that maps the measurement name to a list of values which are the actual results of InfluxDb.

Algorithm 5.1 shows the first join procedure of the response. After this step, the set R contains pairs of distinct measurement-names and their respective values. Wherever there was a duplicate measurement that resulted from a *GROUP BY* Statement, the measurement name is now simply extended by a sequentially increasing numerical postfix.

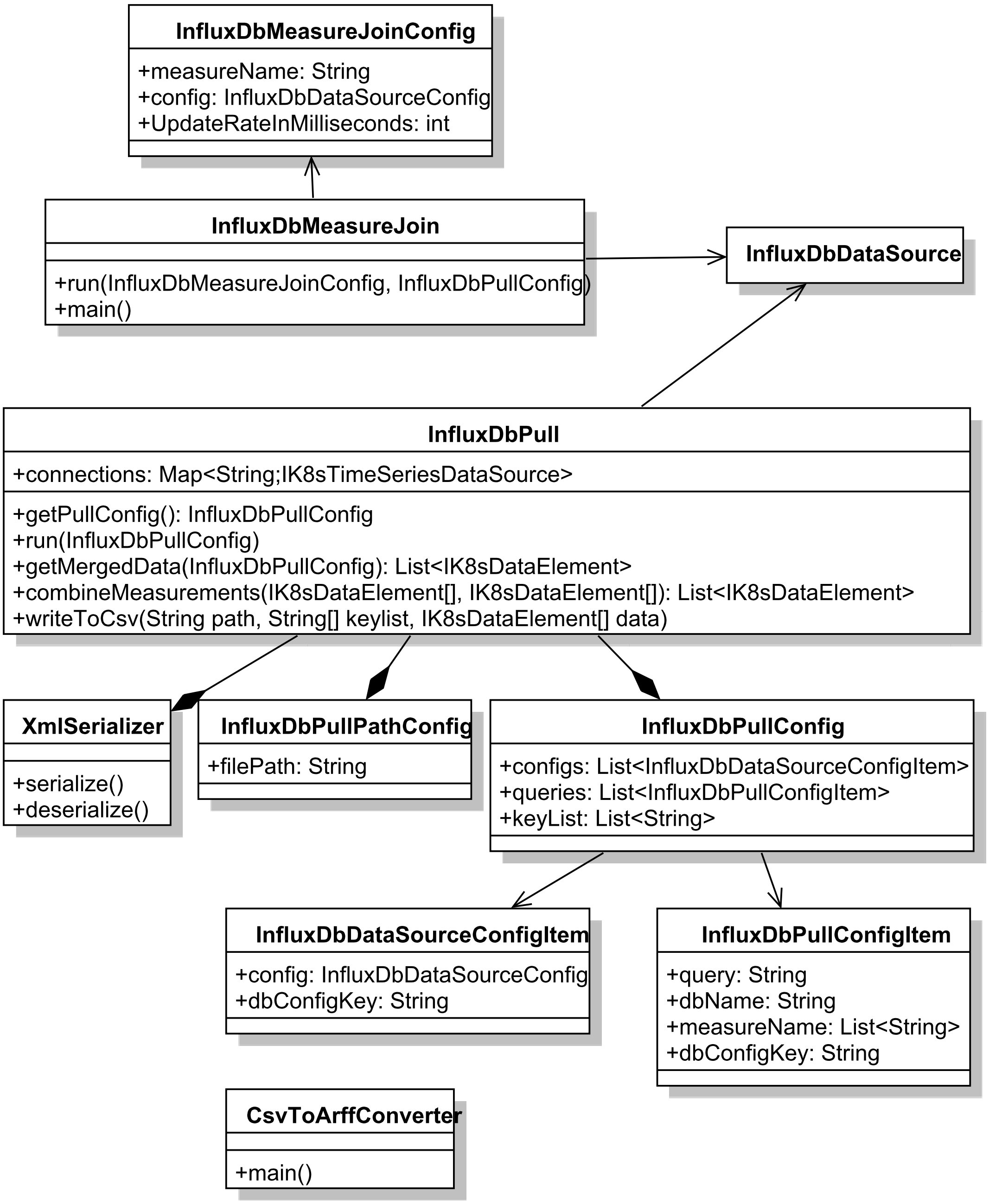

InfluxDbMeasureJoinConfig
+measureName: String
+config: InfluxDbDataSourceConfig
+UpdateRateInMilliseconds: int
InfluxDbMeasureJoin
+run(InfluxDbMeasureJoinConfig, InfluxDbPullConfig)
+main()
InfluxDbDataSource
InfluxDbPull
+connections: Map<String;IK8sTimeSeriesDataSource>
+getPullConfig(): InfluxDbPullConfig
+run(InfluxDbPullConfig)
+getMergedData(InfluxDbPullConfig): List<IK8sDataElement>
+combineMeasurements(IK8sDataElement[], IK8sDataElement[]): List<IK8sDataElement>
+writeToCsv(String path, String[] keylist, IK8sDataElement[] data)
XmlSerializer
+serialize()
+deserialize()
InfluxDbPullPathConfig
+filePath: String
InfluxDbPullConfig
+configs: List<InfluxDbDataSourceConfigItem>
+queries: List<InfluxDbPullConfigItem>
+keyList: List<String>
InfluxDbDataSourceConfigItem
+config: InfluxDbDataSourceConfig
+dbConfigKey: String
InfluxDbPullConfigItem
+query: String
+dbName: String
+measureName: List<String>
+dbConfigKey: String
CsvToArffConverter
+main()

Figure 5.4.: Main classes of the Project *K8sfpEvaluator*.

Algorithmus 5.1 Inter-Query Join

```
function InterQueryJoin(X : DbResponse)
    $R \leftarrow \emptyset$
    for all $g \in X.Results$ do
        $c \leftarrow 0$
        for all $s \in g.Series$ do
            $I \leftarrow \emptyset$
            for all $v \in s.Values$ do
                $I \leftarrow I \cup (measureName + c, v)$
            end for
            $R \leftarrow R \cup (s.Name + c, I)$
            $c \leftarrow c + 1$
        end for
    end for
end function
```

After this, the **Measurement-Join** can be applied onto the result. Assuming we want to join different measurements like the network and the CPU of a specific service, it is currently only possible to query the data by issuing two different queries that will be answered separately by InfluxDb (e.g. R_{CPU} and $R_{network}$).

```
SELECT value as cpu_edge FROM \"cpu/usage_rate\" WHERE pod_name =~ /edge/ GROUP BY
    pod_name
SELECT value as network_edge FROM \"network/rx_rate\" WHERE pod_name =~ /edge/ GROUP BY
    pod_name
```

This means, that in order to compute a single time-series $R_{all} = \{(t, \{R_{CPU,t}, R_{network,t}, \ldots\})\}$ where for every point in time t there is a collection of measurements $R_{all,t}$ that holds the value of all defined measurements at time t, another join operation is needed.

The algorithm $MeasureJoin(R, join\_dist, join\_tol)$ (5.2) shows how this join was implemented. the variable $join\_dist$ is the timespan (in milliseconds) in which two different measurements are to be joined. The variable $join\_tol$ defines another timespan in which measurements are joined if so far no measurement has been found yet. The Function $merge(X, Y)$ simply combines the measurements by adding $X \leftarrow X \cup Y$ where X and Y both contain tuples of names and value-sets. Therefore, the merge-function adds all name-value-set tuples from Y to the tuple-set of X.

The **Database Join** is required when different measurements from different InfluxDb instances have to be combined into a single dataset. This had to be done for the logged data from the workload generator Locust since the other database was already heavily used by the monitoring tool.

Algorithmus 5.2 Measurement Join

```
function MeasurementJoin(R : Set, join_dist : Integer, join_tol : Integer)
    $M \leftarrow R_0$
    for all $N \in R \backslash R_0$ do
        for $i \leftarrow 0, i < |N_I|, i \leftarrow i + 1$ do
            $jt \leftarrow M_i[TIMESTAMP]$
            $before \leftarrow jt - join\_dist$
            for $j \leftarrow 0, j < |R|, j \leftarrow j + 1$ do
                $jf \leftarrow R_{j,I}[TIMESTAMP]$
                $diff \leftarrow jt - jf$
                if $jt > jf$and$jf > before$ then
                    $merge(jt, jf)$
                else if $diff < join\_tol$ then
                    $merge(jt, jf)$
                end if
            end for
            $R \leftarrow R \cup (s.Name + c, I)$
            $c \leftarrow c + 1$
        end for
    end for
end function
```

The database join can now easily be made on top of the current result-sets $R_{all,i}$ that is produced for every database (k-times). The final dataset can be created by computing $R_{joined} \leftarrow dbJoin(R_{all,0}, R_{all,i}) \forall i \in \mathbb{N} \backslash \{1\}$. The function $dbJoin()$ simply iterates over the time-series $R_{all,0}$ and $R_{all,i}$ and joins every item that contains a close enough timestamp.

Chapter 6

Evaluation

This chapter presents and discusses the results which were obtained by the previously described approach.

The implemented microservice architecture provides a wide variety of different interfaces on which statements can be made about the quality and value of different failure prediction strategies. One important research question the evaluation will discuss about is **how individual microservices and a particular failure prediction instance for this service are affected by other, in parallel running applications**. For instance, consider a system in which several hundred containers are running in parallel. Naturally, it would be very unpractical to only deploy as much containers on a node such that the summed up maximum CPU limit of all containers does not exceed the computing capacity of the respective node since it is unlikely that all containers reach their CPU limit at the same time. This is due to the fact that in most systems there are certain bottleneck components through which the performance of the system is limited (critical path). In the rare case in which the system does reach its limit, despite all the effort of load balancing and service distribution, it is crucial that failure predictors are not affected by this circumstance (i.e. they must still be able to predict a faulty application state in this scenario).

This evaluation will also quantify the *degree* to which a failure predictor can be influenced by other, separately running processes that are deployed on the same host and discuss about the metrics that can be influenced by this or similar side effects. For any shown side-effect, an alternative prediction strategy will be investigated that may be used in such scenarios and they will be evaluated for practical use.

6.1. Evaluation Plan

The strategy of evaluation subdivides into the following steps:

1. **Exploration** (Section 6.3). In this step, different simple scenarios will be presented and evaluated that show interesting attributes and unexpected behaviors.
2. **Validation** (Section 6.4). In this step, certain scenarios that were identified in the Exploring step will be validated, i.e., it will be shown that this scenario is repeatable and shows a similar behavior for equal simulations.
3. **Calibration** of failure predictors (Section 6.4). Here, the used failure predictors will be calibrated by using control datasets that were created during the previous steps.
4. **Prediction Evaluation** (Section 6.6). Next, the failure predictors are executed on different scenarios. The quality of the prediction results will be discussed for each individual validated scenario.

6.2. Prediction Measurements

For the analysis of the workload scenarios, we consider the following measurements:

- The **CPU-percentage** of the containers and the nodes. This measurement can easily be accessed by any monitoring component and it can easily be used as well since for every microservice, there is a hard CPU-limit. The CPU is often used as a metric that determines the time range of possible occurring failures. For instance, a failure predictor can simple predict the CPU of a component and increase the likelihood of failures as soon as the CPU hits a threshold limit of 98%. This threshold limit can be optimized by observation: Controlled scenarios where the CPU-Workload is increased until failures occur in a certain system can be used to find the CPU-limit at which failures are likely.
- The **Memory-Percentage** of the containers and the nodes. This measurement can be used in a similar way as the CPU-measurement. However, in contrast to the CPU, if the memory is increased to its limit (i.e. due to a memory leak), in most cases the application is sure to fail.
- The **Network-Usage** of the containers and the nodes. The network-usage is limited by the network capabilities of the hardware setup. Tests have shown that the upper limit at which the network-bandwidth is at full capacity in our scenario is at around 6 Gbps.

- The **Response-Times** of the most vital functions in some containers. This metric indicates how long a service needs to process a single request, however, since there is no hard response-time limit defined for most applications, an upper limit has to be set manually.
- The simulated **User Count**. In the context of the RSS-Reader application, the user count cannot be directly measured if the application would be used by actual users. It may be possible to count the number of currently subscribed users, but the number of users that currently access the system could deviate significantly from this value. Therefore, this measurement cannot be used as a metric for failure prediction. Instead, the value can be used to validate the results of the prediction results.
- The **Success- and Failure Rates** of the workload generator. Identically to the actual number of users, the success and failure rates cannot be determined by any monitoring component due to factors like network delay and response-time deviations. The response-time of a function might not be correlating to the actual response-time due to queuing or asynchronous propagation of requests. This metric will be used to determine the quality of the prediction results.

6.3. Exploration

This section evaluates the behavior of the deployed containers and the respective metrics (like CPU and memory usage) for a generated workload scenario. In section 6.3.1, a linear workload scenario is presented and prediction ideas are discussed. In section 6.3.2, the scenario is modified by placing a service on a host that has already deployed a large amount of containers. To show that the observed effect can also be produced in another part of the application, section 6.3.4 analyzes how the edge-service performs on a stressed host. Finally, section 6.3.5 shows, that the degree to which a stressed host can affect a deployed application can be very high and shows that failure predictors should be able to forecast these scenarios reliably.

