Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

PHILIPP EICHHORN
MASTER THESIS

THE UNI1 IMMUNE SYSTEM FOR CONTINUOUS DELIVERY

Submitted on 28 November 2016

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 28 November 2016

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 28 November 2016

Abstract

In this thesis we propose an immune system for the continuous delivery process of the Uni1 application. We add canary deployments and show how continuous monitoring can be used to detect negative behaviour of the application as a result of a recent deployment. Analyzing the Uni1 application is done via user defined health conditions, which are based on a number of metrics monitored by the immune system. In case of degraded behaviour, the immune system uses rollbacks to revert the Uni1 application to the last stable version. With the help of the immune system, application developers do no longer have to manually monitor whether a deployment completes successfully, but instead can rely on the immune system to gracefully handle deployment errors.

Contents

1 Introduction **v**
1.1 Uni1 v
1.2 Goal of this thesis v

2 Conceptual model **2**
2.1 Continuous software development practices 2
2.1.1 Continuous integration 2
2.1.2 Continuous delivery 3
2.1.3 Continuous deployment 3
2.2 Deployment 4
2.2.1 Types of deployments 4
2.2.2 Feature toggles 7
2.2.3 Deployment orchestration 8
2.2.4 Rollbacks 10
2.3 Monitoring 11
2.3.1 Data collection 11
2.3.2 Data storage 12
2.3.3 Data analysis 13
2.4 The immune system 13
2.4.1 Determining the application health status 13
2.4.2 Performing actions to improve application health 15

3 Architecture and design **16**
3.1 Uni1 application 16
3.1.1 Software stack 16
3.1.2 Deployment setup 18
3.1.3 CI pipeline 19
3.2 Uni1 Immune System 20
3.2.1 Monitoring 20
3.2.2 Deployments 23

4 Implementation **25**

4.1 Software stack 25
4.2 Features 26
4.2.1 Monitoring domain specific metrics 27
4.2.2 Analysis rules 28
4.2.3 Generic graphical user interface 29

5 Evaluation **30**
5.1 Integration 30
5.2 Monitoring 31

6 Related Work **32**

7 Conclusion **33**

Appendices **34**
Appendix A Using the Open Data Service 34
Appendix B The Uni1 immune system GUI 36

References **42**

1 Introduction

1.1 Uni1

This thesis uses the software of the Uni1 startup (http://uni1.de) as a basis for conducting research on the topic of continuous delivery. Uni1 is founded by Prof. Dr. Dirk Riehle of the Open Source Research Group at the Friedrich Alexander University of Erlangen-Nuremberg (FAU), and FAU alumni Matthias Lugert (M.Sc.). The goal of Uni1 is to revolutionize how universities and companies collaborate and conduct business with one another.

Uni1 is made up of a number of different software components. This thesis focuses on the Uni1 market software, which is available at https://app.uni1.de. If you would like access to this platform please reach out to Matthias Lugert (matthias.lugert@uni1.de) for details.

1.2 Goal of this thesis

The goal of this thesis is to improve the process of continuous delivery, by adding an "immune system" to the continuous integration (CI) pipeline of the target application. This immune system, much like the one which can be found in living objects, serves two purposes:

- **Monitoring:** the immune system should be able to detect problems with the target application, for example sales of a webshop dropping below a certain threshold.
- **Countermeasures:** upon discovering problem with the target application, the immune system should deploy countermeasures to solve the problem.

The term target application in this thesis refers to the application which provides the actual business value, whereas the immune system is a meta application for

the target application. In the case of Uni1, the target application is the website hosted at https://app.uni1.de.

The rest of this thesis is structured as follows. Section 2 describes the conceptual model of an immune system, including different types of deployments, how applications can be monitored and what actions can be taken based on the application health status. Section 3 gives a brief overview of the Uni1 application, and analyses the architecture of the Uni1 immune system. Section 4 discusses the implementation of the Uni1 immune system, which is evaluated in section 5. Section 6 lists related works on stabilizing the deployment process and section 7 gives a brief conclusion.

2 Conceptual model

This section gives a brief overview of how the terms continuous integration, continuous delivery and continuous deployment are used in this thesis, before analyzing how an immune system can help stabilize the software deployment process.

2.1 Continuous software development practices

2.1.1 Continuous integration

In his "Continuous Integration" article (Fowler, 2006b), Martin Fowler describes CI as:

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

The focus of CI is on integrating software changes from different developers or machines into a single piece of software, and asserting that this integration happens without problems. Important here is, that CI does not stand for any single technology or specific tool, but rather is a loose term for a collection of practices.

In his book on "Continuous Integration" (Duvall, Matyas & Glover, 2007), Paul Duval describes the bare minimum of a CI system as:

- Integration with a version control system
- A build script
- Some sort of feedback mechanism (such as e-mail), in the case of build errors or failed tests
- A process for integrating source code changes

Most CI system also provide support for automatically running tests when integrating code changes.

2.1.2 Continuous delivery

Drawing a strict line between CI and continuous delivery is not a simple task, as many practices are associated with both terms. In his book "Continuous Delivery" (Humble & Farley, 2011), Jez Humble describes continuous delivery as:

> Continuous delivery provides the ability to release new, working versions of your software several times a day.

The key difference to CI is, that continuous delivery includes generating a deployable artifact (for example a war file for Java applications or an APK file for Android apps) as part of the automated build process. As such continuous delivery can be thought of as a superset of CI.

Common practices which are generally part of continuous delivery are:

- **Deployment pipleline:** an extension to the CI pipeline which allows any built artifact to be deployed to the target machine / customer at any given time. Triggering this process is done manually.
- **Version management:** in a bare CI system, built artifacts do not always relate to a released version of a software. In an continuous delivery system, each released software version has one (or multiple depending on the environment) artifacts which are generated as part of the deployment pipeline. Keeping track of these artifacts makes is possible to deploy any version of the software at any time, including performing rollbacks in case of problems.

2.1.3 Continuous deployment

Continuous deployment holds all the same values and practices as continuous delivery, with the exception that deployments are no longer triggered manually, but rather after each build of the CI pipeline.

This level of automation may not always be desired and might not even be possible for all types of applications. For example Android apps, which are typically released via the Google Play Store, take several hours before becoming available to end users (Google Inc., 2016a), which makes releasing in intervals less than one day impractical. Another example are websites which are distributed via global content delivery networks (CDN), where pushing changes to edge caches of the

network can take up a considerable amount of time. Amazon CloudFront, a CDN service offered by Amazon Web Services (AWS), states that "propagation to all edge locations should take less than 15 minutes" (Amazon Web Services, Inc., 2016b).

2.2 Deployment

Any system that wishes to strengthen the continuous delivery process, requires knowledge of how the continuous delivery pipeline is constructed, and in particular how the target application is deployed. This section presents and compares different types of deployments.

2.2.1 Types of deployments

Full rollout

Probably the most simple and straight forward way to deploy an application is to simply release the software to all customers all at once. In the context of a static website, this means updating all HTML, CSS and JavaScript files and possibly invalidating CDN or browser caches.

The advantages of this approach are:

- **Simplicity:** implementing a full rollout pipeline is very straight forward in most cases, as there is always only a single active application version at any given time. This also makes reasoning about the application state and tracing errors easier, compared to a scenario where multiple application versions are involved.
- **Speed:** deployment of a new application version to all customers is as fast as can be. This is important when fixing critical bugs in the application.
- **Consistency:** if multiple and different application versions have simultaneous write access to the same application state, this state is bound to be modified differently from one version to another. If an application version has access to the state of another application version, which for example might be the case with a single shared user database where one version has renamed a table column, the application requires explicit checks for how it should process this state. Single version environments don't need this kind of logic (ignoring application errors and failed version transitions).

Downsides of a full rollout are:

- **Fault tolerance:** should a software version contain a critical bug, then this bug will be deployed to all customers. Even in case the bug is discovered early on, interrupting a running deployment is not always possible and has to be explicitly supported, otherwise leaving the application state in a potential inconsistent state.
- **Flexibility:** with this all or nothing approach, testing a different version of software, for example via AB testing (section 2.2.1), requires to either have a copy of the target application running a different version, or the usage of feature toggles (section 2.2.2).
- **Upgrade downtimes:** upgrading all servers of an application all at once, will lead to the application becoming unresponsive during the deployment process. One solution to this problem is often referred to as Blue Green Deployment (Fowler, 2006a). With Blue Green Deployments an environment running the old version has to be created first, which handles all requests while the deployment is running. After a successful upgrade, the two environments have to be swapped "atomically" (logically, not necessarily on the machine processor level) for the new version to receive incoming requests. Depending on the number of machines that make up an application environment and the duration and frequency of a deployment process, having a clone environment can cause noticeable additional costs.