6.3.1. Linear Workloads

This scenario analyzes the behavior of the deployed RSS-Reader containers for an increasing amount of users that want to access the service. The number of users are simulated with two locust worker nodes which are deployed in the cluster. The number of users is iteratively increased by the value listed in table 6.1

Time (Minutes)	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Number of users	3	5	10	20	50	100	150	200	250	300	350	400	500	600

Table 6.1.: Simulated users for a linearly increasing workload.

Overall, one execution takes about 42 minutes and the user count is at first sightly increased and increases further until it hits 600. Tests have shown that the CPU-utilization of the application is approximately 1 for every tested scenario when 600 users are simulated. This workload simulation is supposed to emulate a system that gets increasingly overloaded by user requests and therefore for failure prediction, it should be possible to find a metric from which the symptom of the failure can be determined.

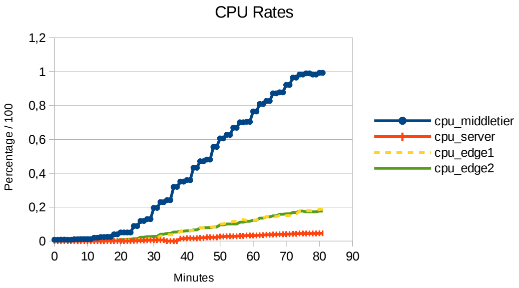


Figure 6.1.: CPU rates of all containers.

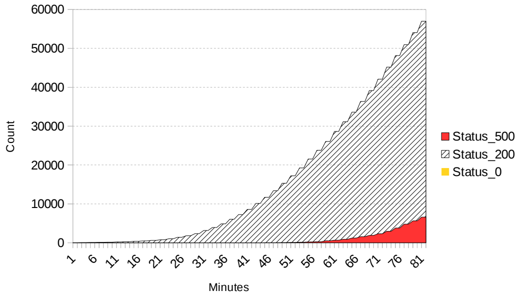


Figure 6.2.: Success (status 200)- and failure-rates (status 500, status 0).

Figure 6.1 shows the increase of CPU-utilization in each observed microservice. As can be seen, the increase is linear and therefore corresponds with the increase of users over time. This scenario was executed after deploying 2 edges and 1 middletier onto the Kubernetes cluster, which results in the middletier being the bottleneck component of the application (the CPU-utilization of every other component is significantly lower). Figure 6.2 shows the total number of successful and unsuccessful requests (fails). The number of fails increases as soon as the CPU-utilization gets high enough. In this simulation, the number of failures start to increase at minute 54 (5 failures, 66.9% CPU-utilization). At minute 68, the failure count has increased to over 100 failures (137 failures, 87.8% CPU) and at minute 76 there are over 1000 failures (1210 failures, 99.0% CPU). At the end of the simulation, there are 2293 failures and the CPU-usage is $99,3\%$.

If there would have been a failure predictor that was monitoring the application, it should have generated an alert just before the occurrence of the majority of the failures logged by the workload generator. Hence, for a time-series failure predictor to be effective at predicting this workload-induced failure, there has to exist a metric that

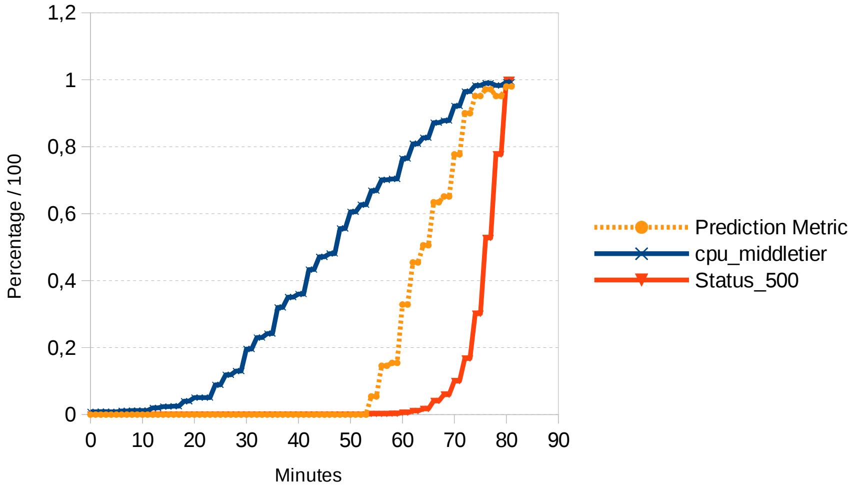


Figure 6.3.: Computed Metric for a specific simulation.

directly correlates to this failure rate which is, as stated earlier, unaccessible in a real scenario. Looking alone at this scenario, it would be very simple to create that metric.

$$P_m(X, \tau_{P_m}) = \bigcup_{i=1}^{|X|} \frac{\max(0, (X_i - \tau))}{1 - \tau}, 0 \leq \tau_{P_m} \leq 1 \tag{6.1}$$

Equation 6.1 shows a simple function $P_m(X, \tau_{P_m})$ which computes such a metric based on the input set X which represents the respective time-series. By setting the threshold-value $\tau_{P_m} = 0.65$, The metric in Figure 6.3 can be created only from the CPU time-series. Of course, in a different scenario, this particular threshold value might not be the best choice, hence it would require more training data to actually calibrate this function.

Assuming that it is possible to create a metric that correlates to the number of failures in every scenario, it is also possible to create a failure predictor that can predict this number by calculating and forecasting this particular metric.

Figure 6.4 shows the results of an *ARIMA* forecast that was applied to the dataset. At any time t, the forecast calculates the future CPU-usage-trend for time $t + t_{th}$ where in this case $t_{th} = 5$. As can be seen in the graph, at exactly $t = 51$ the function $P_m(X_{ARIMA}, 0.65)$ starts to increase (actual value: 6.5%) whereas the actual prediction metric increases at $t = 56$ to 5.4%.

Prediction metrics like the function P_m can now be used to create a binary failure prediction function by calculating $\hat{P}_m(X) = \varphi(P_m, X, \tau_{P_m}, \sigma_{P_m})$ where τ_{P_m} represents the threshold value of the metric up to which the metric does not correlate to the actual

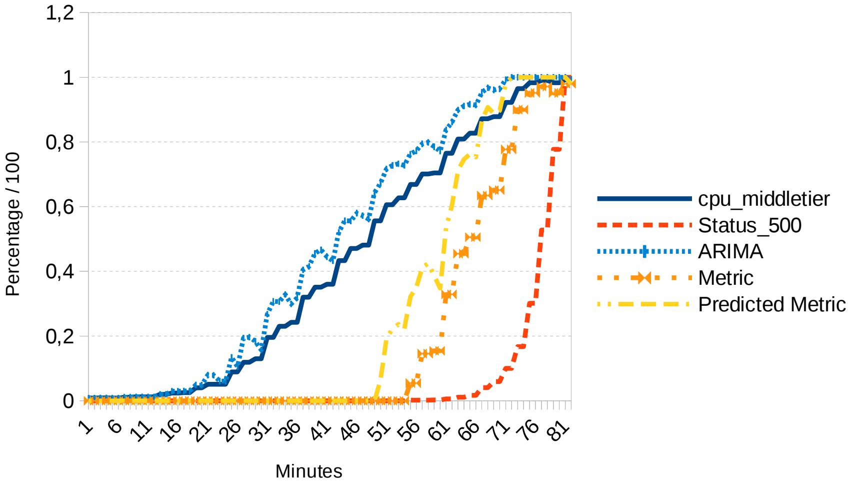


Figure 6.4.: Prediction Results and Predicted Metric of the workload scenario. The prediction forecasts $t_{th} = 5$ minutes into the future. The prediction is not shifted, i.e. the forecast at point t should hold for the time $t + t_{th}$.

failures and σ_{P_m} is the threshold value that determines when P is significant enough to be considered a failure.

$$\varphi(P, X, \tau, \sigma) = \begin{cases} 0 & \text{for } 0 \leq P(X, \tau) < \sigma \\ 1 & \text{for } \sigma \leq P(X, \tau) \leq 1 \end{cases}, 0 \leq \tau, \sigma \leq 1 \tag{6.2}$$

Figure 6.5 shows the predicted binary metric $\varphi(P_m, X, \tau, \sigma)$ for $\tau = 0.65$ and $\sigma = 0.01$. Of course, since the prediction function is optimized to this special scenario, the binary prediction result is perfect in the sense that it would predict a failure exactly t_{th} minutes before it arises.

To evaluate this prediction result, the first step is to calculate the actual failure percentage from the actual failure-rate metric which the failure predictors should be predicting. Equation 6.4 shows how this can be done. The threshold variable τ is set to the constant $\tau = 0.01\% = 0.0001$. X represents in this case the set of successful and unsuccessful requests (X_{eval}).

$$P_{eval}(X, \tau_{P_{eval}}) = \frac{|X_{fails}|}{|X_{fails}| + |X_{successes}|} \tag{6.3}$$

$$\hat{P}_{eval}(X, \tau_{P_{eval}}) = \varphi(P_{eval}, \tau_{P_{eval}}, \tau_{P_{eval}}) \tag{6.4}$$

Figure 6.6 shows the functions P_{eval} (blue line) and $\hat{P}_{eval}$ for this particular scenario.

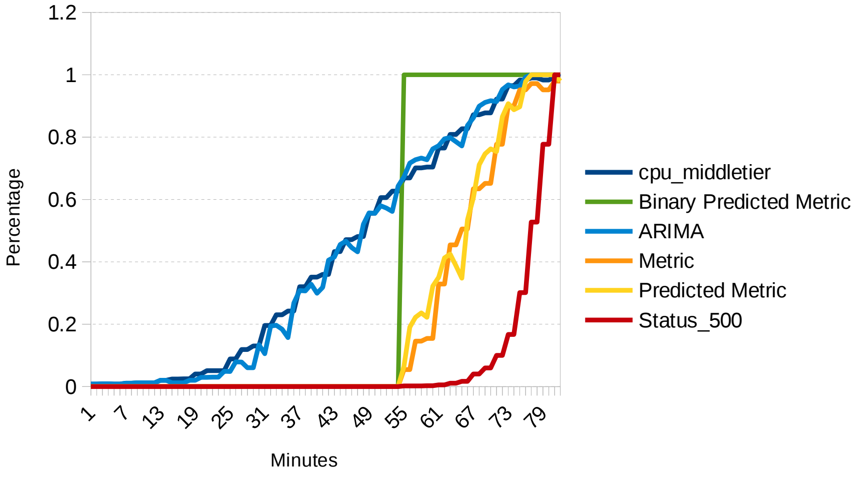


Figure 6.5.: Shifted Prediction Results and the simple binary prediction Metric.

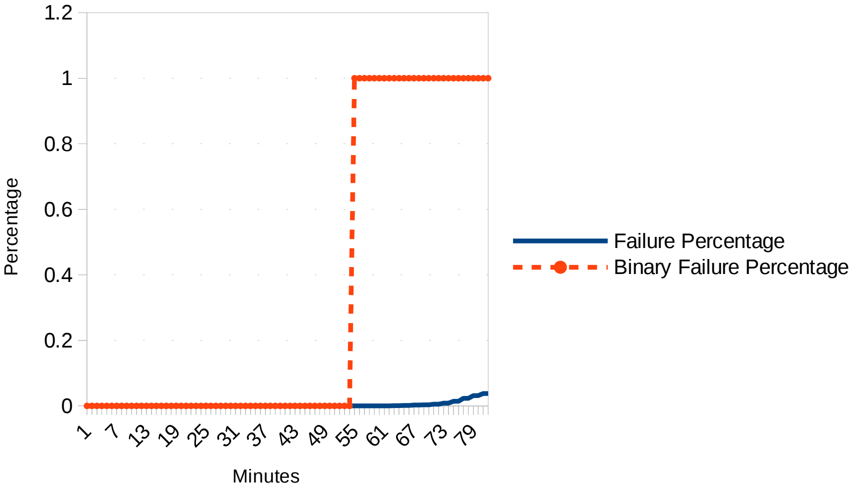


Figure 6.6.: Plotted P_{eval} (blue line) and $\hat{P}_{eval}$ (red line).

For this scenario, $\hat{P}_{eval}(X_{eval}, 0.0001) = \hat{P}_m(X)$ holds and therefore the failure prediction method is able to make a perfect prediction. Also, $TP_{\hat{P}_{eval}} = |fails|$ and $FP_{\hat{P}_{eval}} = 0$ holds and therefore the area under the ROC curve is 1.

6.3.2. Middletier CPU-Workload

In contrast to the scenario above, the middletier component is now deployed on a host that already has a high number of containers that consume a lot of CPU-Resources. For

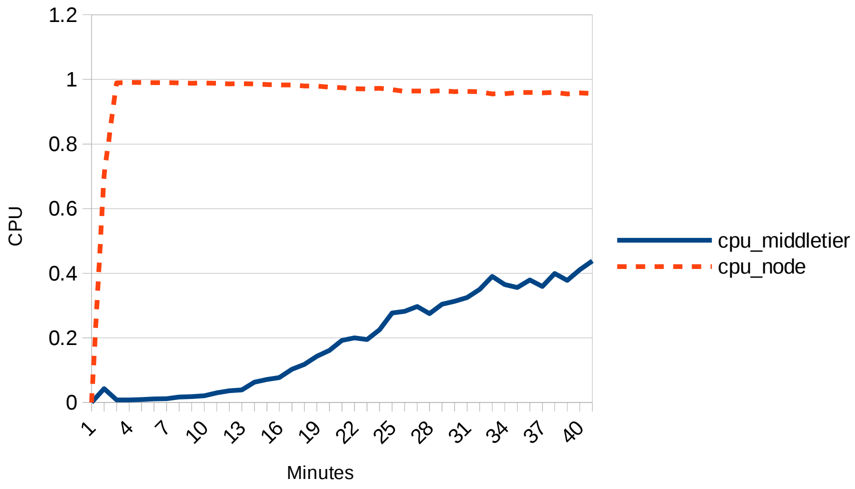


Figure 6.7.: CPU-usage of the middletier-container and the CPU of the host.

this, stress containers are deployed that each run the Linux command *stress* with the parameter *–cpu 100*. Overall, this brings the CPU-utilization of the host to about 95%.

Parameter	Description
Microservice Setup	2 Edges, 1 Middletier, 1 RSS-server, Middletier as bottleneck
Container Configuration	Middletier separated on Node C
Node stress	> 98% CPU workload on Node C
Container stress	None

Table 6.2.: Execution Parameters.

Figure 6.7 shows, that the CPU of the middletier does not increase above a certain threshold (in this case about 0.4) despite that the CPU-limit of the container is set to 1. This scenario will be discussed in more detail in section 6.4.3. While in this scenario, the deployment of the component was intentionally done on the CPU-stressed node, the same effect could in practice be achieved by multiple containers on a host that experience a sudden increase of workload that might be induced by users that want to access the service all at once or by an intentional attack of the system (e.g. *Denial Of Service* attack). In such a case, failure predictors are therefore still required to work and must understand the situation despite of the fact that the CPU-Resources of the service is seemingly not overloaded.

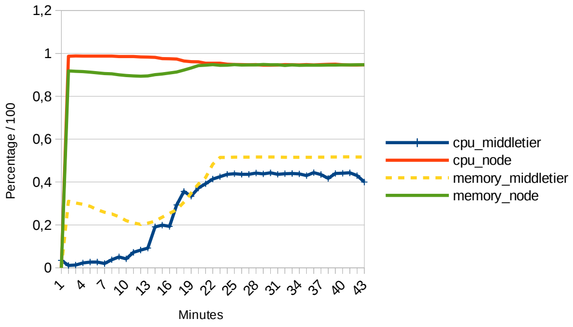


Figure 6.8.: CPU-utilization of the service and the node.

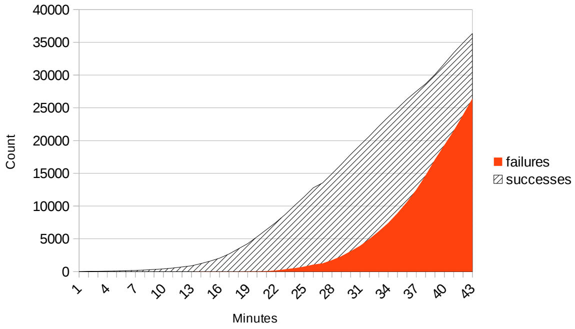


Figure 6.9.: Success- and failure rates.

6.3.3. Middletier Memory-Workload

In this scenario, the effects of memory shortage will be evaluated. Table 6.3 shows the parameters for the simulation. The middletier, which is the bottleneck component in this case, was separated on a physical host where 10 memory-stress-containers were executed on in parallel. The execution of the memory-stress containers also induced a rather high CPU-utilization on the respective host (>90%).

Parameter	Description
Microservice Setup	2 Edges, 1 Middletier, 1 RSS-server, Middletier as bottleneck
Container Configuration	Middletier separated on Node D
Node stress	None
Container stress	> 90% CPU and memory workload on Node D by using only microservices

Table 6.3.: Execution Parameters.

Figure 6.8 shows the CPU- and memory-utilization of the middletier and the respective physical host. Figure 6.9 shows the number of successful and unsuccessful requests over the execution time period. As can be seen, the CPU and memory measures are not increasing above 60% but failures still arise.

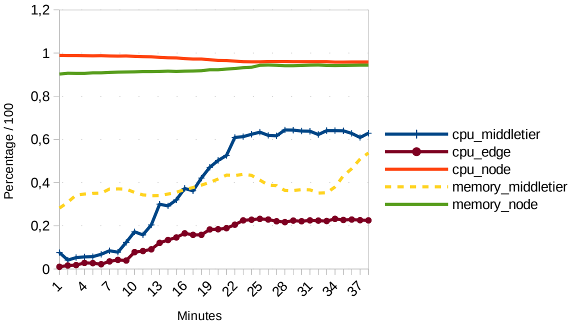


Figure 6.10.: CPU- and memory-usage of the containers and the host system.

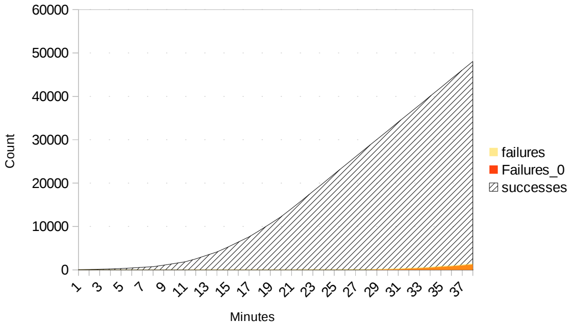


Figure 6.11.: Failure count recorded by the workload generator.

6.3.4. Service CPU-Workload

The stress can also be put onto another part of the application. In this scenario, the edge-container is separated and stressed with multiple CPU-stress services. Table 6.4 shows the parameters for the execution.

Parameter	Description
Microservice Setup	1 Edge, 1 Middletier, 1 RSS-server, Edge as bottleneck
Container Configuration	Edge separated on Node D
Node stress	None
Container stress	> 90% CPU on Node D by using only microservices

Table 6.4.: Execution Parameters.

Figure 6.10 and 6.11 show the CPU-utilization and Failure-Rate for the scenario. As can be seen, very little failures could be produced. Failures that were reported in this simulation were mostly caused by the service being unavailable.

6.3.5. Maximum Workload Complications

This scenario shows to what degree an overloaded system can affect the application. The system was overloaded by creating multiple stress-containers and every component was placed onto the same host.

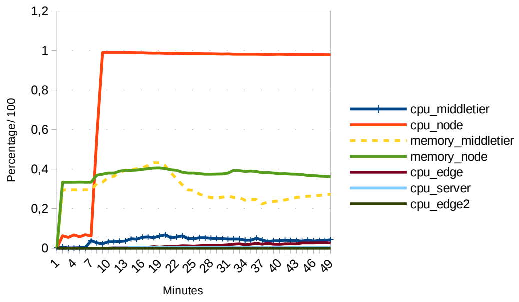


Figure 6.12.: CPU-utilization.

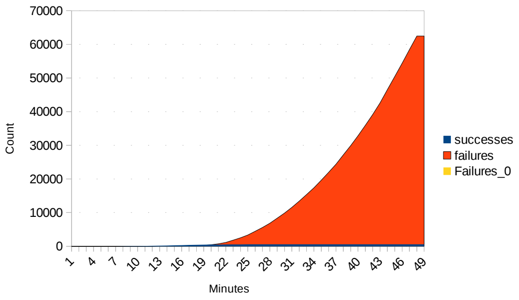


Figure 6.13.: Success- and failure rates.

Parameter	Description
Microservice Setup	2 Edges, 1 Middletier, 1 RSS-server, Edge as bottleneck
Container Configuration	All on Node D
Node stress	None
Container stress	> 90% CPU on Node D by using 15 cpu-stress microservices

Figures 6.12 and 6.13 show the CPU-utilization as well as the success- and failure rates of the application. As can be seen, the failure count can be increased to such a high level that it exceeds the number of successful requests.

6.4. Workload Analysis

The previous section showed different scenarios for the execution of microservices. In this section, these scenarios are further analyzed, discussed and evaluated.

6.4.1. Linear Workload Validation

The above discussed scenario will now be used to calibrate the failure prediction method. For this, there will be multiple executions of the linear workload scenario with slightly differing simulated users. Algorithm 6.1 is used in order to create multiple linear workload profiles that are then being sent to the workload generator. The parameter $count$ determines the amount of workload changes whereas the variable max limits the maximum simulated user count. The illustrated algorithm increases the amount of simulated users over time by a random amount. As can be seen, the amount of users can also slightly decrease and is therefore not entirely monotonically increasing.

Algorithmus 6.1 Workload Increase Algorithm

```
function CreateLinearProfile(count, max)
    $I \leftarrow \{0\}$
    $m, i \leftarrow 0$
    while $i < count$ do
        $m \leftarrow m + random\ from 1\ to\ \frac{2max}{count}$
        $I \leftarrow I \cup m, i \leftarrow i + 1$
    end while
    if $m > max$ then
        $i \leftarrow 0$
        while $i < count$ do
            $I_i \leftarrow max(0, I_i - \frac{(m-max)(i+1)}{count+1})$
            $i \leftarrow i + 1$
        end while
    end if
    if $I_i < max$ then
        $I \leftarrow CreateLinearProfile(count, max)$
    end if
    return $I$
end function
```

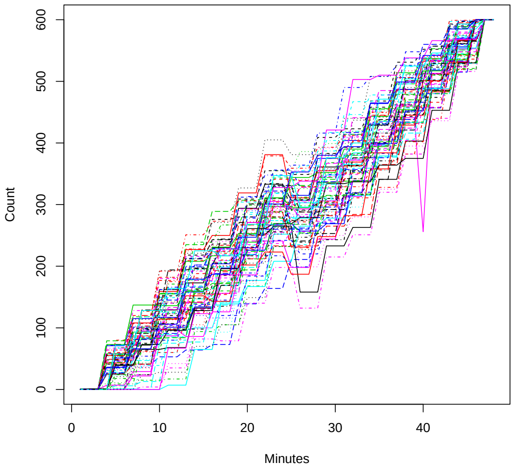


Figure 6.14.: Simulated user Count for all Scenarios.

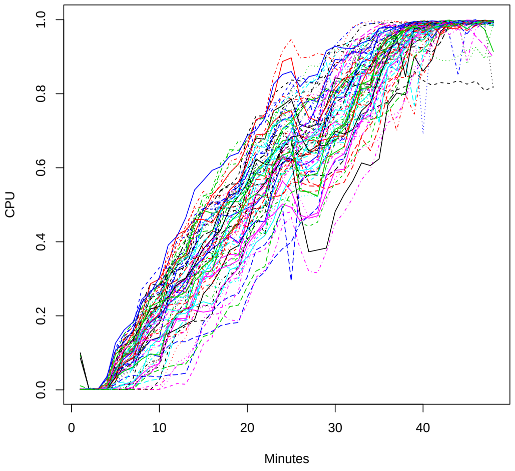


Figure 6.15.: CPU utilization for all Scenarios.

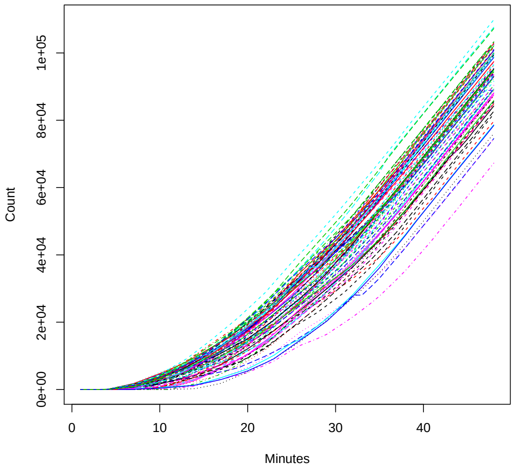


Figure 6.16.: Overall successful user requests.

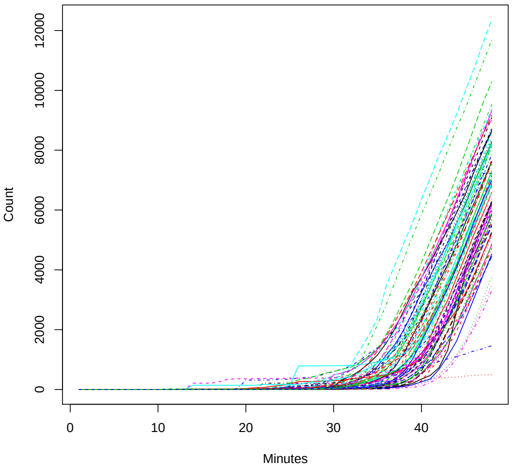


Figure 6.17.: Overall unsuccessful user requests.

Figure 6.14 shows the simulated user count for every execution. The user workload in this execution is based on the function $CreateLinearProfile(15, 600)$, i.e. the maximum user count is 600 and the user count changes 15 times until it reaches 600 users. Each user workload is simulated for 3 minutes so the execution of one single scenario takes about 45 minutes overall. Figure 6.15 shows the plotted CPU-usage for every Execution. As can be seen, the CPU-usage in most cases reaches its limit at around 500 simulated users and becomes 50% for a user count of about 300.

In Figure 6.16, the summed up successful requests are shown for each execution. In most cases, the successful requests are constantly increasing and correlate with the number of simulated users. Figure 6.17 shows the overall unsuccessful requests which begin to occur in some executions already at minute 13 and increase to about 6000 at the end of the execution.