Incremental rollout

An incremental rollout differs from a full rollout only marginally, in the sense that when upgrading software on the target machines (servers, customer devices, etc.), not all machines are updated at once, but rather one after the other. This process only really applies to scenarios where the business has control over the target machines, such as backend servers. A counterexample are mobile applications, where the user of the target device has full control over when and if an update should be processed or not. This freedom of the user doesn't completely eliminate incremental rollouts in mobile scenarios, the Google Play Store for Android applications for example has explicit support for incremental rollouts (Google Inc., 2016b), but it does make the process more indeterminate and hard to properly control. We focus on scenarios with full control over target machines.

For the most part the advantages and disadvantages are the same as with full rollouts, with the following differences.

Advantages:

- **No upgrade downtimes:** this is the primary difference and advantage over full rollouts.

Disadvantages:

- **Inconsistency:** incremental rollouts do away with the idea that a deployment should be an atomic operation, meaning that application environments can now be in a state somewhere between two versions. This introduces the previously mentioned potential for inconsistencies in the application state, which have to be handled on the application level.
- **Error handling:** with full rollouts, the actual transition from one version to another is a simple matter of switching two environments, something that for example AWS BeanStalk supports naturally, by swapping the CNAME of two AWS BeanStalk environments (Amazon Web Services, Inc., 2016a). With incremental rollouts, the switch from one version to another is no longer atomic, but rather continuous, as more and more machines upgrade to the new version. The deployment pipeline needs to explicitly handle errors that can occur when a machine fails to upgrade. There are multiple options how to handle errors, for example discarding the old machine and launching a new instance instead (when using virtual machines), disconnecting the machine from the outside network (for example by removing it from an AWS Load Balancer), or attempting a rollback should the new application version as a whole haven been declared faulty.

There are a number of variations of the incremental rollout pattern. AWS Elastic Beantalk supports additional deployment modes called **rolling** and **rolling with additional batch**. Both modes introduce the concept of deploying fixed size batches of machines instead of single machines. Rolling with additional batch will launch an additional batch of machines prior to the first deployment, which allows the target application to run at full capacity even during the deployment process.

Canary release

In his article about "CanaryRelease", Danilo Sato (Sato, 2014a) describes canary releases as:

> Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.

Because a canary release only gradually deploys a new application version to machines, it can be thought of as another variation of incremental rollout, with the primary difference that the time between upgrading individual (batches) of machines is intentionally kept long enough for the effects of the new version to

be measured.

There are different strategies for routing users to a new version, the simplest one being random selection. More sophisticated approaches include showing the new version to employees in the own company first, or selecting users based on their profile.

Canary releases come with all the up and downsides of regular incremental rollouts, including the following advantages:

- **Testing with live traffic:** having a high test coverage is great, testing an application with real users is better, as no amount of carefully constructed tests will ever replace real users interacting with the application.

Additional disadvantages are:

- **Complexity:** canary releases require selecting users for a new version consistently and deterministically. Even a random user selection strategy will have to be advanced enough, to either always pick a user for the new version or never, otherwise risking to show a different version to the user on each visit.

A/B Testing

In their article about "Network A/B Testing", Gui et al. (Gui, Xu, Bhasin & Han, 2015) describe A/B testing as:

> A/B testing, also known as bucket testing, split testing, or controlled experiment, is a standard way to evaluate user engagement or satisfaction from a new service, feature, or product. [...] The goal of A/B testing is to estimate the treatment effect of a new change [...]

A/B testing is very similar to a canary release, except that the focus is on comparing different versions of an application, and not on how to release a software. A/B testing is listed here for the sake of completeness, but will not be further analyzed in this thesis.

2.2.2 Feature toggles

Feature toggles are a technique which can be used with any of the above deployment types, with the goal of adding additional flexibility to the application release and testing process. It its simplest form feature toggles are conditional statements in the code of an application, which can turn on and off certain features of an application.

Hodgson quantifies feature toggles along two dimensions: longevity and dynamism (Hodgson, 2016). Longevity determines how long a feature toggle will be part of the application software, ranging anywhere from a couple of days to forever. Dynamism determines how feature toggles can be triggered, for example at build or runtime.

Hodgson distinguishes between four types of feature toggles:

- **Release toggles:** usually short lived and static. These toggles aid the deployment processing by releasing software into production with features that should not yet be visible to users, and are disabled by default. Once development on a feature is complete, the feature can be toggled and released to users.
- **Experimental toggles:** slightly longer lived than release toggles and can be configured dynamically, usually on a per request basis. These toggles are most often used for A/B testing.
- **Ops toggles:** medium to long lived and configurable at runtime. These toggles are used to control high level aspects of an application, for example by disabling high performance features in case of degraded performance.
- **Permission toggles:** mostly very long lived and highly configurable. These can be used to implement regular user permissions, which control the features that can be accessed by users. For example paid only features could be controlled with these toggles.

2.2.3 Deployment orchestration

So far this thesis has assumed that the target application is a single component, which can be upgraded atomically on a single machine. In reality this is rarely the case; most modern software architectures are made up of multiple, independent components which are developed and updated independently. This is not necessarily a result of the ever increasing popularity of microservices, as even simple seeming applications often consist of a database, frontend in the form of a website or mobile application, and a backend. Whether all components should use the same deployment pipeline is up for discussion,[1] however keeping these components in sync does add another layer of complexity to upgrading the application as a whole. This section analyses how **parallel change** can help cope with that complexity.

[1] In his article about "Microservices", James Lewis suggests that in a microservice architecture each component should have its own independent build and deploy process (Lewis & Fowler, 2014).

The idea of parallel change builds upon the concept of **published interfaces** which "refer[s] to a class interface that's used outside the code base that it's defined in" (Fowler, 2003). In an application with multiple components, any interface which is used to communicate between components is a published interface and hence cannot be changed without updating multiple components. As previously discussed in section 2.2.1, updating all components atomically or in parallel is often times impractical (because of downtimes etc.), or in some cases simply impossible, such as with mobile applications. As a result applications require some sort of deployment orchestration to successfully transition an application from one version to another.

Danilo Sato describes parallel change as (Sato, 2014b):

> Parallel change, also known as expand and contract, is a pattern to implement backward-incompatible changes to an interface in a safe manner, by breaking the change into three distinct phases: expand, migrate, and contract.

In the **expand phase** an interface is extended with additional methods / endpoints, which present how the interface should look like after the transition. In the **migration phase** all *client components* of the interface are updated to use the new methods / endpoints of the interface. Finally in the **contract phase** the old, and now unused, methods / endpoints of the original interface are removed. What's important about this process is, that it is time independent. The contract phase can be postponed indefinitely should client components or machines take a long time to update or even never update at all. If the contract phase never occurs, the original interface will have deprecated methods / endpoints which can be supported for as long as required.

Parallel change can be used to roll out updates to all components of an application. The following is based on a sample application which consists of an frontend in the form of a website and a backend. The frontend communicates via an interface with the backend. Figure 2.1 shows how a breaking change to the backend interface can be deloyed, by first expanding the backend interface to support both the old and new endpoints, then migrating the frontend to only use the new endpoints, and finally by contracting the backend interface to only support the new endpoints.

Branch by abstraction (Fowler, 2014) is a technique for gradually introducing large scale breaking changes, which is similar to parallel change, with the exception that an interface is first encapsulate in an abstraction layer which supports the breaking changes, before changing the underlying logic.

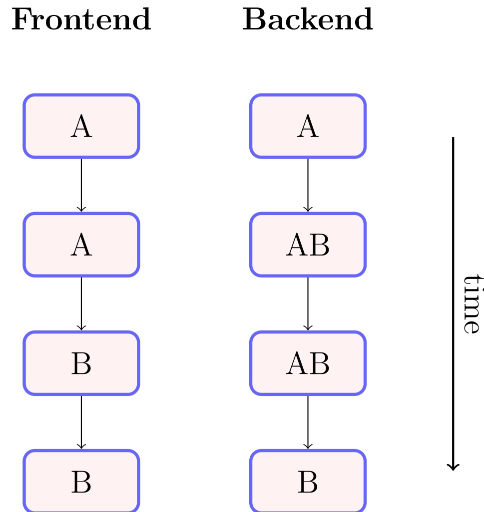

Figure 2.1: Diagram of introducing breaking changes in the interface of an application that consists of a frontend and backend. Each box represents a deployed version of an application component, with A being the initial interface, B the final interface, and AB a version that supports both interfaces A and B.

2.2.4 Rollbacks

Regardless of how software is deployed, there is always the potential for a deployment to fail, for example by failing to restart a service after an update, or by displaying unwanted and potential critical behaviour. Once such a failure has occurred, there are two ways to deal with it: either manually fixing any errors in the application, for example by using SSH to login to a faulty machine and restarting a service manually, or by using the existing pipeline to deploy a different version of the application. We focus on the latter approach of redeploying a previous application version to the environment, which we refer to as **rollbacks**. This section lists the rollback strategies for each deployment type, along with their respective durations.