Figure 6.18 shows the function $P_{eval}(X, 0.0001)$ applied to every execution and Figure 6.19 shows the averaged value $P_{eval,avg}(x) = \frac{\sum_i^n P_{eval,i}(x)}{n}$ along with the average predicted failure metric $P_{m,avg}(X_{ARIMA}, \tau)$ obtained in the same way. The parameter τ is obtained from Figure 6.20 which shows the TP/FP ratio for several different forecasts that were done by using the averaged metrics $P_{m,avg}$ and $P_{eval,avg}$. As can be seen, $\tau = 0.9$ or $\tau = 0.8$ is a decent choice for this kind of scenario. Each gray line in the graph shows a single execution of a scenario with a constant τ. As can be seen, the false positive rate mostly increase only at the beginning of the simulation when the threshold was set too high and no failures had occurred yet. Once the threshold CPU-usage is exceeded,

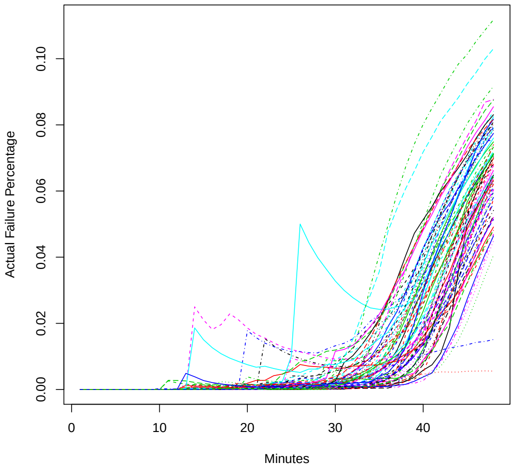


Figure 6.18.: Forecasted failure metric.

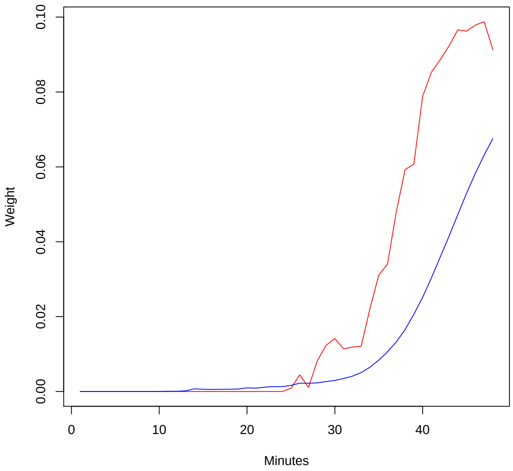


Figure 6.19.: Average forecasted $P_{m,avg}$ (red, upper line) and average actual failure metric $P_{eval,avg}$ (blue, lower line).

there are no more false positives anymore since in this simulation, the CPU-usage only increases and never really decreases.

6.4.2. Response-Time Analysis

Other than the CPU-utilization, the response times of the containers might be an interesting metric to look at. The difficulty for response-time datasets are especially

1. **No hard upper limit.** Unlike CPU- or memory-measurements, the range of response-times vary from application to application. Therefore, in order to use them as a failure prediction metric, it is important to find an appropriate limit before the metric can be used.

2. **Requirement of Context-Knowledge.** Often, a service provides several different functions and interfaces for which it is possible to create a separate response-time dataset. Since it might not be viable to consider all response-time measurements, a list of important functions has to be found manually.

Figures 6.21a and 6.21b show the response times for the middletier component. As can be seen, at the start of the execution, the response-times are at an almost equal level than the response times at the end of the execution. This effect might be caused by the component allocating memory that was stored onto hard-disk in the idle times.

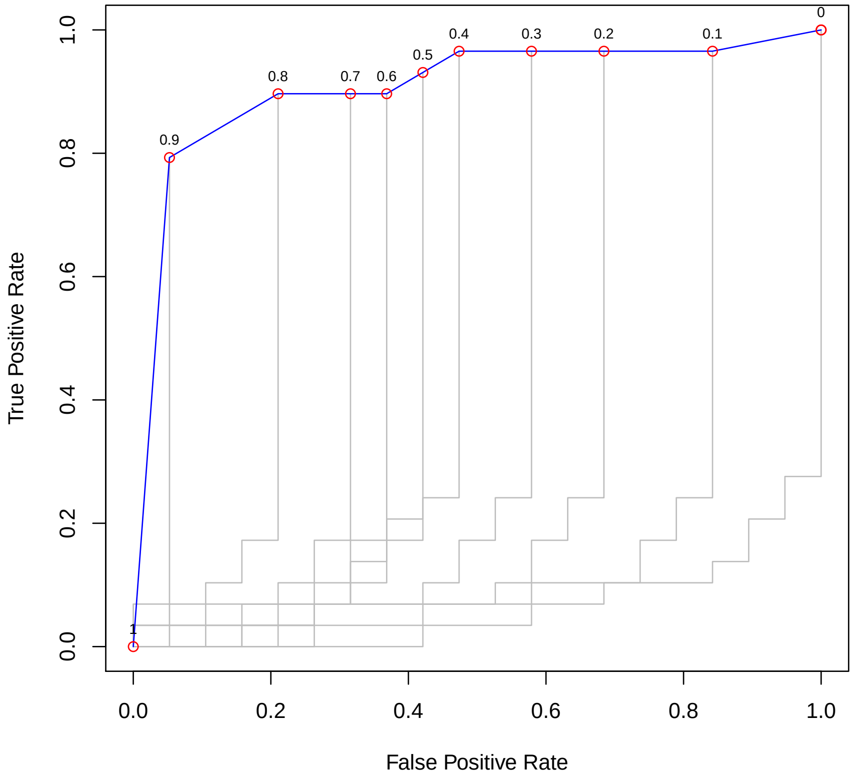


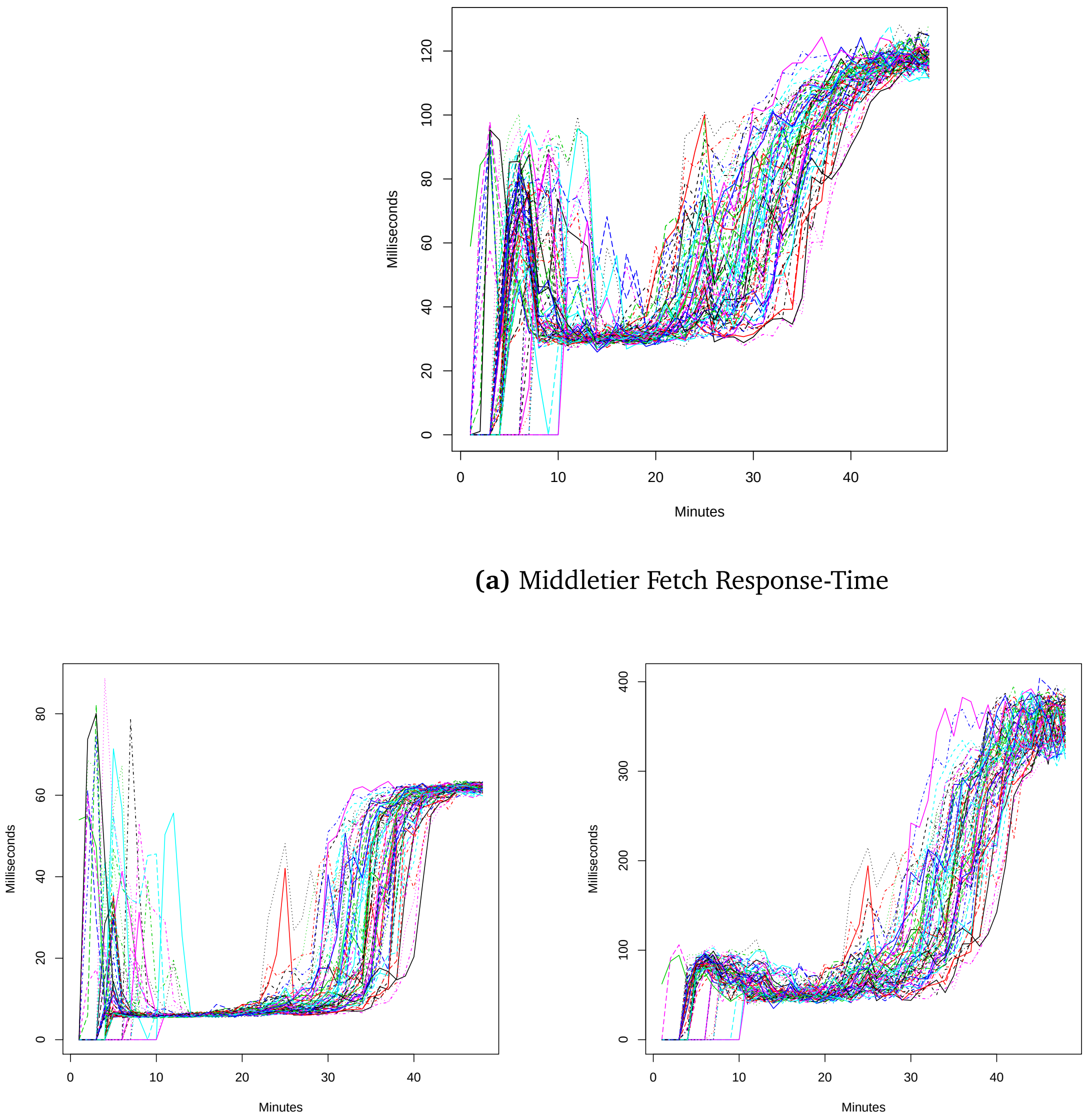
Figure 6.20.: ROC for 10 different values of τ. The data point labels show the value of τ for each point.

However, the component itself was never restarted during the executions and there was only a little time-frame where the component was actually idle.

After this upstarting-phase, the response-times drop to a rather low level and mostly increase suddenly at time 30 which is the time where failures begin to arise. Aside from the upstarting-phase, for a failure prediction metric that is created out of this data, there is the problem that the response-times only really rise up when the failures begin to occur. This is why **the symptom of the failure cannot be predicted easily** by only looking at the response-times of the respective container.

6.4.3. Container-Induced Interferences

Next, it will be evaluated whether the prediction metric remains stable when there are other microservices that are additionally applying workload onto the physical host on

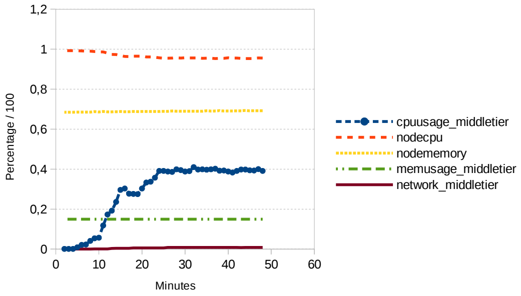


(a) Middletier Fetch Response-Time

(b) Middletier Fetch-Feed Response-Time

(c) Edge 2 View-feed Response-Time

Figure 6.21.: Middletier Response-Times.



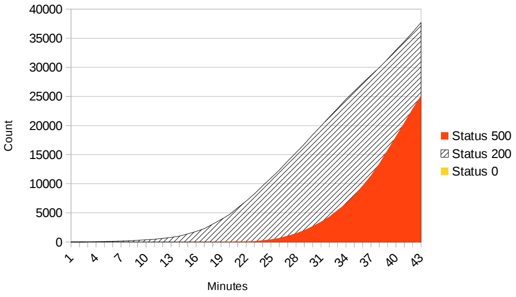


Figure 6.22.: Middletier CPU, memory, and network measurements.

Figure 6.23.: Success (status 200)- and failure rates (status 500, status 0).

which a monitored service is executed on. This scenario shows that the created failure predictor is vulnerable against interference from other containers that are running on the same host. As could be seen in the other scenarios above, a *monolithic* failure predictor can always be created by defining a metric that correlates with the actual, hidden failure metric, which in our case is the number of failures over time logged by the workload generator.

Parameter	Description
Microservice Setup	2 Edges, 1 Middletier, 1 RSS-server, Middletier as bottleneck
Container Configuration	Middletier separated on Node D
Node stress	None
Container stress	> 95% CPU workload on Node D by using only microservices

In this scenario, the CPU-utilization of the middletier component shown in Figure 6.22 does not exceed 45% even though the simulated user count is identical to the normal scenario (Table 6.1). Instead, the amount of Failures produced are strongly increased (see Figure 6.23). This behavior occurs since there are no resources available on the host system which then caps the CPU of all containers to a lower level.

This leads to the fact that the predicted metric P_m with the calibrated τ will never detect any failures that occur when the host machine becomes the bottleneck of the system. Only after adjusting τ it would again be possible to detect the errors, however this would first require the predictor to detect the transition of the bottleneck from the middletier-component to the host.

What now needs to be found would be a new Metric P_H that is able to correlate to the number of failures in the system regardless of the above described problem. When looking only at the metrics available within the container, which is **1)** The CPU-utilization,

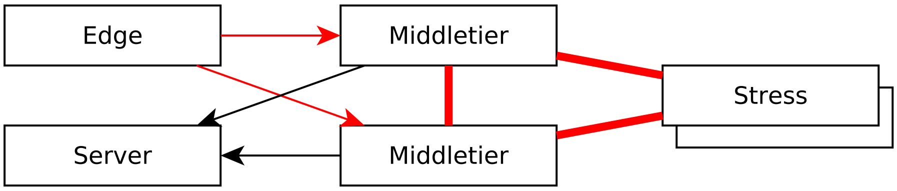


Figure 6.24.: Workload-Dependency Graph. The red lines represent the induced workload that affect the behavior of the containers. The middletier- and stress-containers are hosted on the same node and are therefore CPU-dependent on each other. The edge service influences the CPU-utilization of the middletier component and therefore, the middletier-component is CPU-dependent on the edge service.

2) The memory-usage and **3)** The network usage, Figure 6.22 shows that there there is no obvious metric that correlates with the number of produced failures. In fact, when trying to combine one or more of these metrics, it should be clear that it will never be possible to create an adequate metric that correlates in all discussed cases.

This is the point where it becomes necessary to not only look at the behavior of the container itself but also on the containers that are influencing the system on which the container that is to be observed is deployed on.

6.5. A Hierarchical Approach

So far, the failure prediction was only based on measurements that could directly be obtained without having any knowledge about the function of the container and its dependencies. It has been shown that in this case, this is not enough to obtain a sophisticated failure prediction method that covers the problematic of the CPU-overload in the host system. This problem can also not be solved by simply adding another system-level predictor such that both the host system and the container that is deployed on the system are separately observed since in many cases, it is completely fine if the host system is on a high CPU-workload for a shorter period of time and there would be a significantly higher amount of false positives in the system compared to only observing the containers alone. In fact, the above described problem can even be applied to many other scenarios that can be designed to be a lot more complex than the here discussed scenario (see section 6.7.1).

Now, we want to figure out a way to create a new metric P_H that again correlates with the number of client-failures by using an hierarchical failure prediction approach.

First, a dependency graph needs to be created from which it is possible to conclude the cause of the container interference problem. When considering the architectural structure of the application, it is possible to argue that every container that is deployed on the same host can influence the behavior of every other container within the same host and therefore there is some type of dependency between each container on the same host. The other type of dependency comes from the application-defined dependencies between the services. In this case, the edge-containers put stress on the middletier-containers as well by propagating the client-requests to them. Figure 6.24 illustrates this dependency graph.

By using this context-knowledge, equation 6.5 can be created which is a simple metric that considers the container-interference problem. The idea is that the delta between the CPU-limit of the host and the summed up CPU-workloads of the deployed containers on the host (represented as the vector $CPU_{container}$) except of the container that is to be observed represents the span in which the observed container can still allocate more CPU until the host CPU reaches its limit.

$$P_{H1} = \frac{CPU_{middletier}}{1 - \sum_i CPU_{container,i} + CPU_{middletier}} \tag{6.5}$$

Assuming that the sum of the container-CPU equals the amount of CPU-workload that is put onto the host-CPU, P_{H1} is a metric that determines for any container on which it is applied to the relative CPU-usage that is scaled by the overall host-CPU-limit. Assuming for example, there are 9 containers that each use 10% of the hosts CPU, the 10.th container can only possibly allocate 10% of the hosts CPU until problems arise. for a CPU-usage of 9%, $P_{H1}(container\ 10) = \frac{0.09}{0.1} = 0.9$. This way, the metric P_m could again be applied on P_{H1} in order to get a stable failure predictor and since P_{H1} is scaled like the original CPU-measurement dataset, the parameter τ needs not to be changed.

However, Figure 6.25 shows that the summed CPU does not equal the actual CPU-workload of the host system which might be caused by some overhead from context-switches. Therefore, P_{H1} cannot easily be applied to an actual system.

Instead, the idea of P_{H1} can be used to create Equation 6.6:

$$P_{H2} = \frac{CPU_{middletier}}{\max_i CPU_{container,i}} \cdot CPU_{host} \tag{6.6}$$

Now, P_{H2} uses the relative CPU-ratio of the observed container compared to all other containers as a weight which is then multiplied by the actual CPU-utilization of the host. This has two advantages:

1. The host CPU is directly considered. This means that any overhead induced by non-visible factors can easily be mitigated.

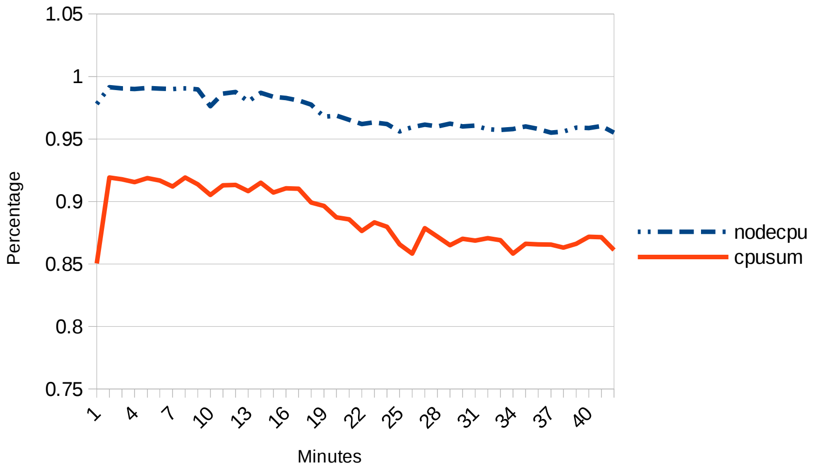


Figure 6.25.: CPU-Workload of the physical node (dashed line) and summed up CPU-workload of all containers that are deployed on the host.

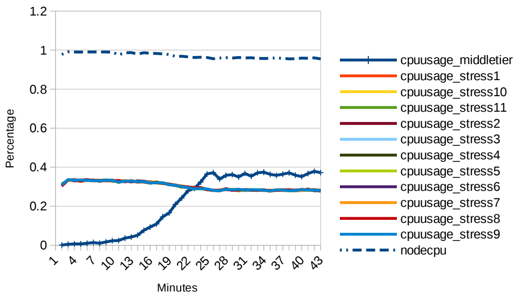


Figure 6.26.: CPU-utilization of all containers and the Host node.

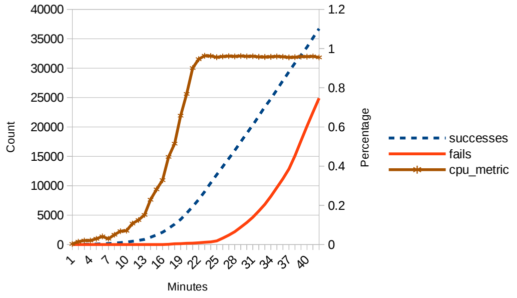


Figure 6.27.: Success- and Failure counts and the computed P_{H2} Metric.

2. The metric scales based on the CPU-usage of all other containers. Tests have shown that the CPU-usage of a new container can be increased up to a certain threshold level which causes all other deployed containers a slight decrease of CPU-share the more the new container requests CPU (i.e. when resources get scarce, all deployed containers get a similar share of CPU regardless of which container was deployed first). Therefore, if the CPU is distributed equally for all containers, the relative CPU serves as an approximate upper CPU-limit in this situation.

Figure 6.26 shows the actual CPU-utilization of the host and the deployed containers. Figure 6.27 shows P_{H2} and how it correlates to the client-failure-count. As can be seen, the CPU increases to 90% at minute 23 which is around the time when failures begin to arise (134 failures at minute 20, 247 failures at minute 23).

Component	Dependencies
DB1	$\emptyset$
RSS1	$\emptyset$
M1	$\{(S1, \frac{1}{10}), ..., (S10, \frac{1}{10})\}$
S1	$\{(M1, \frac{1}{10}), (S2, \frac{1}{10}), ..., (S10, \frac{1}{10})\}$
...	...
S10	$\{(M1, \frac{1}{10}), (S1, \frac{1}{10}), ..., (S9, \frac{1}{10})\}$
E1	{(M1, 1.0)}
E2	{(M1, 1.0)}

Table 6.5.: Architecture Dependency Model.

Component	Dependencies
DB1, RSS1, N1	$\emptyset$
M1, S1 ... S10	$\{(N1, 1)\}$
E1, E2	{(M1, 1.0)}

Table 6.6.: Simplified Architecture Dependency Model.

While there are still several problems with the formula P_{H2}, this example shows that it can get quite complex to manually determine possible metrics that correlate to the failure rate in such scenarios.

6.5.1. HORA Implementation

The above described problem can be modeled by using the *HORA* approach described in Section 2.3.3.

Let $\mathcal{C} = \{E1, E2, M1, DB1, RSS1, S1, ..., S10, N1\}$ be the set of components in the system (E = Edge, M = Middletier, DB = Database, RSS = RSS-server, S = Stress-Container, N = Physical Host). The set ADM is then defined as shown in Table 6.5. Since the middletier and the stress containers are deployed on the same host, they are each CPU-dependent on each other. The table shows a circular relationship between the stress and middletier containers, which is not allowed and therefore the ADM set has to be simplified as shown in Table 6.6. As can be seen, this simplifies the ADM in such a way, that the middletier-component is only dependent on the physical host on which the container is executed on. Figures 6.28 and 6.29 show the graphical representation of the dependencies.

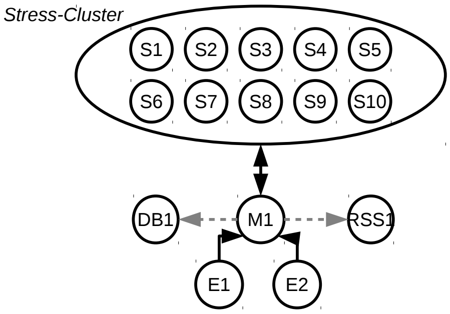


Figure 6.28.: Dependency Graph of the actual ADM. The containers S1 to S10 and the container M1 build a complete Graph. The dependencies for the server and database are not included since in this case, they are not CPU-dependent on each other.

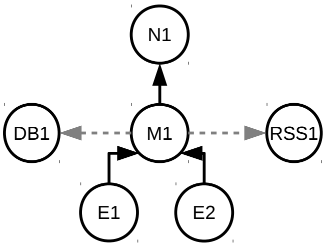


Figure 6.29.: Dependency Graph of the simplified ADM. The dependencies for the server and Database are not included since in this case, they are not CPU-Dependent on each other.

After creating the ADM, the FPM is automatically inferred by the HORA algorithm. Based on the FPM, the HORA predictor generates two prediction results which are:

1. The individual forecast for each component (*CFP*). Currently, the algorithm always produces *ARIMA* forecasts for any single component.
2. The combined forecast for each component. This value represents the results of the Failure Propagation Model (*FPM*).

First, the evaluation is done with the default workload simulation from section 6.3.1. Figure 6.30 shows the *FPM*-prediction result for the observed component and the host.

Next, stress-containers are deployed on the node on which the middletier-container is executed on such that the node-CPU utilization is above 97%. Figure 6.32 shows the prediction of HORA. As can be seen, the combined failure probability (*FPM*) is strongly influenced by the dependency to the executing Node. At minute 25, the failure probability increases to 1.0 which is about the time when failures begin to arise.

However, a more detailed analysis has shown, that the results in Figure 6.32 are actually caused by effects that occur during initialization of the algorithm and therefore, the forecast does only seemingly correlate to the failures in the application.

Figure 6.35 shows, that the forecast also becomes 1 without any workload that is placed onto the service. After the forecast starts at minute 2, the FPM failure-probability of the

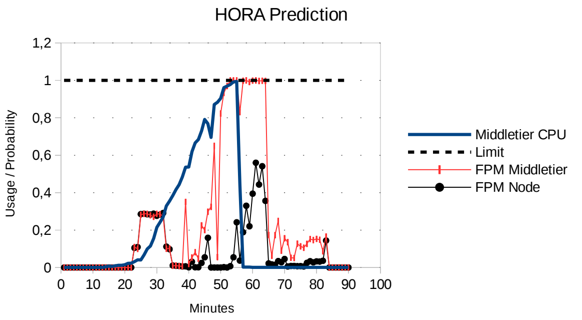


Figure 6.30.: CPU-utilization and the respective HORA prediction for the normal workload scenario.

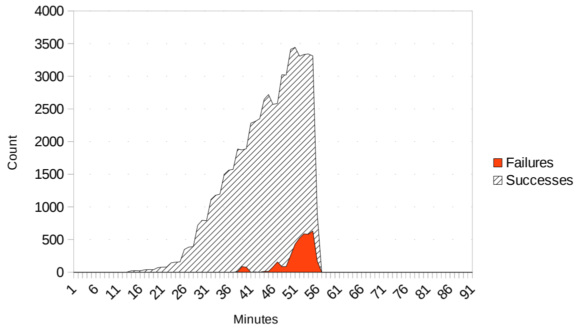


Figure 6.31.: Failures recorded by the workload generator for the normal workload scenario.

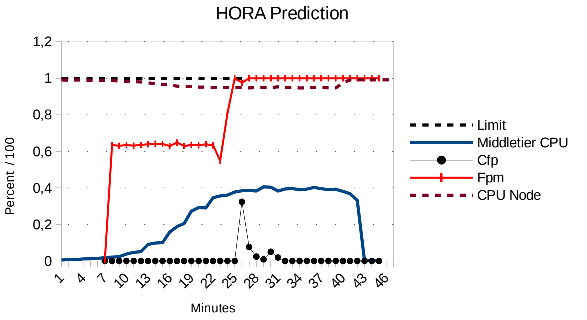


Figure 6.32.: CPU-utilization and the respective HORA prediction for the node-stress scenario.