- **Full / incremental rollout:** simply re-deploys a previous version. Rollback is not instantaneous, the duration depends on the total number of machines in an application environment.
- **Canary release:** users can be routed nearly instantaneously to the stable version.
- **Deployments with feature toggles:** depending on the toggle granularity, shutting down a faulty part of the software can be instantaneous. If a fault cannot be isolated with feature toggles, the rollback strategy depends on how the application was deployed.

2.3 Monitoring

This section discusses how to monitor various parameters of the target application. The result of these monitoring activities will be used in section 2.4 by the immune system to determine the health status of the application. In addition, this section covers some basic mechanisms for constructing a notification system, which can be used by the immune system to be informed about certain conditions, without having to fall back on a polling technique to receive these updates.

Application monitoring consists of roughly three stages: data collection, data storage and data analysis.

2.3.1 Data collection

At this point we do not want to make any assumptions about what data could be relevant to determine the health status of the target application. Instead, we focus on analyzing what kinds of data sources can be tapped and how.

Data sources

We classify the data sources of an application as:

- **Domain specific:** this data can usually be derived from the database of an application and contains domain specific values. Using a two sided marketplace as an example, the number of offered products is a domain specific parameter.
- **User behaviour:** while this information can partially be derived from the application database, it usually stems from a dedicated analytics software such as Google Analytics, which is kept separate from any application logic or data. Examples are the behaviour of users on a website (number of visited pages, total time on website, bounce rate, etc.) or user interaction with newsletters.
- **Machine specific:** data about the state of the machines that run the target application. Sample parameters are the amount of RAM used, CPU utilization or average time to handle a request.
- **Error reporting:** problems with an application are usually logged for later analysis. The number, severity and kinds of errors can be monitored[1].

[1] Google Analytics lists “Crashes and Exceptions” under the category “Behaviour”. We choose to make errors a separate data source, because errors might not always be the direct

- **Finance reporting:** finance related data can sometimes be derived from an application database, but is usually kept separate from the application data due to the sensitive nature of the data.

Accessing data sources

To read and process data sources we propose a system of adapters, where each adapter has specific knowledge about how to access a data source, and then transfers that data into a common format. A simple adapter could depend on polling to fetch data from a source, an advanced implementation should register itself with the data source (where possible) to directly receive updates.

2.3.2 Data storage

To store the data collected from the various data sources for further analysis, the monitoring system requires some form of database. This section does not give an overview of different database management systems (DBMS), but rather lists some of the unique requirements of the monitoring database.

- **Fast write and read operations:** data is written frequently (depending on the sources), and usually read in large batches. Update and delete operations are not required, making the need for transactions obsolete.
- **Retention length:** the immune system in this thesis focuses on the time between deploying an application, and declaring that application version stable enough to not require a rollback. This is a finite time frame, and it is in the interest of all parties involved to keep it as short as possible. As a result the monitoring database does not need to retain the collected data for an indefinite time, but rather only during the time of the deployment.
- **Timestamp support:** each entry in the monitoring database needs to be timestamped for analysis. While not strictly required, explicit support for timestamps and queries based on timestamps is helpful.
- **No entity relationships:** relational database usually support modeling relationships between tables. Entity relationships are not the focus of monitoring and can be excluded for the most part.

result of a user interaction, but could also result from periodic processes.

2.3.3 Data analysis

The monitoring system should not make any assumptions about how to derive the health status of the application from the data it is collecting. Instead, this decision is delegated to the immune system. To support this delegation, and to prevent the immune system from having to query the monitoring system for data at regular intervals, the monitoring system should feature a notification system which supports the registration of analysis rules, which will notify any subscribers in case the conditions specified in those rules are met.

The monitoring system should allow clients to register rules, which perform complex event processing (CEP). In "The Power of Events" (Luckham, 2001), Luckham describes an event as "an object that is a record of an activity in a system". Events are related to another by time. The data collected by the monitoring system can be thought of as a collection of events, each one associated with a single timestamp. Complex events are created by combining multiple events into a single one. As an example, consider the event "the value of the parameter p exceeds the threshold x". In order for this event to happen, two different events need happen first:

- At time t the value of p needs to be below x.
- At time $t + \delta$, with $\delta > 0$, the value of p needs to be above x.

The support for complex event processing in the monitoring system is not strictly required, as subscribes can also split complex events into multiple "simpler" ones as seen above and reconstruct the complex events themselves.

2.4 The immune system

The goal of the immune system is to further stabilize the process of deploying a software version by

- **determining** and monitoring **the application health status** over the duration of the deployment, until the application is considered "stable".
- **performing actions to improve the application health** should the health status decline.

2.4.1 Determining the application health status

We propose two approaches for determining the health status of an application: white box monitoring and black box monitoring.

White box monitoring

With this approach the immune system defines a number of conditions that best describe how a healthy application should behave. These conditions will differ from one application to another and can include domain specific aspects, for example the number of visitors on a particular page of a website.

The advantage of this approach is, that the description “healthy” can be fine-tuned to fit the target application. Additionally the conditions can be adjusted over time or even season to cope with changes.

The primary disadvantage of this approach is, that conditions have to be defined manually and cannot always be applied to applications from different domains. In the case of a business with many applications, this can lead to a fragmented definition of “healthy”.

Black box monitoring

Instead of defining the health status of an application on a software level, which we do with white box monitoring, we can derive the health status from the purpose the application serves for a business. In many cases the ultimate goal of an application is to generate revenue. We can use revenue as a metric to define the health status of an application. An application version is considered unhealthy should it generate no or less revenue than the previous version, and healthy should it generate the same or more revenue.

This simple approach has many drawbacks though, and does not take into account daily / weekly / seasonal fluctuations, the revenue growth gradient, businesses that do not yet generate revenue or non-profit organizations that will never generate any revenue.

Another difficulty with this approach is, that the time to generate a representative amount of revenue limits the the deployment frequency. As an example consider an online car dealership with an average of three sold cars per day. An immune system cannot determine the health status of the shop website with any certainty, if the software is released on a daily basis. Missing revenue can either be the result of a faulty software, or simply be caused by the fluctuations of the business. For the immune system to monitor the health status of the car dealership software, the release frequency has to be either low enough for representative revenue samples to be collected, or a combination of white and black box monitoring could be used.

Health status granularity

Independently of the monitoring approach used, an application is hardly ever completely "healthy" or "unhealthy", instead the actual health status is usually somewhere in between. In this thesis we focus on applications with a binary health status (healthy / unhealthy), a more advanced immune system however should take into account that an application consists of multiple components, which all have their own health status. Additionally, when using white box monitoring, degrading a single metric might be done intentionally to improve several other metrics. For example increasing the number of server machines in an environment, which will increase costs, will reduce latency of client requests and potentially resolve out of memory errors.

2.4.2 Performing actions to improve application health

An immune system is not much more than an advanced monitoring solution, unless it acts upon the information it derives about the target application. The kind of actions that the immune system can perform largely depend on the granularity of the application health status. In the case of this thesis that is "healthy" and "unhealthy", which limits what a monitoring system can do to improve the application health. We focus on rollbacks as a solution to a degraded application health status. If the target application uses feature toggles for releasing new features to customers (in this case called release toggles), the immune system can turn a toggle back off to perform the rollback.

3 Architecture and design

Section 2 introduces a conceptual model for an immune system for continuous delivery. This section applies this model to the case of the Uni1 software, by first giving on overview of the Uni1 software stack and architecture, and then by discussing how to construct an immune system on top of the Uni1 CI pipeline.

3.1 Uni1 application

This thesis focuses on the Uni1 market place software. The purpose of this market place is to create an online community, where members of higher education institutes and companies can come together to organize courses and projects for students. In the following the term Uni1 is used to refer to the market place software.

3.1.1 Software stack

Uni1 uses a classical client / server architecture, where the Uni1 website has the role of the client which interacts with a backend server. The two components are strictly separated and can be developed and deployed independently. Communication between the two components is handled via a RESTful API.

Frontend

The Uni1 frontend is a so called **static website**, which means that all files required to host the website (HTML, CSS, JS) are generated at compile time. Dynamic content is created at runtime via JavaScript in the browser of the user. Hosting a static website can be achieved via any regular file server, such as

Apache HTTPD[1], AWS S3[2] or Netlify[3]. PHP or NodeJS / Express applications are examples for applications where the website is dynamically constructed on the server for each request.

The software stack of the Uni1 frontend consists of a ReactJS [4] application with Redux[5] for managing the application state. ReactJS is a JavaScript framework originally designed by Facebook, for developing rich single page applications[6]. ReactJS is well known for its ability to create highly modular components which can easily be reused, and for its ability to build declarative interfaces which automatically update whenever the underlying application state has changed. Redux is a small framework on top of ReactJS, that advocates a stricter way of organizing the application state, and defines how user interactions should trigger changes in the application state.

Backend

The Uni1 backend consists of a NodeJS / ExpressJS[7] application with MongoDB[8] as a database. NodeJS is a Javascript framework for writing server side applications, and ExpressJS is a routing library used to built the RESTful API on top of NodeJS.

Landing page

The website at http://uni1.de is not to be confused with the previously discussed application. The software running behind uni1.de is a Wordpress installation, serving as a landing page for the Uni1 startup, but otherwise does not interact with the Uni1 marketplace.

[1] https://httpd.apache.org/

[2] https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html

[3] https://www.netlify.com/

[4] https://facebook.github.io/react/

[5] https://github.com/reactjs/redux

[6] The term "Single page application" refers to a JavaScript web application, where the client browser typically only loads a single HTML file with a reference to the main Javascript application. Any further interactions with the website, including navigating to different parts of the application, are handled in JavaScript. Single page applications are frequently combined with the pattern of static pages for their ease of deployment.

[7] https://nodejs.org/en/ and https://expressjs.com/

[8] https://www.mongodb.com/

3.1.2 Deployment setup

This section discussed how the Uni1 application is hosted, and what other external services are required by the application.

Frontend

The Uni1 frontend is hosted in an Amazon Simple Storage Service bucket with public read permissions (Amazon S3). Amazon S3 is a storage solution provided by AWS. One limitation of static single page applications is, that because there is only a single `index.html` at the root of the application, accessing any other page of the application will result in a 404 (not found) HTTP status code. To overcome this limitation, the S3 bucket redirects all 404 requests to the root `index.html`, with the original path as a query parameter. It is then the responsibility of the JavaScript application to read this query parameter and navigate to the appropriate page[1].

In order to enable SSL on the domain app.uni1.de, traffic to and from the AWS S3 bucket is routed through AWS CloudFront, the AWS content delivery network (CDN)[2]. This is due to the limitation of AWS S3, where S3 buckets can have a custom domain name, but not with SSL support. AWS CloudFront on the other hand offers free SSL certificates with custom domain names.

Backend

The Uni1 backend is hosted on AWS Elastic Beanstalk[3]. AWS Elastic Beanstalk is a free supporting service offered by AWS, which provides a simple interface for hosting backend applications on AWS. AWS Elastic Beanstalk does not host applications itself, but rather combines a number of other AWS resources in a convenient and intuitive way, such as AWS virtual machines (EC2[4]), which come preconfigured with auto scaling groups and load balancers. The description "free" only applies to the usage of AWS Elastic Beanstalk itself. Other AWS resources, such as EC2, are billed to the customer as usual.

AWS Elastic Beanstalk is used to host the RESTful API of the Uni1 backend. Other parts of the application, such as resizing uploaded images, are hosted on

[1] This behaviour can be observed with "the naked eye", by directly requesting any path within the Uni1 application (other than the root `index.html`). The URL in the address bar of the browser will change a couple of times before settling on the requested URL.

[2] https://aws.amazon.com/cloudfront/

[3] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html

[4] https://aws.amazon.com/ec2/

AWS Lambda[1]. The AWS documentation describes AWS Lambda as

> AWS Lambda is a compute service where you can upload your code to AWS Lambda and the service can run the code on your behalf using AWS infrastructure. After you upload your code and create what we call a Lambda function, AWS Lambda takes care of provisioning and managing the servers that you use to run the code.

The Uni1 backend uses AWS Lambda for executing code on events that originate from within the AWS ecosystem, for example for resizing an image after it has been uploaded to S3. Future work on the Uni1 application might include migrating the RESTful API from AWS Elastic Beanstalk to AWS Lambda, as this eliminates the need to maintain EC2 servers, and in the case of irregular or low traffic numbers, can reduce costs significantly by only paying for code execution time with AWS Lambda[2].

Other services used by the Uni1 backend are Mailgun[3] for sending transactional emails, MailChimp[4] for managing newsletters, and mLab[5] for hosting the MongoDb database.

3.1.3 CI pipeline

The Uni1 frontend and backend use a continuous integration pipeline for running tests after each commit. Deployments are started manually, the deployment itself however is automated.

The CI pipeline begins with the Uni1 software being hosted in a private repository on GitHub[6]. After each git push to the repository, GitHub notifies TravisCI[7], a continuous integration as a service provider, about the new changes. TravisCI then builds and tests the Uni1 software, and alerts involved parties in case of errors.

Deploying a new version of the Uni1 application is done manually, and involves the following steps:

[1] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[2] This pattern is called **serverless architecture**. Due to recent popularity and the release of AWS Lambda to the general public in April of 2015, a number of frameworks have evolved which try to port legacy (NodeJS) applications to this serverless architecture. Noteworthy mentions are Claudia.JS (https://claudiajs.com/) and serverless (https://github.com/serverless/serverless)

[3] https://www.mailgun.com/

[4] https://mailchimp.com/

[5] https://mlab.com/

[6] https://github.com/

[7] https://travis-ci.com/

- **Frontend:** uploading all HTML, CSS and JS files to the Amazon S3 bucket.
- **Backend:** sending a bundled ZIP package of all relevant files to the AWS Elastic Beanstalk application.

3.2 Uni1 Immune System

This section describes the high level architecture of the Uni1 immune system. Roughly speaking the immune system can be divided into two parts:

- **Monitoring:** how the immune system determines the health status of the application. We discuss a number of metrics that are specific to the Uni1 application.
- **Deployments:** we propose a canary deployment model for the Uni1 application. This part of the immune system is responsible for managing the Uni1 canary and provides a simple interface for deploying new versions of the Uni1 application.

Both parts of the immune system have a frontend in the form of a website.

3.2.1 Monitoring

Before undertaking any actions to modify the behaviour of the Uni1 application, the immune system first needs to assess the health status of the application.

Uni1 metrics

At the time of writing this thesis, the Uni1 startup is still very young and does not yet[1] have a large enough user base or regular income to support **black box monitoring**. This section consequently uses **white box monitoring** to determine the application health status. The following metric types are analyzed:

- **Domain specific:** due to the small number of users, metrics in this category largely serve the purpose of providing "sanity checks", for example that the total number of courses created on the platform should never decline. While this does not give an accurate and detailed picture of the

[1] Go, Uni1, go!

application health status, it does help to assert a level of basic fitness of the application[1].

- **Machine specific:** every application hosted on AWS Elastic Beanstalk is preconfigured to export EC2 performance data to Amazon CloudWatch[2]. Amazon CloudWatch is a monitoring solution for AWS resources, but also features an API for importing data from other sources. The Uni1 immune system uses this EC2 data to detect abnormal behaviour of the application, by defining a range for key metrics, such as CPU and memory consumption, which are considered to be "healthy".

Configuration

In order to cope with changing definitions of what constitutes a healthy Uni1 application, the monitoring service contains two interfaces for configuration: a static configuration file which lists the metrics that should be monitored, and a dynamic set of rules that decide how the metric values should be interpreted.

- **Metrics configuration:** The Uni1 immune system uses a static configuration file[3] to define which metrics of the Uni1 application should be monitored, and how. The contents of this file are used to configure adapters for fetching metric data. Typical configuration parameters for a metric are: the location of the data source (usually an URL), the protocol used for accessing it (HTTP, HTTPS, FTP, etc.), format of the data (JSON, XML, etc.) and the frequency in which this data source should be accessed.
- **Analysis rule configuration:** The immune system delegates the decision about what makes a healthy application to the administrator of the immune system, by providing an interface for registering rules for determining the health status. This interface is similar to that of a CEP rule engine, where each rule can analyze multiple metrics from different timestamps before it triggers an event.

Data storage and analysis

The immune system stores the collected monitoring data in Amazon CloudWatch. Amazon CloudWatch supports this process by letting users defined custom metrics, and by providing an interface for publishing data to those metrics. Each

[1] One could argue that these kinds of metrics could also be covered with regular unit and integration tests. We however argue, that monitoring such values in production can provide an additional level of security which simply cannot be achieved in a regular testing environment.

[2] https://aws.amazon.com/cloudwatch/

[3] A configuration file which is only read once when starting the immune system.