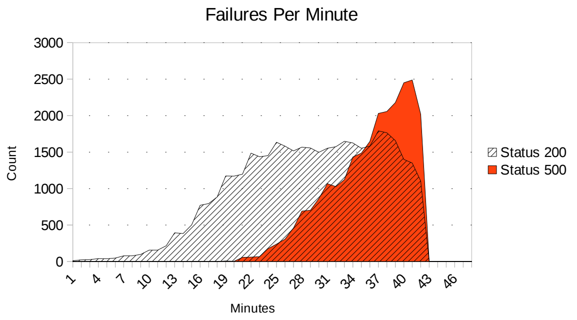


Figure 6.33.: Failures recorded by the workload generator for the node-stress scenario.

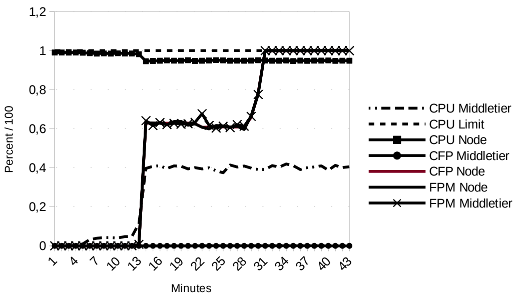


Figure 6.34.: Failure Probabilities for the middletier service while the workload Generator is simulating 600 users and induces high workload.

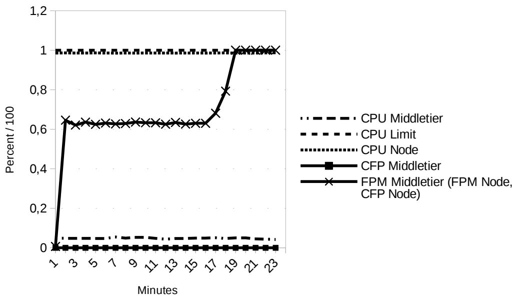


Figure 6.35.: Failure Probabilities for the middletier service while the workload Generator is simulating 0 users and induces no workload.

middletier service increases from about 0.6 to 1 at minute 17 while the workload of the middletier stays constant and *no failures were produced*. Figure 6.32 showed the same failure-probability increase after exactly 15 minutes as well.

The effects that could be observed were caused by a modification that affected the number of data points in the forecast-buffer during initialization. The HORA algorithm was started at minute 5 and first needed 20 data points to start with the prediction.

This becomes clear when looking at the actual HORA prediction algorithm that was described in the the beginning of this thesis. The CFP-prediction of the physical node solely relies on the respective node-CPU-utilization and as can be seen in Figure 6.32, the ARIMA algorithm should have no reason to forecast an increase of the node-CPU since it is almost constantly 100% and since there are no ADM-dependencies for the node, the CFP prediction has to be similar to the FPM prediction.

Modifications

The previous result has shown, that the HORA-algorithm propagates the failure probabilities of the host correctly. However, the FPM forecast does currently not consider the reduced CPU-limit that a service experiences when it is deployed on a stressed host that uses a lot of CPU share on other services.

For this reason, the HORA algorithm is extended by including the idea from section 6.5 into the CFP-prediction of individual components.

Currently, the CFP-threshold value is $\tau_{P_m} = 0.9$. This value is inadequate for scenarios where the CPU share is limited to a reduced amount (in the previous scenario it was limited to 40%) and it has to be reduced in such a case. However, if the threshold is low enough for this scenario, the forecast in the non-stress scenario would become invalid since the threshold would be too low. In consequence, the threshold should be adjusted on-the-fly when a high CPU-workload is observed on the host system. Equation 6.7 shows the new threshold of the algorithm. The constant th_{CPU} needs to be set to the value where the reduced CPU limit takes effect on a stressed host. On the tested machine, this threshold was $th_{CPU} = 0.92$.

$$\tau_{ARIMA} = \begin{cases} \tau_{P_m} \text{when } CPU_{ancestor} > th_{CPU} \\ \frac{3}{4} max_i CPU_{container,i} \text{otherwise} \end{cases}, 0 \leq th_{CPU} \leq 1 \tag{6.7}$$

The threshold modification was tested by repeatedly executing 5 workload scenarios where at the peak, the number of simulated users is 600 which increases the CPU-utilization of the service in a non-stress scenario to 100%. Figure 6.36 shows the FPM

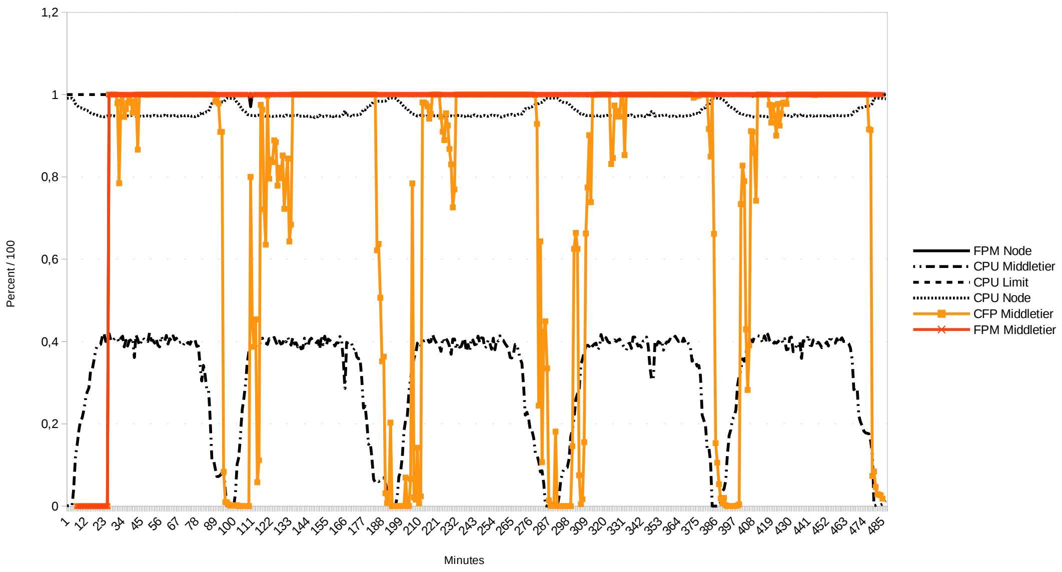


Figure 6.36.: CFP and FPM prediction for multiple executions of the workload scenario.

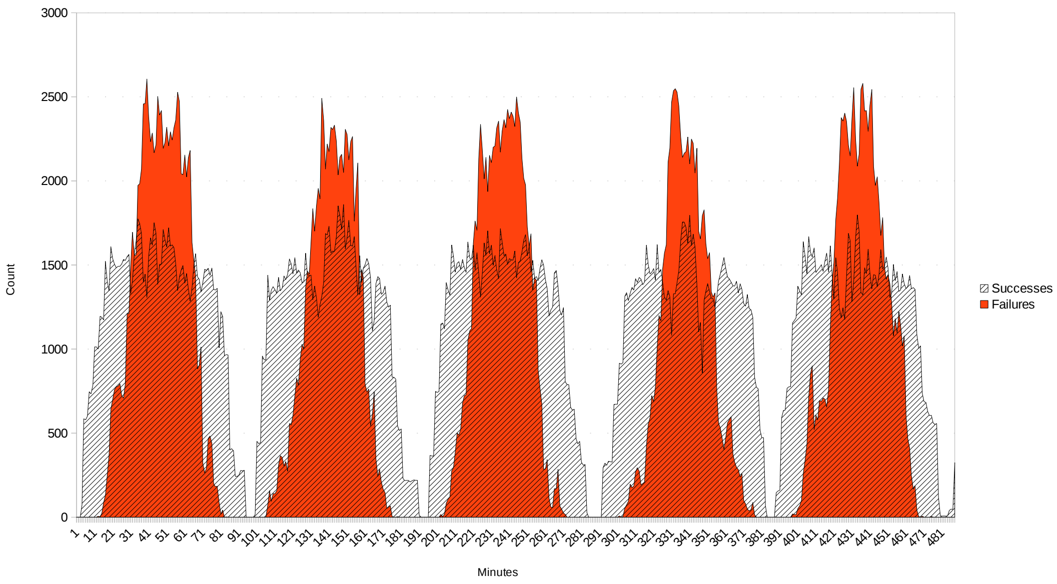


Figure 6.37.: Success- and Failure rates for each workload simulation.

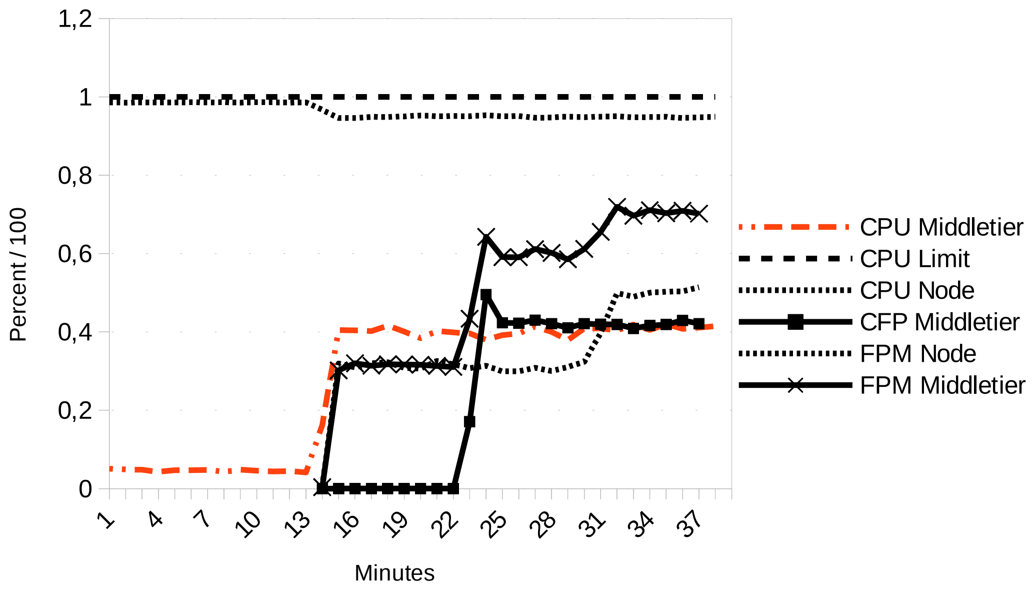


Figure 6.38.: Failure Probabilities of the modified HORA algorithm for the middletier service while 600 users are simulated starting from minute 13. The CFP and FPM predictions at time t were made at time $t - 10$ Minutes.

and CFP predictions of the middletier service for the simulated scenario and figure 6.37 shows the respective successful and unsuccessful requests. As can be seen, the threshold was reduced to a low enough level to classify failures correctly in this scenario. The FPM forecast of the approach is however still too high and needs to be adjusted as well.

Figure 6.40 show a scaled prediction that combines the results from Figure 6.38 and Figure 6.36. Figure 6.41 shows the respective failure- and success counts.

The reason why the FPM prediction is constantly high is, that it is too much influenced by the high CPU load of the ancestor node. Figure 6.38 shows another workload scenario where the workload is suddenly increased to 600 users at Minute 13. This time, the failure probabilities are reduced by $\frac{1}{2}$ for each forecast.

As can be seen in Figure 6.38, the modified algorithm increases the CFP-prediction of the service by reducing the CPU-threshold in the case of high CPU-utilization of the host. Starting at minute 13, the HORA algorithm increases the failure probability of the middletier service by about 35% due to high CPU-utilization of the host. As soon as the workload of the service starts to increase, the forecasted CFP failure probability of the service increases to 45% which leads to an increase of the failure probability in the FPM prediction to over 60%.

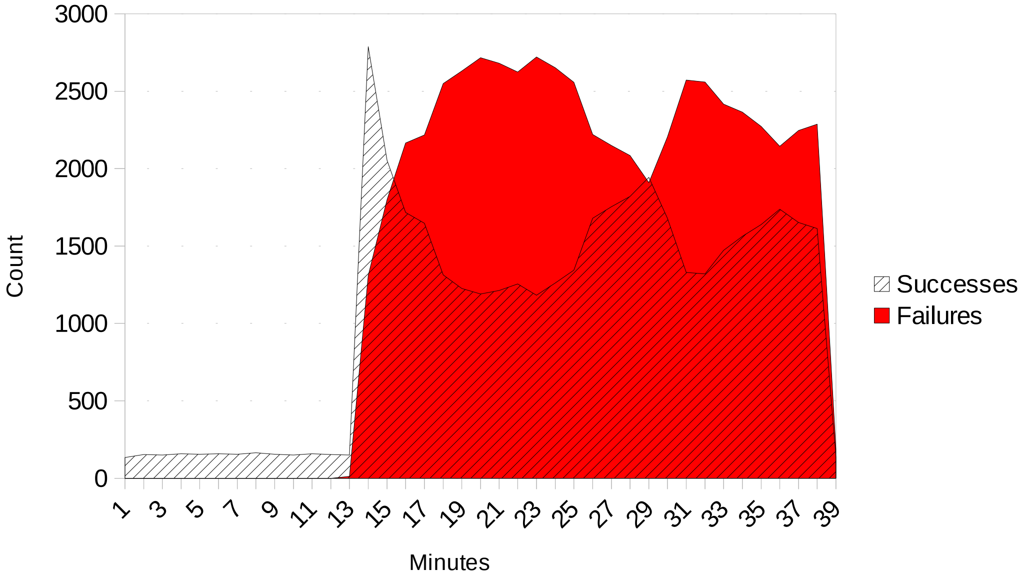


Figure 6.39.: Failure- and Success rates of the modified HORA algorithm.