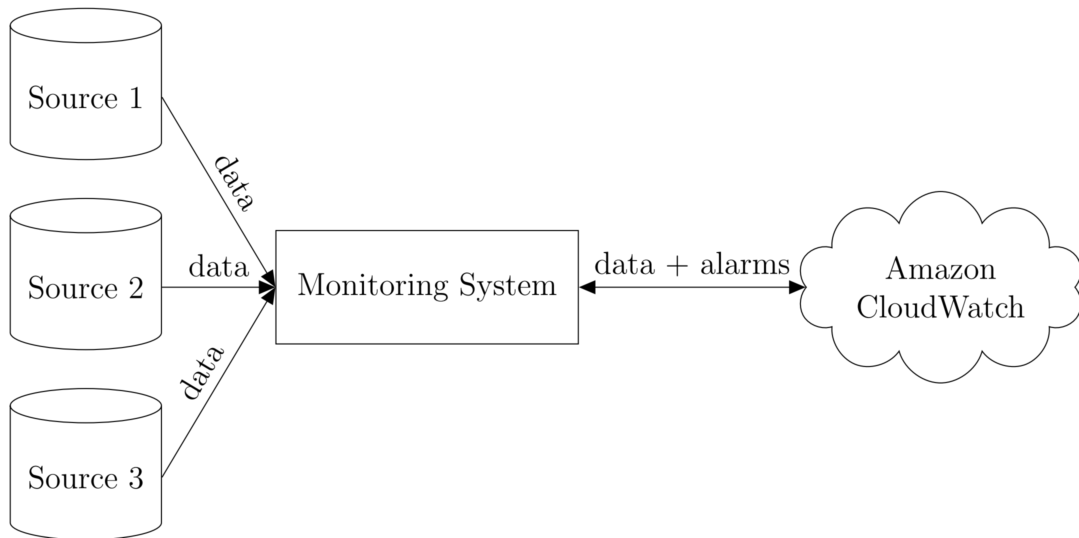

Figure 3.1: This diagram shows a high level overview of the monitoring architecture, with three configured metrics which all come from different data sources. Data flows from the sources, through the immune system into Amazon CloudWatch. To monitor these metrics, clients of the monitoring system can register analysis rules with the monitoring system, which trigger whenever the conditions describes in those rules are met. The monitoring system registers these rules with Amazon CloudWatch in the form of Amazon CloudWatch alarms.

published data point requires the following attributes: a value, a unit and a timestamp from when it was recorded.

Besides acting as a storage solution for the immune system, Amazon CloudWatch is also used for analyzing the recorded data. Administrators of the immune system can define the characteristics of a healthy Uni1 application, by uploading a set of conditions to the immune system. Those conditions define valid ranges for the monitored metrics. If any condition is not met for an extended period of time, the monitoring system will consider the health status of the Uni1 application degraded, and will alert any clients of this change.

The immune system allows clients to register these conditions at runtime. After registration, the conditions are transformed into Amazon CloudWatch alarms[1], which are then registered with Amazon CloudWatch. The immune system periodically checks the state of the alarms on Amazon CloudWatch, and will forward any triggered alarms to its clients when necessary.

Figure 3.1 shows an overview of how data is processed in the monitoring service.

[1]https://aws.amazon.com/blogs/aws/amazon-cloudwatch-alarms/

3.2.2 Deployments

This part of the immune system extends the Uni1 CI pipeline to store created assets, to setup canary releases of the Uni1 application and to manage how new versions are deployed to canary and production machines.

CI pipeline

Prior to the introduction of the immune system, the Uni1 CI pipeline was only responsible for building and testing the application, but did not store created artifacts for future use. The immune system extends the CI pipeline, by uploading all build artifacts to S3 as the last step in the pipeline. Artifacts are named according to the git commit hash, in case a reference back to the original source code is required.

Canary releases

In AWS Elastic Beanstalk, each **application** consists of one or more **environments**, where each environment is an actual instance of running software, which can be addressed via a unique URL. One use case of multiple environments in a single application is to create separate production and testing setups, that are used in different stages of the software development cycle. The Uni1 immune system leverages this feature to create a clone of the original Uni1 environment, to be used for canary deployments.

Managing the deployment process

When deploying a new version of the Uni1 application via the immune system, the following steps are executed:

1. **Deploy artifacts to canary environment:** the immune system will download the build artifacts from S3 and create a new version in the canary environment. The immune system only triggers the deployment to the canary environment, but does not wait for its completion. Instead, it will continue to step 2 as soon as possible.

2. **Start monitoring canary environment:** the immune system will periodically check with the monitoring system to get the health status of the canary environment. The duration of the monitoring period is configurable and should be short enough to release a new version to the customer quickly, but long enough to get a representative sample of how the application is

performing. After the monitoring period is over, the immune system will continue to step 3. Should the monitoring service report a degraded health status in this period, the immune system will proceed to step 4 instead.

3. **Deploy artifacts to production environment:** similar to how artifacts are deployed to the canary environment, the immune system will now deploy the build artifacts to the production environment.

4. **On error, rollback canary environment:** in case the monitoring service reports an unhealthy canary environment, the immune system will regard the deployment as failed, and will redeploy a previous application version to the canary environment.

4 Implementation

This section gives a brief overview of the technologies used to build the Uni1 immune system and discusses a number of implementation decisions.

4.1 Software stack

The Uni1 immune system consists of two separate components: a backend featuring a RESTful API, and a frontend in the form of a static website.

Backend

The backend is a Java application based on the Dropwizard[1] framework. Dropwizard is similar to other Java backend frameworks, for example the Spring framework[2], in the sense that it tries to provide all relevant building blocks for modern backend applications, such as support for RESTful APIs, request validation, request authentication or integration with an object-relation manager (ORM). Unlike Spring however, Dropwizard does not implement all of these features itself, but rather uses existing open source libraries[3], and only provides the necessary "glue" to make all these libraries work in harmony.

Additional services used by the backend are Amazon DynamoDb as a NoSQL database[4], Amazon CloudWatch for storing and processing the monitoring data, and AWS Simple Notification Service[5] for receiving push notifications whenever alarms on Amazon CloudWatch are triggered.

[1] www.dropwizard.io

[2] https://projects.spring.io/spring-framework/

[3] Standard Dropwizard libraries are Jetty (HTTP server), Jersey, (RESTful web framework), Jackson (JSON library), Metrics (metrics library), Guava (utility library), Logback (logging framework) and Hibernate Validator (bean validation support).

[4] https://aws.amazon.com/dynamodb/

[5] https://aws.amazon.com/sns/

Frontend

The immune system frontend is a static website based on a ReactJS / Redux architecture, and uses the RESTful API of the backend for communication. The website is written using the ECMAScript 6 standard[1], uses ESLint for linting[2] and Semantic UI[3] as the CSS framework.

4.2 Features

This section covers implementation details of the Uni1 immune system.

Canary release configuration

One key characteristic of applications with canary releases is, how users are routed to the canary environment. Conceptually there are three ways of doing this:

- **Application based routing:** the logic of how users gets routed to which environment is part of the application itself. This approach provides maximum flexibility, but comes at a performance cost, such as additional roundtrips when requests are forwarded to the target environment.
- **Network based routing:** the network layer is responsible for routing users to the correct environment. This has the benefit of coming with little to no performance costs (a request needs to pass through the network layer in any case), but limits the routing logic to information which is publicly available in the request, such as IP address of the sender.
- **DNS based routing:** even more performant than network based routing, this approach lets the DNS provider decide how to route clients, by mapping the domain name to different environments depending on the DNS lookup request. While this makes implementing routing very easy with DNS providers such as Amazon Route53[4], it ultimately makes the DNS client (browser, mobile application, etc.) responsible for correctly routing users to the appropriate environment. This can lead to inconsistent behaviour should the client not honor the DNS settings. In their paper titled "On the Responsiveness of DNS-based Network Control", Pang et al. (Pang, Akella, Shaikh, Krishnamurthy & Seshan, 2004) discovered that

[1] http://es6-features.org/
[2] http://eslint.org/
[3] http://semantic-ui.com/
[4] https://aws.amazon.com/route53/

up to 47% ob web clients did not adhere to the DNS time-to-live (TTL) settings.

Because we do not wish to modify the Uni1 application to accommodate the immune system, and because reconfiguring the network layer of the AWS Elastic Beanstalk application would ultimately require the Uni1 application to use plain EC2 instances without the Elastic Beanstalk environment, the immune system uses a DNS based routing approach for managing the canary environment.

The Uni1 application uses Amazon Route53 as its DNS provider. Amazon Route53 supports three DNS routing policies which are relevant to canary environments[1]:

- **Simple Routing Policy:** this is the default policy and maps a domain name to a single target address. With this policy the production and canary environment have distinct endpoints, which makes it the responsibility of the Uni1 business to manually distribute users between the two environments.
- **Geolocation Routing Policy:** this policy is typically used for performance reasons, by routing users to servers which are closest to them. The policy allows multiple target addresses to be associated with a single domain name, and each target address to be associated with a location. Routing is then based on the location of the incoming request.
- **Weighted Routing Policy:** with this policy, multiple target addresses can be associated with a single domain name. Each target address can have a weight assigned, which determines the likelihood of a request being routed to that address. When applied to canary releases, this approach essentially implements a random distribution of clients between the canary and production environment, where the ratio between the two groups can be configured.

The Uni1 immune system uses a weighted DNS approach for routing users to the canary and production environments, where 1% of users are being shown the canary environment.

4.2.1 Monitoring domain specific metrics

The two types of metrics monitored by the immune system are domain specific and machine specific ones. AWS Elastic Beanstalk takes care of exporting any EC2 and load balancer specific data to CloudWatch, which leaves the domain specific metrics to be handled by the immune system. There are a number of

[1] https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy.html

strategies to access domain specific data of a target application, such as directly reading from the application database or using existing interfaces. We propose a third solution by implementing a "monitoring endpoint" in the Uni1 application, which returns a mixed set of domain data points. This interface is a RESTful API which returns data in JSON format. Listing 1 shows a sample output of this endpoint.


```
1   {
2       "dbConnected": true,
3       "courseCount": 10,
4       "projectRequestCount": 42,
5       "projectCount": 0,
6       "users": {
7           "companyUserCount": 31,
8           "eduUserCount": 17,
9           "studentUserCount": 0
10      },
11      "timestamp": 1479822696306
12  }
```


Listing 1: This JSON data is returned by the Uni1 monitoring endpoint. The timestamp defines when this data was last generated, in case a cached result is returned by the Uni1 application.

The immune system periodically queries this endpoint and forwards the data to Amazon CloudWatch.

4.2.2 Analysis rules

To configure how the health status of the Uni1 application is determined, the immune system has an interface for registering "health conditions", which define valid ranges for the metric values of the Uni1 application.

The immune system supports two kinds of health conditions:

- **Single threshold:** this condition compares the value of a metric with a single threshold. The comparison operators are greater, greater than, lesser, lesser than and equal to.
- **Range threshold:** this condition defines two thresholds, which determine the valid range of a metric value. The range can either be inclusive (the metric should be within the range) or exclusive (the range should be outside the metric).

Internally the immune system converts these conditions to Amazon CloudWatch alarms. Single threshold conditions can be mapped directly to a Amazon CloudWatch alarms, where range thresholds require two alarms to be created. In order reconstruct the mapping between conditions and alarms, the immune system uses Amazon DynamoDb for storing meta data about conditions. Amazon SNS is used to receive push notifications whenever the state of an alarm on CloudWatch has changed, by implementing an API endpoint in the immune system which is called by SNS with the ID of the alarm and its current state.

4.2.3 Generic graphical user interface

The user interface of the immune system has three main pages:

- **Deploying applications:** this UI shows an overview of all build artifacts that were generated as part of the Ci pipeline, which of those artifacts are currently deployed to what environment, and controls for deploying an artifact.
- **Monitoring the canary health status:** for each metric configured in the immune system, a graph with the latest measurements is show. This UI is dynamically created at runtime, using an endpoint of the backend which lists all configured metrics.
- **Defining health conditions:** this enables administrators to configure health conditions for the Uni1 application.

A visual overview of the UI is shown in appendix B.

5 Evaluation

This section evaluates how the immune system can be integrated with other applications, and the strengths and weaknesses that come with the monitoring solution of the Uni1 immune system.

5.1 Integration

The immune system we propose is heavily dependent on the AWS ecosystem. While the concepts of the immune system can easily be ported to different hosting environments, the software itself is specifically built for applications using AWS Elastic Beanstalk.

Because Elastic Beanstalk applications are highly standardized, and because the immune system does not require the target application to be modified, the Uni1 immune system can be regarded as a "plug and play" immune system for any application hosted on Elastic Beanstalk. This is especially interesting for businesses that wish to provide a basic guard against failed deployments, something which Elastic Beanstalk does not provide at this point[1].

At the same time the tight integration with Elastic Beanstalk makes it difficult to extend the immune system beyond what AWS supports. For example the data retention length of Amazon CloudWatch is limited to two weeks[2]. This is sufficient for monitoring canary deployments which are expected to last only a couple of days. Should be monitoring system be extended to continuously monitor the production environment as well, CloudWatch would no longer be suitable as a data storage. Other limitations include the policy for routing users to the canary environment, how deployments are performed within an environment or extracting machine specific metrics which are not provided by Elastic Beanstalk.

[1] AWS Elastic Beanstalk requires customers to manually monitor the status of the deployment, and in case of errors, to manually start a rollback. See https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.deploy-existing-version.html for details.

[2] https://aws.amazon.com/cloudwatch/faqs/

5.2 Monitoring

The Uni1 immune system uses a **white box monitoring** approach for determining the health status of an application, and requires the administrator of the immune system to configure conditions that define a healthy application. This approach provides maximum flexibility and even makes short term adjustments possible. In addition, domain specific metrics allow for a very high level, business driven definition of how the application should behave, which might even be considered a domain specific language (DSL).

The cost of this configuration freedom is the required knowledge to configure these health conditions. Misconfiguration can lead to accidental rollbacks with negative user experience. Depending on how deployment orchestration of multiple components is handled, an unexpected rollback can also introduce new and very real errors in the application, for example by rolling back the backend to a version which is incompatible with the frontend.

Another limitation of this approach is how difficult it is to apply to applications with low traffic volumes. Because of the limited data available on how the application behaves under load, administrators are not able to experiment how different health conditions affect the deployment process. Instead when traffic volumes increase, the application might perform very differently than it did with low traffic volumes.

6 Related Work

An alternative approach to using an immune system for deployments is, to first deploy a new application version to a testing environment, which ideally is an exact clone of the production environment. This environment can then be used to run extensive integration tests. Neely describes this approach in his work about “Continuous Delivery” (Neely & Stolt, 2013).

Facebook uses a similar method, but additionally requires the developers of the changes which are about to be deployed to be on call during deployment. To verify the availability of the engineers, an IRC bot will contact the engineer before the deployment, and should the engineer not be reachable, will automatically revert all changes of that engineer (Feitelson, Frachtenberg & Beck, 2013).

The term “immune system” in computer science is most often associated with dynamically detecting and fixing security vulnerabilities in applications. Sathre proposes rolling back a software to the last know uncompromised state in the case of security breach (Sathre & Zambreno, 2008). “Immune system” can also be used to describe a system which tries to mask runtime errors caused by mistakes made during development. Sidiroglou et al. present such a system for the x86 platform in their article “Building a reactive immune system for software services” (Sidiroglou, Locasto, Boyd & Keromytis, 2005).

We have not found any work on detecting and fixing problems caused by deployments without the need for manual intervention. We believe the Uni1 immune system to be unique in this regard.

7 Conclusion

This thesis discusses a number of models for strengthening the software deployment process, by continuously monitoring the application under development, and deploying countermeasures in case of degraded application performance. As an example we study the Uni1 immune system, which is used to deploy the Uni1 software. The immune system extends the Uni1 CI pipeline by introducing canary deployments and a monitoring service, which tracks the state of the canary environment. The immune performs automated rollbacks on the canary environment should the health status decline. A user defined set of conditions is used by the immune system to determine the health status of the Uni1 application.

Appendix A Using the Open Data Service

The immune system we propose consists of roughly two core components: a **monitoring system** and a **deployment component**. The three primary objectives of the monitoring system are data collection, data storage and data analysis. Data storage and data analysis is handled by Amazon CloudWatch, which leaves data collection to be implemented by the monitoring system. This section discusses how the Open Data Service (ODS), developed by the Open Source Software Research Group in Erlangen, can be used as a replacement for the monitoring system.

Data collection

The ODS features a fully customizable adapter framework for fetching data from various sources, which is far superior and more flexible than that of the immune system. Migrating the immune system to use the ODS would require the ODS to support more advanced authentication protocols required AWS services, but otherwise does not require any code changes. Data sources in the ODS can be configured dynamically via a RESTful API, as long as the data source uses an HTTP or FTP protocol, and returns the data in JSON or XML format.

Data storage

The ODS relies on Apache CouchDb[1], a document oriented NoSQL database, for storing data. This means that any data regardless of the data schema can be stored. If the immune system monitors data source with heterogeneous schema and stores this data as is in the ODS, it becomes the responsibility of the immune system to deal with these possibly conflicting schema. The adapter framework of the ODS can help with this problem, by configuring adapters which perform schema transformation. This logic is hard coded in the immune system; the ODS allows configuration of these transformations at runtime.

Data analysis

The ODS itself only allows clients to fetch data as is from CouchDb. In order to perform more advanced analysis, the complex event processing service (CEPS) was developed at the Open Source Software Research Group, which nicely integrates with the ODS. This solves the need of the immune system to dynamically

[1] https://couchdb.apache.org/

register health conditions, but does not provide the same range of statistical analysis tools which are offered by Amazon CloudWatch.

Appendix B The Uni1 immune system GUI

This section gives a brief visual overview of the Uni1 immune system graphical user interface. The UI is roughly divided into three pages: deployments, monitoring metric values and a health conditions editor.

Deployments page

Figure 7.1 and 7.2 show the deployment page. At the top of the page, the UI shows an overview of which version, referenced by the shortened git commit hash, is currently deployed to which environment, followed by a high level health status taken from AWS Elastic Beanstalk.

Below that, figure 7.2 shows a list of all build artifacts created by the CI pipeline, as well as an option to deploy those artifacts to the canary environment. The "bug" icon indicates that this version is currently deployed to the canary environment, the "rocket" icon represents the production environment.

Monitoring metric values page

Figures 7.3 and 7.4 show current and past values of the configured metrics, divided into the categories domain specific and machine specific metric types. This part of the UI is dynamically created at runtime from the metric configuration of the immune system backend.

Health conditions editor page

This page contains the editor for creating and modifying health conditions which should be monitored by the immune system. Figures 7.5 and 7.6 show how different kinds of health conditions can be configured. Figure 7.7 shows which conditions are currently enabled on the system and their current status (green indicating healthy values, red indicating unhealthy values).

The immune system UI uses the term "monitoring alarms" to refer to health conditions.

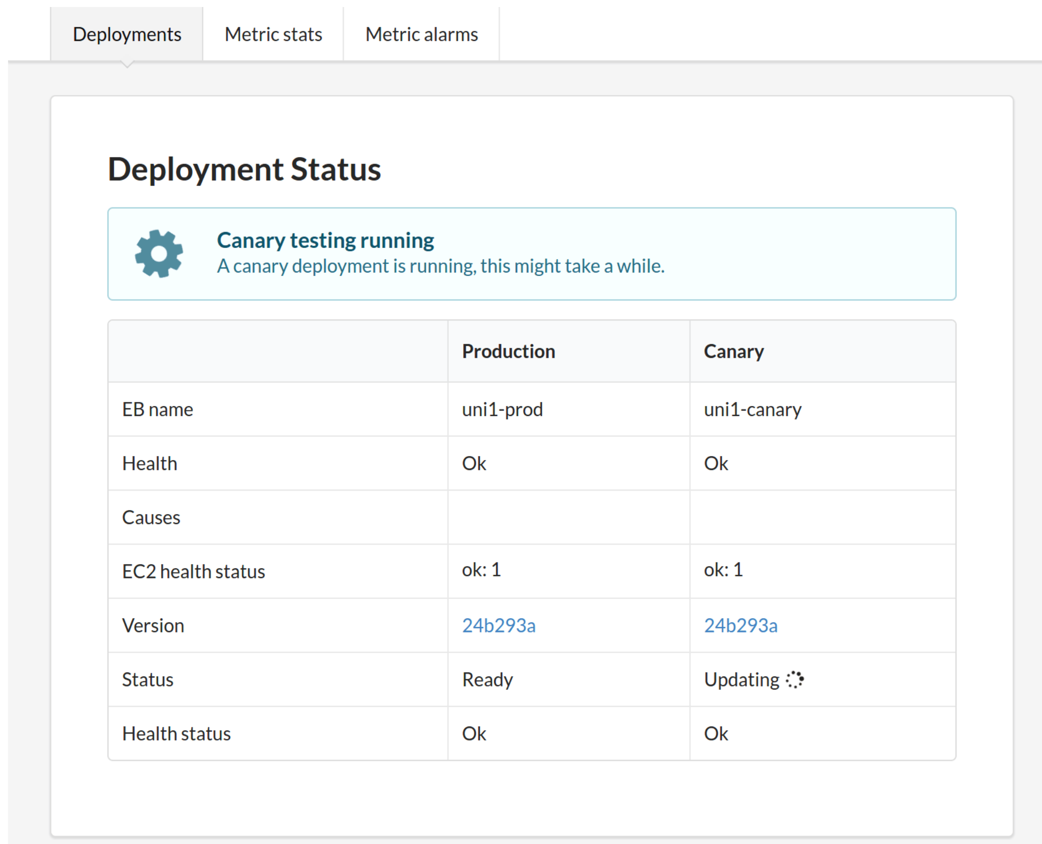

Figure 7.1: The top of the deployments page shows which version is currently deployed to which environment, along with a high level health report. In this screenshot a new version is currently being deployed to the canary environment. The deployment is not yet complete; both environments are showing the same version number as a result.

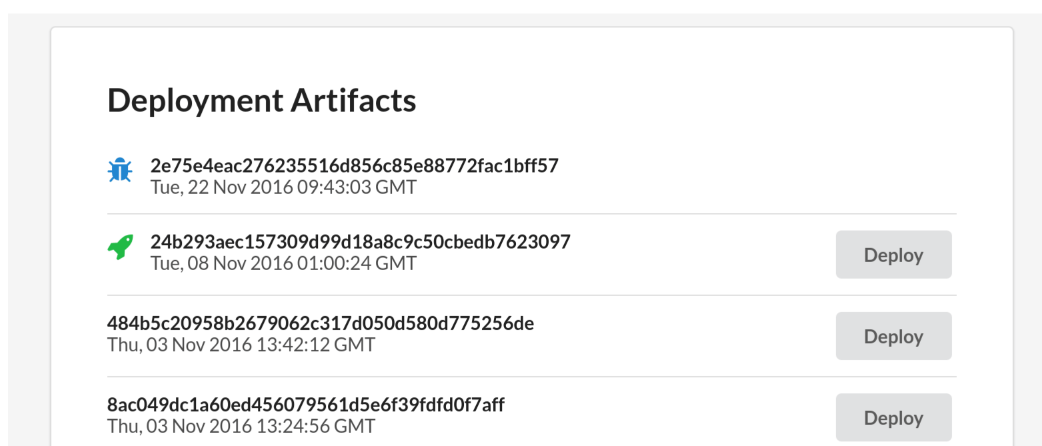

Figure 7.2: This graphic shows a list of all build artifacts, together with a small indicator saying if a version is currently deployed or not.

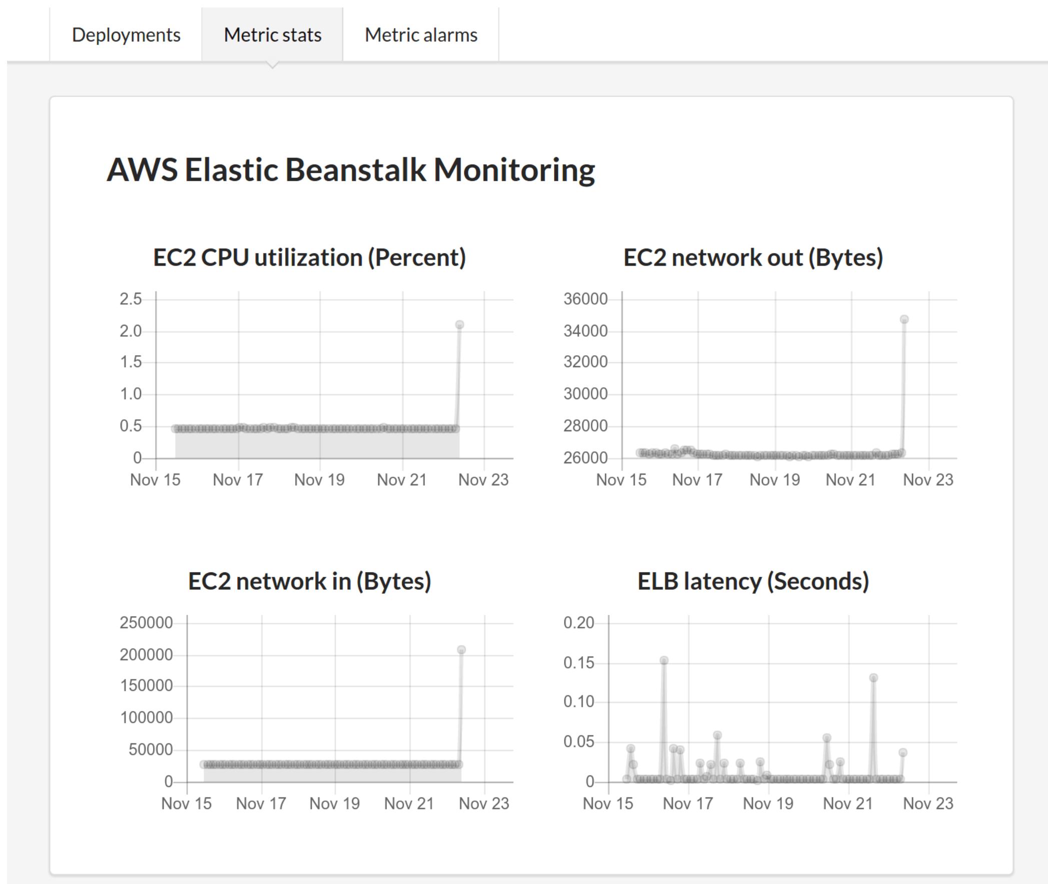

Figure 7.3: This screenshot shows machine specific metric values for the canary environment. The unit of the x-axis is time, while the unit for the y-axis is displayed in the graph header.

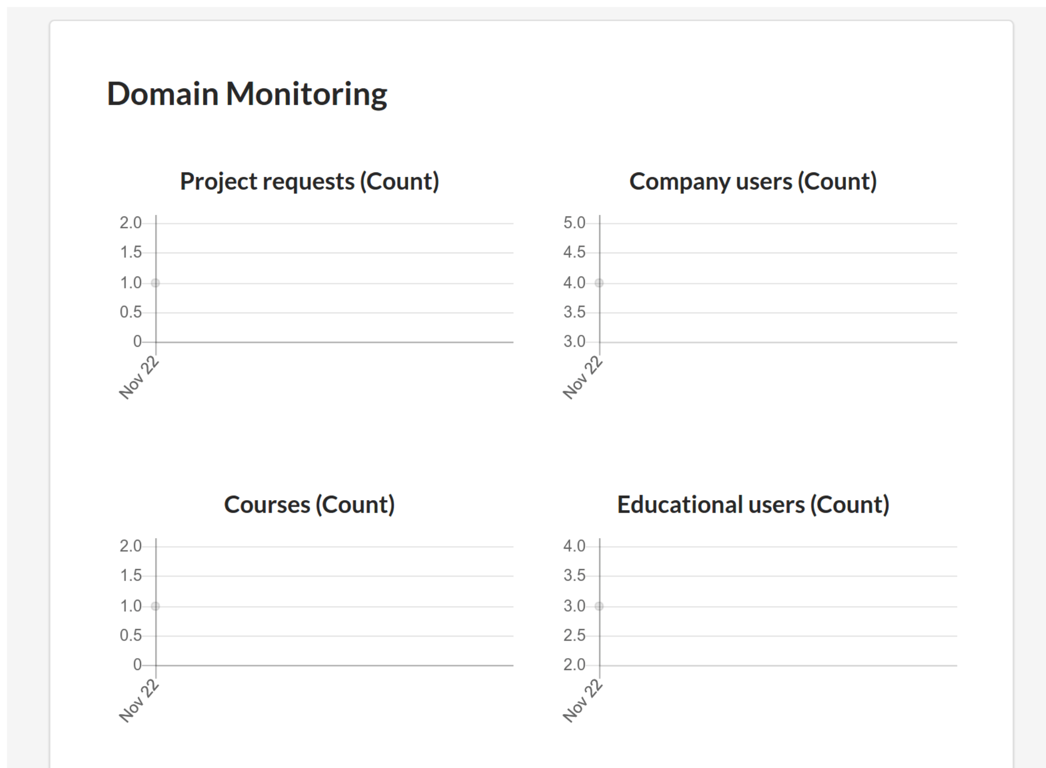

Figure 7.4: This screenshot shows domain specific metric values for the canary environment. The empty appearance of the graphs comes from a very low update frequency of the domain values, along with no recorded changes during the monitoring period.

Deployments Metric stats Metric alarms

Alarm editor

Metric

Courses

Alarm type

Single threshold

Alarm when value is

<

than threshold

42

Submit

Figure 7.5: A screenshot of how single threshold health conditions can be configured. The metric dropdown includes all domain and machine specific metrics.

Deployments Metric stats Metric alarms

Alarm editor

Metric

EC2 CPU utilization

Alarm type

Inside band

Alarm when value is

>=

than lower threshold

5

and

<=

than upper threshold

80

Submit

Figure 7.6: Similar to single thresholds, this screenshot shows the configuration of a health condition value range. The types "inside band" and "outside band" refer to whether the value should be within, or outside of the range.

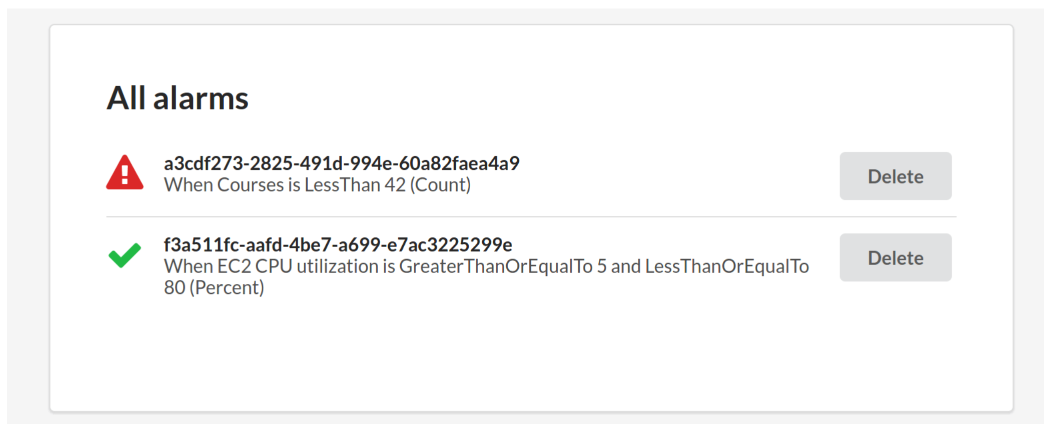

Figure 7.7: This list shows all configured health conditions in the system, here referred to as “alarms”.

References

Amazon Web Services, Inc. (2016a). Blue/green deployments with aws elastic beanstalk. Retrieved November 14, 2016, from https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.CNAMESwap.html

Amazon Web Services, Inc. (2016b). Listing, viewing, and updating cloudfront distributions. Retrieved November 11, 2016, from https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/HowToUpdateDistribution.html

Duvall, P. M., Matyas, S. & Glover, A. (2007). *Continuous integration*. Pearson Education, Inc.

Feitelson, D., Frachtenberg, E. & Beck, K. (2013, July). Development and deployment at facebook. *IEEE Internet Computing*, *17*(4), 8–17. doi:10.1109/MIC.2013.25

Fowler, M. (2003). Publishedinterface. Retrieved November 14, 2016, from http://martinfowler.com/bliki/PublishedInterface.html

Fowler, M. (2006a). Bluegreendeployment. Retrieved November 14, 2016, from http://martinfowler.com/bliki/BlueGreenDeployment.html

Fowler, M. (2006b). Continuous integration. Retrieved November 11, 2016, from http://martinfowler.com/articles/continuousIntegration.html

Fowler, M. (2014). Branchbyabstraction. Retrieved November 15, 2016, from http://martinfowler.com/bliki/BranchByAbstraction.html

Google Inc. (2016a). Launch checklist. Retrieved November 11, 2016, from https://developer.android.com/distribute/tools/launch-checklist.html

Google Inc. (2016b). Release app updates with staged rollouts. Retrieved November 14, 2016, from https://support.google.com/googleplay/android-developer/answer/6346149

Gui, H., Xu, Y., Bhasin, A. & Han, J. (2015). Network a/b testing: from sampling to estimation. In *Proceedings of the 24th international conference on world wide web* (pp. 399–409). WWW '15. Florence, Italy: International World Wide Web Conferences Steering Committee. Retrieved from http://dx.doi.org/10.1145/2736277.2741081

Hodgson, P. (2016). Canaryrelease. Retrieved November 14, 2016, from http://martinfowler.com/articles/feature-toggles.html

Humble, J. & Farley, D. (2011). *Continuous delivery*. Pearson Education, Inc.
Lewis, J. & Fowler, M. (2014). Microservices. Retrieved November 14, 2016, from http://www.martinfowler.com/articles/microservices.html
Luckham, D. C. (2001). *The power of events: an introduction to complex event processing in distributed enterprise systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
Neely, S. & Stolt, S. (2013, August). Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 agile conference* (pp. 121–128). doi:10.1109/AGILE.2013.17
Pang, J., Akella, A., Shaikh, A., Krishnamurthy, B. & Seshan, S. (2004). On the responsiveness of dns-based network control. In *Proceedings of the 4th acm sigcomm conference on internet measurement* (pp. 21–26). IMC '04. Taormina, Sicily, Italy: ACM. doi:10.1145/1028788.1028792
Sathre, J. & Zambreno, J. (2008). Automated software attack recovery using rollback and huddle. *Design Automation for Embedded Systems*, *12*(3), 243–260. doi:10.1007/s10617-008-9020-4
Sato, D. (2014a). Canaryrelease. Retrieved November 11, 2016, from http://martinfowler.com/bliki/CanaryRelease.html
Sato, D. (2014b). Parallelchange. Retrieved November 14, 2016, from http://martinfowler.com/bliki/ParallelChange.html
Sidiroglou, S., Locasto, M. E., Boyd, S. W. & Keromytis, A. D. (2005). Building a reactive immune system for software services. In *Proceedings of the annual conference on usenix annual technical conference* (pp. 11–11). ATEC '05. Anaheim, CA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1247360.1247371