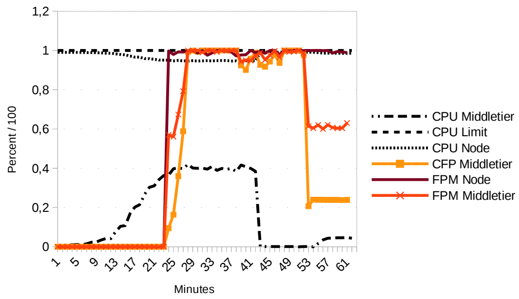


Figure 6.40.: Failure probabilities and CPU-utilization of the middletier component and the host.

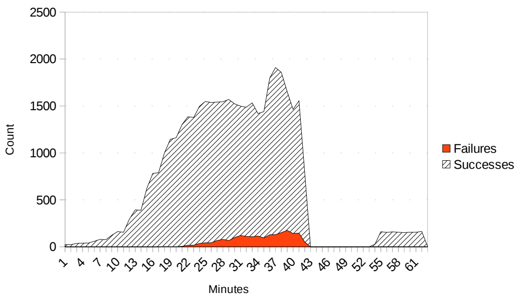


Figure 6.41.: Failure- and success rates of the modified HORA algorithm.

This algorithm shows, that it is indeed possible to forecast failures on both a stressed and a not stressed host by using a predictor that considers the architecture of the application like HORA.

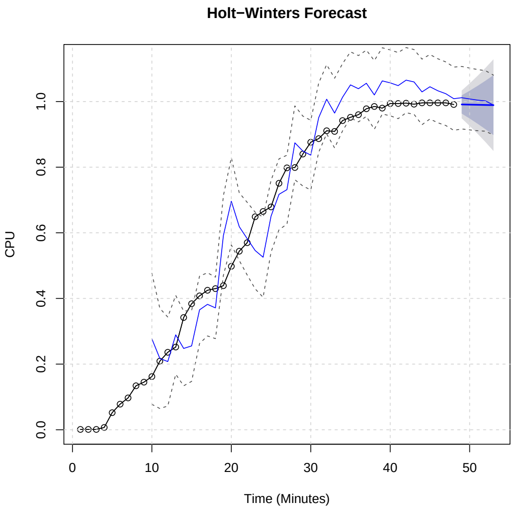


Figure 6.42.: Shifted Holt-Winters 5-minute forecast. The dotted line represents the original data, the blue, solid line is the mean of the prediction and the dashed lines represent the minimum and maximum predictions.

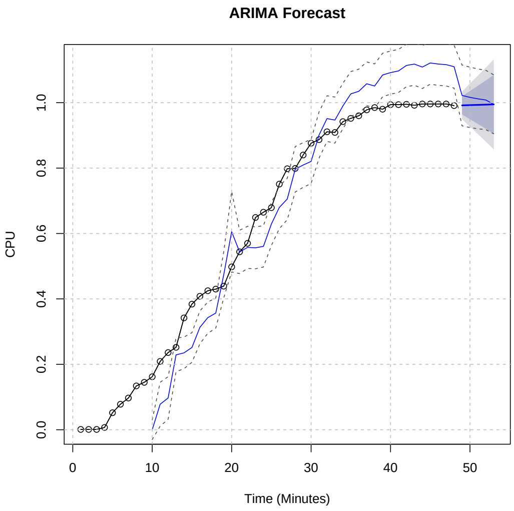


Figure 6.43.: Shifted ARIMA 5-minute forecast. The dotted line represents the original data, the blue, straight line is the Mean of the prediction and the dashed lines represent the minimum and maximum predictions.

6.6. Failure Prediction Results

In the above shown simulations, the quality of the failure prediction influences the result of the correlation metrics. This section gives a short overview about the prediction quality of the used failure predictors. The Figures 6.42 and 6.43 show the prediction results of a simulation with increasing workloads. The prediction was done with a window size of 60, which means that the whole history of the graph at any point in time t is used to predict the next data point.

Table 6.7 shows the *Mean Squared Error* and other metrics for this prediction. As can be seen, the MSE is relatively small for both failure predictors but the value for Holt-Winters is slightly better than the results from the ARIMA prediction.

Figure 6.44 shows the Mean-Squared-Error for different sliding window sizes. As can be seen in the graph, the prediction quality of ARIMA is comparatively low with a window size lower or equal than 5 (which means the algorithm considers the last 5 minutes of

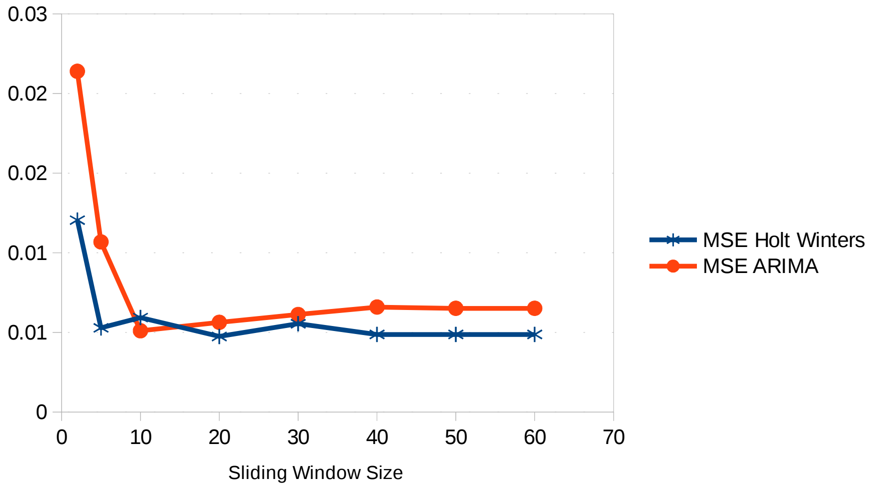


Figure 6.44.: Mean-Squared-Error for different Sliding Window Sizes.

history to make the next prediction) and remains more or less the same for any higher window size.

Metric	Holt-Winters	ARIMA
Mean Squared Error (MSE)	0.00486	0.00650
Mean Absolute Percentage Error (MAPE)	0.20253	0.26308
Mean Absolute Derivation (MAD)	2.55110	3.06573
Mean Squared Derivation (MSD)	0.23370	0.31237

Table 6.7.: Error-Metrics for the prediction in Figure 6.42 and 6.43.

6.7. Results and Discussion

The main goal of this thesis was to analyze and quantify the degree to which an isolated, containerized service can affect other services that are executed on the same machine. For this, different scenarios were evaluated and it has been shown, that containers can in fact be influenced by other containers that run completely independent from another in terms of software architecture.

First, a workload scenario was created where the user count for the test service rises monotonically up to the point where the number of service requests induce failures in the system.

After that, a failure metric was developed that derives the failure likelihood from the CPU-utilization of the service which showed, that those failures do indeed correlate with the CPU-utilization of the service. In order to compute a general threshold value for this metric, the workload scenario was extended such that the user count of the service is not exactly monotonically increasing but is only on average increasing as one would expect a workload scenario in practice to resemble. The scenario was slightly randomized and executed 100 times to show the validity of the approach and to analyze the measurements that can be used for failure prediction. Based on this, the threshold for failures was estimated for further analyses.

The developed monolithic failure predictor has been shown to forecast service failures that result from CPU over-utilization in the described workload scenario. Therefore, the working hypothesis is, that the produced failure predictor is able to forecast most of the failures that are generated due to CPU over-utilization for any isolated service in the system.

However, this hypothesis was falsified by deploying several other services on the host that consume a significant amount of CPU resources. The occurrence of this scenario in practice cannot be excluded since it would be impractical to only deploy as many services on a host as the summarized upper CPU-limit of the containers would allow. The probability of all services consuming their full CPU-utilization share at the same time may not be high but in times where these services are especially requested, failure predictors should still be able to forecast a CPU shortage in the system.

The simplest solution to this problem would be to observe the host CPU as well, additionally to every container within the host. As can be seen for example in Figure 6.22 and 6.23, this may not be appropriate since a high CPU-usage of the node does not automatically correlate with the number of failures in a container that is executed on that host. Therefore, this approach would produce a large amount of false positives in the system which would make it difficult e.g. for unsupervised monitoring systems to understand the current situation.

The insight that was obtained by this analysis lead to the next hypothesis, which states that it it possible to forecast CPU-over-utilization failures by considering the indirect dependencies between the nodes, i.e. the dependency of free CPU-resources on the host. Therefore, the failure metric was extended by considering architectural components of the system. The CPU-Dependency can be formulated for all containerized services by considering the current CPU-utilization of all containers and the CPU-Limit of the host and by setting them in relation to each other. The analysis of this new failure metric has shown, that for the given scenario, it would be possible to compute a metric that correlates with the number of failures again.

The problem with the computed metric was, that it was highly dependent on this particular use-case and that it potentially has to be modified for every other use-case. Also, it seemed quite artificial and unnatural.

As a consequence, a new approach that also considers architectural dependencies between components, called *HORA* was evaluated. The dependencies could however not be modeled as a graph between containerized services only (circular dependencies) and therefore the dependency graph has been simplified such that the executing component is CPU-dependent only from its host and not from any other containers. The results have shown, that in case of a normal execution without any other containers that affect the CPU of the host, the HORA forecast is almost identical to the forecast that was created by the manually computed failure metric. In the case where there are other containers that additionally stress the host CPU, the HORA forecast is however significantly better compared to the monolithic approach. Where the monolithic approach had shown a true positive rate of 0% for any amount of false positives (i.e. no failures were detected whatsoever), the HORA forecast shows a true positive rate of 80% natively, without any modifications of the algorithm itself.

6.7.1. Hierarchical Failure Prediction Approaches

The evaluation shows that for microservice environments, there are new challenges like the existence of interference and interaction of services within a single physical host. We saw, that failures can propagate along the topological graph of dependencies within a system - In the scenario where the microservices interfered with each other by inducing a CPU-shortage on the host system, the dependency graph can be illustrated as shown in Figure 6.24. In this case, the workload itself can be seen as a structural dependency induced by the executing system as the behavior of a single container also depends on the containers that affect the CPU of the host system at the time of execution.

Therefore, this scenario can also be applied to other applications with a different dependency graph than the one in this study: As an example, consider a system that consists of multiple services and resources where each service uses one or more resources. Each use of a resource at time t_x fills a slot for the used resource. A resource holds only a limited number of slots x_{lim} and each filled slot delays any other request from a service by a time t_{delay}. Figure 6.45 illustrates such a system for 3 services and 3 resources.

In this system, each request should be answered within a time t_{th} up to which the response is considered to be non-faulty. Now consider a faulty service B that due to a failure consistently uses Resource 1, thus always putting workload on the resource. In order to process a request from one of the clients, service A accesses Resources 1 and 2 where $t_{delay,A} < t_{th}$. After this, service D processes the response from service A

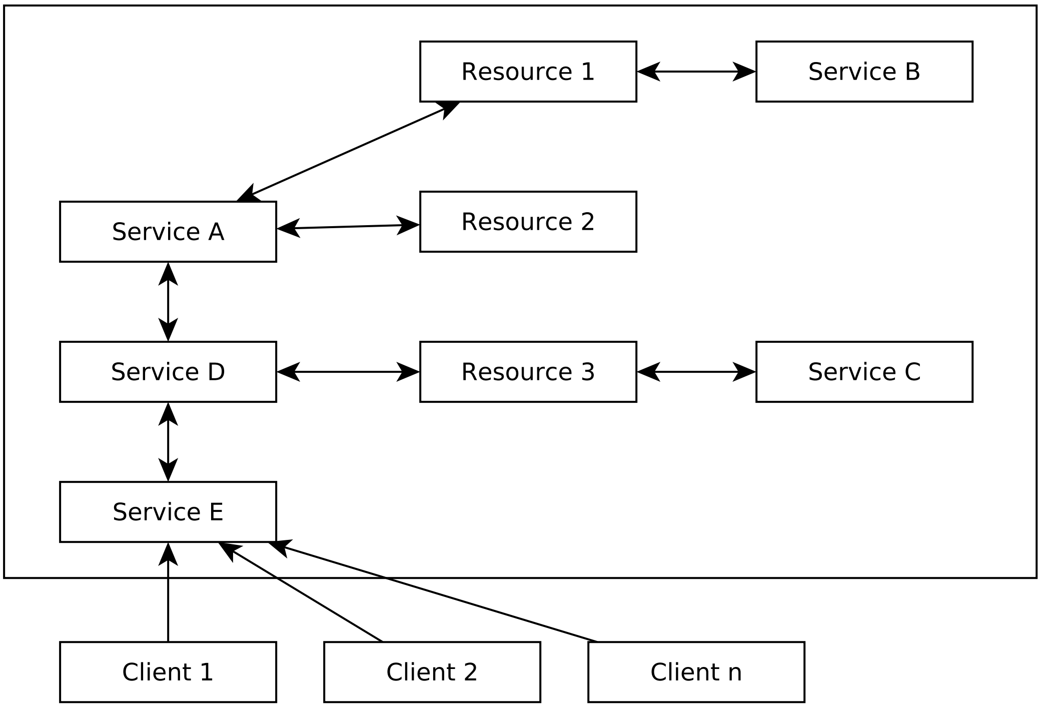


Figure 6.45.: A more complex Dependency Example.

and accesses another Resource. Finally, service E recognizes that the response time $t_{delay,E} > t_{th}$ and produces a failure alert.

If a failure predictor is to be deployed onto this system in order to prevent such a failure by replicating bottleneck resources if needed, it would first have to detect the respective problematic resource and calculate a metric P_h that correlates to the number of failures produced by service E.

A global failure predictor would consider the system or parts of it as a black box which would make it difficult to create such a metric. One possibility would be to measure the response time of service E but this might not serve as an indication metric for failures since when service B would detach from Resource 1 occasionally, the response time does not indicate the moment in time where service B attaches to Resource A again which actually leads to the high response time. The same problem applies for predictors that only predict failures on a single service.

A Hierarchical failure predictor on the other hand is capable of considering the architectural structure of the system and might therefore be able to calculate the metric P_h by forecasting the times where services attach to Resources and estimating the processing time $t_{delay,critical}$ that goes along the critical path of the system which in this case would be $B \mapsto 1 \mapsto A \mapsto D \mapsto E$.

Chapter 7

Conclusion

Containerized system architectures like Docker and Kubernetes have become quite popular in recent years and are on the rise especially in PaaS areas. In contrast to services that are executed on a fully virtualized host, microservices are dynamic, small and independent and can be deployed and removed within seconds. They can be executed in thousands on a single host and often run completely natively on the CPU of the host OS.

Failure predictors, which are often used for failure prevention strategies, have been researched for over 40 years and there are many interesting approaches that can be used to forecast time-series, system event logs and other data regarding a single application like for example the actual program code. Many of these failure predictors are designed for the use in a static environment where the number of components is a known constant. However, in a microservice architecture, many failure predictors have become obsolete and new approaches have to be discovered.

This thesis shows, that additionally to the heavy interaction within services in a Microservice environment, there are also several other effects that have to be considered in failure prediction algorithms. One effect that is discussed in this thesis is the over-utilization of CPU resources on a physical host that is induced by a temporary rise of user requests for the deployed services on a single host. The thesis suggests that forecasting failures in a system under stress produces invalid results for failure prediction algorithms that only consider measurements like the CPU-utilization of single services for failure prediction.

The approach of the thesis can be subdivided into three steps: *1)* The Microservice environment is created. This includes the creation of the system, the choice of the Service on which further tests are run and the choice of monitoring software and measurements for further testing. *2)* The fault generation. In this step, it is discussed about which failures will be analyzed and how they can be produced in the deployed microservice.

3) The data extraction. In this step, a program is developed that automatically extracts and joins entries from the database on which monitoring data is stored. *4)* The failure prediction. Based on the data that was gathered for the executed service, failure prediction algorithms and metrics are discussed that can be used for forecasting the generated failures. *5)* failure evaluation. In the last step, the failure prediction results are compared and a conclusion is drawn.

The results show, that a failure predictor which only analyzes one single component at a time may not be able to classify the overall state of the system. Likewise, a failure predictor that tries to analyze the system as a whole may not infer the state of single components. On the other hand, failure predictors that consider the architecture of the system and combine failure prediction on single components according to their hierarchy are capable of predicting failures in this environment and should therefore be preferred over monolithic failure predictors.

Future Work

This thesis researched the effects of microservices on failure prediction on only a small set of metrics, measurements, failure prediction algorithms and applications. In order to study the effects in more depth, further research would require a broader spectrum to cover more areas and find more implications of Microservice environments. Also, The example in section 6.7.1 could be implemented and extended to show, that the discussed problems can in practice be applied to a more complex system structure.

Appendix A

RSS-Reader Deployment

A.1. RSS-Edge Service

```
apiVersion: v1
kind: Service
metadata:
  name: edge
  labels:
    name: edge
spec:
  type: NodePort
  ports:
  - port: 9090
    name: http
    nodePort: 31000
  selector:
    name: edge
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: edge
  labels:
    name: edge
spec:
  replicas: 1
  selector:
    name: edge
  template:
    metadata:
      labels:
        name: edge
    spec:
```

```
containers:
- name: edge
  image: hora/recipes-rss-edge-kieker:0.4
  imagePullPolicy: Always
  ports:
  - containerPort: 9090
  resources:
    requests:
      cpu: "0.5"
      memory: 500M
    limits:
      cpu: "1"
      memory: 2000M
nodeSelector:
    nodeassignment: 10.0.11.62
```

Listing A.1: REST-Call Generation by Locust

Appendix A

Bibliography

[ALR+01] A. Avizienis, J.-C. Laprie, B. Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001 (cit. on p. 1).

[And15] C. Anderson. "Docker." In: *IEEE Software* 32.3 (2015) (cit. on p. 7).

[BB04] S. Brahim-Belhouari, A. Bermak. "Gaussian process for nonstationary time series prediction." In: *Computational Statistics & Data Analysis* 47.4 (2004), pp. 705–712 (cit. on p. 25).

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. "Xen and the art of virtualization." In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177 (cit. on p. 26).

[Ber14] D. Bernstein. "Containers and cloud: From lxc to docker to kubernetes." In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84 (cit. on p. 27).

[Ber96] L. M. Berliner. "Hierarchical Bayesian time series models." In: *Maximum entropy and Bayesian methods*. Springer, 1996, pp. 15–22 (cit. on p. 26).

[BHJ16] A. Balalaie, A. Heydarnoori, P. Jamshidi. "Microservices architecture enables DevOps: migration to a cloud-native architecture." In: *IEEE Software* 33.3 (2016), pp. 42–52 (cit. on p. 27).

[cAd17] cAdvisor. *cAdvisor*. https://github.com/google/cadvisor. Accessed: 2017-06-12. 2017 (cit. on p. 18).

[Cha00] C. Chatfield. *Time-series forecasting*. CRC Press, 2000 (cit. on pp. 11, 12).

[Doc16a] Docker. *Docker*. http://www.docker.com. Accessed: 2016-11-01. 2016 (cit. on pp. 6, 7).

[Doc16b] Docker. *Docker Swarm*. https://github.com/docker/swarm. Accessed: 2016-11-01. 2016 (cit. on pp. 7, 8).

[DRK14] R. Dua, A. R. Raja, D. Kakadia. “Virtualization vs Containerization to Support PaaS.” In: *2014 IEEE International Conference on Cloud Engineering*. Mar. 2014, pp. 610–614. DOI: 10.1109/IC2E.2014.41 (cit. on p. 5).

[FFRR15] W. Felter, A. Ferreira, R. Rajamony, J. Rubio. “An updated performance comparison of virtual machines and linux containers.” In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 171–172 (cit. on pp. 5, 6).

[FSS07] A. Fedorova, M. Seltzer, M. D. Smith. “Improving performance isolation on chip multiprocessors via an operating system scheduler.” In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society. 2007, pp. 25–38 (cit. on p. 26).

[GCGV06] D. Gupta, L. Cherkasova, R. Gardner, A. Vahdat. “Enforcing performance isolation across virtual machines in Xen.” In: *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc. 2006, pp. 342–362 (cit. on p. 26).

[Has16] W. Hasselbring. “Microservices for scalability: keynote talk abstract.” In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM. 2016, pp. 133–134 (cit. on p. 27).

[Hea17] Heapster. *Heapster*. https://github.com/kubernetes/heapster. Accessed: 2017-06-12. 2017 (cit. on p. 18).

[HK+07] R. J. Hyndman, Y. Khandakar, et al. *Automatic time series for forecasting: the forecast package for R*. Tech. rep. Monash University, Department of Econometrics and Business Statistics, 2007 (cit. on pp. 14, 16).

[HS17] W. Hasselbring, G. Steinacker. “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce.” In: *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE. 2017, pp. 243–246 (cit. on p. 27).

[HTI97] M.-C. Hsueh, T. K. Tsai, R. K. Iyer. “Fault injection techniques and tools.” In: *Computer* 30.4 (1997), pp. 75–82 (cit. on p. 19).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pp. 247–248 (cit. on p. 18).

[Jul89] D. Julong. “Introduction to grey system theory.” In: *The Journal of grey system* 1.1 (1989), pp. 1–24 (cit. on p. 25).

[Kal04] P. S. Kalekar. “Time series forecasting using holt-winters exponential smoothing.” In: *Kanwal Rekhi School of Information Technology* 4329008 (2004), pp. 1–13 (cit. on p. 13).

[Kim03] K.-j. Kim. “Financial time series forecasting using support vector machines.” In: *Neurocomputing* 55.1 (2003), pp. 307–319 (cit. on p. 25).

[KJP15] A. Krylovskiy, M. Jahn, E. Patti. “Designing a smart city internet of things platform with microservice architecture.” In: *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE. 2015, pp. 25–30 (cit. on p. 30).

[KS02] N. K. Kasabov, Q. Song. “DENFIS: dynamic evolving neural-fuzzy inference system and its application for time-series prediction.” In: *IEEE transactions on Fuzzy Systems* 10.2 (2002), pp. 144–154 (cit. on p. 25).

[KSS03] J. C. Knight, E. A. Strunk, K. J. Sullivan. “Towards a rigorous definition of information system survivability.” In: *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*. Vol. 1. IEEE. 2003, pp. 78–89 (cit. on p. 1).

[Kub16] Kubernetes. *Kubernetes*. https://kubernetes.io/. Accessed: 2016-11-01. 2016 (cit. on pp. 7, 8).

[KUK10] E. Kayacan, B. Ulutas, O. Kaynak. “Grey system theory-based models in time series prediction.” In: *Expert systems with applications* 37.2 (2010), pp. 1784–1789 (cit. on p. 25).

[MN15] K. Meinke, P. Nycander. “Learning-based testing of distributed microservice architectures: Correctness and fault injection.” In: *International Conference on Software Engineering and Formal Methods*. Springer. 2015, pp. 3–10 (cit. on p. 27).

[NBZ06] N. Nagappan, T. Ball, A. Zeller. “Mining metrics to predict component failures.” In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 452–461 (cit. on p. 23).

[Net] Netflix. *Netflix Rss Recipes*. https://github.com/Netflix/recipes-rss. Accessed: 2017-03-11 (cit. on p. 35).

[OB15] H. Ooghe, S. Balcaen. “Are failure prediction models widely usable? An empirical study using a Belgian dataset.” In: (2015) (cit. on p. 2).

[POVG16] T. Pitakrat, D. Okanovic, A. Van Hoorn, L. Grunske. “An architecture-aware approach to hierarchical online failure prediction.” In: *Quality of Software Architectures (QoSA), 2016 12th International ACM SIGSOFT Conference on*. IEEE. 2016, pp. 60–69 (cit. on p. 16).

[RGP08] P. P. Rodrigues, J. Gama, J. Pedroso. “Hierarchical clustering of time-series data streams.” In: *IEEE transactions on knowledge and data engineering* 20.5 (2008), pp. 615–627 (cit. on p. 26).

[RVR+08] I. Rojas, O. Valenzuela, F. Rojas, A. Guillén, L. J. Herrera, H. Pomares, L. Marquez, M. Pasadas. "Soft-computing techniques and ARMA model for time series prediction." In: *Neurocomputing* 71.4 (2008), pp. 519–537 (cit. on p. 24).

[SFS12] D. Shue, M. J. Freedman, A. Shaikh. "Performance Isolation and Fairness for Multi-Tenant Cloud Storage." In: *OSDI*. Vol. 12. 2012, pp. 349–362 (cit. on p. 26).

[SLM10] F. Salfner, M. Lenk, M. Malek. "A survey of online failure prediction methods." In: *ACM Computing Surveys (CSUR)* 42.3 (2010), p. 10 (cit. on pp. 2, 10, 11).

[Sol02] S. Soltani. "On the use of the wavelet decomposition for time series prediction." In: *Neurocomputing* 48.1 (2002), pp. 267–277 (cit. on p. 25).

[Sve11] R. Svenningsson. *Model-Implemented Fault Injection for Robustness Assessment*. QC 20111205. 2011 (cit. on pp. 18, 19).

[Thö15] J. Thönes. "Microservices." In: *IEEE Software* 32.1 (2015), pp. 116–116 (cit. on p. 6).

[XXHW13] J. Xiao, Z. Xu, H. Huang, H. Wang. "Security implications of memory deduplication in a virtualized environment." In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE. 2013, pp. 1–12 (cit. on p. 6).

[Yan15] C. Yang. "Checkpoint and Restoration of Micro-service in Docker Containers." In: (2015) (cit. on p. 27).

[Zha03] G. P. Zhang. "Time series forecasting using a hybrid ARIMA and neural network model." In: *Neurocomputing* 50 (2003), pp. 159–175 (cit. on pp. 12, 14).

All links were last followed on June 17, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature