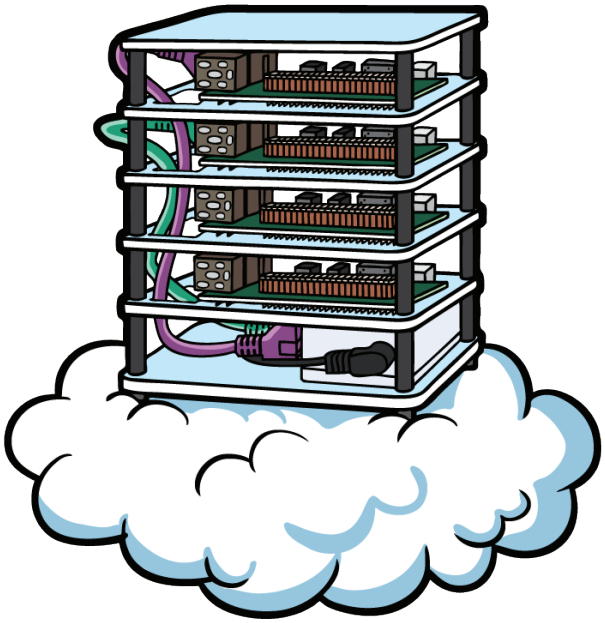
KubeCloud

A Small-Scale Tangible Cloud Computing Environment

Master's Thesis in Computer Engineering
Aarhus University, Department of Engineering



Authors:

Kasper Nissen - KN87372

Martin Jensen - 20106561

Supervisor:

Christian Fischer Pedersen - cfp@eng.au.dk

Date: June 6th, 2016

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Abstract

Cloud computing and container technology have experienced widespread interest for the last couple of years. Academia, however, is challenged in providing courses and material to embrace this new paradigm. Currently, most research into cloud computing has evolved around the use of software simulation, expensive data centers, and large setups of low-cost Raspberry Pi clusters unable to be handed out to students. We introduce KubeCloud, a small-scale tangible cloud computing environment. KubeCloud allows students to experiment with cloud computing concepts and theories with a hands-on, practical learning object. KubeCloud is a controlled test environment that can be used by researchers to experiment with a low-cost, small-scale model of a cloud computing environment. KubeCloud consists of four Raspberry Pi 2 model B and allows for easy extension of multiple KubeClouds to form a larger scale-model. In this thesis, we present the design of KubeCloud and a designed learning activity, the execution of the learning activity with KubeCloud, and an evaluation of KubeCloud as a learning and research object. The evaluations suggest that KubeCloud can help students break down the abstractions of cloud computing and make them visible in a physical cluster.

Keywords: Cloud Computing, Microservices, Kubernetes, Docker, Containers, Teaching strategies, Learning Objects, Resilience

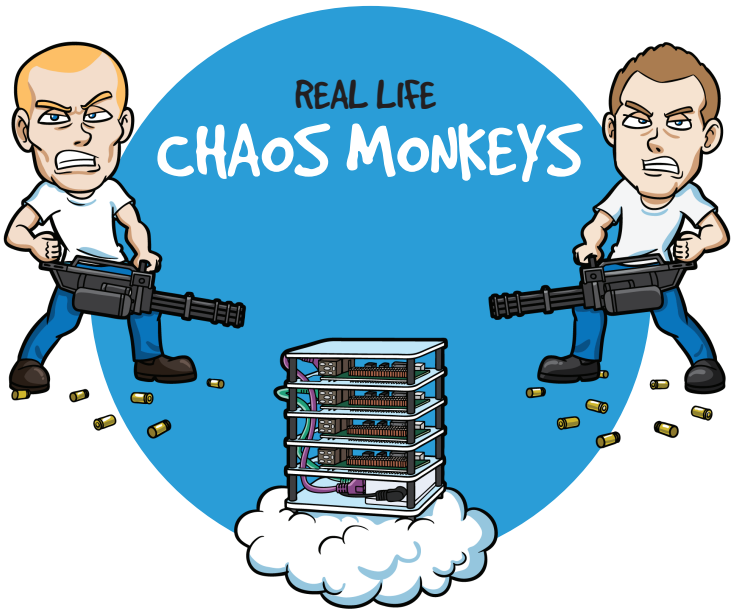
Acknowledgements

We would like to thank our academic supervisor, Christian Fischer Pedersen, for the attention and guidance during the preparation and execution of this master's thesis.

We would also like to thank the students participating in the Object-Oriented Network Communication in the spring of 2016 at Aarhus University School of Engineering for their attention and hard work.

Additional thanks goes to Arjen Wassink, Manager at Quintor (NL), Ray Tsang, Sr. Developer Advocate at Google, and Lucas Käldström, founder of the Kubernetes-on-ARM project for their help and guidance with Kubernetes for ARM-devices.

Finally, we would like to thank our friends and family for their patience and support as well as constructive criticism in the final stages of writing.



"*Yesterday's best practice is tomorrow's antipattern*"
- Neal Ford, Director, Software Architect, and Meme Wrangler at ThoughtWorks

Table of Contents

1

Introduction

Patterns, architectures, paradigms, frameworks, and languages change and evolve over time. The way we built software yesterday may not be suitable to serve the customers of tomorrow. In the 1990s, when the internet got traction, it was relatively easy to understand how servers and applications were connected. Usually, an application would run as a single server running on a single physical machine. If the network or power cable were disconnected, the application would become unreachable.

As the popularity of the Internet grew, companies discovered a new opportunity and started to embrace the new market. These new markets included platforms such as Amazon (an online book retailer), eBay (an online auction service), and Hotmail (an online email service). The Internet boom in 1995 to 2000 led to more complex requirements for system development and architecture. The way software engineers usually dealt with these requirements were typically by adding functionality to an existing monolithic application. A monolithic architecture is an application where all functionality is packaged together as a single unit of compute or application. Chapter 5 covers the topic of monolithic applications in further detail.

At the beginning of the 00s businesses faced new challenges by increasing demand for online services. The rapid growth and increased competition forced businesses into cutting costs and maximizing the utilization of existing technology [1, p. 18]. Meanwhile, businesses still had to *"continuously strive to serve customers better, be more competitive, and be more responsive to the business's strategic priorities"* [1, p. 18]. The two underlying factors behind these forces were the need to combine different technologies, *heterogeneity*, and the need for *change*. As a result, enterprises in the Internet business domain started adopting the idea of *Service-Oriented Architectures (SOA)* instead of the old, homogeneous and monolithic architecture. Chapter 5 covers the topic of SOA in further detail. The paradigm shift in the architectural style is visualized in Figure 1.1.

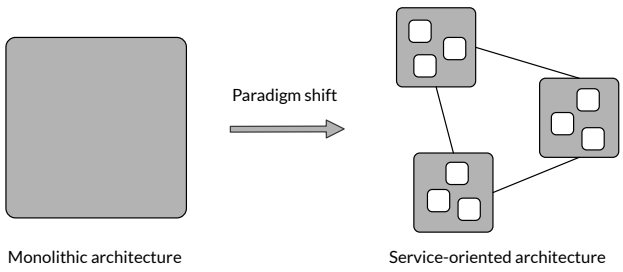


Figure 1.1: Paradigm Shift from Monolithic to Service-Oriented Architecture

The need for heterogeneity arose because different types of systems had to be interconnected. Older systems and new services had to co-exist in this new architecture. Furthermore, this allowed businesses to de-

velop new services across different platforms and languages. This new architectural style introduced the complexities of non-deterministic network calls. The paradigm shift from intra-process to inter-process communication added complexity to the developer's task, and the understanding of possible failures in distributed systems became crucial to ensure resilient systems.

The other main force was the need for *change*. Globalization and e-business were, at the beginning of the 00s, accelerating the pace of change. *"Globalization leads to fierce competition, which leads to shortening product cycles, as companies look to gain advantage over their competition"* [1, p. 18]. The increased competition forces businesses to be able to change quickly e.g. by adding new features.

The offerings in the market control consumers' expectations. The fierce competition raises the expectations for online services. The increased expectations generate requirements to a service's solution. Developers are forced to improve techniques, tools, languages, etc. to satisfy these ever-increasing requirements, and in the end, meet the expectations of the consumers. How these forces impact a business is visualized in Figure 1.2.

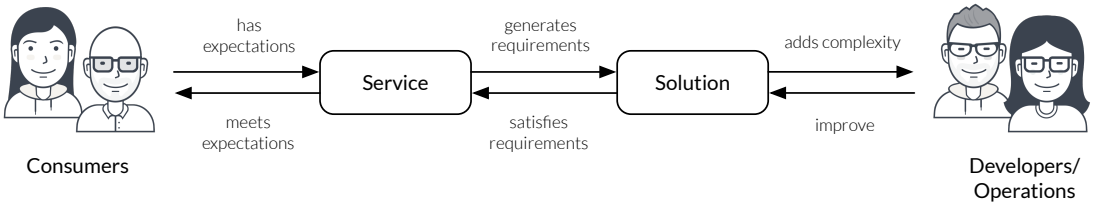


Figure 1.2: Market Forces' Impact on Businesses

Consumers do not have much loyalty to Internet services [2]. If a competitor offers a product of better quality or higher availability, the consumers will probably choose them instead. Furthermore, professor Klaus Schwab describes this need for change:

> *"In the new world, it is not the big fish which eats the small fish, it's the fast fish which eats the slow fish."* ***- Professor Klaus Schwab, Founder and Executive Chairman of the World Economic Forum*** *[3]*

Through the 00's the service-oriented architecture trend continued to grow, and much new technology was developed and implemented. Popular protocols such as *Simple Object Access Protocol (SOAP)* and the infamous *Enterprise Service Bus (ESB)* were adopted. Chapter 5 discusses these protocols in further details. In 2011, a group of software architects started to discuss what they saw as a common architectural style they recently had been exploring. In 2012, James Lewis, Principal Consultant at ThoughtWorks, presented the ideas as *microservices* the UNIX way. Around the same time Fred George, Independent Consultant and former Vice President at ThoughtWorks, presented the same idea, and Adrian Cockcroft, Cloud Architect at Netflix, presented the ideas that Netflix had been working with as *'fine-grained services'*.

The reason for these new enterprise architectural styles was the direction that the SOA community had taken. Too much logic centralized in middleware components, too little focus on a decomposition of the business domain, and focus on reusability rather than isolation and autonomy.

The introduction of cloud computing (widely accessible) in 2006 by Amazon Web Services Elastic Compute Cloud (AWS EC2) as a commercial web service allowed companies to rent computer resources online to host their infrastructures and applications [4]. The key enabling factors behind this were the improve-

ment of virtualization technologies and proper abstractions of the underlying hardware. The evolution of cloud computing along with the service-oriented paradigm shift allowed a more fine-grained architectural trend. The fine-grained trend was further fostered by the spread of container technology in 2013 when Docker was released. Internally Google has used container technology together with a cluster management system, Borg, since 2004 [5, p. 1]. Kubernetes, an open source cluster management system built on the experiences of running and managing containers for a decade, was released by Google in 2014. The next evolutionary step, according to Google, is to transform data centers from being machine-oriented to being application-oriented by using containers [6, p. 5]. Bill Baker, Distinguished Engineer at Microsoft, described the transformation with an analogy of treating servers as cattle instead of pets.

The demand for engineers with skills in cloud computing and this new way of building distributed applications is increasing [7]. This new paradigm demands different skills and an understanding of distributed systems. The demand for graduates with knowledge and expertise in these topics is increasing. Unfortunately, many educational institutions have not offered courses within this new paradigm. As Breivold and Crnkovic point out:

> *"The existing Cloud Computing courses reported in literatures do not address enough details and cloud-related topics that are required in terms of building cloud-based architectures, developing and migrating applications in a cloud. They also lack utilization of real-world systems."* ***- Breivold and Crnkovic, 2014*** *[8, p. 36]*

The lack of education is partly because of the rapid pace of change in cloud computing and the the fact that the industry mainly drives the research. The absence might lead to students graduating without the skill set required by the industry.

Abstraction makes it easier for developers and operation personnel to manage cloud solutions. Abstractions are good - however failing to understand the premise of the abstractions and the underlying environment may result in catastrophic consequences.

Before cloud computing, it was relatively easy to understand how an application would run on a single physical machine. If the machine was unreachable, the service would be unreachable. Today's massive distributed systems are now powering the cloud. Monolithic applications are still widespread. However, microservices and containers are on the verge of hitting mainstream adoption according to a survey done by Nginx with 1800 IT professionals [9]. Therefore, there is a need to align the mismatch between the mental model of the graduates and the demand in the industry. This mismatch is visualized in Figure 1.3.

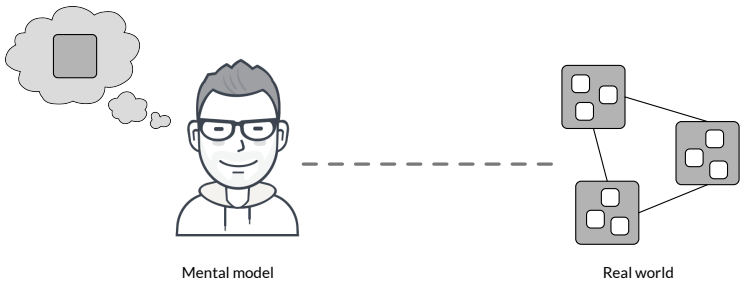


Figure 1.3: Mental Model Mismatch

Even though the interest in academia is growing, the demand for students with an understanding of this paradigm is currently not met. Furthermore, Breivold and Crnkovic express the need for graduates who un-

derstand the non-deterministic behavior, faults in a distributed system, and the need for designing highly-available and resilient systems. Nygard describes these undesired effects in distributed systems as antipatterns and presents patterns to avoid these undesired effects [10, p. 31-114].

Designing for resilience in microservice architecture is therefore of utmost importance to meet consumers' expectations and thereby to the business.

1.1 Motivation

The cloud computing paradigm changes the way consumers and businesses interact. Marc Andreessen, co-founder of Venture capital firm Andreessen-Horowitz[1], stated in 2011 that *"software is eating the world"* [11]. Meaning that every company is becoming a software company to some degree. Even though the interest of cloud computing in academia is on the rise, it does not meet the demand of the industry. Cloud computing will have a bigger role in the industry in the coming years, and therefore, it becomes a part of the mandatory skill set for professionals [8, p. 30]. Education in this field needs to follow the technological advances and required skill set from the industry. There is thus a need for alignment of the mental model of graduates and the real world.

The motivation for this master's thesis is, therefore, to investigate how these two can be aligned. Furthermore, this master's thesis will investigate how educators can be able to demonstrate and highlight the non-deterministic behavior of distributed systems.

1.2 State of the Art

Cloud Computing Education Strategies [8]

Breivold and Crnkovic propose education strategies for cloud computing to meet the demand from the industry. They emphasize a strong need in course design to fulfill the following requirements: *"(i) clarify cloud computing concepts, (ii) include underlying technologies targeting different roles in a cloud, (iii) integrate both educator's goals and practitioner's objectives"*.

Furthermore, they stress the importance of academia/industry partnerships and the multiplicity of learning styles such as readings, class discussions, guest lectures, and hands-on projects.

TeachCloud: A Cloud Computing Educational Toolkit [12]

Jararweh et al identified the lack of teaching tools in cloud computing education. They conducted a survey, after completing a cloud computing course, to identify the challenges students found most important. 93% of the students answered that the lack of hands-on experience was the largest challenge.

Jararweh et al propose a modeling and simulation environment to accommodate this lack of teaching tools. A toolkit called TeachCloud was developed. TeachCloud is an extension of CloudSim that enables experimenting with different components such as processing elements, data centers, virtualization, and web-based applications. TeachCloud made it easy for students to modify cloud computing components and gave them deeper insights into the factors that affect cloud computing environments.

[1] investor in Facebook, Skype, Twitter, among others

The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures [13]

Tso et al describe that most research into cloud computing *"relies on either limited software simulation, or the use of a testbed environment with a handful of machines"*. They present a scale-model of a data center consisting of Raspberry Pis (PiCloud). The Raspberry Pi has made the construction of a scale-model affordable because of its low cost and its low power consumption. PiCloud emulates all layers of a cloud stack, and provides a full-featured cloud computing research and education environment. PiCloud is divided into four racks each containing 14 Raspberry Pi model B which sums to a total of 56 nodes. Tso et al conclude that the PiCloud enables them to carry out practical research without the limitations of simulation.

Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment [14]

Abrahamsson et al state that *"today's students who are the researchers of tomorrow, are not getting enough exposure to real cloud computing infrastructures"*. They emphasize that students need hands-on experience in this area. However, acquiring such an infrastructure is too expensive for universities, and they propose building a Raspberry Pi consisting of 300 nodes instead.
Furthermore, Abrahamsson et al highlight the two most important use-cases for the cluster: being an inexpensive and green testbed for cloud computing research, and a robust and mobile data center for operating in adverse environments.

Iridis-pi: a low-cost, compact demonstration cluster [15]

Iridis-Pi cluster consists of 64 Raspberry Pi model B. Cox et al emphasize the benefits as low power consumption, easy portability, affordability, and ambient cooling. Iridis-Pi is suitable for educational use and provides students with a platform for inspiration and experimentation with high-performance computing. Cox et al have built Iridis-Pi and performed benchmarks on the computation performance with e.g. Hadoop.

1.3 Problem Formulation

The demand for software engineers who understand cloud computing and the complexities of distributed systems is strongly increasing, meanwhile many universities do not offer education within these areas. Teaching in this fairly new discipline is, therefore, important in order to prepare students for the challenges in the job market, but realistic experimenting with the technologies behind the cloud providers is expensive. Is it possible to build a small-scale model of a cloud computing environment using modern technologies? Does it make sense to build a small-scale cloud computing environment for teaching? How can a small-scale tangible cloud computing environment support students' learning? How can a small-scale tangible cloud computing cluster clarify the complexities and possible faults in a distributed system? Moreover, can such a cluster be used for research and experiments?

The answer to these questions will impact how cloud computing courses are designed and taught, and thereby disclose the value of a small-scale tangible cloud computing cluster as an inexpensive learning object.

Courses have been designed using simulations or leveraging cloud providers, but, to our knowledge, there have not been other attempts to incorporate hands-on experience in cloud computing courses. The effects of hands-on teaching methods are, thereby, missing.

This present master's thesis will examine how a small-scale tangible cloud computing cluster can support students' learning. In order to do so a course will be designed and conducted, furthermore a tangible cluster will be designed and built. The clusters will be handed out in order to enable the students to get a hands-on experience with cloud computing and conduct experiments with different software architectures. Several experiments will be designed and conducted in order to determine the students' learning outcome, clarify the importance of designing for failure, and the cluster's role therein.

1.4 Our Contribution

We argue that the evolution of cloud computing needs more focus in the academic community. There is a need for finding a better way to learn and grasp the complexities of cloud computing and microservice architectures. TeachCloud proposed the idea of using simulations as an educational tool, and other unconventional methods such as using a Raspberry Pi cluster has been taken. We propose a small-scale tangible cloud computing cluster, KubeCloud, as a small scale model of a cloud computing infrastructure to be part of the education of engineering and computer science students. KubeCloud helps to emphasize the problems and fallacies of distributed computing in a cloud computing environment.

Building on the pedagogy and theories of how to optimize student learning, specifically within engineering education, we see a need for a hands-on tangible computing cluster to educate and foster further research within the area of cloud computing. KubeCloud acts as a mediator object to align the students' mental model with the real world as depicted in Figure 1.4

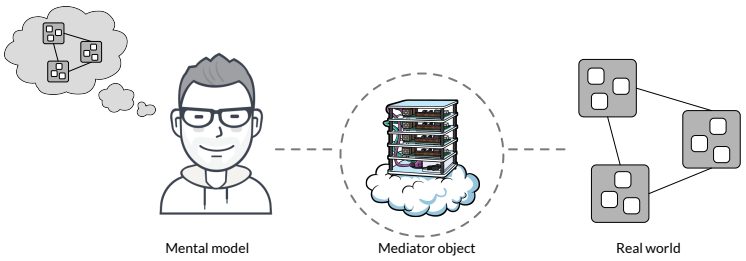


Figure 1.4: Mental Model after Designed Learning Activity

The mediating role of KubeCloud will help students understand the underlying assumptions and concepts of cloud computing. KubeCloud will furthermore provide a basis for experimentation and research in a controlled test environment. Allowing for verification of e.g. Nygard's stability patterns to highlight the importance of building resilient systems and designing for failure.

1.5 Reading Guide and Structure

This master's thesis consists of ten chapters, as visualized in Figure 1.5. The solid lines indicate the suggested order of reading. Furthermore, seven appendices are created to assist the reader and provide more detailed descriptions of specific topics. The dashed lines in Figure 1.5 indicates the order in which the Ap-

pendices should be read.

Chapter 2 - Fundamentals of Learning Theory: Learning theory is an important part of aligning students' mental model with the real world. Knowing how engineering students learn becomes a vital part of designing a learning activity and a learning object.

Chapter 3 - Designing the Learning Activity: Designing a learning activity is necessary to investigate the effects of KubeCloud. Learning theories must be applied within the constraints of the course format.

Chapter 4 - Fundamentals of Cloud Computing Infrastructure: An understanding of cloud computing infrastructure fundamentals is necessary to design and build a learning activity and a tangible cloud computing cluster.

Chapter 5 - Fundamentals of Cloud Computing Architecture: Cloud computing has enabled new ways of designing architectures. An understanding of cloud architecture fundamentals is, therefore, necessary to design and build a learning activity and a tangible cloud computing cluster.

Chapter 6 - Defining Resilient Cloud Computing: Microservices have introduced the non-determinism of distributed network communication. It is therefore of utmost importance to understand the complexities of distributed systems to build resilient systems.

Chapter 7 - Building a Tangible Cloud Computing Cluster: Visualization and tangibility are important in conveying cloud computing concepts to accomodate engineering students' learning style preferences. Designing and implementing a tangible cloud computing cluster as a mediating learning object to align the students' mental model to the real world is needed.

Chapter 8 - Experiment: Evaluating Application Level and Infrastructure Level Resilience: KubeCloud allows for physical experimentation that cannot be done using cloud providers. Experimenting with resilience at the infrastructure and application level is a key feature in order to demonstrate the new paradigm of embracing failure.

Chapter 9 - Experiment: Evaluation of KubeCloud and Course Design: An evaluation is needed to determine whether KubeCloud and the designed course is a viable cloud computing teaching strategy for engineering students.

Chapter 10 - Conclusions: A presentation of the lessons learned is needed in order to draw the overall conclusion of the effects of the designed learning activity and KubeCloud as a learning and research object.

Appendix A - Designing the Learning Activity: Presents an overview of materials created and selected for designing the learning activity.

Appendix B - Additional Kubernetes Concepts: Presents further concepts of Kubernetes, such as canary deployments, scaling, YAML examples, and database opportunities.

Appendix C - Characteristics and Principles of Microservice Architecture: Describes the characteristics and principles of Microservices in greater detail.

Appendix D - Physical Design of KubeCloud: Presents additional images and drawings of the physical design of KubeCloud.

Appendix E - Application Layer: An Introduction to Spring Boot & Cloud: Introduces the microservice chassis framework, Spring Boot and Spring Cloud, and provides examples of usage of components introduced during the designed learning activity.

Appendix F - Experiment: Application Level and Infrastructure Level Resilience: Presents additional results, including verification experiments performed by the students in Module 4 of the designed learning activity.

Appendix G - Experiment: Overall Evaluation of KubeCloud and the Learning Activity: Presents a more detailed result of the overall evaluation of KubeCloud.

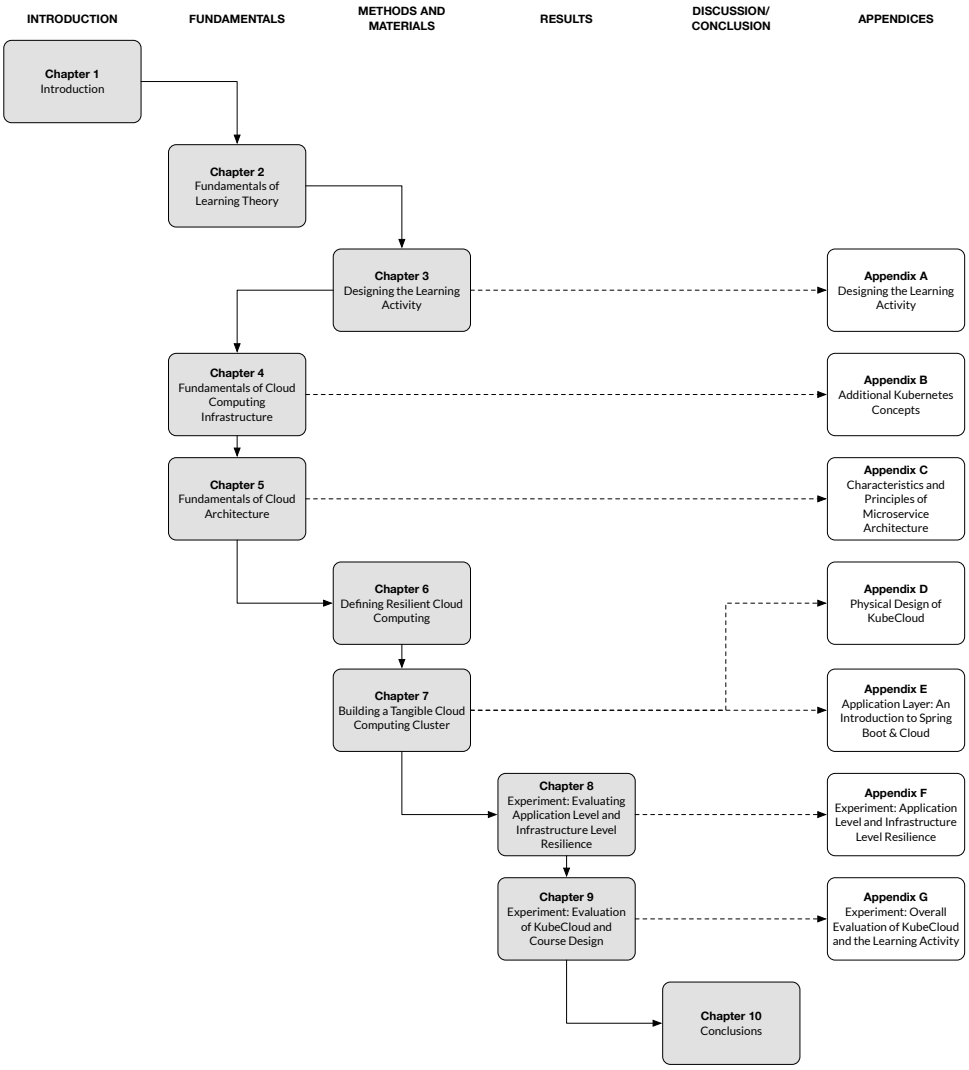


Figure 1.5: Thesis Structure

2

Fundamentals of Learning Theory

Learning theory is an important part of aligning students' mental model with the real world. Knowing how engineering students learn becomes a vital part of designing a learning activity and a learning object.

Learning can include a wide variety of different activities such as reading a chapter of a book, listening to a presentation, having a discussion, taking notes, browsing the Internet, seeing a demonstration, playing a game, etc. There are many different ways of learning, and the learning activity that individuals get the most out of may vary.

Designing a learning activity involves three elements (Figure 2.1): learning outcomes, means (hence the teaching methods and learning experience), and the assessment tasks [16, p. 7]. The learning outcome determines the student's desired achievement, the teaching and learning experiences must be designed to achieve the desired outcome, and the assessment task evaluates to what degree the outcome has been achieved.

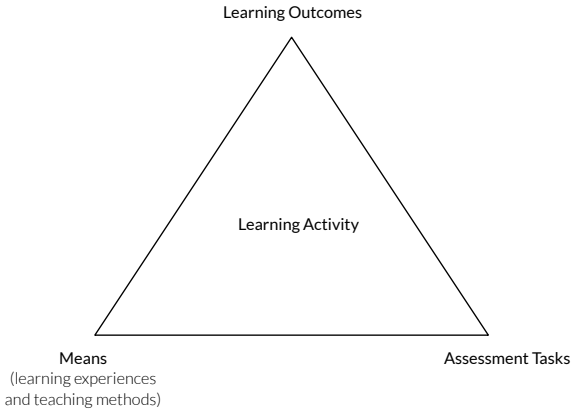


Figure 2.1: Essential Components in a Learning Activity [16, p. 7]

The following two sections cover the design of learning outcomes and assessment tasks described with Bloom's and SOLO taxonomies. The subsequent sections cover the means using learning theories.

> *The teaching methods are the means; the learning outcomes are the ends* ***- Potter and Kustra, 2012*** *[16, p. 7]*

2.1 Bloom's Taxonomy

In 1956 Bloom, Engelhart, Furst, Hill, and Krathwohl, published a new taxonomy, Bloom's taxonomy. The purpose of the taxonomy was to *"promote higher forms of thinking in education"* [17, p. 1]. The original taxonomy included six major categories in the cognitive domain; *Knowledge*, *Comprehension*, *Application*, *Analysis*, *Synthesis*, and *Evaluation* [18, p. 212]. The ordering of categories are in the degree of difficulty, that is, one must normally be mastered before the next. Bloom's taxonomy has most frequently been used to classify curricular objectives and assess students to verify whether or not the objectives are met.

A revision of Bloom's taxonomy was published in 2001 by a former student of Bloom, Anderson, and Krathwohl. The revision included a change of the six categories from nouns to verbs. Further, a rearrange of the Evaluation and Synthesis steps in the original taxonomy was implemented. *"The new taxonomy reflects a more active form of thinking and is perhaps more accurate"* [17, p. 4] The revision also introduced an extra dimension that in the original taxonomy was hidden in the subcategories. Hence, the revised version contains the Cognitive dimension (box below) and the Knowledge dimension, which includes; Factual Knowledge, Conceptual Knowledge, Procedural Knowledge, and Metacognitive Knowledge [18, p. 214].

Bloom's Revised Taxonomy
"Structure of the Cognitive Process Dimension of the Revised Taxonomy" [18, p. 215]

- **Remember** - Retrieving relevant knowledge from long-term memory
- **Understand** - Determining the meaning of instructional messages, including oral, written, and graphic communication
- **Apply** - Carrying out or using a procedure in a given situation
- **Analyze** - Breaking material into its constituent parts and detecting how the parts relate to one another and to an overall structure or purpose
- **Evaluate** - Making judgments based on criteria and standards
- **Create** - Putting elements together to form a novel, coherent whole or make an original product.

Bloom's taxonomy gives teachers, instructors, and educators a convenient way to describe the degree to which they want their students to understand and use concepts, to demonstrate particular skills, and to have their values, attitude, and interests affected.

2.2 SOLO Taxonomy

In 1982, Biggs and Collis described a taxonomy they named SOLO. SOLO stands for **S**tructure of **O**bserved **L**earning **O**utcomes and describes the increasing complexity in a student's understanding of a subject through five stages. Potter and Kustra describe SOLO as: *"Understanding is conceived as an increase in the number and complexity of connections students make as they progress from incompetence to expertise"* [16, p. 9]. The difference between SOLO taxonomy and Bloom's taxonomy is according to Biggs and Collis: *"Bloom Taxonomy is really intended to guide the selection of items for a test rather than to evaluate the quality of a student's response to a particular item"*, and mention that this *"makes it difficult to apply the Taxonomy meaningfully to open-end responses"* [19, p. 13]. Whereas SOLO, on the other hand, is designed for assessing open-ended responses.

SOLO and Bloom's taxonomy are not mutually exclusive, and can be used in conjunction to design curriculums in terms of intended learning outcomes and assessments of learning outcomes.

The five stages in SOLO taxonomy (Figure 2.2) are: pre-structural, unistructural, multistructural, relational, and extended abstract.

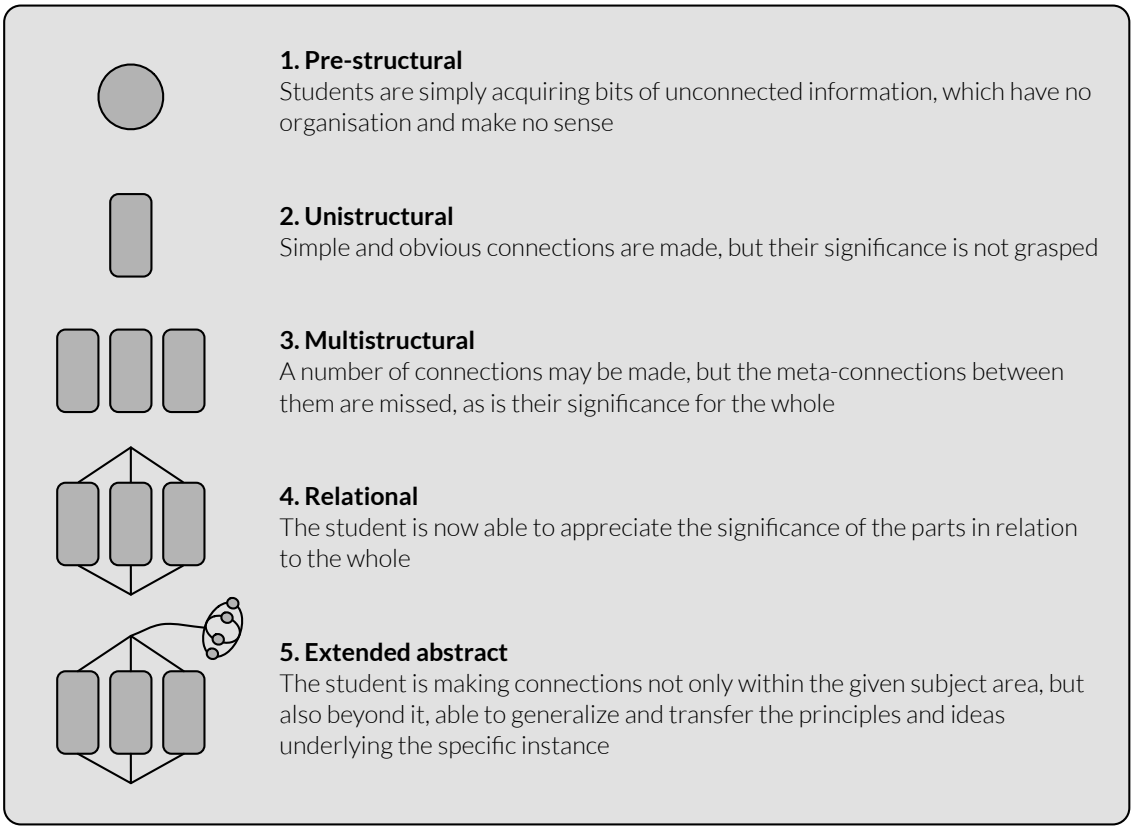


Figure 2.2: Stages in SOLO Taxonomy [20]

The pre-structural stage in SOLO taxonomy refers to a stage of ignorance, where the presented information barely makes any sense. An example of an answer to the question of *"Why use the SOLO taxonomy to create learning outcomes?"* from a student in the pre-strucrutal stage could be: *"I don't know"* [16, p. 10].

The two following stages, unistructural and multistructural, are typically described to be levels of superficial understanding, in which simple and obvious connections are made. The significance of the connections is not fully grasped. An example of an answer from a student in the unistrucrutal stage could be: *"It helps you choose appropriate expectations"* [16, p. 11]. The multistructural level refers to a few numbers of additional connections made compared to the unistructural level, but the significance is still not grasped. A deep understanding of a topic can be achieved, but without connections to other topics, "the big picture" is still missing. An example of an answer from a student in the multistrucrutal stage could be: *"It helps you choose appropriate expectations, plan an assessment strategy, organize a useful sequence of learning experiences, and use outcomes to encourage deeper approaches to learning"* [16, p. 12].

The last two stages in SOLO taxonomy refers to the understanding and grasping of the concepts and abstractions. The first and last stages of SOLO taxonomy can be viewed as existing outside of the learning

activity. The pre-structural stage refers to the knowledge prior to entering the learning experience, whereas the extended abstract stage relates to the usage of the knowledge gained after the end of a learning activity. An important fact to remember is that students go through this learning cycle iteratively within each topic the instructor introduces. The student may be at different levels of SOLO taxonomy within the different subjects of the curriculum.

2.3 Learning and Teaching Styles

There are different styles of learning - and students learn in a variety of ways. To mention a few; seeing, hearing, reflecting, experimenting, acting, memorizing, and visualizing. Likewise, there exist many different styles of teaching. Extensive research with a focus on optimizing students' learning outcome both in general and specialized areas of education has been carried out. Felder and Silverman described in 1988 a mismatch between learning and teaching styles in engineering education [21, p. 674]. In the following section, we will discuss how engineering students learn and discuss the teaching methods matching these styles of learning.

Learning Styles

Felder and Silverman define a learning style model that *"classifies students according to where they fit on a number of scales pertaining to the ways they receive and process information"* [21, p. 674]. They propose a conceptual framework of 32 different learning styles in their original article, based on among others Jung's theory of psychological types. However, the article went through a revision 14 years after its first publication in 1988 with two significant changes in the model. The revised model removes the category of inductive and deductive learning from the original model because of extensive research suggesting that induction is the "best" method of learning. Induction refers to styles such as problem-based learning which will be covered later in this chapter. The original article defined a Visual and Auditory category, which was changed to Visual and Verbal to include both written and spoken words in the verbal category.

Model of Learning Style [21]
A summary of the four questions in the revised version to answer in order to define a student's learning style:

- *What type of information does the student preferentially perceive?*
  - **Sensory** (external): Sights, sounds, physical sensations
  - **Intuitive** (internal): Possibilities, insights, hunches
- *Through which sensory channel is external information most effectively perceived?*
  - **Visual**: Pictures, diagrams, graphs, demonstrations
  - **Verbal**: Spoken and written words, sounds
- *How does the student prefer to process information?*
  - **Actively**: Through engagement in physical activity or discussion
  - **Reflectively**: Through introspection
- *How does the student progress toward understanding?*
  - **Sequentially**: In continual steps
  - **Globally**: In large jumps, holistically

The revised version defines 16 different combinations of learning styles. To focus our efforts towards the

majority of preferences in learning style, we need to understand which styles are most dominant.

Sensing and Intuitive Learning

Sensing and Intuitive learning refer to the two ways in which people tend to perceive the world. *"Sensing involves observing, gathering data through senses"* [21, p. 676] whereas *"intuition involves indirect perception by way of the unconscious – speculation, imagination, hunches"* [21, p. 676]. *"Sensors like facts, data and experimentation"* [21, p. 676] whereas *"intuitors prefer principles and theories"* [21, p. 676]. Felder and Silverman's research suggests that the primary focus is on lectures and readings instead of facts and practical exercises. They point out a mismatch in the style of teaching that favors intuitive learners whereas the majority of engineering students have a preference of a sensory learning style. The learning activity should reach both types, to embrace engineering students' learning styles. Targeting the learning activity can be done by blending concrete information such as facts, data, and observable phenomena with abstract concepts such as principles, theories, and mathematical models.

Visual and Verbal learning

Visual and Verbal refer to the ways people receive information. Visual involves sights, pictures, diagrams, symbols, etc. whereas verbal involves sounds and words (both written and spoken). *"Visual learners remember best what they see"* [21, p. 676], and verbal learners *"remember much of what they hear and more of what they hear and then say"* [21, p. 676]. Felder and Silverman's research shows that most people of college age and older are visual while most teaching is verbal. To optimize learning to include both types, it is important to include visual material such as pictures, diagrams and sketches accommodating e.g. textual slides.

Active and Reflective learning

Active and Reflective learning refers to *"the complex mental process by which perceived information is converted into knowledge"* [21, p. 678]. Active involves experimentation with the information in the external world whereas *"reflective observation involves examining and manipulating the information introspectively"* [21, p. 678]. Felder and Silverman's research suggests that there are indications that engineers are more likely to be active than reflective learners. By altering lectures with occasional pauses for reflection and brief discussion or problem-solving activities, the learning outcome can be targeted towards both learning style preferences.

Sequential and Global learning

Sequential and Global learning refers to the order that students will learn. Some students are comfortable with the sequential process typically used in universities where *"mastering the material more or less as it is presented"* [21, p. 679]. Others, however, do not learn in this sequential manner, but suddenly they will "get it". Felder and Silverman describe the global learners as: *"They may then understand the material well enough to apply it to problems that leave most to the sequential learners baffled"* [21, p. 679]. Basically, sequential learners follow linear reasoning when solving problems whereas global learners make intuitive leaps. To address both types of students, instructors should provide "the big picture" or goal of the lesson to reach the global learners.

Felder and Silverman's research shows that engineering students prefer **sensory**, **visual**, **active**, and **sequential** learning styles [21, p. 680].

Teaching Styles

The previous section defined the learning style preferences of engineering students, and what instructors can do to address the individual types. Looking at teaching styles from a higher perspective reveals two main approaches to teaching; teacher-centered approach, and student-centered approach.

Teacher-Centered Approach

The teacher-centered approach to learning has the teacher as the main figure of authority. The student is passive and viewed as an *"empty vessel"* [22] that has to be filled with knowledge via lectures or direct instruction. The goal of a learning approach with a teacher-centered approach is a test with an assessment based on e.g. a scored test.

Student-Centered Approach

In contrast to the teacher-centered approach, the focus is shifted to include the students to a greater extent. The role of the instructor shifts to a more coaching and facilitating role. The student-centered approach connects teaching and assessment and students are continuously evaluated by doing group projects, class participation, presentations, etc.

The relation between engineering education and the learning style/teaching approaches are summed up in Felder's revision notes to the original article. Felder describes that *"the 'best' method of teaching"* (at least below graduate school level) [21] is a student-centered approach, like problem-based learning as we will discuss in the following section. Furthermore instructors within engineering education need to focus on teaching styles that match the most prefered learning styles.

2.4 Problem-Based Learning

Problem-Based Learning (PBL) is a student-centered approach to learning. PBL was invented by Dr. Barrows in the 1960s in an attempt to *"solve a problem he encountered while teaching medical students: they did not remember what they had learned and did not use reasoning processes characteristic of physicians"* [23, p. 1].

> **Problem-Based Learning** Problem-Based Learning is an instructional (and curricular) learner-centered approach that empowers learners to conduct research, integrate theory and practice, and apply knowledge and skills to develop a viable solution to a defined problem. [23, p. 7]

PBL facilitates problem-solving and is centered around a specific real-world problem with no single correct answer. PBL enforces group work and self-learning in order to solve the problem and apply new knowledge, and reflect on the learning outcome. Furthermore, the instructor's role is to *"facilitate the learning process rather than to provide knowledge"* [24, p. 235]. The box below sums up the main goals of PBL.

The Goals of Problem-Based Learning
"PBL was designed to help students" [24, p. 240]

- construct an extensive flexible knowledge base
- develop effective problem-solving skills
- develop self-directed, lifelong learning skills
- become effective collaborators
- become intrinsically motivated to learn

Problem-based approaches have been used for many years, and Santos and Monte point out the relevance to software engineering education. *"Given the demand [...] for solutions that actually contribute to modern organizations, the search for qualified professionals who have considerable practical experience has been growing day by day"* [25, p. 1]. Furthermore, Santos and Monte describe, and set out to investigate the effectiveness of, how PBL can improve the skill set of software engineers. *"PBL have merged in higher education as an approach to foster changes in teaching and learning processes, which are aligned to the new requirements of the labour market"* [25, p. 1].

2.5 Activity-Based Learning

Activity-based learning (ABL) is an approach that has elements similar to PBL incorporated. The approach focuses on the activity-first, instead of the problem-first approach as in PBL. Churchill describes the goal of activity-based learning as the construction of *"mental models that allow for 'higher-order' performance such as applied problem solving and transfer of information and skills"* [26, p. 1]. Fallon, Eimear and Walsh point out the outcome of activity-based education as students become *"more actively involved in the learning process through acts of 'doing', 'being' and 'critically reflecting' than in traditional, didactic education that is more centered around the passive act of 'knowing'"* [27, p. 4-5].

The following sections will describe some of the theories behind these ABL approaches and how learning objects can be an important part of designing a learning experience.

Activity Theory

Activity theory is a *"philosophical framework for studying different forms of human praxis as developmental processes, both individual and social levels interlinked at the same time"* [28, p. 62]. It is a way to analyze and reason about human activity. It is used to analyze activity in many domains such as education, human-computer interaction, activities of everyday living etc. [28, p. 62]. In activity theory, activities do not make sense without a context. Engström describes these activities as happening in an *activity system*.
Figure 2.3 shows Engström's triangle depicting an activity system. The *subject* can be a person or a group, and the *object* is the product of the activity. The object can be physical or mental, and during the activity the object is transformed. *Tools* are what help the subject transform the object and, in the end, reach a given goal. These tools can be physical or mental, for instance a hammer or a model of something. Jonassen describes tools as follows *"Tools can be anything used in the transformation process [...] The use of culture-specific tools shapes the way people act and think"* [28, p. 63].

The bottom layer contains the constraints of the activity. *Rules* determine what actions and activities the *community* allows e.g. which tools, models, and methods that are used to mediate the process [28, p. 64]. *Devision of labor* refers to the division between the individuals in a group who attend to the task in their

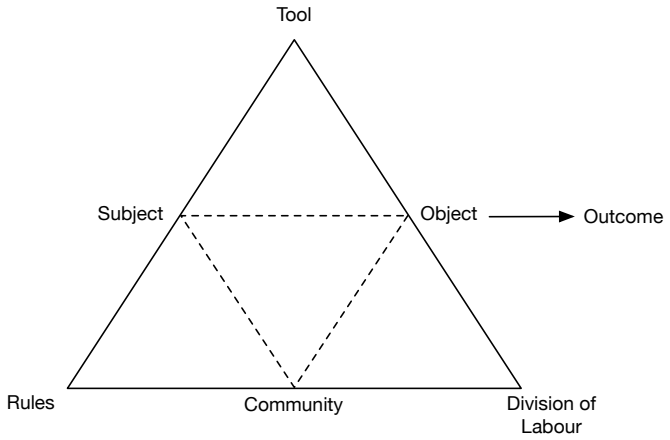


Figure 2.3: Activity System

domain. Designers and developers will, for instance, attend to different areas of the activity. Finally the *outcome* is the end result that can be physical or mental.

Tool mediation is a concept in activity theory about how tools and humans' mental models are connected. Jonassen describes tool mediation as follows: *"tools mediate or alter the nature of human activity and, when internalized, influence humans' mental development"* [28, p. 66-67]. Furthermore, the tool cannot be understood without looking at the context of human activity. Nardi [28, p. 66] points out that cognitive psychology traditionally ignores artifacts (mediating tools) and says that *"activity cannot be understood without understanding the role of artifacts in everyday existence, especially the way that artifacts are integrated into social practice"*. According to Prawat on tools *"[...] they are looked upon as extensions of the individual"* [29, p. 39].

Constructivist Learning Theory

Constructivist theory is based on Piaget, Dewey, and Vigotsky's view of epistemology: *"There is no knowledge out there independent of the knower, but only knowledge we construct as we learn"* [30, p. 1]. From their perspective, learning is not remembered ideas but *"a personal and social construction of meaning out of the bewildering array of sensations"* [30, p. 1-2].

Piaget did not agree with the earlier educational philosophies on how much value play and exploration would impact learning. *"Constructivism is relevant in engineering classrooms because of the set of skills learners get to practice and how it helps engineering students build their own knowledge and solutions to problems"* [31, p. 144]. McKenna adds to this view in engineering education: *"programming skills are important to help students develop engineering "habits of mind" such as how to break a large complex problem into manageable parts, how to isolate effects, test, and debug problems"* [32, p. 1].

The constructivist learning theory is one of the key epistemological views in problem-based learning.

Learning Objects

The idea of learning objects has experienced an increase in attraction in the education communities along with the growth of contemporary pedagogies that promote student-based approaches. Churchill proposes

a classification of learning objects with the goal of defining the term learning objects in his article *Towards a Useful Classification of Learning Objects* [33, p. 2].

Types of Learning Objects
"Churchill proposes a classification that contains six different types of learning objects:" [33, p. 2]

- **Presentation object**
  - Direct instruction and presentation resources designed with the intention to transmit specific subject matter
- **Practice object**
  - Drill and practice with feedback, educational game or representation that allows practice and learning of certain procedures
- **Simulation object**
  - Representation of some real-life system or process
- **Conceptual model**
  - Representation of a key concept or related concepts of subject matter
- **Information object**
  - Display of information organized and represented with modalities
- **Contextual representation**
  - Data displayed as it emerges from represented authentic scenario

The classifications proposed by Churchill emerge from an extensive amount of attempts to define the term *Learning Object*. He discovers that all types of learning objects appear to have the following common characteristics [33, p. 4]:

- they are digital, utilizing different media (and often interactive) to represent data, information, ideas, knowledge, or reality
- they are designed to afford educational reuse

Based on the observations above Churchill proposes a general definition of the term *Learning Object*.

> **Learning Object** *"a learning object is a mediated representation designed to afford uses in different educational contexts"* [33, p. 4]

Learning objects in activity-based theory within engineering education is described by Churchill as: *"Engineering content can effectively be presented with learning objects"* [26, p. 5]. He further argues that *"an active interaction with a learning object [...] enables construction of learners' mental model"* [26, p.1]. As an example he mentions the use of models of simulating and visualizing objects as digital and interactive models, where the user can change parameters and see the effects of these changes. Churchill describes the use of a learning object as:

> *If a picture is worth thousands of words, a learning object is worth thousands of pictures.* ***- Daniel Churchill, 2003*** *[26, p. 5]*

3

Designing the Learning Activity

Designing a learning activity is necessary to investigate the effects of KubeCloud. Learning theories must be applied within the constraints of the course format.

Chapter 2 covered some of the fundamentals of learning theory. In this chapter, we will apply these theories and concepts to design a learning activity focused on the use of a tangible cloud computing cluster. We propose a design of a learning activity using a tangible cloud computing cluster as a learning object. The focus of the learning activity is to motivate and engage students in practical problem solving in cloud computing. The following chapters will give a detailed explanation of the concepts and theories the designed learning activity is based on.

The designed learning activity has been taught in a course, Object-Oriented Network Communication[1] at Aarhus University School of Engineering, to verify the learning outcome of the learning activity and the impact of the tangible cloud computing cluster. These results will be presented in Chapter 8, 9.

3.1 The Need for a Learning Object

Activity-based theories describe the use of learning objects and the importance such an object can have in a learning experience. We see a need for designing a learning object that can help students understand the context from which many abstractions in cloud computing have been made, and propose *KubeCloud - A Small-Scale Tangible Cloud Computing Environment*.

Abstractions are one of the key concepts in cloud computing, and these abstractions have made it easy for developers to provision and deploy applications in the cloud. Failing to understand the premise of these abstractions can have great consequences. The paradigm shift from monolithic applications (using intra-process method calls) to multiple small services (using inter-process communication over a network) introduces a lot of complexity that has to be dealt with. The fallacies of distributed computing have to be addressed and made visible for students. KubeCloud will help students learn about the fallacies of distributed programming and provide a platform for experimentation and exploration in a controlled environment. An active process, in which the learner uses sensory input and constructs meaning out of it, is important (Chapter 2). This view is also pointed out by Satterthwait, *"Some of the most productive, and common, science activities are those that involve the manipulation of objects. This factor plays a significant role in motivating and focusing our students on the learning of Science through the use of objects in an*

[1] `http://kursuskatalog.au.dk/en/course/63876`

activity in which they can be engaged" [34, p. 8]. The need for hands-on experience was found in a survey conducted at Jordan University of Science and Technology during an investigation into the challenges of teaching Cloud Computing. 93% of the students answered that there was a lack of hands-on experience [12, p. 2].

Chapter 7 covers the details of the design of the KubeCloud. The course presented in this master's thesis will be designed around an active, student-based approach and the use of the tangible cloud computing cluster as an object for learning.

3.2 Designing the Overall Learning Activity

The course format is constrained, by an external requirement from the university, to have a duration of seven weeks. These seven weeks are derived into seven modules, each consisting of two half days (4x45min) with respectively lectures and project work. The designed course will be tested inside the scope of an already existing course; Object-Oriented Network Communication[2] taught by Christian Fischer Pedersen. Object-Oriented Network Communication restricts the design of the course since Domain Name Service (DNS) is presented in module 6 by Pedersen. DNS is, though, related to the rest of the course and plays an important role in the understanding of how the cluster management system, Kubernetes, performs service discovery.

As described in Chapter 2, a learning activity involves three elements: learning outcomes, means, and assessment. The following section will first provide a high-level overview of the design of the course in its entirety, and afterward describe the design of the individual module, using these three elements as the primary structure.

Learning Outcomes

Learning outcomes refer to the outcomes that the students should be able to achieve from the designed course. The topics of the course have been chosen to give the students an introduction to cloud computing, the movement towards a service-oriented architecture, and containerization of applications. The topics will be discussed later in this master's thesis. We will first describe the qualifications of the course designed using Bloom's taxonomy to assess the students' performance in the exam. Lastly, a more detailed description of the course content is supplied.

Description of Qualifications

The participants will after the course have insight in the possibilities with state-of-the-art microservice architecture and the elements involved in deploying containerized services in a cloud computing platform. The course will focus on designing a microservice architecture based on the Spring Boot and Spring Cloud frameworks, containerizing the services with Docker and deploying the containerized architecture into a Container as a Service platform, Kubernetes. Emphasis will be on designing distributed systems in regards to resilience and error handling.

The participants must at the end of the course be able to:

- Judge and select appropriate strategies for handling perturbations in a distributed architecture

[2] `http://kursuskatalog.au.dk/da/course/63876`

- Design and construct a distributed microservice architecture, containerize and deploy the system in a cloud computing platform
- Validate the distributed microservice architecture in relation to resilience
- Explore and compare different antipatterns in faults for distributed systems
- Reflect upon the state-of-the-art microservice architecture compared to a monolithic architecture

Contents

The designed course presents an overview of fundamental principles of distributed systems and cluster computing with a particular focus on resilience in a microservice architecture. Containerizing services will be presented and discussed throughout the course with Docker as the enabling technology. Different deployment models will be presented and a Container as a Service platform will be used as the infrastructure for deploying containers. The course will give practical hands-on experience with running a microservice architecture in a Raspberry Pi Kubernetes cluster, KubeCloud. Resilience will be presented and discussed. Practical experience will be achieved using KubeCloud by pulling network cables and attacking the cluster. The DNS and service discovery part of the course will give an introduction to the fundamentals of DNS and explain how service discovery is an important concept when abstracting the underlying hardware with a cluster management system such as Kubernetes. The underlying theme of the course is a real-life problem that the students must solve using the presented concepts and theory.

Means

Means refer to the actions instructors can take to provide the best possible learning experience for the students to achieve the learning outcomes.

Determining Students' Learning Style

The course is designed to foster the highest possible learning outcome for the students. This raises the need for insights into how the participating students describe their learning preferences. To gain this insight, the students are asked to hand in a questionnaire before the start of the course. The questionnaire is created by Felder[3], and the goal is to determine the students' learning style according to the theory described in Chapter 2. The questionnaire consists of 44 questions. Each question has two possible answers and an answer to each question is required. Felder and Silverman describe engineering students to be sensory, visual, active and sequential learners. The result of the questionnaires filled out by the participating students shows that three out of four learning styles match the expected style. However, the students participating in the course lean towards a global learning style preferences instead. This preference must be addressed in the course design. Chapter 9 gives a more detailed description of this experiment.

Teaching Style

Knowing about the students' preference of learning styles, we can design the course to optimize their learning. The experiment shows that the students have a preference for the sensory learning style, which includes facts, data, and experimentation. This suggests that the use of a learning object for experimentation is the right approach. Using multiple sources of learning is a good way to cover the learning style

[3] `http://www.engr.ncsu.edu/learningstyles/ilsweb.html`

preferences. For instance, this is achieved through a combination of presenting the topics in the form of facts and data, and then having the students experiment in workshops and project work. These methods point in the direction of a student-centered approach, which will include the students in a much higher degree. This approach is also supported by the students preferring an active learning style which involves using the information they obtain actively. A problem-based learning approach is chosen to apply the context of the real world. We will discuss PBL later in this chapter.

The results of the experiment show that the students are visual learners. To address this learning style several pictures and diagrams are presented during lectures. Furthermore, a tool for visualizing KubeCloud is needed to support this preference. Lastly, the experiment shows that the majority of the students are global learners. To address this preference, each module will start with a recap of the previous module and emphasize how the new topic relates to the previous topic with a particular focus on how the topic relates to the context of the entire course.

Course Structure

As mentioned, the course has an external requirement of lasting seven weeks which has led to seven modules. Each module introduces a new topic and associated exercises or workshops. The exercises build upon the previous exercises, and most of them will involve KubeCloud to support the students' understanding of cloud computing. During the exercises, instructors will be present to answer questions the students may have. The final result will be a project report and a project presentation for the class followed by an individual examination.

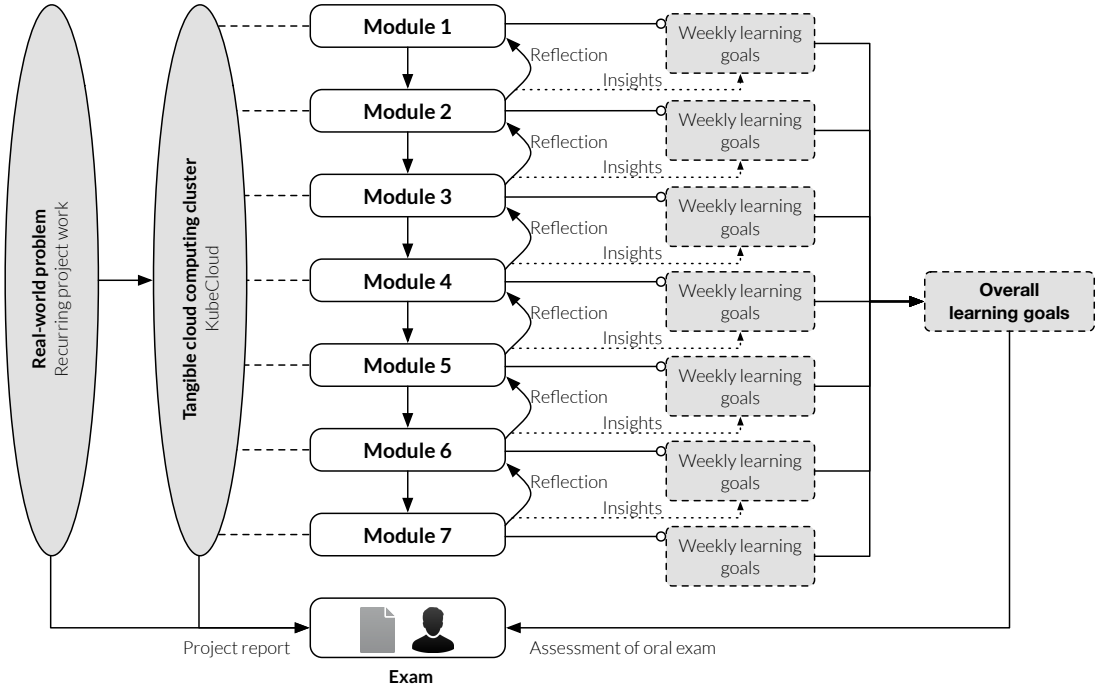


Figure 3.1: Course Structure

Figure 3.1 shows the course structure that consists of modules with learning goals connected to an ongoing project work phase called "Tangible cloud computing". The project is derived from a real life problem following the principles of PBL. The real-world problem is presented to drive the students' group work.

The problem is the e-commerce challenges on Black Friday and is relatively loosely described to allow investigation and experimentation. The problem formulation is shown below.

Problem Formulation
You are working as a development team within a big e-commerce website company selling goods to all of Scandinavia. Black Friday is getting more and more traction in Scandinavia and for the past two years, you and your team have experienced an increasing traffic on your e-commerce website at midnight when Black Friday begins. Last year 10 minutes past midnight, your application crashed because of the heavy traffic load. You and your team have been assigned a total rewrite of your applications architecture to ensure scalability and resilience. You and your team have chosen to implement the solution as a microservice architecture leveraging the Spring Boot and Spring Cloud frameworks running on the Kubernetes Infrastructure platform where your application is packaged and deployed in containers. The requirements for the solution is fairly simple, but the main focus should be on resilience and scalability.

The minimum requirements for this problem are as follows:

- The application should contain a simple graphical user interface
- The application should leverage the gateway API pattern to distribute requests to specific services
- The application should contain at least three data delivering services
- Services have to be configured remotely so that every client fetches configuration at boot from a central configuration server

It is seen (Figure 3.1) that the learning goals for each module have been created to facilitate the students' learning in order to make it possible to evaluate after each module. The goals are designed with Bloom's taxonomy in mind, with the focus on covering different levels of expertise.

In order to support the problem-based group work, a tangible cloud computing cluster will be handed out to each group. The cluster acts as a learning object, and it spans several of Churchill's classification of learning objects. Cloud computing is presented as a tangible physical model which both serves as a simulation object and as a conceptual model. Furthermore, it is possible to interact with the cluster and try things out while getting visual feedback. The cluster is supposed to ease the transition from understanding software on a single machine to the scale of a cloud infrastructure. Using the terminology from activity systems, the cluster will work as a mediating tool that, hopefully, is easier to learn and reason about than an actual cloud service provider. Hein describes that the physical actions and hands-on experiences can help learning and that motivation is a key concept. Furthermore, social activity is described as an important factor.

Questionnaires for each module have been created from the learning goals to evaluate the students' progression and to enforce reflection. The questionnaires contain assessable questions linked to each module's learning goals. The result of the experiment is presented in Chapter 9.

Additional Tools and Methods

Besides the means described above, additional means have been applied to achieve the best possible learning outcome for the students. This section will, in short, present some of the additional methods and tools applied. For further descriptions of these tools, we refer to Appendix A.

Blog: www.rpi-cloud.com A blog has been created with the purpose of sharing the findings during this master's thesis. Furthermore, this blog features articles about different tools such as the Spring Boot and Spring Cloud frameworks used in the course. These articles provide a good basis for the students to get acquainted with the tools used.

Workshops Specifically tailored workshops have been created to the topics of Docker (17 pages), Kubernetes (13 pages), and Resilience & Load testing (19 pages). The workshops supplement the presentations and provide a hands-on approach to the presented topics. Furthermore, these workshops dig deeper into the theory and provide concrete, hands-on experience.

External Presentation In order to relate the concepts taught in the course, an external presenter was invited to bridge the curriculum to the real-world. The external presenter's focus was Continuous Delivery and how tools such as Docker helps in the transitioning to a more service-oriented architecture and the benefits of a continuous delivery pipeline. The presenter, Martin Amdisen, Continuous Delivery Consultant at Praqma, gave the presentation in Module 5.

Additional Activities Additional activities cover, among other things, a presentation and demonstration given at the Google Developer Group in Aarhus. Furthermore, a Kubernetes workshop at Praqma in Aarhus was held to introduce some of their employees to Kubernetes. These additional activities support the course's validity and emphasize the demand in the industry for knowledge within these topics.

Assessment

Assessment refers to determining if the provided means have resulted in the desired learning outcome for the individual student. As mentioned the students are asked to hand in different assignments. On a weekly basis, the students hand in answers to a questionnaire. The questionnaires are designed to give insight into the students' progression along with the possibility of reflecting upon the week's topic seen from the students' perspective. These weekly hand-ins are evaluated using SOLO taxonomy. SOLO taxonomy is chosen because of the nature of the responses given by the students. The answers are fairly short, but they provide insights into the students' progression and understanding of the week's topic and their context in relation to the entire course. SOLO taxonomy provides a useful basis for determining the students' development. The result of these weekly experiments will be presented in Chapter 9.

Besides the weekly reflection hand-ins, the students must work with the given project during the entire course. In each week, new tools and theories are added that can be applied in the project work. The students are asked to hand in a report describing the project, how the different topics relate, and how they are implemented in the project. This project will be the foundation for the assessment of the course. The students will present the result individually at an oral exam with a duration of 20 minutes. The students will be assessed by the demonstrated knowledge during these 20 minutes. Bloom's taxonomy will then be used to evaluate the performance, and the outcome of this evaluation will be a grade in the Danish 7-scale system. The date of the exam will be after hand-in of this master's thesis, June 7th, 2016.

3.3 Designing the Weekly Learning Activity

The design of each module will be described in the following using the same structure presented in the previous section. A module consists of 8x45 minutes. Table 3.1 shows each module's materials, both in terms of reading material, that the students must prepare before class, and the additional resources

created within this present master's thesis. Details about the learning material and additional resources can be found in Appendix A.

Module:	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Number of slides	123	21	62	58	-	-	20
Reading materials (pages)	28	10	44	32	5	-	17
Video (min)	61	10	50	51	0	-	0
Votiee questions	12	0	1	0	0	-	0
Workshop (pages)	0	17	13	19	0	-	0
Number of reflection questions	3	5	7	7	3	-	3

Table 3.1: Overview of Material

Module 1 - Course Introduction and Java Frameworks

Learning Outcome

The first module will introduce cloud computing, microservice architecture, Raspberry Pi clusters, Spring Boot, and Spring Cloud. The students will get applications up and running locally during the project work.

On successful completion of this module, students will be able to:

- Understand challenges in cloud computing and microservice architectures by providing sufficient examples
- Understand the fundamental anatomy of a cluster by providing sufficient examples
- Apply theory of microservice architecture patterns in practice by implementing Java applications using Spring Boot and Spring Cloud
- Evaluate monolithic and microservice architectures by comparing and reflecting upon their benefits and drawback

Means

To achieve the learning outcome of the first module, much material has to be covered. The first four lectures are a combination of presentations combined with small breaks with questions. The students will use a voting platform[4] to answer these reflection questions. This approach has been chosen to reach the active learners who need room for reflection and small discussions. Much material is presented in this first module. However, students have 4x45 minutes in the second part of the module to start working with the topics. This distribution of lecturing versus group work is also seen in Table 3.1 with the number of slides presented.

[4] `http://www.votiee.com/`

Assessment

Assessment of the weekly reflection assignment will be evaluated using SOLO taxonomy. The results are described in Chapter 9.

Module 2 - Docker Lightweight Containers

Learning Outcome

The second module will cover virtualization by introducing hypervisors and containers, particularly Docker. The project work includes an exercise to create a Docker image of the project from Module 1 to familiarize the students with the technology.

On successful completion of this module, students will be able to:

- Understand virtualization by providing sufficient examples
- Understand Docker images and containers by providing sufficient explanations
- Apply Docker images and containers by building and running images and containers
- Evaluate containers and hypervisors by comparing and reflecting upon their benefits and drawbacks

Means

The topic of the second module is, to a larger extent, hands-on with the use of Docker. The presentation is limited to cover only 21 slides as Table 3.1 shows. Instead, the students are given a workshop to familiarize them with Docker. The workshop supplies the students with a deeper understanding of the use of Docker as a container technology.

Assessment

Assessment of the weekly reflection assignment will be evaluated using SOLO taxonomy. The results are described in Chapter 9.

Module 3 - Kubernetes Container Cluster Management

Learning Outcome

The third module will cover container cluster management by introducing Kubernetes. The project work will include deployment of Docker containers in Kubernetes.

On successful completion of this module, students will be able to:

- Understand the concept of orchestration by providing sufficient explanations
- Apply cluster management by deploying Docker containers in a Kubernetes cluster
- Analyze Kubernetes' scheduling capabilities by comparing it to manual scheduling
- Evaluate Kubernetes' capabilities to sustain uptime by discussing the components of Kubernetes

Means

The topic of the third module is cluster management with Kubernetes. Much new material has to be covered because it involves many new concepts. Kubernetes is a central part of the project work, since this is the platform that the students must deploy their microservice architecture in. The presentation of the material is set to approximately two lectures (2x45 min) and includes a live demonstration of a project including visualization of the concepts in KubeCloud. In order to construct the skills needed, the workshop format is used again to familiarize the students with Kubernetes.

Assessment

Assessment of the weekly reflection assignment will be evaluated using SOLO taxonomy. The results are described in Chapter 9.

Module 4 - Resilience and Load Testing

Learning Outcome

The fourth module will cover resilience and load testing by introducing and discussing definitions of resilience and load testing.

On successful completion of this module, students will be able to:

- Understand the concept of resilience by providing sufficient examples
- Apply patterns to improve resilience by implementing these using Spring Boot and Spring Cloud
- Analyze the effect of patterns for resilience by conducting tests agains a cluster using load testing tools
- Evaluate testing techniques by comparing stress testing, load testing, and unit testing

Means

Resilience and Load testing is covered in a similar manner as in module three. Approximately two lectures (2x45min) consist of presentations and live demonstration of load testing using KubeCloud. Afterwards the students are given a workshop that will highlight the covered topics through experiments.

Assessment

Assessment of the weekly reflection assignment will be evaluated using SOLO taxonomy. The results are described in Chapter 9.

Module 5 - External Presentation and Project Work

In the fifth module an external lecturer from industry is invited to present the topic of Continuous Delivery and how its being applied at their customers. The external lecturer is Martin Amdisen, Continuous Delivery Consultant at Praqma. Praqma is a consultancy focusing on helping the industry adopt the principles of Continuous Delivery. The abstract of the talk can be seen below:

Continuous Delivery is the art of actually getting software to the end users.

Praqma is helping many companies large and small get better at releasing their software. With Continuous Delivery we strive to make sure that all code changes end up in the product, thoroughly tested and documented. The key part of this is automation.

In this talk we will give an introduction to the underlying concepts of Continuous Delivery, as well as a brief overview of some of the tools we use to enable this, such as Docker and Jenkins.

The remaining time in this module will be used for further development of the project.

Module 6 - Domain Name Service

The sixth module will introduce DNS. As mentioned earlier this topic will be presented by Pedersen. The content of this talk and the learning goals are specified by Pedersen. The content of this topic is however of utmost importance and is included in the design of the course as part of this master's thesis. The weekly reflection of this module will not be presented in this master's thesis.

Module 7 - Service Discovery and Project Presentation

Learning Outcome

The seventh module will introduce service discovery, and in the end of the module each group will present their project for the rest of the class. The project work will focus on how to leverage service discovery in Kubernetes on the tangible cloud computing cluster. The topics of Bonjour and Avahi will be covered by another master's thesis student researching these topics.

On successful completion of this module, students will be able to:

- Understand service discovery by providing sufficient examples
- Analyze service discovery models by comparing when different models are appropriate
- Evaluate Bonjour, Avahi and KubeDNS by comparing the similarities and differences between them

Means

The topic of this last week of the course cover the topic of Service Discovery and focuses on why it is important and how it is achieved within Kubernetes.

Assessment

Assessment of the weekly reflection assignment will be evaluated using SOLO taxonomy. The results are described in Chapter 9.

4

Fundamentals of Cloud Computing Infrastructure

An understanding of cloud computing infrastructure fundamentals is necessary to design and build a learning activity and a tangible cloud computing cluster.

Until now, we have stressed the importance of learning theory and applied it to design a learning activity for engineering students. The focus of the next three chapters (including this chapter) will change direction by diving into the details of the course content and the necessary knowledge to design and implement a tangible cloud computing cluster, KubeCloud. This chapter will provide a definition of the enabling technologies in the cloud computing stack and infrastructure. The next chapter (Chapter 5) will describe the fundamentals of cloud architecture and how to design applications in a service-oriented manner. Lastly, the implications of applying a service-oriented architecture will be discussed in Chapter 6 in regards to faults and resilience.

4.1 Defining Cloud Computing

Cloud computing represents a paradigm shift in how applications are run and managed. Sosinsky describes the paradigm shift as: *"Cloud computing makes the long-held dream of utility computing possible with a pay-as-you-go, infinitely scalable, universally available system"* [35, p. 3]. The National Institute of Standards and Technology (NIST) defines cloud computing as:

> **Definition** *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [36, p. 2]

Cloud computing is distinguished from other types of computing by the notion that resources are virtual and (almost) limitless. The underlying details of the physical systems on which the software runs are abstracted from the perspective of the user. According to Sosinsky the two essential concepts in cloud computing are:

Abstraction Cloud computing abstracts the details of system implementation from users and developers. Applications run on physical systems that aren't specified, data is stored in locations that are unknown, administration of systems is outsourced to others, and access by users is ubiquitous. [35, p. 4]

Virtualization Cloud computing virtualizes systems by pooling and sharing resources. Systems and storage can be provisioned as needed from a centralized infrastructure, cost are assessed on a metered basis, multi-tenancy is enabled, and resources are scalable with agility. [35, p. 4]

According to NIST, cloud computing can be divided into two distinct sets of models [36, p. 2-3]:

- **Deployment models:** The location and management of the cloud's infrastructure.
  - *Public clouds* The cloud infrastructure is for open use by the general public and exists on the premises of the cloud provider.
  - *Private clouds* The cloud infrastructure is provisioned for exclusive use for a single organization and may exist on or off premises.
  - *Hybrid cloud* The cloud infrastructure is a composition of two or more deployment models.
  - *Community cloud* The cloud infrastructure is provisioned for exclusive use by a specific community and may exist on or off premises.
- **Service models:** The types of services accessible on a cloud computing platform.
  - *Infrastructure as a Service* (IaaS): The cloud provider manages the virtualization layer and below (Figure 4.1). You manage from the OS layer and above.
  - *Platform as a Service* (PaaS): The cloud provider manages the runtime layer and below (Figure 4.1). You manage from the data/application layer.
  - *Software as a Service* (SaaS): The cloud provider manages the complete operating environment with applications, management, and a user interface (Figure 4.1).

The cloud stack (Figure 4.1) is comprised of several enabling technologies that can be combined in various ways. As previously mentioned, virtualization has played an important role in the software stack.

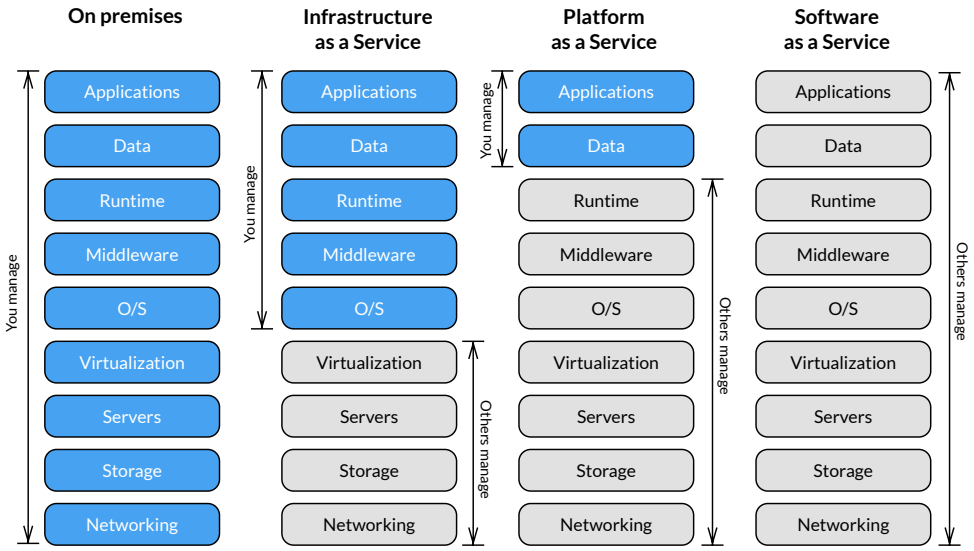


Figure 4.1: Cloud Stack and Deployment Models

Google has challenged NIST's definition of service models by introducing the concept of Container as a Service (CaaS). Burns et al describe this new service model: *"Containers encapsulates the application environment abstracting away many details of machines and operating systems from the application developer and the deployment infrastructure"* [6, p. 74]. By this definition, Container as a Service is placed between Infrastructure as a Service and Platform as a Service. Containerization will be discussed later in this chapter.

NIST describes some of the essential characteristics of cloud computing as: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [36, p. 2]. Sosinsky describes additional advantages being: lower costs, ease of utilization, quality of service, reliability, outsourced IT management, simplified maintenance and upgrade, and low barrier to entry [35, p.17-18]. An example of the elasticity in cloud computing is coping with the additional traffic on Black Friday. If a retailer is required to run the hardware needed for Black Friday peak hours, much energy would be wasted the rest of the year. Companies have different peak hours, and by utilizing the elasticity of cloud computing, it becomes possible to scale resources up and down dynamically.

Among the disadvantages of cloud computing are: less customizability, latency, privacy, and security.

An often used analogy is that computational power from cloud computing corresponds to electricity from the power grid. This illustrates the benefit of the pay-as-you-go model instead of everyone acquiring a power plant. It is, though, important to note that you trust a cloud vendor with your data, as pointed out by Subramanian [37]. This also justifies the hybrid cloud model in which you keep some of the hardware under your own control e.g. for personal data.

4.2 Host Operating System and Hypervisors

The operating system (OS) plays a different role in cloud computing than in traditional desktop computing. The physical machine and the OS becomes a simple resource in a shared pool of resources that is utilized by multiple tenants. Gartner defines multitenancy as:

> **Multitenancy [38]** "Multitenancy is a reference to the mode of operation of software where multiple independent instances of one or multiple applications operate in a shared environment."

The OS in cloud computing must ensure isolation and multitenancy. According to Bernstein, primarily two technologies ensure this in cloud computing, namely hypervisors and containers [39, p. 81]. A hypervisor, also called a Virtual Machine Monitor, is a layer on top of a host machine for running virtual machines. There are different types of hypervisors differentiating on whether or not there is an OS between the hypervisor and the host machine.

- **Type-1 native**: Hypervisors without an OS running on bare-metal
- **Type-2 hosted**: Hypervisors running on top of an OS (Figure 4.2)

One of the early commercial type-1 hypervisors was VMware ESXi[1], which is used for both private and public clouds. An example of a type-2 hypervisor is VMware Workstation[2] that can run on a regular computer. The biggest player in cloud hosting, Amazon Web Services (AWS), use XEN Hypervisor[3], and Microsoft uses their own Hyper-V for Microsoft Azure and their private cloud. The benefits of using hypervisors are

[1] https://www.vmware.com/products/esxi-and-esx/
[2] http://searchvmware.techtarget.com/guides/VMware-Type-2-hypervisor-comparison-Workstation-vs-Player-vs-Fusion
[3] www.xenproject.org

the isolation, dynamism, and the flexibility of running different operating systems. Among the drawbacks are the start-up time for the guest OS, and the size of each running instance.

Containers share the host OS and eliminate the guest OS as seen in Figure 4.2. This makes containers smaller to deploy than virtual machines. Furthermore, it is possible to run numerous containers on a single host. There have been several similar technologies through the time. Bernstein summarizes the inspiration as being the Unix command *chroot* from 1979, *jail* in FreeBSD from 1998, and *zones* in Solaris 10 from 2004. In 2006, Google contributed *cgroups* to the Linux kernel. Google has used containers for their internal services since 2004. As of 2014 Google deployed over two billion containers per week and over 3000 per second [40]. Burns et al describe one of Google's motivations for using containers: *"The resource isolation provided by containers has enabled Google to drive utilization significantly higher than industry norms"* [6, p. 4].

In 2008 Linux Containers (LXC) were created which layered a more user-friendly tooling around *cgroups* and *namespaces*. However, containers were not popularized until Docker in 2013 provided an even more user-friendly interface for LXC.

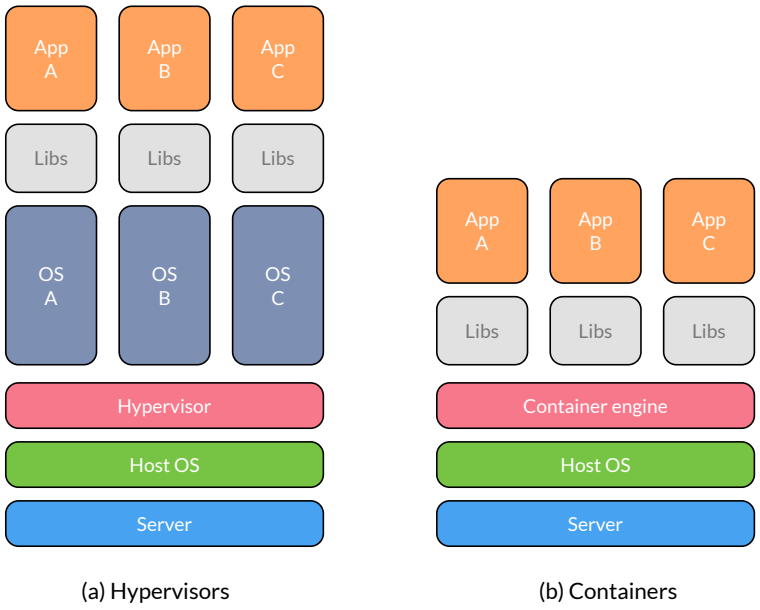


Figure 4.2: Hypervisors (type 2) and Containers [39, p. 82]

In summary, the difference between hypervisors and containers is that virtual machines on hypervisors run an entire OS whereas containers extend a running kernel. According to Joy's performance comparison between Linux Containers and virtual machines, *"containers have outperformed virtual machines in terms of performance and scalability", and continues "because of its better scalability and resource utilization, containers can be used for application deployments to reduce resource overhead."* [41, p. 346].

Docker

In 2013, the container project *Docker* was released. Docker is an API on top of Linux Containers (LXC) that has gained a lot of popularity. Bernstein explains Docker as: *"Docker extends LXC with a kernel- and application-level API that together run processes in isolation"* [39, p. 82]. Docker helps standardize working

with containers by providing a uniform interface. The two main concepts of Docker are containers and images. A container is the name of a running instance that is constructed from an image. Images are either downloaded from a registry, build manually, or a combination starting from an image and extending with additional layers. Images are like LEGO bricks. They can be stacked on top of each other (Figure 4.3). A Docker image contains an application and its needed dependencies, which includes the runtime environment. The layers in a Docker image are filesystem layers as seen in Figure 4.3. Docker provides a declarative definition of an environment (Dockerfile) which ensures that an application always runs in the same environment. Docker removes the need to maintain and worry about different versions of languages and runtimes on servers since it is contained in the image. Docker eliminates the classic problem of differences in the developer and production environment. Turnbull explains this problem between developers and operations as: *"It works on my machine; it's operations' problem now"* [42, p. 103].

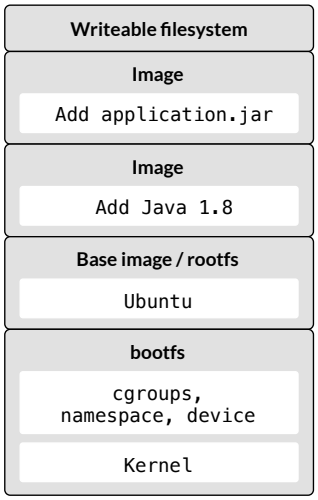


Figure 4.3: Docker Image Layering

Turnbull further describes the layered filesystem in Docker as [43, p. 2-3], the bottom layer consists of typical Linux boot filesystems called *bootfs* which is mounted in a read-only mode. When the boot has finished, the container (notice the transition from image to container) is moved into memory, and the boot filesystem is unmounted. The next layer is a root filesystem called *rootfs* which can be different base images such as Debian or Ubuntu. This layer is also referred to as the base image. Rootfs is mounted and stays in read-only mode unlike a traditional Linux boot, where the mode is changed to read-write after successful boot. This is made possible by *union mount* which is a way of combining multiple filesystems while maintaining the view of a single filesystem. Files and directories are, in other words, added on top of each other. In this way, several read-only layers are combined, and on top, a writeable layer is added. This permits changes in the container while sharing images between containers since modifications are kept in a small writeable layer on the top. Images are stacked on top of parents as seen in Figure 4.3.

Docker consists of a command line client and a daemon running on a host machine. The daemon runs on a Linux machine but the client can run on Linux, Windows and OS X. The communication between client and daemon can either be a TCP connection or a Unix socket which makes it possible to point clients to different daemons. Docker's architecture[4] is depicted in Figure 4.4.

[4] `https://docs.docker.com/engine/understanding-docker/`

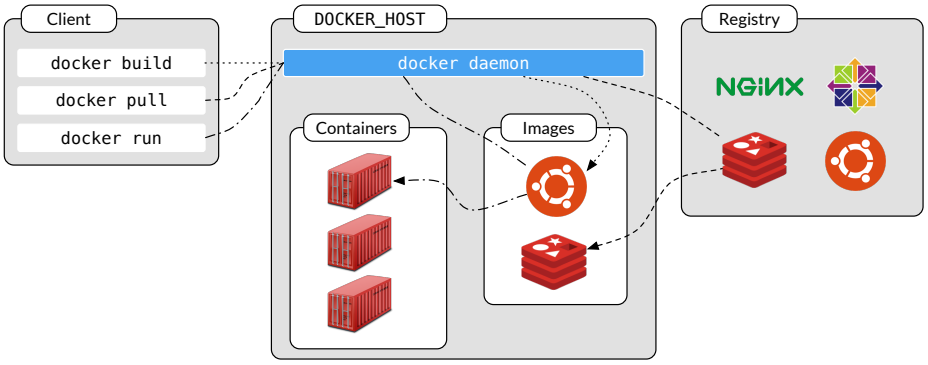


Figure 4.4: Docker Architecture

Registries, such as Docker Hub[5], contain images with prepared environments. This enables diverse environments and changes without having to install and maintain different runtime versions on different servers. Images can be pulled and pushed similar to Git. Because of the layering, only the difference is pulled or pushed, which is the new layer(s). One thing to keep in mind, when using other's images, is whether or not they contain security issues.

The challenges of management, scheduling, and service discovery arise when moving from one container to multiple containers that communicate with each other. These challenges will be described in the following section.

4.3 Cluster Management

A cluster can be viewed as an abstraction comprised of multiple machines (virtual or physical). Combined, the machines (nodes) form a resource pool that can be viewed as a single entity e.g. three machines (Figure 4.5) with 2 CPU cores and 8GB RAM would have the total compute power of 6 cores and 24GB RAM. Machines in cloud computing are degraded to a simple utility. Bill Baker, Distinguished Engineer at Microsoft, describes this shift with an analogy of treating servers like cattle instead of pets.

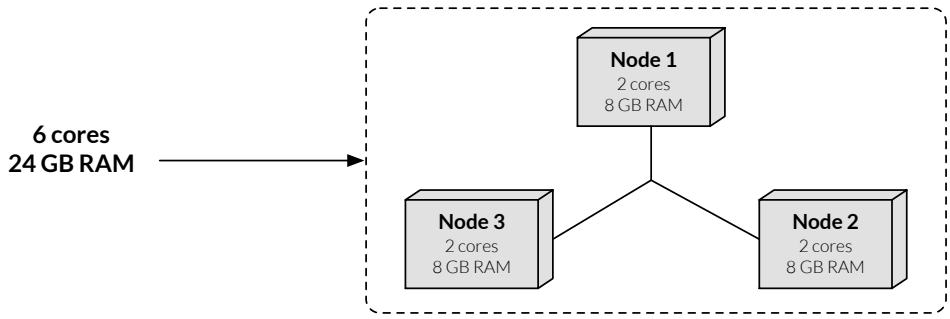


Figure 4.5: Cloud Abstraction

The view of machines as simple, homogeneous resources allows for a more dynamic environment. The dynamic environment makes it easier to scale the number of running instances (e.g. containers) to the current load. Taking care of resource allocation in an intelligent way while dynamically connecting instances

[5] `https://hub.docker.com/`

becomes a complex task. Furthermore, the paradigm shift towards service-oriented architectures makes it even harder to orchestrate the increasing amount of services. This leaves an immense task on cluster management systems to orchestrate instances of services. These tasks include creating an appropriate cluster abstraction of multiple machines, optimizing resource scheduling, keeping services running and to some degree sustaining resilience.

The Bin Packing Problem

Kelsey Hightower, Staff Developer Advocate from Google, explains[6] the challenge of resource scheduling optimization as *the bin packing problem*. The concern of the bin packing problem is to pack workloads of different sizes into a finite amount of bins (nodes) while minimizing the number of bins. In other words, optimizing the utilization of resources in terms of e.g. CPU and memory within each bin without increasing the amount of bins as the only solution. Scheduling applications in an intelligent way can avoid resource waste but the bin packing problem is classified as having an NP-hard computational complexity. If an application requires a lot of CPU and only little memory then there is a risk of blocking memory resources for other applications (Figure 4.6 - node b).

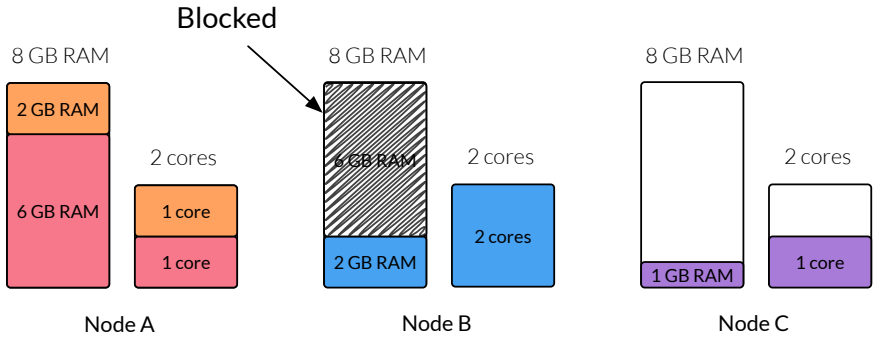


Figure 4.6: The Bin Packing Problem

There exist many cluster management systems created by big web-scale companies. We will in the following discuss Docker Swarm, Apache Mesos, and Kubernetes.

Docker Swarm

Docker Swarm is an abstraction on top of Docker. Docker Swarm utilizes the standard Docker API, but instead of running all containers on the same Docker host they are distributed across a swarm of Docker hosts. For scheduling the containers across different hosts, a Swarm manager is introduced. The rest of the hosts run Swarm agents that communicate with the manager. The strategies for scheduling containers are: spread, BinPack and random. Spread favors a low amount of containers on each node while BinPack favors packing according to the memory usage. Furthermore, Docker Swarm can create replica instances across a cluster. The first non-beta version of Swarm was released in September 2015, which makes it a relatively new option.

[6] `https://www.youtube.com/watch?v=9v8LReolgZ8`

Apache Mesos

Apache Mesos, on the other hand, is a well-tested option that Twitter has used and contributed to since 2010, and currently a lot of successful cloud companies such as Netflix, Airbnb, and Uber use Mesos[7]. Mesos is designed with high-availability and resilience in mind [44]. Mesos is described as a distributed systems kernel that can be used to schedule many different types of workloads. *Mesos Agent Nodes* do the work and report their free resource to a *Mesos Master*. A *Mesos Master* sends tasks to Agents and offers resources to *Frameworks* according to an allocation strategy. *Frameworks* consist of an executor process on each Agent to handle the execution. The Mesos architecture is depicted in Figure 4.7.

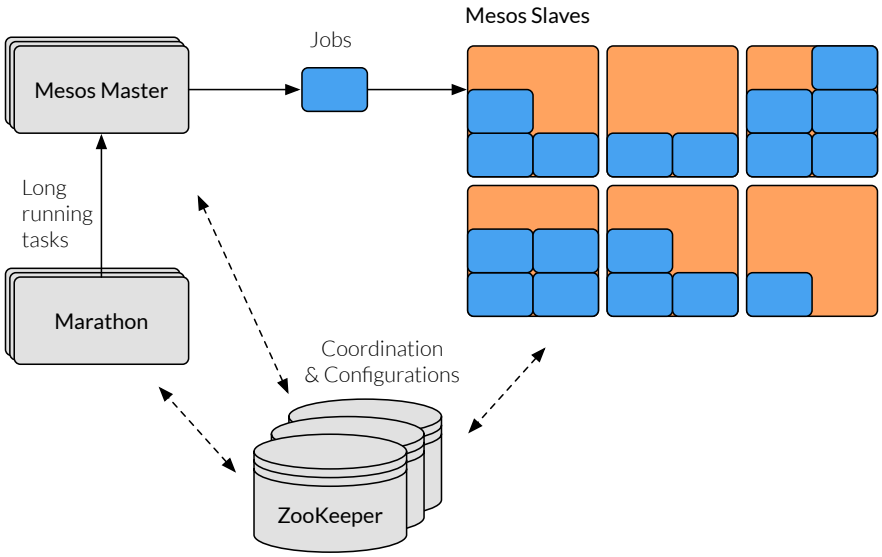


Figure 4.7: Mesos Architecture

Since Mesos is designed for resilience, the master is replicated with an elector, called *ZooKeeper*, in front of it. *ZooKeeper* makes sure that agents can find the master, and the ZooKeeper itself is also usually replicated. Marathon is a container orchestration platform that runs on top of Mesos.

Kubernetes

Google is a pioneer in container technology. Burns describes: *"Containerization transforms the data center from being machine-oriented to being application-oriented"* [6, p. 74]. Google's internal container management platform, called Borg [5], takes care of the challenges involved running web-scale cluster management. In 2014, Google announced and released an open source cluster management system called Kubernetes that is similar to Borg. Kubernetes builds upon many of the concepts in Borg and is *"an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure"* [45]. Kubernetes is chosen for KubeCloud. The choice will be discussed in Chapter 7. A high-level overview of Kubernetes is depicted in Figure 4.8.

[7] http://mesos.apache.org/documentation/latest/powered-by-mesos/

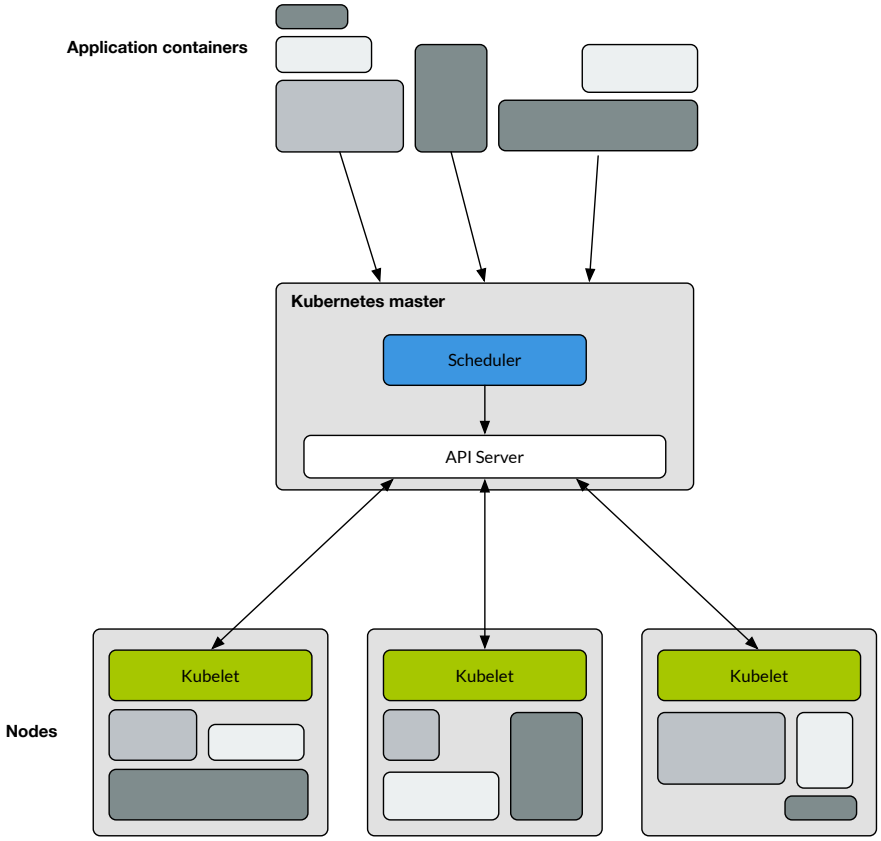


Figure 4.8: Kubernetes Architecture

From a high-level perspective, we see three main parts in the Kubernetes workflow. First, we have the application containers or the workload that Kubernetes needs to handle. Next, we have the Kubernetes master, who is in charge of scheduling the workload, communicating with the workers (nodes), keeping track of the cluster, etc. Lastly, the workers consume the workloads and execute given tasks.

The Kubernetes master is the controlling unit in Kubernetes. It serves as the main management contact point for administrators, and it provides a range of cluster-wide services for the Kubernetes workers. Figure 4.9 shows an overview of the main components of the Kubernetes master deployed on a single node.

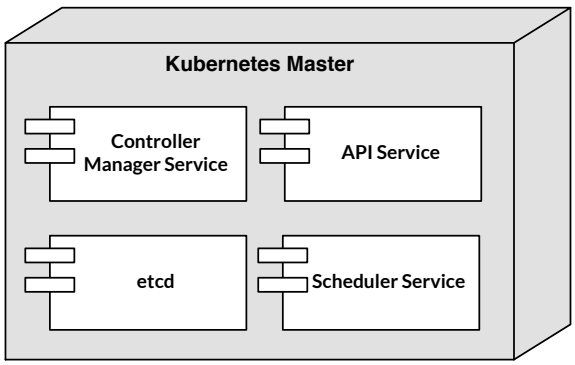


Figure 4.9: Kubernetes Master

Figure 4.9 displays the four main components involved in the responsibilities of the Kubernetes master.

etcd

In Kubernetes, etcd is a fundamental component. It is a globally available configuration store developed by CoreOS. Ellingwood defines etcd as: *"etcd is a lightweight, distributed key-value store that can be distributed across multiple nodes"* [46, p. 3]. Kubernetes stores configuration data in etcd. This data is accessed by the nodes and can be used for service discovery. Furthermore, the etcd store represents the state of each node and the cluster in its entirety.

Controller Manager Service

The Controller Manager Service handles the replication processes used in Kubernetes defined by the Replica Sets (introduced later). The controller manager uses etcd to write the details of these operations. Furthermore, the controller manager watches for changes in etcd. Ellingwood describes this concept as: *"When a change is seen, the controller manager reads the new information and implements the replication procedure that fulfills the desired state"* [46, p. 4].

API Service

The API service is the main management point of the entire cluster. The API service allows users to deploy applications and control many of the Kubernetes workloads and configurations. Furthermore, the API service is responsible for aligning the etcd store with the service details of the deployed containers. The API service is implemented as a RESTful interface, which allows ease of communication for different tools and libraries.

Scheduler Service

The scheduler is responsible for assigning workloads to specific nodes. The scheduler determines which node to deploy a workload on by reading the workload's operating requirements, analyzes the current infrastructure environment, and then places the work on the node best fit for the workload.

Add-On Services

Kubernetes' architecture is loosely coupled and allows for easy add-on of additional services.

- **Heapster:** Heapster enables Container Cluster Monitoring and Performance analysis (cAdvisor aggregation).
- **KubeDNS / SkyDNS**: DNS service for resolving services instead of using IPs directly.
- **Registry**: Local Container Registry instead for depending on Docker Hub.

The Kubernetes worker performs the work in the cluster. Kubernetes workers have few requirements necessary to communicate with the Kubernetes master. Furthermore, the container network has to be configured in order for the worker to be able to run the workloads assigned to them. Figure 4.10 shows the deployment diagram of a Kubernetes worker and the components it is comprised of.

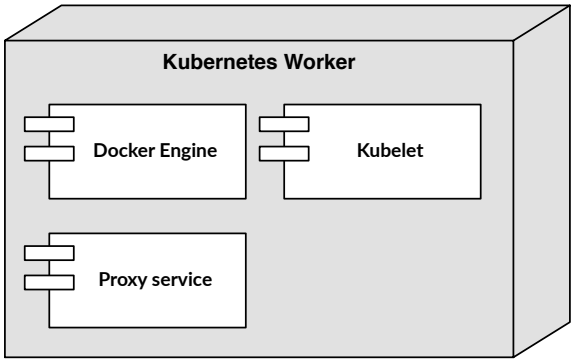


Figure 4.10: Kubernetes Worker

The Kubernetes worker consist of three main components as visualized in Figure 4.10. These three components will be described below.

Docker Engine

Since Kubernetes runs and manages containers, specifically Docker containers (other container formats can be used as well), Docker has to run on each worker. Docker is used to run the containers of the specific hosts that they are being scheduled to. Kubernetes makes the key assumption that each node has a dedicated subnet available to the Docker engine in order for pods to communicate. Different approaches can be used to fulfil this requirement. The approach in KubeCloud will be discussed in Chapter 7.

Kubelet

The Kubelet is a small service that acts as the main contact point for the Kubernetes worker with the Kubernetes master. Ellingwood describes the Kubelet's responsibility as: *"This service is responsible for relaying information to and from the master server, as well as interacting with the etcd store to read configuration details or write new values"* [46, p. 5].

Proxy Service

The proxy service is responsible for dealing with the host subnetting and making services available by forwarding request to the correct containers. The general responsibility of the proxy service can be summarized to making sure the networking environment is predictable and accessible, but still isolated.

Lessons Learned in Running Borg

Verma et al (Google) describe their lessons learned with Borg, *"We've learned from operating Borg in production for more than a decade, and describe how these observations have been leveraged in designing Kubernetes"* [5, p. 13], as:

- **The good:**
  - Allocs are useful
  - Cluster management is more than task management
  - Introspection is vital
  - The master is the kernel of a distributed system
- **The bad:**

- Jobs are restrictive as the only grouping mechanism for tasks
- One IP address per machine complicates things

The first good lesson learned in running Borg is that **Allocs are useful**: *"A Borg alloc is a reserved set of resources on a machine in which one or more tasks can be run"* [5, p. 3]. In Kubernetes, an alloc is equal to a *pod*. A pod is the central unit that Kubernetes schedules, and a pod consists of a group of one or more containers e.g. Docker containers). This means that Docker containers themselves are not assigned to a host, instead they are nested inside another container, called a pod. A pod can contain one or more containers, and cohesive containers are usually deployed together in a single pod. When containers are deployed together inside a pod, they can leverage a shared IP address and port space, and communication via localhost is possible. Effective container-to-container communication is, thereby, a solved problem. This is the first of four identified network problems that Kubernetes solves [8]. Figure 4.11 shows two scenarios; (left) a single container deployed in a pod, (right) two containers deployed in the same pod.

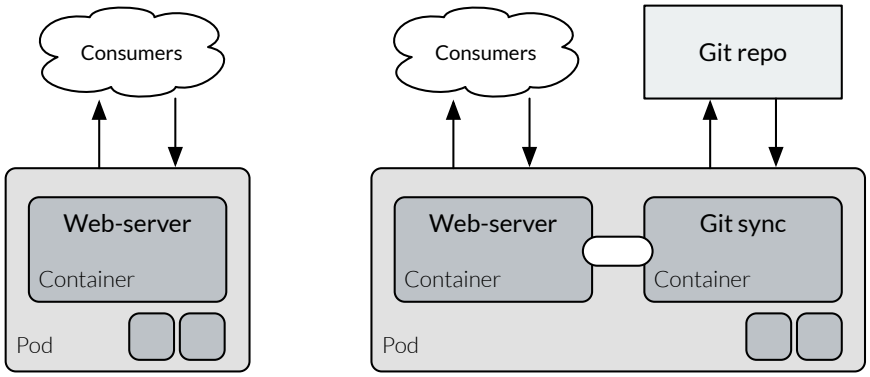


Figure 4.11: Pods

The figure on the right in Figure 4.11 shows two applications running in the same pod. The example contains a worker application fetching configurations from a Git repository and streams it to a web-server. The communication between these two containers could be over localhost. A pod is always scheduled on the same host, meaning a pod cannot span multiple hosts. Different pods can be located on different machines, and pod-to-pod communication over a network (networking problem number two) is thereby necessary. Pod-to-pod communication is achieved by leveraging that each pod has an IP address in a flat, shared networking namespace. The IP is either assigned by a cloud provider or a software defined network. In this way, all pods can communicate with each other, and conflicting port allocation among pods on the same host is not a problem.

The second good lesson learned in running Borg is **Cluster management is more than task management**. *"The applications that run on Borg benefit from many other cluster services, including naming and load balancing"* [5, p. 14]. In Kubernetes, naming and load balancing is supported using the *service* abstraction. A service acts as a load balancer in front of replicated pods. The definition of a service is: *"A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service"* [45]. Pods are targeted using a label selector, as visualized in Figure 4.12. The label selector in Figure 4.12 is type=FE. This results in all pods having this label will receive traffic from the service.

[8] `http://kubernetes.io/docs/admin/networking/`

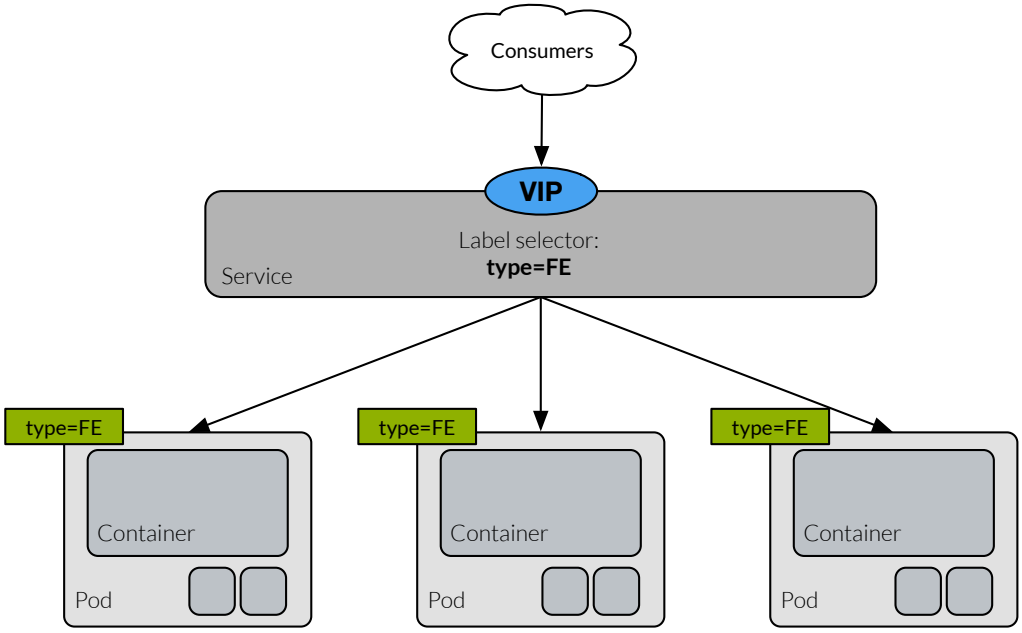


Figure 4.12: Services

Services are assigned a static Virtual IP (VIP) in the range of service IPs defined in the Kubernetes API server. Opposite pods, services will keep the IP throughout its lifetime whereas pods dynamically can be started and killed. Services solve the third networking problem of pod-to-service communication by using the Virtual IP that transparently proxies to pods. Furthermore, services solve the fourth networking problem of external-to-service communication. Cloud providers can integrate load balancers with services, but solutions using the proxy service in the workers also exist.

The third good lesson learned in running Borg is **Introspection is vital**. *"An important design decision in Borg was to surface debugging information to all users rather than hiding it"* [5, p. 14]. Kubernetes includes some of Borg's introspection techniques such as the resource monitoring tool, cAdvisor. cAdvisor allows for individual monitoring of a node's resources e.g. CPU and memory.

The last good lesson learned in running Borg is **The master is the kernel of a distributed system**. *"Borg was originally designed as a monolithic application"* [5, p. 14], but over time some parts of Borg (e.g. the scheduler) was split into services in order to scale up the workload and feature set. Kubernetes was designed as composable microservices from the beginning. All Kubernetes' microservices are clients of the API server (Figure 4.9 and Figure 4.10).

The first bad lesson learned in running Borg is **Jobs are restrictive as the only grouping mechanism for tasks**. *"Borg has no first-class way to manage an entire multi-job service as a single entity, or refer to related instances of a service"* [5, p. 14]. Users of Borg hacked their way around this by encoding their service topology in the job name. To avoid these workarounds, Kubernetes instead organizes its pods using labels. *"Labels are key/value pairs that are attached to objects such as pods"* [45]. Kubernetes connects objects together with labels and can be viewed as a grouping mechanism. The use of labels reduces coupling. Furthermore, labels can be added or removed during the lifecycle of the object. An example of labels used as meta-data for a pod can be seen in Figure 4.13. A further example of labels is shown in Appendix B.

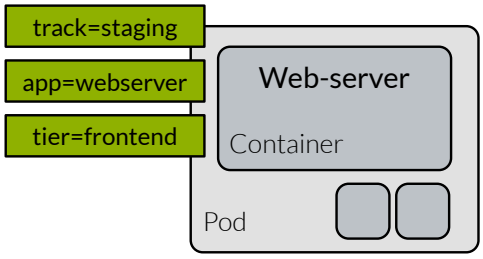


Figure 4.13: Labels

The second bad lesson learned in running Borg is **One IP address per machine complicates things**. *"In Borg, all tasks on a machine use the single IP address of their host, and thus share the host's port space"* [5, p. 14]. This results in port allocation management because of a limited number ports. Kubernetes solves this problem by leveraging software-defined networking that allows every pod to get its own IP address. This allows *"developers to choose ports rather than requiring their software to adapt to the ones chosen by the infrastructure and removes the infrastructure complexity of managing ports"* [5, p. 13].

Other important concepts in Kubernetes are *replica sets* and *deployments*. A replica set ensures that a specified number of pods "replicas" are running at any given time. Replica sets are a lower level concept that is not controlled directly, instead replica sets are controlled through a deployment. Figure 4.14 shows an example with three pods and a replica set managed by a deployment. Users can specify any given desired number of replicas wanted. The replica set will work towards the desired state, and keep it. If a node suddenly is removed, all pods running on this node will be rescheduled to a new node, and the desired number of replicas will be maintained.

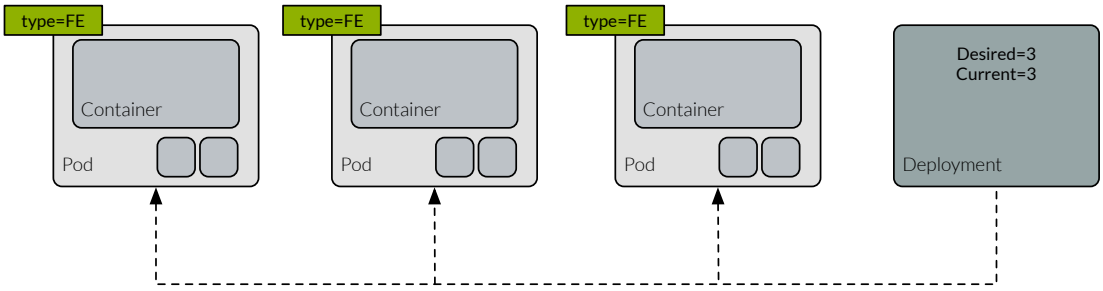


Figure 4.14: Replica Sets

A deployment is, as mentioned, a higher-level concept that manages replica sets, pods and provides declarative updates to pods along with a lot of other useful features [45]. It is only necessary to describe the desired state in a deployment object, and the deployment controller will then change the actual state to the desired state at a controlled rate. Deployments can be described in a declarative manner using YAML files leveraging the ideas of *infrastructure as code*[9]. The deployment of a new version is done in a controlled manner at a controlled rate. One pod will be replaced at a time with the new version until all of the replicas are replaced with a new pod.

[9] `http://martinfowler.com/bliki/InfrastructureAsCode.html`

5

Fundamentals of Cloud Architecture

Cloud computing has enabled new ways of designing architectures. An understanding of cloud architecture fundamentals is, therefore, necessary to design and build a learning activity and a tangible cloud computing cluster.

The previous chapter introduced cloud computing infrastructures including cluster management systems such as Kubernetes. This chapter will focus on the architectural decisions for applications designed to run on a cloud computing infrastructure. Furthermore, this chapter describes the architectural paradigm shift from monolithic application architecture towards a finer grained service-oriented architecture.

5.1 Monolithic Architecture

The term *monolithic* was used to describe software applications that grow into monoliths by Raymond in 2003.

> *"Many programmers with excellent judgment about how to break up code into subroutines nevertheless wind up writing whole applications as monster single-process monoliths that founder on their own internal complexity"* ***Raymond, 2003*** *[47, p. 188].*

The word *monolith* originates from Greek and means *single stone*. In software, this corresponds to using a single process instead of splitting services into multiple processes that each does one thing well. Splitting services into several processes is used extensively in microservice architecture which is why monolithic architectures often are used to describe the opposite.

In a monolithic architecture, all functionality of the application is packaged together as a single unit. This single unit can e.g. be a JAR or WAR file for Java applications. Monolithic architecture style is described by Lewis and Fowler as: *"The server-side monolithic application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML view to be sent to the browser"* [48, p. 2]. Communication between components is accomplished through intra-process method invocations. Villamizar et al describe monolithic applications: *"In monolithic applications all services are developed on a single codebase shared among multiple developers, when these developers want to add or change services they must guarantee that all other services continue working"* [49, p. 584]. The benefit of easily communicating between components has the drawback of complexity for e.g. enterprise developer teams.

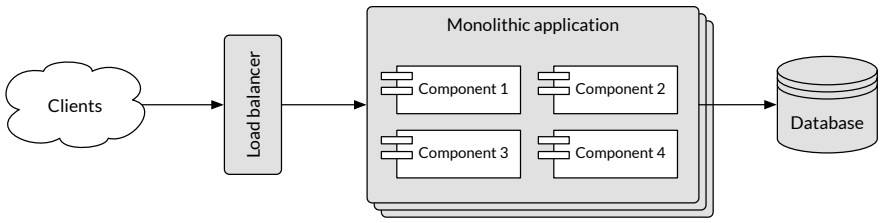


Figure 5.1: Monolithic Architecture

Running a single instance of a monolithic application is easy, and mapping between one application and one running process is easy to understand. The simplicity, though, comes at the price of scalability regarding the ability to innovate rapidly and scale computing resources efficiently. Scaling a monolithic architecture leads to duplicates of the entire application (Figure 5.1). Even though only one of the components is stressed, all of the components scale up which can lead to wasted resources.

Gupta states some of the advantages of a monolithic architecture as: well known, IDE-friendly, easy sharing, simplified testing, and easy deployment [50]. Monolithic architectures are widespread, and developers are schooled to work in such environments. IDEs such as Visual Studio, eclipse, and IntelliJ IDEA are built around solutions and projects that easily can grow big. Testing is easy since everything is in one place. Even though there are many benefits of applying the monolithic application architecture, at some point it will evolve into what Gupta refers to as a *big ball of mud*. This happens over time, when teams grow, when experienced developers leave and new ones join, and when there is a constant pressure to deliver. Gupta describes the following disadvantages of a monolithic architecture as: limited agility, an obstacle for continuous delivery, "stuck" with technology stack, and technical debt [50]. Limited agility refers to the deployment the monolith in its entirety every time a change is made. Over time monoliths will grow and build times will get long and slow, and therefore become an obstacle for continuous delivery. The choice of technology happens before the process of building the application begins. This results in a "technology lock-down" that can require a rewrite of the entire application to break out of.

5.2 Service-Oriented Architecture

In an enterprise setting, applications often start with a monolithic approach. As the application grows, the need for separation arises. Service-oriented architecture (SOA) gained traction in the beginning of the 00s as organizations started to adopt the new paradigm of online business. The drivers for adopting SOA was, among others, the challenges IT executives were facing of *"cutting costs and maximizing the utilization of existing technology at the same time continuously striving to serve customers better, be more competitive, and be more responsive to the business's strategic plan"* [1, p. 18]. The two main drivers for this type of pressure are the need for *heterogeneity* and the need for *change*. Endrei et al describe the characteristics needed: *"In order to alleviate the problems of heterogeneity, interoperability and ever changing requirements, such an architecture should provide a platform for building application services with the following characteristics: Loosely coupled, Location transparent, Protocol independent"* [1, p. 19].

In order to address these requirements, *functional* decomposition of applications into services is an important factor. Endrei describes services as: *"A service is generally implemented as a course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled, message-based communication model."* [1, p. 21].

Communication between services is done in various ways, but popular choices in the SOA community are by using the *Simple Object Access Protocol (SOAP)* over HTTP or using middleware bus-systems like the *Enterprise Service Bus*. SOAP is a protocol for inter-service communication over HTTP. Each SOAP service is bound to a contract using the *Web Service Definition Langauge (WSDL)*[1] that adheres to an *XML Schema Definition (XSD)*[2]. SOAP messages are rather large because they wrap the message in a *SOAP Envelope*, this envelope has a SOAP Header and SOAP Body. The Enterprise Service Bus, on the other hand, acts as a bus that hides complexity and simplifies access, which basically handles the complex details in the background. The ESB has been criticized by, among others, Jim Webber, Global Head of Architecture at ThoughtWorks, and Martin Fowler, Chief Scientist at ThoughtWorks, in their talk *Does My Bus Look Big in This?*[3] from 2008.

> *"Except, I don't think ESB is Enterprise Service Bus. I think it is Erroneous Spaghetti Box."* - ***Jim Webber, 2008***

The reason for this criticism is the outsourcing of vital pieces of the architecture into middleware components. Webber describes the branding of *"the ESB as the panacea for all your ills"*. However, for many it was not the promised silverbullet. Newman describes SOA as: *"at its heart is a very sensible idea. However despite many efforts, there is a lack of good consensus about how to do SOA well"* [51, p. 8]. In his opinion, *"much of the industry has failed to look holistically enough at the problem and present a compelling alternative to the narrative set out by various vendors in this space"* [51, p. 8]. He further states that the problems with SOA consist of problems with communication protocols, vendor middleware, and lack of guidance about service granularity.

A simplified service oriented architecture is visualized in Figure 5.2. Opposite the monolithic architecture seen in Figure 5.1 the architecture is now transformed into an inter-process architecture. This results in communication over network instead of intra-process method invocations.

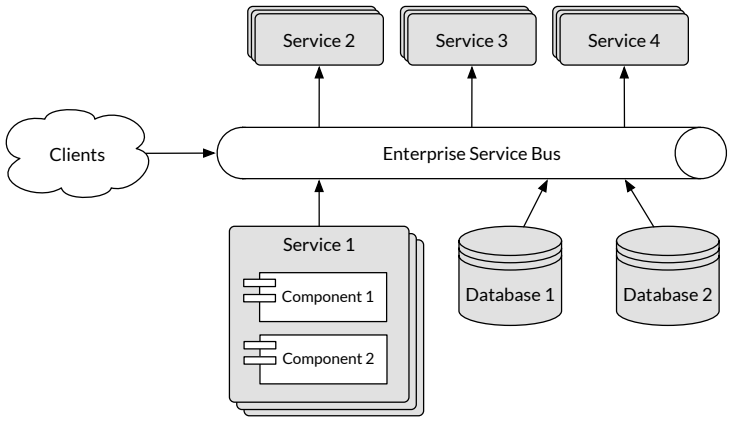


Figure 5.2: Service-Oriented Architecture (SOA)

The benefits of applying a service-oriented architecture are, among others, independent development of services, independent choice of technology stack for services, increased reuse, more responsive and faster time-to-market. Villamizar describes the shift towards service-oriented architectures and the next step as: *"To avoid the problems of monolithic applications and take advantage of some of the SOA ar-*

[1] `http://www.w3schools.com/xml/xml_wsdl.asp`
[2] `http://www.w3schools.com/xml/schema_schema.asp`
[3] `https://www.infoq.com/presentations/soa-without-esb`

chitecture benefits, the microservice architecture pattern has emerged as a lightweight subset of the SOA architecture pattern" [49, p. 584].

5.3 Microservice Architecture

Microservices have been one of the most hyped architectural styles for the past couple of years. A survey done by Nginx in November 2015 showed that: *"nearly 70% of organizations are either using or investigating microservices, with nearly 1/3 currently using them in production*" [9, p. 6]. The respondent pool consisted of 1,825 NGINX community members such as developers, application architects, DevOps, CIO/CTOs, and system administrators.

Microservices are not a completely new way of architecting software. This architectural style has emerged from web-scale companies as a need to be more efficient and cope with a world of ever more connected devices and traffic on a global scale. Microservices build on the Unix design principle of *"splitting large programs into multiple cooperating processes"* [47, p. 188]. This result is an architectural style that favors a suite of small services in separate processes communicating using lightweight mechanisms. Benefits of splitting monolithic applications into separate services are, among other, *"to innovate quickly, reduce complexity, scale computing resources efficiently, and grow development teams in a controlled way"* [49, p. 584] according to Villamizar et al.

Newman defines microservices as: *"Microservices are small, autonomous services that work together"* [51, p. 2]. Adrian Cockcroft, former Cloud Architect at Netflix, extends the definition: *"service-oriented architecture composed of loosely coupled elements that have bounded contexts"* [52, p. 2]. Fowler and Lewis summarize the two definitions:

> *"Microservice architectural styles is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."* ***- Martin Fowler and James Lewis, 2014*** *[53, p. 2]*

In summary, microservices are heterogeneous, loosely coupled services cooperating by lightweight communication. Figure 5.3 depicts a simplified microservice architecture. API Gateway is explained in Appendix E.

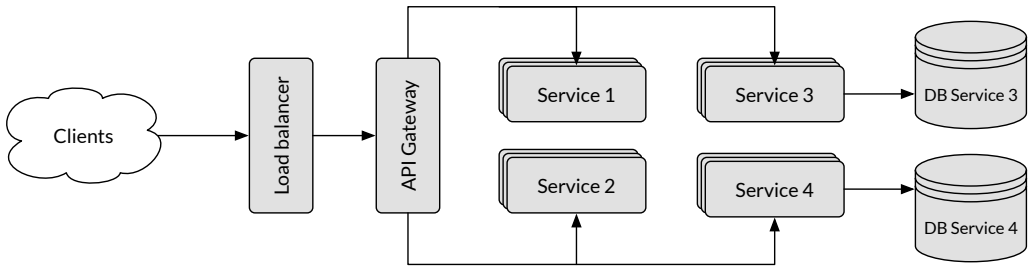


Figure 5.3: Microservice Architecture

Fowler and Lewis describe nine characteristics of a microservice architecture [53]. In the following section we will look into the principles and elements that characterizes a microservice architecture. A more

detailed explanation of the characteristics defined by Fowler and Lewis combined with Newman's seven principles of microservices can be found in Appendix C.

Componentization via Services

Componentization via Services refers to breaking down the architecture into services. This decomposition allows services to be independently deployable and communicate inter-process (inter-service) via 'dumb pipes' as Fowler and Lewis describe it.

Organized around Business Capabilities

The concept of a *bounded context* from Domain Driven Design is important when organizing services. *Bounded context* focus on separating entities and grouping separate context among high cohesive services, which leads to high decoupling. Microservice architecture organizes teams around functional business capabilities instead of technology layers such as UI, middleware, and database specialists. Conway described in 1968 that an organization's communication structure will be reflected in the way teams design their systems.

> *"[...] organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations"* ***- Melwyn Conway, 1968*** *[54, p. 31]*

Products not Projects

Microservice architecture style tries to avoid the project model (software is considered done after delivery) by preferring a team owning a product over its full lifetime. Rather than looking at software as a set of functionality to be completed, there is an on-going relationship between the business and its users.

Smart Endpoints and Dumb Pipes

Enterprise Service Bus puts a lot of *smarts* into the middleware layer. The microservice community favors putting *smarts* into the services instead of in the middleware. They build decoupled and highly cohesive services that own its domain logic

Decentralized Governance

Standardizing on a single technology platform restricts the available technologies. Lewis and Fowler state: *"Not every problem is a nail and not every solution is a hammer"* [53, p. 8].

Decentralized Data Management

One central database (Figure 5.1) implementing the entire domain model couples services through the database. Microservice architecture embraces bounded contexts owning their own data (Figure 5.3), domain models, and only exposing necessary data through its external interfaces.

Infrastructure Automation

The evolution of infrastructure automation techniques has reduced the operational complexity of building, deploying and operating applications. Microservices are built with the concepts of *Continuous Delivery*[4]

[4] `http://martinfowler.com/bliki/ContinuousDelivery.html`

and its precursor *Continuous Integration*. Two key concepts of *Continuous Delivery* worth mentioning are the extensive use of automated tests and the promotion of working software pipelines to automate deployments.

Design for Failure

A consequence of transitioning from a monolithic architecture to a microservice architecture is the shift from an intra-process integrated application to a distributed system. Any call between services can and will at some point fail, due to the non-deterministic behavior of the underlying network. Microservice architecture, therefore, needs to be designed for failure and to respond as gracefully as possible in case of error.

Evolutionary Design

Evolutionary design refers to the ability to cope with change. Controlling change does not necessarily mean a reduction in how frequent changes are implemented. When decomposing applications into services, the decision of how to slice up the application has to be made. Fowler and Lewis emphasize that the key property here is the notion of independent replacement and upgradability.

5.4 Inter-Service Communication

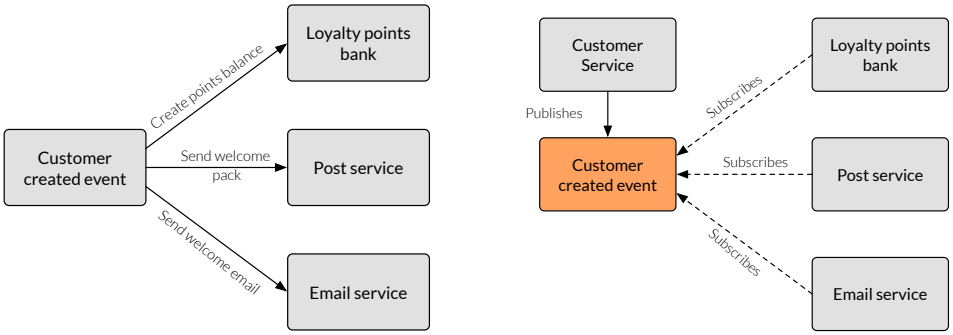
In the previous section, we described the three core application layer architectural styles. We now turn the focus toward the microservice architecture style and describe some of the possibilities of inter-service communication. One of first decisions, an architect of microservice oriented architectures has to make, is whether communication between services should be *synchronous* or *asynchronous*.

Synchronous communication refers to calls made to a remote server, which blocks the caller until the operation completes. Synchronous calls are usually implemented using the *request/response* communication style. The essence of request/response is that the caller makes a request and waits for the response.

Asynchronous communication refers to calls made to a remote server, where *"the caller does not wait for the operation to complete before returning"* [51, p. 42]. Asynchronous calls are usually implemented using event-based communication styles. With an event-based communication style, the communication path is inverted. Newman describes it as, *"instead of client initiating a request asking for things to be done, it instead says this thing happened and expects other parties to know what to do."* [51, p. 43].

Newman describes a benefit of synchronous communication *"Synchronous communication can be easier to reason about. We know when things have completed successfully or not."* [51, p. 42]. Bonér, CTO at Lightbend, published in March 2016 *Reactive Microservices Architecture* as a proclamation for a more reactive asynchronous communication style. He describes *"REST is widely considered as the default microservice communication protocol. It's important to understand that REST is most often synchronous which makes it a very unfitting default protocol for inter-service communication"* [55, p. 21]. Bonér further describes the benefits of an asynchronous communication style as services being decoupled in time and space.

As the complexities in the business processes flow increase, management across the boundary of the individual services becomes hard to grasp. Two possible choices to handle these problems are: *Choreography* [51, p. 45] and *Orchestration* [51, p. 44].



(a) Orchestration (b) Choreography

Figure 5.4: Orchestration Versus Choreography

Orchestration

Orchestration relies on a single central point to guide and drive the process in the implementation. This central point informs each part of the system of its job and lets it work out the details. In Figure 5.4 (a) an example of orchestration is shown. The Customer service interacts with its services in a synchronous request/response communication style. It is easy for the Customer service to determine whether or not each dependent service has completed its task. A downside to this approach is that the Customer service potentially can become a central governing authority - which is not desirable in a microservice architecture.

Choreography

Choreography is an asynchronous event-based approach where the customer service emits an event: *Customer created*. The three services know exactly how to respond to this type of event. This results in a much more decoupled architecture, where services instead subscribe to events like in a usual *publish/subscribe* approach. The downsides are that the overall view gets harder to understand and reason about, because of the implicit nature of the architecture. The choreography approach is visualized in Figure 5.4 (b). Overall, according to Newman: *"systems that tend to be more toward choreographed approach are more loosely coupled, and more flexible and amendable to change"* [51, p. 45].

5.5 Service Discovery

When microservices are deployed to a cloud computing infrastructure, the problem of knowing where each particular service is located arises, e.g. their network location being an IP and port. This problem arises because of the dynamic infrastructure where services can be scaled, fail and new ones scheduled on different nodes, which leads to different network locations. Because of this dynamic architecture, a solution to locate and discover services is needed.

Client-Side Service Discovery

Client-side service discovery refers to clients being responsible for determining the network locations of available services and load balance requests across them. To do this, the client queries a service registry, which has an internal database with the available services. The client uses a load-balancing algorithm to select the appropriate available service and makes a request. This pattern is shown in Figure 5.5.

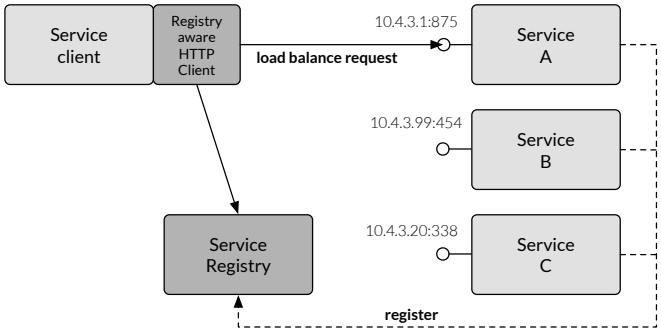


Figure 5.5: Client Side Service Discovery [56]

Before the service registry can be used to full effect, each service in the architecture needs to be registered. When a service starts up, it is announced on the network, and the service registry adds the service to the database. If the service for some reason gets terminated, the service registry needs to be notified. The service registry maintains the database by using a heartbeat mechanism to check if services still are available and ready to respond to requests. An example of the client-side service discovery is Netflix's open source tool, Eureka. Eureka provides an HTTP REST interface and together with another Netflix tool, the client side load balancer called Ribbon, the service registration and load balancing is handled.

Server-Side Service Discovery

In server-side service discovery, it is no longer the client's responsibility to query the service registry. Instead, it is the load balancer that queries the service registry and routes each request to an available service. Registration and deregistration are handled in the same way as the client-side service discovery.

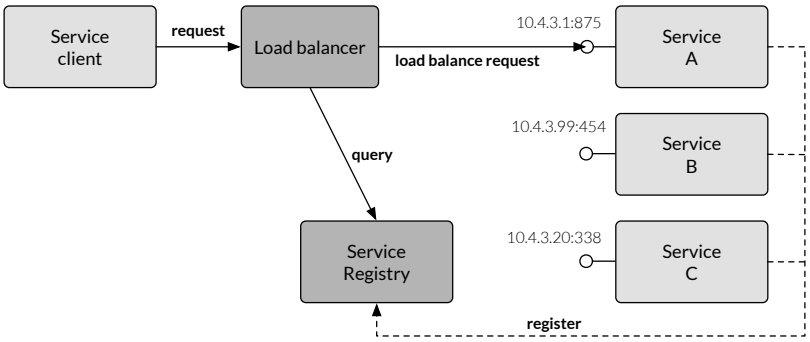


Figure 5.6: Server Side Service Discovery [56]

An example of applied server-side service discovery is Kubernetes. Kubernetes runs a proxy on each host in the cluster. Richardson describes the role of the proxy: *"The proxy plays the role of a server-side discovery load-balancer"* [56, p. 7]. Nommensen describes the server-side service discovery flow of Kubernetes: *"in order to make a request to a service, a client routes the request via the proxy using the host's IP address and the service's assigned port"* [57]. This results in the request transparently being forwarded by the proxy to an available service running somewhere in the cluster.

6

Defining Resilient Cloud Computing

Microservices have introduced the non-determinism of distributed network communication. It is therefore of utmost importance to understand the complexities of distributed systems to build resilient systems.

Google's *Site Reliability Engineering - How Google Runs Production Systems* and Nygard's *Release It! - Design and Deploy Production-Ready Software* try to shift the traditional focus of design and development towards systems in production.

> *"Software design as taught today is terribly incomplete. It talks only about what systems should do. It doesn't address the converse—things systems should not do."* ***- Nygard, 2007*** *[10, p. 1]*

Nygard describes a lack of focus on how to run systems in production. Even if a system is perfectly designed and built, the software does not bring any value unless it is available. Beyer et al agree with this and further add the analogy below.

> *"Software engineering has this in common with having children: the labor before birth is painful and difficult, but the labor after the birth is where you actually spend most of your effort. Yet software engineering as a discipline spends much more time talking about the first period opposed to the second, despite estimates that 40-90% of the total costs of a system are incurred after birth."* ***- Beyer et al (Google), 2016*** *[58, p. xv]*

We will throughout this chapter try to define the term resilience and describe how to handle the fact that failures eventually will happen. Embracing and designing for failure are the new mantras.

6.1 Towards a Definition

The term resilience is used in several different sciences e.g. material science, engineering, psychology, and ecology. The Latin translation of resilience is *"leap back"* or *"bounce back"*. The Oxford dictionary equates resilience with toughness and elasticity, and an often used metaphor for illustration is a spring bouncing back. A spring is both tough and elastic. The opposite of resilience is brittleness and fragility.

Resilience (Oxford) [a]

1. The capacity to recover quickly from difficulties; toughness
2. The ability of a substance or object to spring back into shape; elasticity

[a] `http://www.oxforddictionaries.com/definition/english/resilience`

Occurrences force a resilient system to "bounce back". These occurrences are described in the literature with different terms such as a *disturbance*, *disruptions*, or *perturbation* [59, p. 13, 14, 19].

Narrowing the focus down to information and communication technology (ICT), multiple terms of resilience have a similar meaning, and it can be difficult to distinguish between them. The term *resilience* in ICT has been described as the *"ability to deliver, maintain, improve service when facing threats and evolutionary changes"* from a study on the usage of the word resilience in literature [60, p. 27]. In this case *evolutionary changes* in current and future ICT is the important distinction from *fault tolerance* [61, p. 4]. Strigini points out that ICT systems are more interconnected, open, and exposed to more change than previously, which is why old methods for dependability do not work. This underpins the premise of this thesis and the need for new techniques to achieve dependability.

> *While existing practices of dependable design deal reasonably well with achieving and predicting dependability in ICT systems that are relatively closed and unchanging, the tendency to making all kinds of ICT systems more interconnected, open, and able to change without new intervention by designers, is making existing techniques inadequate to deliver the same levels of dependability.* ***- Strigini, 2012*** *[61, p. 4-5]*

There are different ways of addressing the risk of faults in a system. *Fault avoidance* is about lowering the probability of components having or developing faults, while *fault tolerance* is functioning even though a fault occurs. Drawing a line between fault tolerance and *resilience* can be hard, but fault tolerance covers e.g. a hardware component continuing to function in spite of an fault or error, whereas resilience also is about recovering fully (bouncing back) from failure. The step beyond resilience is the principle of *antifragility* (Figure 6.1) which was introduced by Taleb [62, p. 1]. The idea is that a system gets stronger from errors (or faults) and learns from it. But as Monperrus points out this does not fit well with traditional dependability in software systems [62, p. 1]. Steps toward antifragility are, though, being made as we will explain using an example from Netflix later in this chapter.

Figure 6.1: Fault Handling Scale

6.2 Characteristics

Resilience is related to concepts such as dependability and availability which often is measured in percentages of uptime over a time period. Bruneau and Reinhorn's *"resilience triangle"* [59, p. 21], describing seismic resilience in infrastructure, illustrates the same concept. A quality of service function defining how well a system is performing is plotted over time (Figure 6.2) the quality of service function being e.g. whether responses are succeeding or failing, or to what extend the latency is acceptable. A resilient system can bounce back into a fully functioning shape after a disruption.

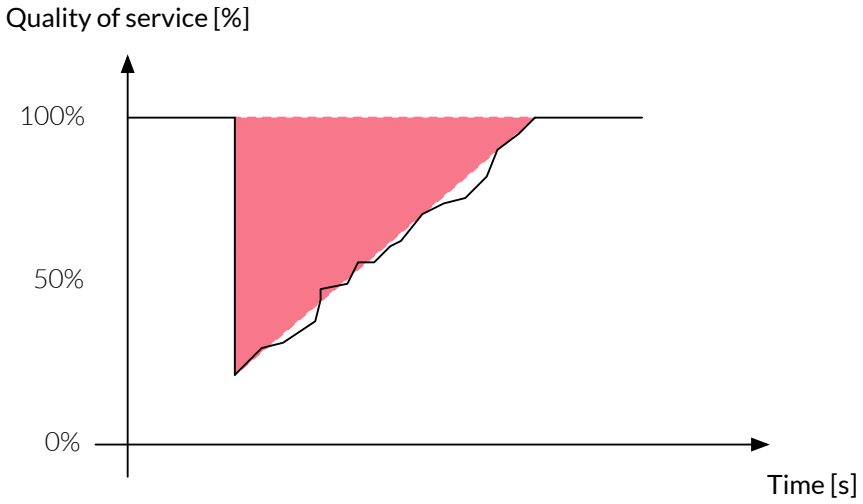


Figure 6.2: Resilience Triangle

Quantitative metrics of resilience are difficult to find since it involves external and unpredicted disturbances. Some attempts to define metrics are: time to recovery, ratio of performance before and after a disruption, and the quality of service over time (resilience triangle) [59, p. 21-22].

Bruneau and Reinhorn define four characteristics borrowed from the domain of resilience in physical infrastructure [63, p. 2].

- **Robustness** is the strength to withstand stress without losing functionality
- **Redundancy** is a way for the system to continue by using a substitute despite a disruption.
- **Resourcefulness** is the capacity to identify problems, establish priorities, and mobilize resources when a disruption happens. It can furthermore be described as the ability to recover while meeting priorities and achieving goals.
- **Rapidity** is the capability of containing losses, recovering functionality and avoiding future disruptions in a timely manner.

In addition to these characteristics *adaptive capacity* is an important concept: *"The adaptive capacity of a system can be viewed as the capacity to adapt and to reconfigure in the face of disruptions without losing functionality"* [59, p. 25]. Another definition involves *"[...] the ability to cope with novel situations"* [59, p. 25]. This is in agreement with Strigini's view describing the need for techniques working in open and interconnected environments to ensure dependability.

6.3 Our Definition

A clarification of what is meant by resilience in distributed systems is needed since the term is getting more attention. In this section, we will define a subset of resilience in the context of distributed systems and cloud computing from our perspective and experiences.

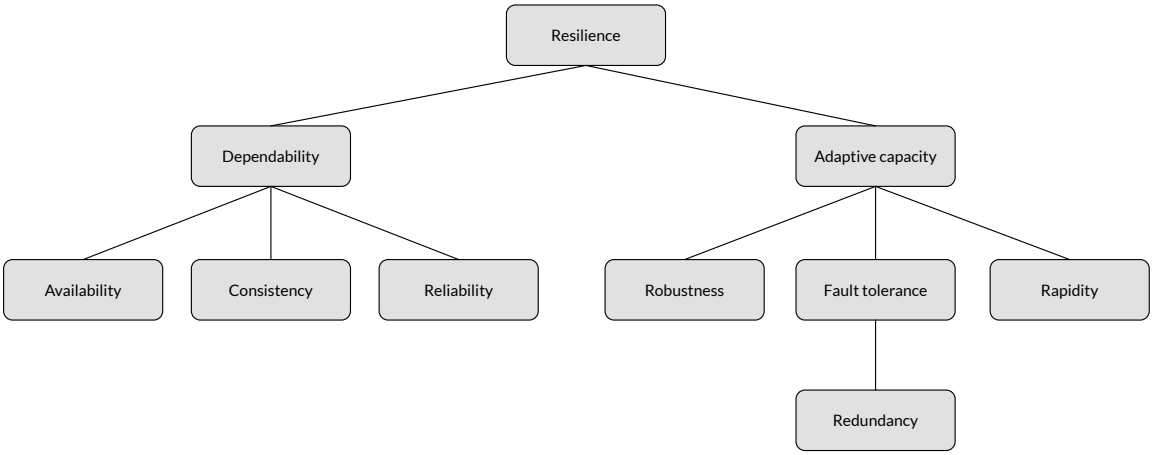


Figure 6.3: Our Resilience Definition

The illustration of resilience (Figure 6.3) shows characteristics of resilience with interconnected concepts. Dependability has been described for software for many years. An attribute such as *availability* has been described as the *"readiness of a service"*. Similarly *reliability* has been described as the *"continuity of a correct service"* [64, p. 2]. We argue that there is a lack of focus on the granularity of consistency. Ensuring high availability is often traded off by prioritizing consistency lower. Section 8.1 investigates to what extent this trade-off can be done, and quantifies the effect. The experiment is conducted by introducing errors while measuring how a circuit breaker, explained later, can help.

The adaptive capacity, the right-hand side (Figure 6.3), is another important factor for resilience, which is similar to resourcefulness. Being able to adapt is important in an ever-changing environment, which leads to the following desired characteristics. Robustness refers to standing against stress or potential disruptions and avoid failure. Fault tolerance refers to continuing when faults or errors happen, and a mean to this is redundancy. The last concept is rapidity which refers to mitigating the impact of failure and recovering to normal service. Section 8.2 investigates these concepts by quantifying how an infrastructure, in this case Kubernetes, can detect, mitigate, and handle faults, when a service becomes unavailable due to a network error. Furthermore, the effect of running multiple replicas for robustness is investigated and quantified.

6.4 Fallacies and Antipatterns in Distributed Systems

Many problems can occur if challenges are overlooked or wrong assumptions made in distributed systems. Not being aware of the challenges lead to brittleness, fragility, and in resilience terms disruptions and perturbations. Among the sources of disruptions are, according to Mansouri et al, human factors, natural factors, organizational factors, and technical factors [59, p. 16]. In 1994 eight fallacies often assumed by programmers in distributed computing were identified by Deutsch [65, p. 1]. Despite it is over 20 years ago these fallacies are still sometimes assumed (or overlooked). This is an interplay between human and technical factors in which the lack of knowledge about somewhat predictable technical factors shifts it to a human error.

Fallacies of distributed computing [65, p. 1]

- The network is reliable.
- Latency is zero.

- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Similar to these fallacies are Nygard's *antipatterns* of recurring undesired patterns in software systems. Many of these patterns are derived from integration points and the non-deterministic behavior of the underlying network. These patterns can be hard to see up front because they first occur when a service is in production. Nygard describes the connection between stability and resilience as:

> *"A resilient system keeps processing transactions, even when there are transient impulses, persistent stresses, or component failures disrupting normal processing. This is what most people mean when they just say stability. It's not just that your individual servers or applications stay up and running but rather that the user can still get work done."* ***- Nygard, 2007*** *[10, p. 24].*

Nygard's stability antipatterns are described in Table 6.1. These antipatterns are potential threats that can be hard to catch with traditional testing procedures. Several of the antipatterns happen in production, and are not that easy or obvious to test in a development environment.

Stability antipattern	Description
Integration Points	Integrations points are the external services a service is using. These require an integration over the network through e.g. a socket, which introduces a stability risk. Nygard calls this *"the number-one killer of systems"*.
Chain Reactions	Chain reactions happen if one service crashes and the remaining services must serve an increased load. The increased load increases the risk of another failure being triggered.
Cascading Failures	Cascading failures happen when a service's undesired state is propagated to another service, which then again affects a third service and so on. This can take down the entire system. Cascading failures are often caused by integration points without timeouts.
Users	Users can take down your system in various ways. More users lead to increased resources requirements, but there are also malicious users deliberately trying to break the system.
Blocked Threads	Blocked threads are often found near, but not limited to, integration points. Integration points can be guarded with timeouts, but blocked threads can also appear in the form of e.g. deadlocks which are harder to catch.
Attacks of Self-Denial	Attacks of self-denial are e.g. when special offers on a web shop are made and the traffic suddenly increases drastically. If the marketing department has not cleared this with the operations team.

Continued on next page

Table 6.1: Stability Antipatterns, Michael T. Nygaard, "Release It!"[10]

Table 6.1 – *Continued from previous page*

Stability antipattern	Description
Scaling Effects	Scaling effects can happen when you have a many-to-one or many-to-few relationship. A database is, for instance, a case where the database can reach a client limit. Scaling effects are hard to test.
Unbalanced Capacities	Unbalanced capacities are about not being able to scale up and down dynamically. When an attack of self-denial is executed you need to be able to scale up, but when it is done you do not want to have a lot of idle resources. Since Nygard's "Release It!" was released in 2007 a lot has happened with cloud providers, virtual machines and containers that makes this a lot faster. As in 2007 you are still being billed for increased load, though.
Slow Responses	Slow responses are, as mentioned, a cause for cascading failures. They can also lead to frustrated users who clicking a reload button, thereby generating even more traffic. Slow responses can furthermore lead to memory leaks and increased garbage collection.
Unbounded Result Sets	Unbound result sets are when you ask for more data than you need and discard the rest. As your amount of data grows this becomes worse and lead to slow responses.

Table 6.1: Stability Antipatterns, Michael T. Nygaard, "Release It!"[10]

6.5 Application vs Infrastructure Level Resilience

Throughout our work with resilience, we noticed the recurring distinction between application level and infrastructure level resilience. Some measures to improve resilience can be implemented in source code written in e.g. Java and Python. Other measures can be applied at the infrastructure level by handling the applications' surroundings in an intelligent way. Nygard presents several patterns to handle the previously described antipatterns. Some of these patterns will be described and put in the context of application vs. infrastructure level resilience.

Application Level Resilience

At the application level, software architecture and implementation are in focus. The software is, of course, run in some environment, but the following will describe general improvements of architectures.

Use Timeouts

The timeout pattern is concerned with the first fallacy about trusting the network blindly. The risk of blocking a thread forever and eventually draining a thread pool is present without timeouts. Slow responses can lead to cascading failures and chain reactions. The solution is to limit the acceptable waiting time and have a fallback strategy. Using the terms from Figure 6.3, this is a way of prioritizing availability over reliability by tolerating a lower level of consistency. A lower level of consistency could be a degradation in the returned content e.g. returning a top10 movie list instead of a personalized list.

Circuit Breaker
A circuit breaker protects a service from an unhealthy integration point by applying rules for when it is allowed to call the integration point. When an integration point fails too many times (e.g. timeout) it is blocked and transitions from the state *closed* to *open* (Figure 6.4). Given a configuration, the circuit breaker transitions to the *half-open* state in which it attempts a reset its state to see if the integration point is safe to call again. If the attempt succeeds the state transitions from *half-open* to *closed*, and if the attempts fails it returns to the *open* state.

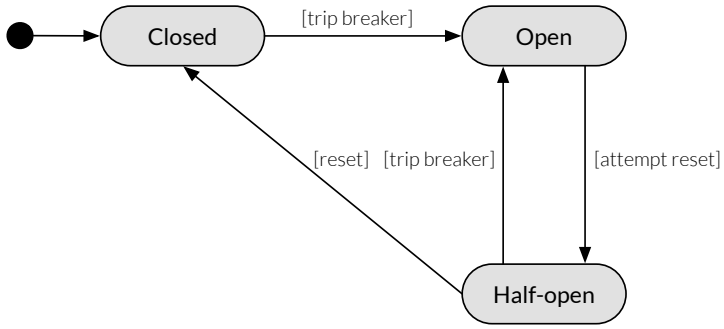


Figure 6.4: Circuit Breaker States

Fallback methods can be used together with circuit breakers to degrade gracefully instead of failing. As with timeouts, this is trading higher availability at the cost of consistency and reliability. An experiment involving circuit breakers, fallback methods and avoidance of cascading failures is presented in Section 8.1.

Fail Fast
The fail fast pattern summarizes the similarity in the two previous patterns. Failing slowly is worse than failing fast, and several antipatterns are linked to waiting. If services are synchronously chained (A calls B who calls C) much time may be wasted in the form of blocked threads. Timeouts and circuit breakers exemplify how this pattern can be implemented.

Bulkheads
Another recurring theme in the previous patterns has been the containment of damage. In ships, bulkheads are used to contain damage to a certain part of a ship (Figure 6.5).

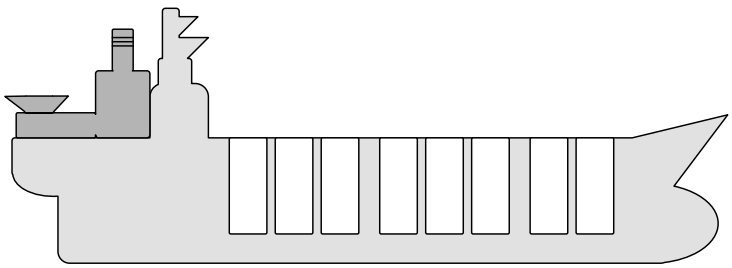


Figure 6.5: Bulkheads

Bulkheads can be applied at different levels. At the application level, the architecture can be split into multiple services such as microservices and gain the containment through multiple processes. When interacting with different integration points, different thread pools within the service can be used. The ben-

efit comes with the price of less flexibility to share resources when one of the integration points is passive.

Decoupling Middleware
Nygard describes different degrees of coupling in middleware and points out that some of the previously described problems can be mitigated by decoupling middleware. A step in this direction could e.g. be to choose a message queue instead of HTTP for communication between services.

> *"Message-oriented middleware decouples the endpoints in both space and time. Because the requesting system doesn't just sit around waiting for a reply, this form of middleware cannot produce a cascading failure."* ***- Nygard, 2007*** *[10, p. 115]*

Furthermore, Nygard states that synchronous (tightly coupled) middleware's advantage is *"its logical simplicity"* opposite of asynchronous processes that are *"inherently harder"* [10, p. 115]. Newman points out the same increase in complexity in asynchronous programming. The inherent complexity will most likely result in an increase in Mansouri et al's [59, p. 16] disruption caused by a human factor.

> *"The associated complexity with event-driven architectures and asynchronous programming in general leads me to believe that you should be cautions in how eagerly you start adopting these ideas."* ***- Newman, 2015*** *[51, p. 57]*

On the other hand, Bonér, the inventor of the Akka framework, expresses asynchronous communication as a requirement for isolation and, thereby, resilience.

> *Isolation is a prerequisite for resilience and elasticity and requires asynchronous communication boundaries between services to decouple them in: Time (Allowing concurrency), Space (Allowing distribution - and mobility – the ability to move services around)* ***- Bonér, 2016*** *[55, p. 7]*

Throughout the rest of this master's thesis, the focus will primarily be on synchronous communication. The designed course (Chapter 3) already contains many new concepts. Netflix is an example of a company using a synchronous architecture. However they have created a reactive extension called RxJava that introduces asynchronous behavior.

Infrastructure Level Resilience

At the infrastructure level, the environment around the application is in focus. The cloud stack described in Figure 4.1, from server to runtime, plays an important role when software transitions from developer machines to a production environment.

Docker has decreased the hurdle of recreating the same environment. Encapsulating everything in a lightweight container image assures identical dependencies on developer machines and in the production environment. The hardware and network topology remains different and risk failing in one way or the other. Some of these errors can be handled at the application level, but starting a new process in the environment outside the application requires something from the infrastructure level.

Test Harness
A test harness can help prepare for production in addition to existing tests. It is a way to make the application more robust, and whether it belongs to the application level or infrastructure level resilience can be discussed. The idea is to inject faults in the external network calls in the application by calling a malicious server that simulates network errors. The reaction of the system under test will show whether it is "cynical" enough [10, p. 111]. Netflix uses similar concepts presented later in this chapter.

Steady State
The last Nygard pattern we will present is Steady state which is a pattern for minimizing human fiddling and keeping servers running without intervention.

> *Every single time a human touches a server is an opportunity for unforced errors.* ***- Nygard, 2007*** *[10, p. 100]*

Compared to the previously mentioned human factor, this is about minimizing the risk of the human factor in the production environment. A similar concept is found from continuous integration in which "hurtful" tasks are automated and thereby become less error-prone. The quote behind this is: *"if it hurts, do it more often"* [66]. This results in a declarative definition of how to perform a challenging task instead of relying on memory.

Replication and Redundancy
If a service instance crashes or fails, an action must be taken. In this regard, fault tolerance and redundancy can switch to a backup instance. A resilient reaction when an instance fails is replacing the failed replica (instance) with a new one. By running redundant instances, service failures can be mitigated.

Replication also serves the purpose of improved performance by adding a horizontal load balancer in front of multiple replicas. In this way, a given workload is distributed across several replicas. The performance can be improved by adding more replicas.

These mechanisms are obtained at the infrastructure level, and a cluster management system such as Kubernetes can maintain a number of running replicas. In Kubernetes, a desired state can be specified. A control loop then measures the observed state and compares it to the desired state. If the two states are unequal, an action will be triggered to align the two states. In this case, the infrastructure must provide a mechanism for only directing traffic to the 'healthy' replicas. Directing the traffic is related to *rapidity* which refers to how fast an error is identified and an action taken. An experiment in how Kubernetes handles this will be performed in Section 8.2.

Master Election
High availability (HA) systems require resilience. Single point of failure does not harmonize with resilience. When running a cluster management system, such as Kubernetes or Mesos, the concept of a master is used. To ensure availability when a master, for some reason, disappears a strategy to elect a new master is needed. The concept is: the current leader sends out a heartbeat to maintain the position as the leader while several candidates race to become the leader [45]. The difficult part is reaching consensus, and several tools such as ZooKeeper, etcd and Consul exist for this purpose. We will not focus more on this topic throughout the thesis, but it is an important part of ensuring high availability.

6.6 Different Approaches to Resilience

Other approaches achieving or assisting the degree of resilience can be applied. These approaches will be described in this section and compared to the previously mentioned methods.

Formal Methods

Formal methods are for instance used to ensure safety-critical systems against faults. In that sense, formal methods provide fault avoidance. On embedded devices the software stack does not contain as

many layers as in cloud computing. Candea and Fox questions the need for formal models in Internet systems since the validity of the model does not take the underlying complexity into account.

> *Computer scientists have tried to use prescriptive rules, such as formal models and invariant proofs, to reason about software. These rules, however, are often formulated relative to an abstract model of the software that does not completely describe the behavior of the running system (which includes hardware, an operating system, runtime libraries, etc.).* ***- Candea and Fox, 2003*** *[67, p. 68]*

For this reason and the need to respond to changes quickly, we have not worked with formal methods.

Simulations

CloudSim is a modeling and simulation framework for cloud computing infrastructures and service created at the University of Melbourne that can simulate components such as data centers, virtual machines, power consumption, memory provisioning, and hosts. Workloads can be simulated on different models of cloud infrastructures. CloudSim has also been used for teaching purposes [12].

Chaos Engineering at Netflix

Netflix has an interesting approach to ensure resilience and to some extend even antifragility. They introduce faults in their production environment in different ways to learn how to handle failures. Bennet and Tseitlin explain the approach in the following way: *"By frequently causing failures, we force our services to be built in a way that is more resilient." [68]*
They embrace failures and apply an approach of 'chaos engineering'.

Netflix uses several different tools and have created a 'simian army' that contains several destructive tools named after monkeys. These monkeys are, from time to time, let loose to harden their software, check that everything is as expected, and test specific components. Some of the tools are explained in the following. *Chaos Monkey* is a tool to kill virtual machine instances randomly on their cloud provider (AWS). *Janitor monkey* cleans up by shutting down the unused resources. *Conformity Monkey* shuts down instances not following the best practices, leading to a relaunch. *Chaos Gorilla* is similar to Chaos Monkey, but instead of killing a virtual machine it simulates an outage in an AWS availability zone. *Chaos Kong* does the same as Chaos Gorilla, but simulates outages in AWS regions which consist of multiple availability zones. In this case, all traffic must be directed to another region e.g. from US-WEST to US-EAST regions. The idea behind these concepts is described by the Netflix Chaos Team: *"We knew that we could rely on engineers to build resilient solutions if we gave them the context to *expect* servers to fail." [69]*. This approach is very similar to *crash-only software* in which software is designed to *"crash safely and recover quickly" [67, p. 1]*.

On September 20th 2015 an AWS region became unavailable and a lot of large Internet sites were down, but Netflix's preparations helped them - similar to antibody in medicine:
"Netflix did experience a brief availability blip in the affected Region, but we sidestepped any significant impact because Chaos Kong exercises prepare us for incidents like this." [69].

7

Building a Tangible Cloud Computing Cluster

Visualization and tangibility are important in conveying cloud computing concepts to accomodate engineering students' learning style preferences. In order to align the students' mental model with the real world, a tangible cloud computing cluster needs to be designed and implemented as a mediating object.

The previous three chapters introduced many cloud computing concepts. In order to teach and convey these concepts, we introduce a small-scale cloud computing cluster, KubeCloud, as a mediating learning object. The focus of this chapter is the process of specifying, designing, and implementing KubeCloud (Figure 7.1).

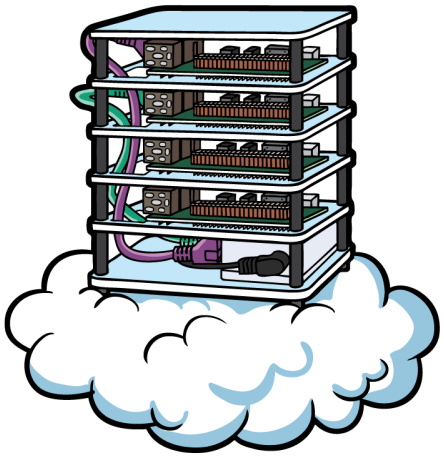


Figure 7.1: Tangible Cloud Computing Cluster

KubeCloud is designed to accommodate the learning style preferences identified by Felder and Silverman (Section 2.3): sensory, visual, active, and sequential. These learning style preferences confirm the need for a learning object. Our experiment, presented later, confirmed three out of four of these preferred learning styles for the students participating in the course presented in this master's thesis. To accommodate the students' preferences and to create an active, sensory learning environment, KubeCloud shall be able to visualize and present concepts and allow for practical hands-on group work to foster social interaction and experimentation. In agreement with Churchill's classification of learning objects, KubeCloud shall be used as a: Presentation object, Practice object, Simulation object, and Conceptual object. KubeCloud has to be a practice object to foster learning by doing. The workshop format involves the use of KubeCloud

as a practice object. Presentations of concepts in cloud computing have to be demonstrated using KubeCloud. KubeCloud will, therefore, act as a presentation object. KubeCloud shall, furthermore, improve the students' skills of real life technologies, and must, therefore, act as a simulation object representing a small-scale real-life system. Lastly, KubeCloud further acts as a conceptual model e.g. by visualizing a small-scale data center.

7.1 Requirements for a Learning Object

In order to specify the requirements for KubeCloud, we must understand the constraints. The course budget constrained the total cost to DKK 13,000 (USD $1.957[1]). The budget should allow for a minimum of five clusters (20 students with four students per cluster) to be handed out. Another major constraint is the limited implementation time before the course start (thesis start: January 25th, 2016; course start: April 7th, 2016). Before the implementation, a set of requirements are specified for KubeCloud within the limited constraints:

- KubeCloud shall be tangible and allow for practical hands-on experimentation
- KubeCloud shall allow for deployment of microservice architecture
- KubeCloud shall allow for containerization with Docker
- KubeCloud shall allow for container management with Kubernetes
- KubeCloud shall allow students to pull out cables to inject failures
- KubeCloud shall be able to visualize the state of the cluster
- KubeCloud shall visually have characteristics of a server rack
- KubeCloud shall be transportable
- KubeCloud shall have a small physical form factor and be able to be carried around in a bag
- KubeCloud shall allow for shutdown and restart with a frequency twice a week
- KubeCloud shall allow for configuration of the cluster size

The following sections will describe and discuss the choices made in order to satisfy these requirements.

7.2 Physical Design

The physical design of KubeCloud is very important in order to satisfy the requirements. The process of building and designing the cluster started out with sketching the possible physical designs. After the initial process of sketching, 3D models of the cluster with the components were made to visualize the cluster and compare it to images of real datacenters. Lastly, an exact 3D model of the individual parts needed to be manufactured was created and sent to the fabricator. The process of the physical design can be seen in Figure 7.2.

[1] DKK-USD currency: 664

(a) Sketch (b) 3D model (c) KubeCloud

Figure 7.2: Sketch, 3D, and Cluster

The requirement *"KubeCloud shall visually have characteristics of a server rack"* should afford visual associations of working on a scale-model of a data center. KubeCloud consists of stackable layers of Raspberry Pis similar to servers stacked in a server rack in a real data center. Figure 7.3 shows the similarity between a server rack and KubeCloud.

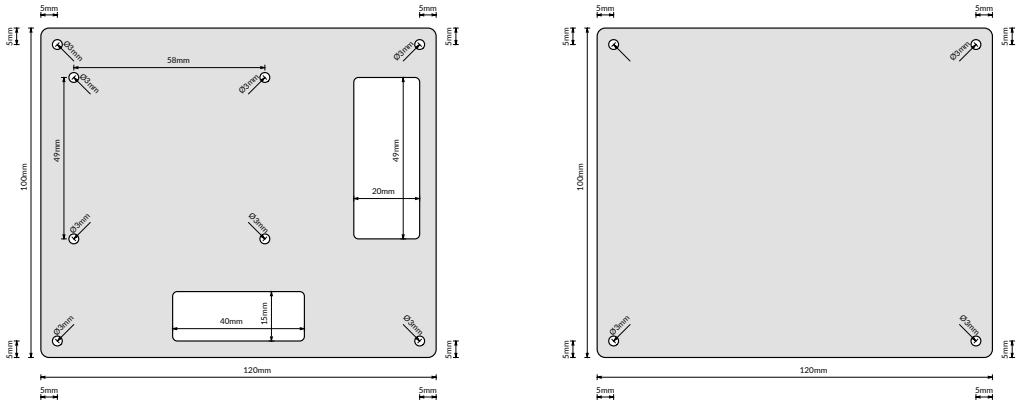
(a) Rack (b) KubeCloud (c) KubeCloud

Figure 7.3: Rack Versus Cluster

Figure 7.3 (a) shows a real server rack in a data center. Figure 7.3 (b) shows multiple KubeClouds comprising a larger scale data center compared to a single KubeCloud. Figure 7.3 (c) shows the dynamism of KubeCloud and the ability to configure the cluster size from a physical perspective. Multiple KubeClouds can easily be mounted together to form a larger small-scale cloud computing environment. This satisfies the requirement of *"KubeCloud shall allow for configuration of the cluster size"*.

To satisfy the requirement of *"KubeCloud shall have a small physical form factor and be able to be carried around in a bag"*, the power and network connectors are enclosed in the body of the physical design in order to avoid damage during transportation. This further satisfies the requirement *"KubeCloud shall be transportable"*. The physical size of KubeCloud is made as small as possible within the constraints of the size of the used components (Raspberry Pi and dlinkgo) and the concealment of cables. Figure 7.4 shows the technical drawings of the two different layers the cluster is comprised of. (a) is the layer the Raspberry Pi is mounted on, (b) is the top and the bottom layer. The 3D model of the two layers can be downloaded from our blog[2].

[2] `http://rpi-cloud.com/guide-how-to-build-a-raspberry-pi-cluster/`



(a) Raspberry Pi layer

(b) Top bottom layer

Figure 7.4: Technical Drawings 1:2.5

One of the important purposes of KubeCloud is to demonstrate the effect of failures e.g. network failure. The hole, seen on the right side of Figure 7.4 (a), makes the ethernet port of the Raspberry Pi accessible and allows for pulling the plug through this hole. This satisfies the requirement of *"KubeCloud shall allow students to pull out cables to inject failures"*.

7.3 Hardware

In order to build KubeCloud, choices of hardware need to be determined within the constraints of the budget. This section will describe the chosen hardware and present the total cost of constructing KubeCloud clusters. A single KubeCloud will contain four nodes and a switch.

Server

The physical computers (servers) in the cluster have to be chosen. Previous efforts in building small-scale cloud computing environments have chosen the Raspberry Pi as computational units [13, 14, 15]. The Raspberry Pi is a cheap, yet powerful little machine (900MHz Quad Core Processor, 1GB RAM, ARMv7), that does not require extra cooling, and that easily can be mounted on a surface. Moreover, the Raspberry Pi has the ability to run many different Linux distributions. The low price tag makes it ideal for building a small cluster. Each Raspberry Pi has a 16 GB Kingston Micro SDHC card and is powered by the official Raspberry Pi 2A power supply.

Switch

The Raspberry Pi's Ethernet port is attached via the USB 2.0 bus, which results in the upstream bandwidth not supporting gigabit speeds. The speed requirements for the switch, therefore, did not play a significant role in the selection. A 10/100 Mbps switch is sufficient, and the dimensions and the number of ports become the most significant details of the switch. The smaller the better. The chosen switch is a Dlinkgo Fast Ethernet Switch GO-SW-5E with five Ethernet ports (10/100 Mbps). The cables were chosen to be Ethernet Cat5e UTP. The most significant part of choosing cables was length, price, and color of cables. The colors make it easy to distinguish between the nodes in the cluster which is important during exercises.

Cost

The actual cost (excl. VAT) of building a KubeCloud cluster is shown below.

Item	Description	Price (DKK)	Qty	Total (DKK)
1	Raspberry Pi 2 Model B	244.15	4	976.60
2	Raspberry Pi 2 power supply	34.11	4	136.44
3	Kingston MicroSDHC 16 GB	37.96	4	151.84
4	Dlinkgo Switch GO-SW-5E	53.16	1	53.16
5	Cat 5e UTP Network cable 0.25m - Orange, Violet, Yellow, Green	4.00	4	16.00
6	Cat 5e UTP Network cable 1.50m - Blue	9.60	1	9.60
			Total	1343.64

In addition to the KubeCloud clusters, an additional 16-port switch and a router is purchased. The router is needed to create a WiFi network in the classroom for easy access to the clusters. The rack is manufactured at the local tool shop at Aarhus University School of Engineering. To ensure each student gets hands-on time with the cluster, a total of 8 clusters are built. The total cost of the materials used within this present master thesis is:

Item	Description	Price (DKK)	Qty	Total (DKK)
1	Raspberry Pi Kubernetes cluster	1343.64	8	10749.12
2	D-Link DIR-816L AC750 Router	239.20	1	239.20
3	Sempre Switch 16-port 10/100	185.00	1	185.00
4	Materials at toolshop (approx)	200.00	1	200.00
			Total	12716.96

The total amount in USD excl. VAT is approximately $1915[3]. The price per student assuming four students per cluster is USD $60. The total cost is within the constrained budget.

7.4 Network Topology (Physical)

The physical network topology of a KubeCloud is set up in a star-topology with the switch as the center of the cluster. Figure 7.5 illustrates this setup.

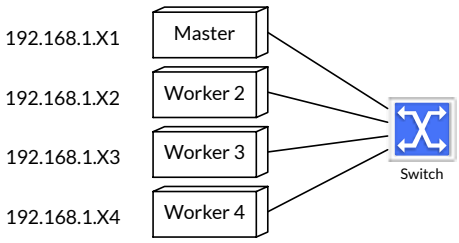


Figure 7.5: Network Topology

[3] DKK-USD currency: 664

The static IP convention in each KubeCloud cluster is depicted in Figure 7.5. The static IP convention assumes clusters of four nodes with the last of the four octets representing a group number and the location of the node in the cluster. The first digit of the last octet determines the group number while the second digit determines the location. The location follows a convention of the master node being the topmost physical location. The following worker nodes and IPs are increasing downwards, e.g. group 1 will have the following nodes: 192.168.1.11 (Master), 192.168.1.12 (Worker 2), 192.168.1.13 (Worker 3), 192.168.1.14 (Worker 4).

An alternative to static IPs is to use zero-configuration networking implementation such as Avahi. A test was conducted in the beginning of the project, but the stability of using Avahi was not satisfying in our case. Therefore, the static IP convention was chosen instead. Static IPs are rigid and can limit the number of nodes in a cluster following the naming convention, but stability was the crucial parameter. This way of configuring the IPs of the cluster allows for easily connecting the single cluster to the entire network of clusters used in the classroom. This decoupling of clusters allows for custom configuration of cluster sizes, which will be explained in the software section of this chapter. When putting it all together, the clusters are again set up in star-topology as depicted in Figure 7.6.

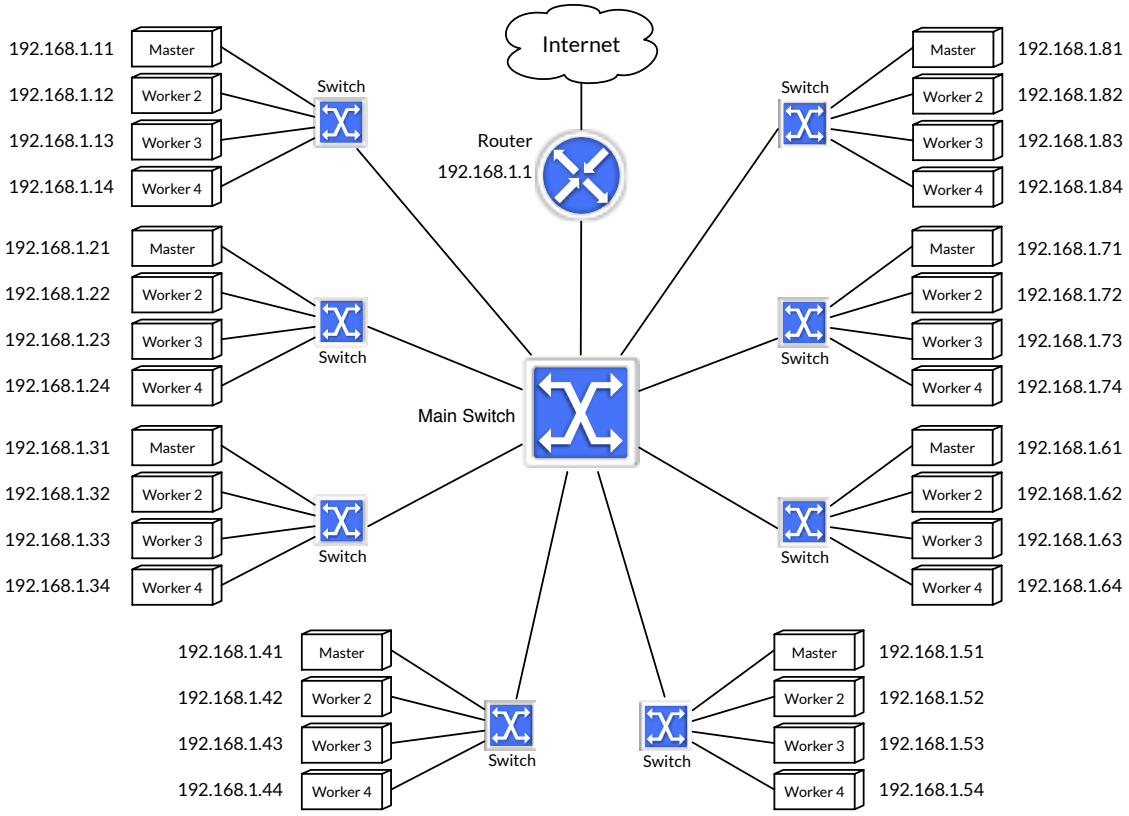


Figure 7.6: Overall Network Topology

The router act as the gateway for the internet connection needed to access the Docker Hub repositories.

7.5 Software

The previous sections described the process of designing and building a physical cluster. We now turn the focus to the decisions made regarding software in building KubeCloud. The involved layers will be described and discussed from the operating system layer to the application layer.

Operating System

Hypervisors are often used in data centers (Chapter 4) and could be an interesting topic to investigate. The limited resources on a Raspberry Pi, however, does not allow for powerful enough partitions of the resources into virtual machines. Furthermore, the partitioning contradicts the purpose of the physical representation of the concepts in cloud computing. KubeCloud is, therefore, a bare-metal cluster without virtual machines.

The choice of operating system (OS) is important since the rest of the software need to run on top of it. Stability is one of the most important properties of the OS for KubeCloud. A Linux distribution is needed to run Docker and Kubernetes. We did not have a particular distribution preference but looking at the available options in the ARM community led to the choices of HypriotOS, Raspbian, or Arch Linux. The intention was to build on top of already configured distributions to keep the focus on the usage of the cluster.

The Road to a Stable OS

Several different configurations of operating systems combined with different Docker and Kubernetes versions have been tried out on the road to stability to address the requirement *"KubeCloud shall allow for shutdown and restart with a frequency twice a week"*.

HypriotOS with k8s-on-rpi
At first, HypriotOS[4](v0.6.1) was tried out. HypriotOS is an extension of the default Raspberry Pi OS (Raspbian). Installation of HypriotOS is easy, and it has Docker installed out of the box. HypriotOS seemed like a good solution since Kubernetes was easy to get up and running on the clusters using k8s-on-rpi[5]. Unfortunately, stability issues arose after restarting the clusters, which became evident when Kubernetes became unresponsive. The time used for scheduling of pods experienced an increase and failed in many cases.

HypriotOS with kubernetes-on-arm
Another Kubernetes project called kubernetes-on-arm[6] was installed on top of HypriotOS (v0.6.1), but the stability issues were still present. The debugging was time-consuming since we had to flash SD cards, assign static IPs, install Kubernetes, and wait for the errors to appear.

ArchLinux with kubernetes-on-arm
Since the common factor in the two previous configurations was the operating system, a new operating system was investigated. An ArchLinux installation from the kubernetes-on-arm project turned out to (almost) be the end of the road of our stability issues. In order to automate and adjust the project to our setup, scripts were created for automation purposes. These scripts are described in further detail in the

[4] `http://blog.hypriot.com/downloads/`
[5] `https://github.com/awassink/k8s-on-rpi`
[6] `https://github.com/luxas/kubernetes-on-arm`

upcoming section on Kubernetes.

Docker

As described in Section 4.2, containers such as Docker have gained much attention the last couple of years because of the benefits they provide. Figure 7.7 shows Docker's role in the cluster. Applications (Spring Boot) are containerized into a Docker image with the needed dependencies. The Kubernetes infrastructure is using the Docker images to manage the cluster.

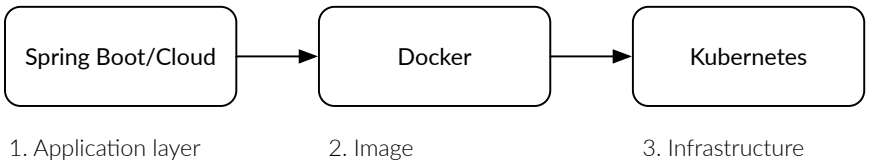


Figure 7.7: Flow Overview

Installing Docker on a Raspberry Pi is not difficult, but since the Raspberry Pi runs on ARM architecture, all Docker images must be built for ARM. Most developer machines do not use ARM, and cross-compilation or building on an ARM device is, therefore, necessary. One of the ideas behind Docker is that applications and dependencies can be packaged together and recreated in the production environment. The ARM architecture limitation is unfortunate, but it has not been a big problem. Two very similar base images of Java 8 have been used for the ARM and amd64 architectures to accommodate this challenge.
The previously described ArchLinux configuration comes with the installation of Docker used in KubeCloud.

Kubernetes

Kubernetes (v1.2) was chosen as cluster management system for several reasons. The previously mentioned transformation from being machine-oriented to application-oriented combined with the increased tendency of containerization led to the decision of using containers and Kubernetes. Furthermore, containers are more lightweight than virtual machines, and since the Raspberry Pi has restricted resources compared to a server it seemed like a good idea to reduce the overhead. In order to illustrate concepts of resilience, Kubernetes offers several interesting approaches to adaptive capacity (Figure 6.3) on the infrastructure level resilience. Among those are replication, heartbeats, and automatic rescheduling when pods fail, which leads to rapidity. Many of Kubernetes' concepts can be used to illustrate resilience on a physical cluster.

Kubernetes is used to manage Docker containers on the Raspberry Pis in KubeCloud. Each Raspberry Pi node is registered as a node in the cluster and is thereby a small-scale bare-metal server running Kubernetes. On managed solutions such as Google Container Engine, a virtual machine is used per node instead of a Raspberry Pi. In KubeCloud, Kubernetes schedules workloads (containers) across the different Raspberry Pis (nodes).

As previously described, the road to stability led to a configuration with ArchLinux and the kubernetes-on-arm project. Kubernetes-on-arm furthermore comes with many features such as a command-line tool to enable masters and workers and to manage add-ons. One of the most important add-ons has been the KubeDNS add-on, which makes it possible to use a service's name as the hostname instead of using

IP addresses. The command-line tool makes it possible to change the role of any of the nodes seen in Figure 7.6. A master can with a single command be configured as a worker of another master. In our setup, each cluster has a master and three workers, but this structure can easily be changed if a cluster with e.g. seven workers is needed. The command-line tool has furthermore served as a building block in the previously mentioned automation scripts. Scripts for startup and shutdown of the clusters were created to ease the startup process and, more importantly, to make sure that the cluster was shut down correctly. The shutdown script disables Kubernetes on all nodes, waits for them to finish, and shuts down the Raspberry Pis.

One of the essential assumptions in Kubernetes is that pods can communicate with other pods regardless of which host they are scheduled on. As described in Section 4.3, all pods have their own IP address, which frees the user of Kubernetes, in most cases, for port mappings between container ports and host ports. In order to fulfill this assumption, Kubernetes can use different implementations depending on the platform it is deployed to. KubeCloud leverages an open source project called flannel which is included in kubernetes-on-arm.

Flannel is a software defined overlay network, also called a virtual network, that assigns a subnet to each host for use with the container runtime, which in this case is the Docker engine. An example of the virtual network provided by flannel is shown in Figure 7.8.

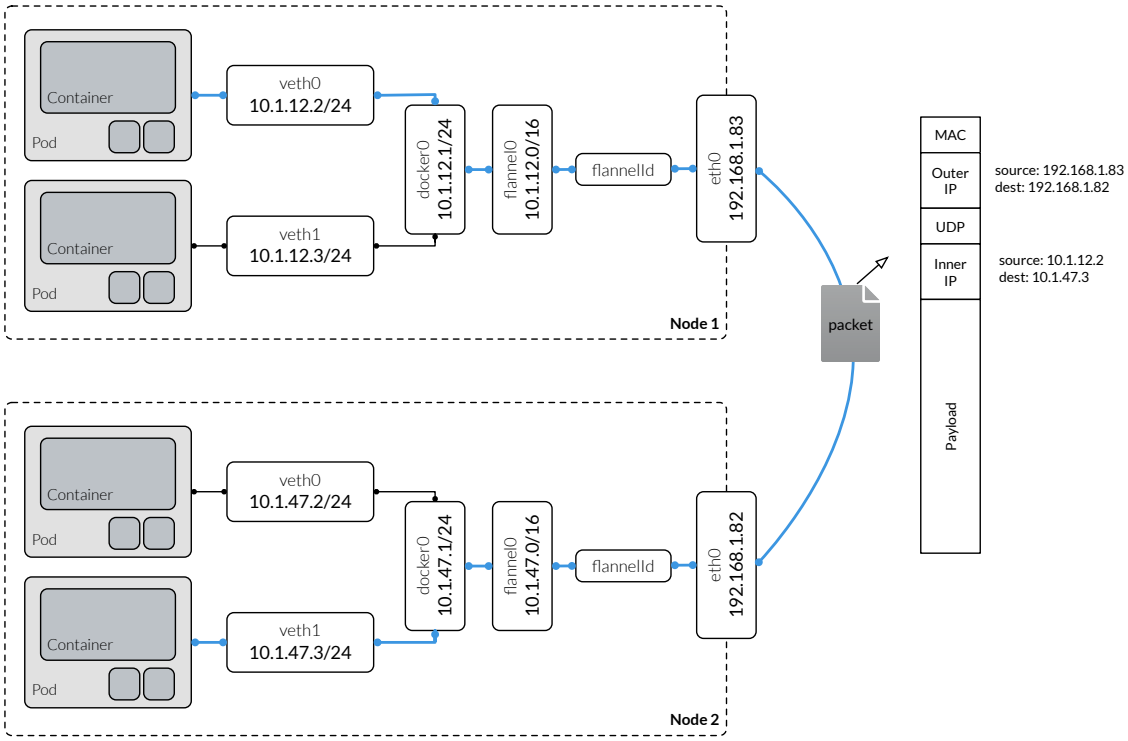


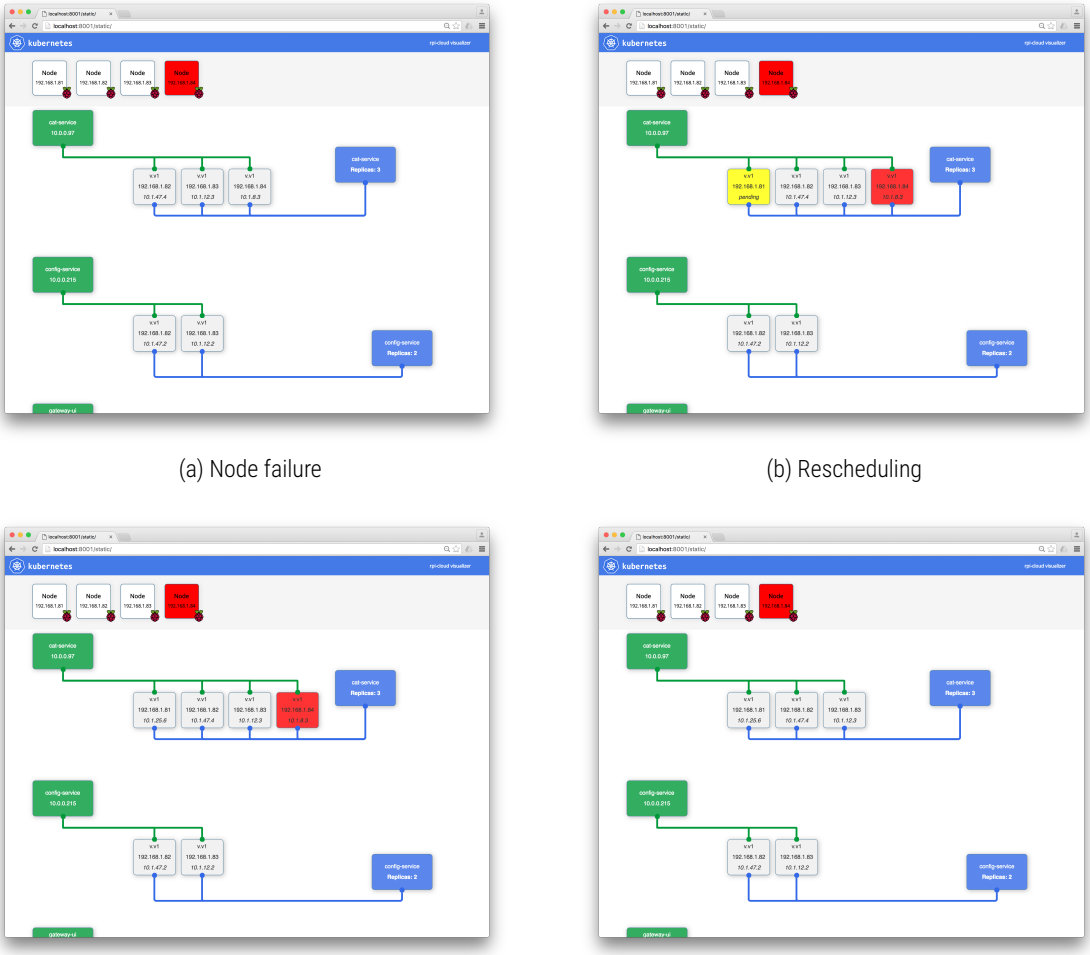
Figure 7.8: Flannel Networking

Figure 7.8 shows two nodes in a Kubernetes cluster. Each node has the Docker engine and a flannel agent running. The example shows how the pod-to-pod communication is achieved. Every pod is assigned a virtual IP provided by the docker0 subnet. The docker0 subnet is controlled and assigned by flannel. Flannel stores these configurations in etcd, which makes it possible to map between host IP and virtual IP. As seen from the payload, Node 1's packet contains an inner IP destination with the virtual IP of the pod

and an outer IP destination with the host machine's IP (Node 2). This mapping helps decouple pod and machine, and multiple pods on the same machine can expose the same port.

Visualizer

Visualization is expected to play an important role in students' learning and the bridging between cloud computing and the physical machines, especially since engineering students tend to be visual learners. This fulfils the requirement *"KubeCloud shall be able to visualize the state of the cluster"*. KubeCloud allows for injecting failures e.g. by pulling the network cables (Figure 7.9). However, it can be hard to understand and grasp what is going on in Kubernetes. Hence, a visualization tool is needed.



(a) Node failure

(b) Rescheduling

(c) Recovered

(d) Clean up

Figure 7.9: Visualization of Node Failure

The top row indicates the status of each node, and the red color indicates that the last node is not responding. Each green box corresponds to a running *service*. The services are connected to *pods* of varying colors. The pod color indicates whether a pod is running, pending or terminating. The pods are controlled by *deployments* that are shown as blue boxes.
As seen in Figure 7.9, the overview of the cluster changes dynamically when the state of the cluster is changed. Figure 7.9(a) shows that one of the nodes is not reporting heartbeats to the API server. (b)

shows Kubernetes rescheduling the pod running on the failing node to maintain the desired state of the deployment. (c) shows the recovery to the desired state. Lastly (d) everything is cleaned up and fully recovered.
Brendan Burns, co-founder of the Kubernetes project, has created the original visualization tool. This tool has been open sourced and a lot of contributions have been made by, among others, Ray Tsang (saturnism) and Arjen Wassink (awassink). We have forked their repository and made some additions. The additions to the visualizer made during this master's thesis are, among others, fixing a memory leak in the original code. Drawing of nodes in the DOM kept being overlayed every 3 seconds. The result of having the visualizer open for longer periods made it heavy to interact with. Furthermore, the upgrade from Kubernetes 1.1 to 1.2 resulted in issues. ReplicationController were substituted with replica sets and deployments. We updated the visualizer to work with 1.2.

Application Layer

The two previous sections described how Kubernetes runs and manages containerized applications (Figure 7.7). Now we will focus on the applications running on top of the infrastructure. The application layer is not limited to a single language or framework since Docker containers are used. In order to experiment with microservices, a microservice chassis framework is chosen: Spring Boot and Spring Cloud. These frameworks make it easy to get up and running with a microservice architecture in Java. Furthermore, Spring Boot and Spring Cloud offer many relevant cloud computing libraries that easily can be included in projects. Multiple Netflix Open Source libraries are accessible through Spring Cloud e.g. an implementation of a circuit breaker for synchronous communication (Hystrix) and an implementation of the API Gateway pattern (Zuul). These libraries and examples of implementing them are described in further details in Appendix E.

KubeCloud is used as a presentation item as well as a practice item. In order to accommodate the role as a presentation item a demo project demonstrating some concepts of microservices, Docker and Kubernetes has been implemented. Secondly, the demo project supports the role as a practice item by providing a reference point for the students to get started. The demo project is a simplified microservice architecture to keep the initial complexity and understanding of the system low. The architecture contains four services as depicted in Figure 7.10.

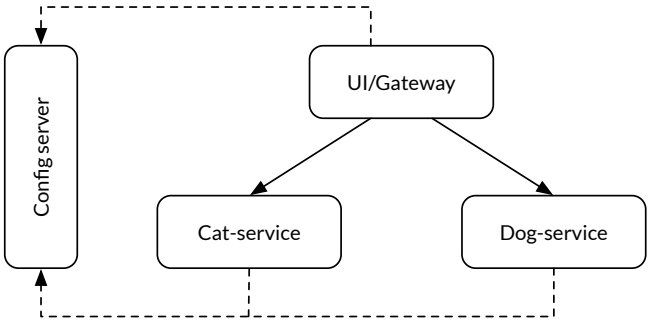


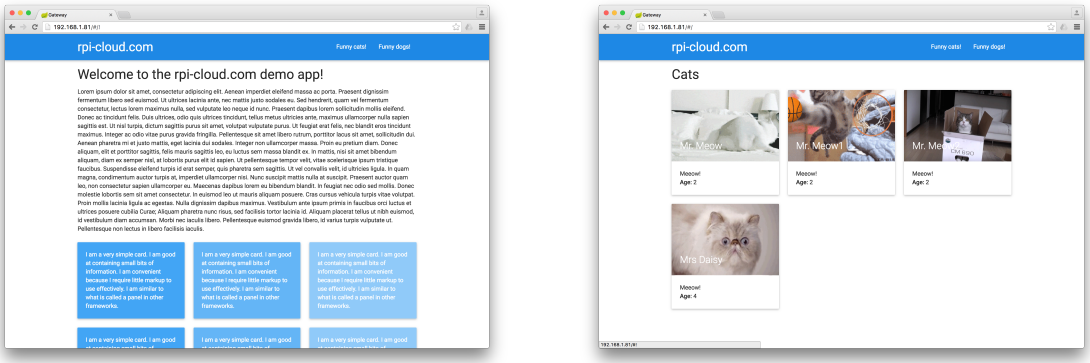
Figure 7.10: Architecture of Demo Application

The four services are:

- **Config-server**: The config server is implemented with the Spring Cloud Config, and acts as a central configuration management hub.

- **UI/Gateway**: The UI/Gateway is an implementation of the API Gateway pattern along with the graphical user interface (GUI) of the application.
- **Cat-service**: The cat-service is a RESTful service, returning a list of cat-objects
- **Dog-service**: The dog-service is a RESTful service, returning a list of dog-objects.

The graphical user interface makes requests to the API Gateway that then directs the calls to the specific service. The application is shown in Figure 7.11 in which the data from cat-service is shown in Figure 7.11(b).



(a) UI Gateway (b) Cats v2

Figure 7.11: Screenshots of Demo Application

The previously mentioned DNS add-on for Kubernetes is used for service discovery for the individual microservices in this dynamic environment. The DNS add-on allows inter-service communication by using the Kubernetes services' names instead of IPs. Figure 7.12 shows an example where a service makes a request to **http:/config-service**. This service name will then be attempted resolved on the host by looking up the designated DNS services. The resolv.conf file, from the host machine, contains the IPs of the DNS services. The KubeDNS virtual IP is added to resolv.conf by kubernetes-on-arm. KubeDNS is then contacted and looks up **config-service** in etcd. If found, the resolved IP will be sent back, and the service name is resolved to a virtual IP that the request is sent to as specified in Figure 7.8.

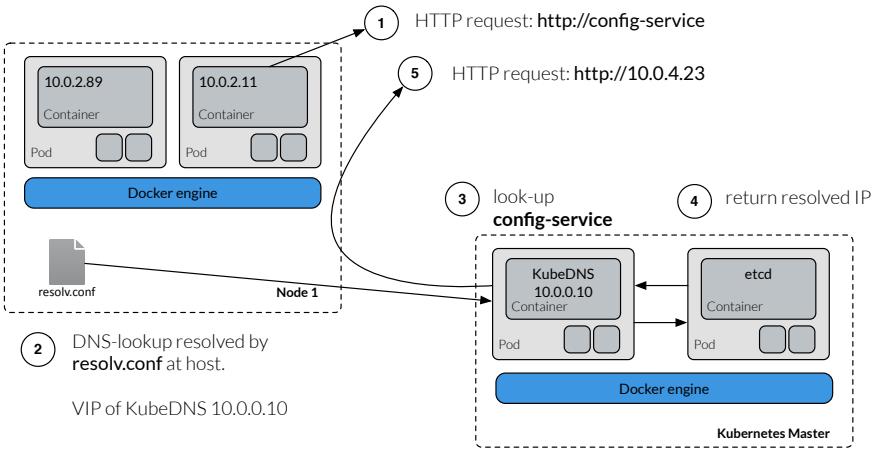


Figure 7.12: KubeDNS

7.6 Load and Stress Testing

When applications are developed and deployed in KubeCloud, we need to simulate user traffic on the cluster to apply real use cases. KubeCloud then becomes a controlled test environment for cloud computing research. In order to simulate traffic and experiment with how applications respond to different traffic loads, different attacks can be made. There are two different ways of attacking the application: *load* and *stress testing*. A load test is *"a test focused on determining or validating performance characteristics of the software under test when subjected to workload models and load volumes anticipated during production operations"* [70]. In a stress test the subject to workload is pushed beyond the anticipated production operations. Both types of tests have been performed in this thesis. In order to perform load and stress testing Vegata[7] and Gatling.io[8] have been used. Vegeta allows for a constant request rate against a target whereas Gatling records usage scenarios and replay them.

7.7 Benefits and Limitations

The benefits and limitations met in the process of designing and implementing KubeCloud will be described in this section.

Benefits

Hands-on
The cluster allows students to interact and experiment with a physical model of the concepts. Raspberry Pis are used as server machines (Kubernetes nodes) instead of spinning up virtual machines on a cloud provider. This enables the students to inject failures such as pulling the network cable, attacking with a large traffic load, etc.

Visualization
The visualization tool is not restricted to Kubernetes in a Raspberry Pi cluster. The representation of the nodes makes it easier to understand where workloads are scheduled and how the cluster maintains the desired state.

Affordable Test Environment
Buying a test environment is expensive in terms of hardware, cooling, installation and power consumption. An example from the University of Helsinki spending around one million Euros without cooling systems [14, p. 170-171] illustrates how expensive this solution is. By contrast, KubeCloud is cheap, portable, and has low power consumption. KubeCloud is not as performant as a full-size data center, but the same concepts and network errors can be experimented with in a closed test environment.

Transportable
The fact that KubeCloud is transportable allows students to use the cluster whenever they want to. No expensive installation of a large data center is necessary.

Configurable
Since KubeCloud can be configured in numerous ways, the clusters can also be used with a varying amount of workers per master. The physical design allows the physical clusters to be assembled with a varying

[7] `https://github.com/tsenart/vegeta`
[8] `http://gatling.io/`

amount of Raspberry Pis.

Flexible

Another benefit is KubeCloud's flexibility in terms of what software it can run. Docker allows everything that can be packaged into a Docker image (on ARM) to run in the cluster. This makes it easy to have different runtimes available without causing conflicts between different versions among the users. Since Docker can be used without Kubernetes it is not needed to understand Kubernetes to use the cluster.

Limitations

ARM Architecture

The largest limitation is that Raspberry Pis use the ARM architecture. The benefit of low price and lower power consumption comes with trade-offs. Most developer machines do not use ARM architecture which leads to an unfortunate situation concerning Docker. Containers must be built on an ARM machine or cross-compiled, but this introduces the issue of different execution environments that Docker is trying to solve in the first place. Running ARM-built containers, on the other hand, might be accessible from a developer machine in the near future[9]. The fragmentation that ARM architecture introduces is, though, still unfortunate in regards to what images can be applied.

Build/Deployment Process Slow

When a microservice is updated with new functionality, it takes some time before it is running in the cluster. In the process a new Docker image must be built and pushed to a registry, configuration files updated, and a deployment with the new image started. Because of the Raspberry Pi's performance pulling and pushing Docker images take some time. This manual process could be automated by leveraging concepts from continuous delivery.

Power Supply

In KubeCloud, each Raspberry Pi has its own power supply connected to a power strip. This results in many cables, and a better solution would have been a power hub built into the physical construction. However, due to budget constraints, this was not possible.

Stability Concerns

In order to sustain a stable Raspberry Pi cluster, all nodes must be shut down correctly. The portability of the cluster introduces a risk of stability issues opposite a datacenter in a basement. Pulling the power plug can lead to corrupt SD cards or invalid state in the Kubernetes cluster.

[9] `http://blog.hypriot.com/post/first-touch-down-with-docker-for-mac/`

8

Experiment: Evaluating Application Level and Infrastructure Level Resilience

KubeCloud allows for physical experimentation that cannot be done using cloud providers. Experimenting with resilience at the infrastructure and application level is a key feature in order to demonstrate the new paradigm of embracing failure.

Until now, we have covered many different topics. In the introduction, we described the paradigm shift from monolithic applications to distributed microservice architectures and the increased complexity this architectural style brings. This increased complexity requires new methods to ensure resilience, and thereby new ways of teaching are needed. As described in Chapter 6, we outline methods to achieve resilience as application level and infrastructure level. Nygard describes multiple patterns for achieving a higher degree of resilience in production systems. However, to our knowledge, proper experiments have not been performed to verify the effects of these patterns. KubeCloud provides a controlled test environment in which it is possible to conduct experiments with these patterns e.g. pulling a network cable in a cloud provider's data center is not possible. Furthermore, these experiments will underpin the importance of designing for failure and support the students' understanding of resilience. The students have further conducted the following experiments to verify the results: application level resilience and infrastructure level resilience regarding recovery time.

All tests in this section are performed using a load testing tool, Vegeta.

8.1 Application Level Experiments

Design of Experiment: Integration Points in a Synchronous Architecture

Application level resilience refers to methods that can be applied by the developer to achieve a higher degree of resilience. The experiments described in this section will focus on integration points in a synchronous architecture and the derived antipatterns. We will investigate the effects of applying patterns as a solution to these antipatterns.

Integration points in a synchronous architecture must be handled with care. Nygard describes integration points as *"[...] the number-one killer of systems"* [10, p. 46]. He points out that timeouts and circuit breakers can be a solution. This experiment illustrates and documents the effects of circuit breakers when a

service becomes unresponsive.

The test setup consists of three services running in KubeCloud. The three services form a chain in which Service 0 requests resources from Service 1 through HTTP, and Service 1 requests resources from Service 2 also through HTTP. A delay of 15 seconds is introduced in Service 2 to simulate an unresponsive and strained service. The services are implemented in Spring Boot using Netflix's circuit breaker (Hystrix). Further details and implementation examples can be found in Appendix E. This experiment will investigate three scenarios with a varying amount of circuit breakers. The scenarios are the following:

1: No circuit breakers
In the first scenario, all requests between services are synchronous calls without timeouts or circuit breakers (Figure 8.1). The expected result is requests piling up between Service 1 and Service 2 because of slow responses leading to blocked threads. The blocked threads should lead to a cascading failure making Service 1 unavailable and, at some point, Service 0 unavailable.

Figure 8.1: Services without Circuit Breakers

2: One circuit breaker
In the second scenario, a timeout at 1 second between Service 1 and Service 2 is added. Furthermore, a circuit breaker is placed between Service 1 and Service 2 (Figure 8.2). The expected result is an improved overall response time since Service 1's timeout decreases time spent in blocked threads. Furthermore, the circuit breaker shuts off Service 2 at some point because of its slow responses.

Figure 8.2: Services with One Circuit Breaker

3: Two circuit breakers
In the third scenario, a timeout at 1 second between Service 0 and Service 1 together with a timeout at 1 second between Service 1 and Service 2 is added. Furthermore, a circuit breaker is placed between Service 0 and Service 1 together with a circuit breaker between Service 1 and Service 2 (Figure 8.3). The expected result is lower latencies since the timeout between Service 0 and Service 1 is reduced to 1 second.

Figure 8.3: Services with Two Circuit Breakers

The expected result is that more circuit breakers lead to lower latencies and a higher success rate for requests. The results are expected to be degraded responses in the form of more fallback answers from Service 0 and Service 1.

Without circuit breakers, the requests are expected to block all threads because of Service 2's delay leading to blocked threads in the rest of the services while more requests are piling up.

When applying a circuit breaker between Service 1 and Service 2, the latencies are expected to be lower than the scenario without circuit breakers because Service 1 fails fast - the timeout is set to 1 second.

With two circuit breakers, the lowest latency is expected since the test runner's requests are directed towards Service 0 that has timeout is set to 1 second. If the circuit breaker can keep up with the request rate, the latency should not exceed much more than 1 second. Another interesting aspect is the amount of fallback responses from Service 0 and Service 1.

Test Protocol
For each iteration in each scenario, the same state must be created. In order to do so, a single instance of each service is started. In Kubernetes, this is achieved with a deployment with a single pod and an exposed service in front. When all three services are ready, two requests to Service 0 are made with 10 seconds delay. This ensures all services are ready, and that the circuit breakers, if present, are in the *closed* state.
After 15 seconds, which is the duration of the delay in Service 2, the test runner can be started. The test runner generates 100 requests/second for 30 seconds which leads to 3000 measurements.

When the test runner has finished, the Kubernetes pods are deleted and fresh instances created. The test protocol steps can then be repeated. Tests are executed from a MacBook Pro running an HTTP load testing tool called Vegeta which makes requests to Service 0 through a cabled network.

Metrics
The main metrics from the experiments are *latency*, *success rate* and *level of degradation*.

The latency is a value between 0s and 30s since the test runner's timeout is set to 30s. The success rate is based on the HTTP status codes returned. 200 denotes a success, and 206 denotes a success from a circuit breaker. The level of degradation is measured as the distribution of the responses' origins. Response from Service 2 is the desired and best response. Subsequently, Service 1 is preferred over Service 0.

Data Basis
Three iterations of each scenario are run to reduce the impact of random factors. In each iteration, 3000 measurements will be made which sums up to 9000 measurements per scenario.

Results of Experiment: Integration Points in a Synchronous Architecture

The results of the experiment show that circuit breakers lead to lower latencies, fewer timeouts and a different distribution of the responses' origins.

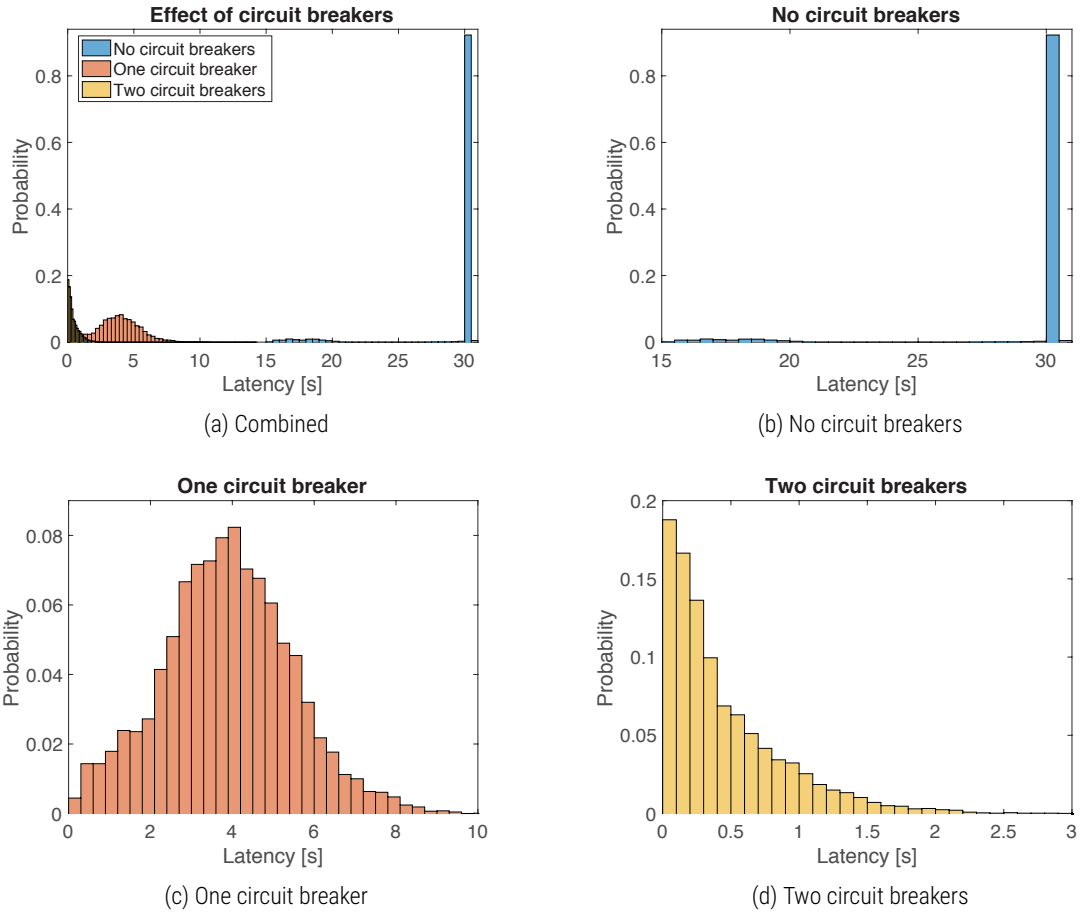
Latency
The histograms (Figure 8.4) show normalized distributions of the response times for each scenario and a combination (a).

Without circuit breakers (b), more than 9 of 10 latency measurements are measured as a value around 30 seconds, which is the test runner's timeout. The small section between 15 and 20 seconds are successful responses. These responses are in all three iterations around the first 200 responses after which the occurrences drastically decrease. The high amount of timeouts is most likely caused by blocked threads

in Service 2 leading to a cascading failure.

Adding a circuit breaker in Service 1 (c) leads to lower response times, though, still several seconds. This is expected because of the reduced time spent in blocked threads in Service 1. The reduced time is caused by the timeout and a state change in the circuit breaker from *closed* to *open* state. No timeouts in the test runner were triggered after the addition of a circuit breaker.

The last scenario with two circuit breakers (d) led to drastically reduced response times. As expected, these values were mostly lower than the timeout of Service 0 which is 1 second. It makes sense that moving the circuit breaker closer to the caller reduces the latency since the undesired external factors are shielded by the circuit breaker.



(a) Combined

(b) No circuit breakers

(c) One circuit breaker

(d) Two circuit breakers

Figure 8.4: Effects of Circuit Breakers

Response Types

The second part of the experiment is concerned with the degradation of the received content. This is an important factor in the experiment since the origin and success rate of the responses determine their usefulness. The distribution of the responses is sorted from best to worst (Table 8.1), and it is seen that responses from Service 2 only were obtained without using circuit breakers. This makes sense since circuit breakers, in the other scenarios, have a timeout at 1 second and Service 2 sleeps for 15 seconds for each request. The 7.2778% succeeding answers in the first scenario came at the expense of 92,7222% timeouts.

Applying a circuit breaker in Service 1 led to almost nothing but responses from Service 1's fallback method, but a few server errors with HTTP status code 500 were also received. The fallback method is invoked when the circuit breaker's state is open. The 0,0111% server errors correspond to 1 of 9000 responses.

Applying two circuit breakers, the distribution of response origins is mainly split between Service 0 and Service 1's fallback methods.

	No circuit breaker	One circuit breaker	Two circuit breakers
Service 2	7.2778%	0%	0%
Service 1	0%	99.9889%	36.8111%
Service 0	0%	0%	62.7333%
Server error	0%	0.0111%	0.4556%
Timeout	92.7222%	0%	0%

Table 8.1: Distribution of Response Types

In general, the scenario without circuit breakers failed 92.72% of the time but was the only scenario getting a response from Service 2. The scenario with a single circuit breaker reduced the response time significantly, and the degradation of the response was better than the last scenario. The scenario with two circuit breakers was even faster and shifted the degradation of the responses to Service 0's fallback method in most cases.

Discussion of Experiment: Integration Points in a Synchronous Architecture

Consistency in the responses is traded for availability, as seen from the results (Table 8.1). The scenario without circuit breakers was the only experiment obtaining answers from the desired service (Service 2), but 92.7% of the responses timed out, which leads to an availability percent of 7.3%. Adding a circuit breaker in Service 1 shifts the availability to 99.99% with a timeout at 30 seconds. Consistency in the response is degraded since all the responses come from Service 1 instead of Service 2. Adding another circuit breaker in Service 0 results in 99.5% availability. The degradation of the responses contained 36.8% responses from Service 1 and 62.7% responses from Service 0.

The shift in latency (Figure 8.4 (a)) clearly shows an improvement in latency by using a circuit breaker. A way of obtaining resilience is, thereby, by protecting against slow responses from integration points.

An important aspect of the experiment is to design the test protocol to start from the same state for each iteration. The services had a startup delay that caused the first request to time out. Therefore, it was necessary to ensure that the services were ready before each test. Two requests to Service 0 were made before starting the test to make sure that all three services had started. Furthermore, executing two tests without resetting the state led to different results. The reason for this were probably Service 2's blocked threads and state in the circuit breakers. Therefore, the services were removed and created between each testrun.

8.2 Infrastructure Level Experiments

The previous section focused on the effects of application level patterns to achieve a higher degree of resilience. This section will instead look into designing experiments to verify the different initiatives possible at the infrastructure level. Fault tolerance and the ability to detect failures are among the most notable features to ensure a higher degree of resilience at the infrastructure level, as described in Chapter 6.

Design of Experiment: Effects of Replication

Fault tolerance is a key feature in Kubernetes. The goal of this experiment is to investigate the effects of replication to determine the how number of replicas affects the latency under varying traffic loads.

The test setup consists of a simple application written in Java with the Spring Boot framework. The application is packaged into a Docker container and deployed into Kubernetes. The application is returning a string when it receives a HTTP request at '/'. The experiment performs ramp up of requests, which refers to continuing to apply an increasing amount of load, to stress the system. 10.000 requests are performed at each step with a step size at 100 requests/second until the system experience a drastic increase in latency. Calculating the duration of each step is done using the following formula:

$$duration_{step} = \frac{10.000 + (rate_{current} - 1)}{rate_{current}}$$

This experiment is performed with three scenarios: replicas=1, replicas=5, and replicas=10. Each scenario runs three times to eliminate randomness and the worst outliers. The average response time of the three runs of each step is applied. All load is generated by a MacBook Pro through a cabled network directly at the service endpoint '/'.

The expected result is that replicating the application will move the boundary for failure.

Test Protocol

Every test shall be run with a 'clean' environment. The test protocol of this experiment is as follows:

1. Run the application in Kubernetes (start the pod)
2. Expose the application in Kubernetes (add a service)
3. Scale the deployment to the desired number of replicas
4. Start the test script [1]

Steps 1-4 shall be run in total of three times.

Metrics

The main metric in this experiment is *latency* and *the number of replicas*. Especially the point at which the latency will have a drastic increase is the point of interest.

Data Basis

Three iterations of each scenario are run to reduce the impact of random factors. The result of each step for each scenario is an average of 30,000 measurements.

[1] Attachment 6

Results of Experiment: Effects of Replication

The results of running the described test protocol verify the boundary for failure is moved when increasing the number of replicas. Figure 8.5 shows the result of the experiment. Each scenario is depicted as a fitted line to each step's average of the latencies for each iteration.

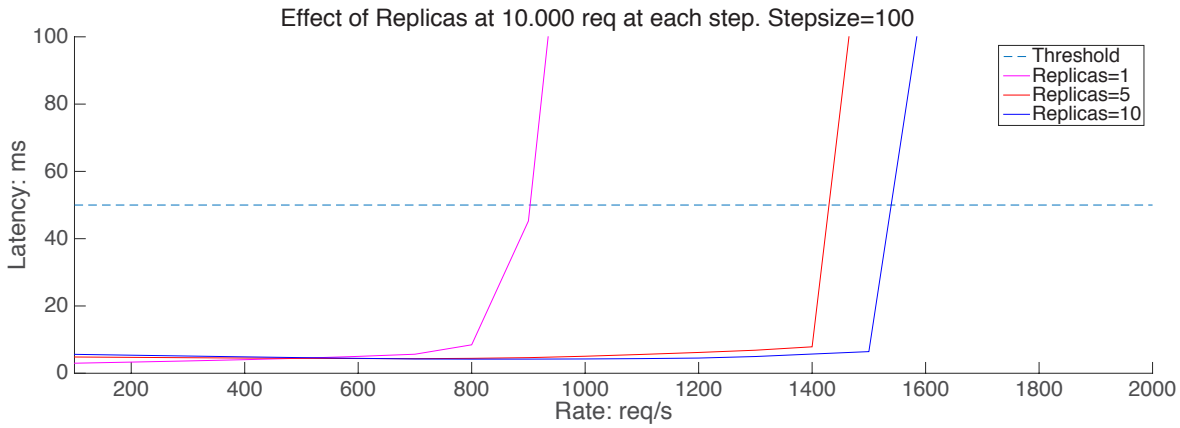


Figure 8.5: Effects of Replicas at 10.000 Req. each Step

Figure 8.5 shows the effect of using an infrastructure with replication. The threshold applied to Figure 8.5 is a reference point, determining an unsatisfiable latency. At some point, a drastic increase begins, and the server becomes unresponsive shortly hereafter. The iteration of the scenario is stopped when timeouts pile up. The results can be summarized to:

	Breaking point
Replicas=1	∼900 request/second
Replicas=5	∼1400 request/second
Replicas=10	∼1500 request/second

Table 8.2: Effects of Replicas Summarized

Increasing the number of replicas from 1 to 5 shifts the amount of concurrent request that the system can handle from 900 request/second to 1400 request/second, which is an increase of 55.6% in the amount of load that the application can handle. The improvement of using 10 replicas instead of 5 replicas is significantly lower, and it only provides an increase in requests/second of 7% compared to 5 replicas.

Discussion of Experiment: Effects of Replication

From the experiment, it is seen that the number of replicas has a correlation with the throughput. The robustness of a system is concerned with how well it withstands stress, and from the results, it is seen that adding more replicas pushes the limit of requests/second it can handle (Figure 8.5).

The improvement in performance from 1 to 5 replicas is significant, but from 5 to 10 replicas the improvement is less significant. A reason for this might be that the cluster consists of four Raspberry Pis, and the resources are more or less utilized.

During the tests, the system was pushed beyond its boundaries which resulted in peculiar behavoir. Each iteration of each scenario required a shutdown of Kubernetes and a shutdown of the Raspberry Pis. This resulted in cumbersome waiting times.

Design of Experiment: Recovery Time

Another key feature of Kubernetes as a platform is the ability to detect failures, e.g. a container failing or a node being removed from the cluster. The goal of this experiment is to look into this ability and try to determine the recovery time of an application, by only deploying one instance of the application. The second part of this experiment is to verify that by increasing the number of replicas the infrastructure will be able to handle a node failure with a minimum of lost requests.

The test setup consists of the same application used in the previous experiment. Since the goal of this experiment is to determine the effects of losing a node, the network cable of the node running the application will be pulled. The experiment will run with a total duration of 3 minutes (180 seconds) for each iteration, and the network cable is removed after 30 seconds. The experiment is run with replicas=1, replicas=2, and replicas=5 and with rate=100 request/second, and rate=200 request/second.

KubeCloud is configured with a timeout for unresponding nodes of 10 seconds (–node-grace-period) and a further timeout for the pods running on a unresponding node of 5 seconds (–pod-eviction-timeout). The expected result with one replica is that the application will be unresponsive for about 5 seconds (–pod-eviction-timeout) + 10 seconds (–node-grace-period) + 25 seconds (Spring Boot startup time on the Raspberry Pi) = 40 seconds. Whereas the expected result for running more than one replica will result in no period of unresponsiveness.

Test Protocol
Every iteration of a scenario has to be run with a 'clean' environment. The test protocol is as follows:

1. Run the application in Kubernetes (start the pod)
2. Expose the application in Kubernetes (add a service)
3. Scale the deployment to the desired number of replicas
4. Determine where the application (pod) has been started
5. Start a stopwatch and test script at the same time
6. After 30 seconds, remove the network cable of the node where the pod is running

Steps 1-6 shall be run three times. If the application is scheduled on the master node, it is necessary to delete the pod and start it again to be rescheduled on a worker node. We only pull the plug of nodes with one application (pod) running. All load is generated by a MacBook Pro through a cabled network directly at the service endpoint '/'.

Metrics
The two main metrics in this experiment are *the recovery time in seconds* and the *success rate in percent*.

Data Basis
Three iteration of each scenario is run to reduce the impact of random factors.

Results of Experiment: Recovery Time

The effect of applying replicas is clearly visible in regards to the recovery time and the ability of Kubernetes to detect a failure. Table 8.3 summarizes the results of replication for the six scenarios run: replicas=1, replicas=2, and replicas=5 for rate=100 and rate=200. The results are averages of the three iterations of each scenario.

Success rate [% (success/error)]	**rate=100**	**rate=200**
replicas=1	74.65% (13,437/4,563)	72.75% (26,189/9,811)
replicas=2	99.87% (17,977/23)	99.88% (35,957/43)
replicas=5	99.98% (17,996/4)	99,98% (35,994/6)

Table 8.3: Comparison of Success Rates with Rate=100 and Rate=200

The results show that the rate does not play a significant role. Furthermore, an improvement of approximately 25% percentage points in the success rate between one replica and two replicas. The effect of running a single instance of the application is also clearly visible. Figure 8.6 shows a plot of one of the scenarios with rate=200 and one replica. The human factor involved in pulling the network cable resulted in slightly different timestamps for the unresponsiveness, therefore, a randomly selected scenario is shown to illustrate the differences. The plot in Figure 8.6 shows a fitted line indicating successful requests (1) and failing requests (0) over time.

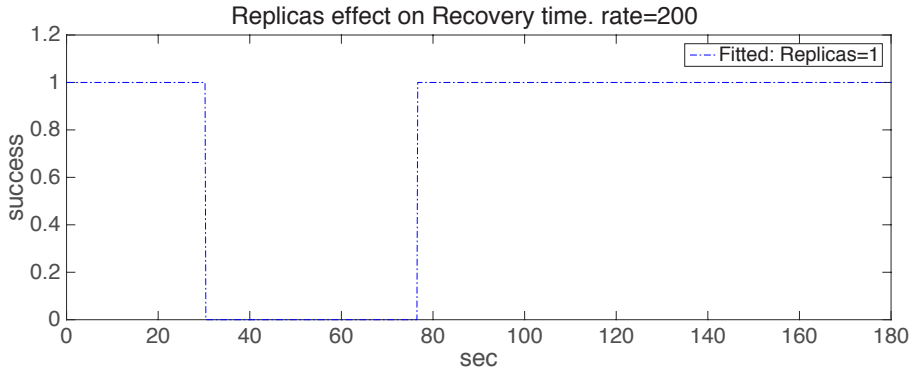


Figure 8.6: Replicas Effect on Recovery Time: Replicas=1, Rate=200

It is clearly visible that the application is unresponsive, and that the client does not receive any responses for a total period of approximately 47 seconds. The 47 seconds is close to the expected recovery time of 40 seconds when running one replica.

Figure 8.7 shows the effects of increasing the number of replications to a total of two. The effect of replication is, as stated in Table 8.3, clearly significant compared to only running a single instance as depicted in Figure 8.7.

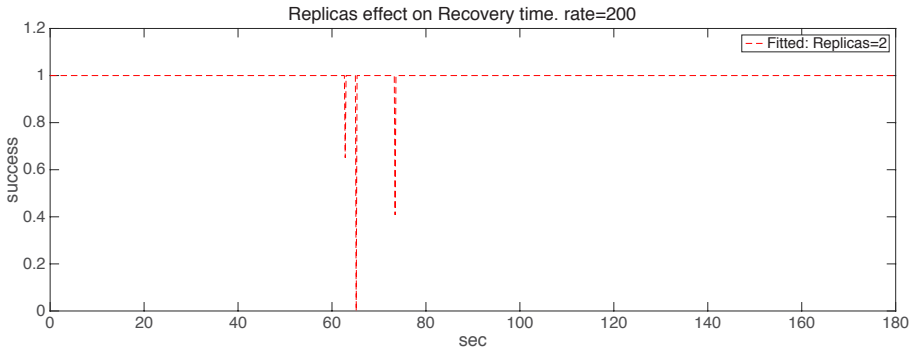


Figure 8.7: Replicas Effect on Recovery Time: Replicas=2, Rate=200

When running two instances the number of errors are on average reduced to 43 from 9,811 at rate=200.

Lastly, Figure 8.8 shows the effect of five replicas. The fitting of the line removes the on average six unsuccessful responses and the uptime is close to 100%.

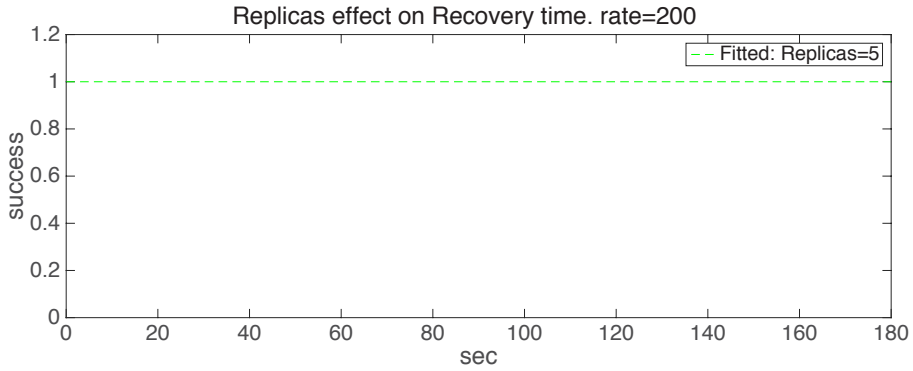


Figure 8.8: Replicas Effect on Recovery Time: Replicas=5, Rate=200

Discussion of Experiment: Recovery Time

The main conclusion is the importance of having more than one running replica at a time. The results show (Table 8.3) that an improvement from one to two replicas is very large, from around 73% to around 99.9%, while the further improvement is very small, but though present. It makes sense that an extra running instance can provide fault tolerance.

Another desired attribute, in this case, is the rapidity of the infrastructure to identify and mitigate further errors. From the experiments, it is seen that Kubernetes quickly stopped traffic to the disconnected node. The configuration of Kubernetes' timeouts (–pod-eviction-timeout and –node-grace-period) especially played a role in the first scenario in which no other node could respond instead.

9

Experiment: Evaluation of KubeCloud and Course Design

An evaluation is needed to determine whether KubeCloud and the designed course is a viable cloud computing teaching strategy for engineering students.

This chapter will cover several experiments made to determine the effects of KubeCloud and the designed course. The first experiment clarifies the students' learning style preferences. The subsequent two experiments are centered around formative aspects of the weekly evaluation of KubeCloud and the designed course. Lastly, summative investigations are made to evaluate the overall aspects of KubeCloud and the designed course.

Evaluation of KubeCloud and the designed course have been performed in an existing course, Object-Oriented Network Communication (ITONK), at Aarhus University School of Engineering. The course curriculum has previously covered technologies such as Java RMI, DNS, DDS, CORBA, etc., and the course needed a renewal of the topics.

The class participating in these experiments primarily consists of students in the last year of their bachelor's degree in Information- and communication technology (ICT). Furthermore, students at Aarhus University Computer Science and Department of Engineering are allowed to enroll. The class consists of a heterogeneous group of around 60% ICT students while the remaining 40% is a mix of Computer Science, M.Eng. Computer Engineering, and Health Technology students.

9.1 Determining the Students' Learning Style Preferences

Design of Experiment

In order to target the learning experience towards the class, we need to understand how the students learn. Before starting the learning activity, we need to determine how the students themselves rate their learning style. As described in Section 2.3, Felder and Silverman describe four categories of learning styles. Felder further created a questionnaire to determine individual students' preferred styles of leaning within the four categories. This questionnaire[1] contains 44 questions in which the student must choose between

[1] `http://www.engr.ncsu.edu/learningstyles/ilsweb.html`

two possible answers to each question.

To determine the students' learning style preferences, the students are asked to hand in the results of the questionnaire before the course starts. The expected results of this experiment are to confirm the results of Felder and Silverman's research stating that engineering students are active, sensing, visual and sequential learners.

Test Protocol
A week before the course starts, the questionnaire is sent out. The students hand in an image of the summarized results of the questionnaire.

Metrics
The questionnaire returns a result with a rating of the individual student's preferences in the four categories. The four categories are rated on a scale from 1 to 11 on the side of the student's preference as seen in Figure 9.1.

Data Basis
The data basis consists of 16 students' responses. Two students did not hand in their assignment.

Results of Experiment

In order to determine the result, a negative weight is assigned on the right side of the axes of the four categories. The resulting average is displayed in Figure 9.1.

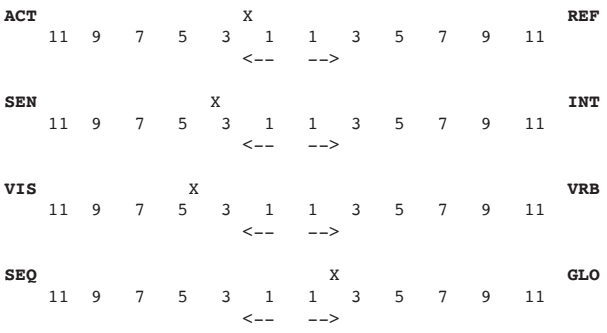


Figure 9.1: Aggregated Learning Style Preferences of the Class

Discussion of Experiment

The results show that the students in the class prefer **active**, **sensing**, **visual**, and **global** learning styles. Three out of four categories match the statements made by Felder and Silverman. The difference is in the last category, where our experiment shows that the students have a small preference for a global learnering style. The visual/verbal preference contains the most dominant preference.

9.2 Weekly Evaluation of the Learning Activities

Design of Experiment

After determining the students' learning style preferences, we need to measure the students' progression. An evaluation of the results of the weekly reflection assignments according to SOLO taxonomy is made in the context of the entire course. Part of the course design is the weekly learning goals, and to evaluate the students progression, a weekly reflection assignment is handed in. The goal of this reflection assignment is for the students to reflect upon the weeks' learning activities and, in the first week, determine their initial knowledge of relevant topics. Furthermore, these assignments will provide insight into the students' progression. The assignment contains between three and seven questions. The questions are all asked as open-ended questions, and the students are provided with a text field to input their thoughts.

Test Protocol
The questionnaires are released to the students in the last hour of the last lecture of every module. Below is the determined test protocol for each module. The lecture content of Module 6 is prepared by Christoffer Werge, another master's thesis student, and is, therefore, not a part of this master's thesis.

Reflection before the course

1. What is cloud computing?
2. What is a distributed system?
3. What is cluster computing?
4. What is virtualization?
5. Describe some benefits and liabilities of sequential vs. parallel execution
6. Describe your understanding of possible error in communicating over a network
7. Describe your understanding of resilience in a system

Module 1 - Course introduction, Cloud computing, frameworks

1. What challenges do you see in cloud computing and microservice architecture?
2. What would you say a cluster consist of?
3. What benefits and drawbacks do you see in a microservice architecture compared to a monolithic architecture?

Module 2 - Docker lightweight containers

1. Describe with your own words the benefits and drawbacks of virtualization in general
2. Describe with you own words, the difference between Containers and Virtual Machines on Hypervisors
3. Describe with your own words, benefits and drawbacks of Docker containers
4. Describe with your own words how Docker fits into a microservice architecture

Module 3 - Kubernetes cluster manangement

1. Describe the connection between Docker and Kubernetes

2. Describe with your own words how Kubernetes orchestrates containers
3. Describe with your own words the benefits and drawbacks of using a cluster management system such as Kubernetes
4. How did workshop help you understand this week's topic

Module 4 - Resilience and load testing

1. Describe in your own words what resilience is and provide a couple of examples
2. How did circuit breakers and replications affect the resilience of the services in the workshop?
3. How did the workshop help you understand this week's topic?
4. For what purposes would you use unit testing, load testing, and stress testing respectively?

Module 5 - Project work and external presentation

1. Describe the process of deploying applications into Kubernetes, and how (if) the ideas of automation pipelines can help?
2. Describe with your own words the relationship between Microservices, Docker, Kubernetes, and Continuous Delivery
3. Compare the course content to the real-world perspective described by the external presenter. How does it relate?

Module 7 - Service Discovery and project presentations

1. Describe the role of service discovery in relation to Docker, Kubernetes, and microservices
2. Describe the benefits and drawback of having an IP per pod (Kubernetes)
3. Compare how communication between two pods is performed with or without Kubernetes DNS

Metrics
Since the questions are open-ended, the metrics will be a formative evaluation. The goal is to get an insight in the students' mental models and categorize them using SOLO taxonomy. The metric is a classification of the answers in relation to the entire course content.

Data Basis
The data basis is varying from week to week (Table 9.1).

Results of Experiment

The overall evaluation of the students' progression in the context of the overall course content is displayed in Table 9.1. As mentioned, Module 6 is out of the scope of this master's thesis. Furthermore, the hand-in process of Module 7 reflections is due on the date of handing in this master's thesis, and is, therefore, not included in this report.

Module	Responses	Stage
0: Before the course	17	Prestructural
1: Course introduction, Cloud computing, frameworks	13	Unistructural
2: Docker lightweight containers	11	Unistructural
3: Kubernetes cluster manangement	10	Multistructural
4: Resilience and load testing	9	Multistructural
5: Project work and external presentation	10	Relational

Table 9.1: Weekly Evaluation of Questionnaires (SOLO)

Evaluation of Module 0: Before the course
As seen from Table 9.1 we evaluate the students' stage before the course to be in the *prestructural stage*. The basis for this result stems from examining and evaluating the answers of 17 students to a total of seven asked questions. A common theme, when asked about what cloud computing is, was to some degree wrong answers and misconceptions of what cloud computing is, e.g. *"As far as i know, cloud computing is a system (and/or) application hosted in the cloud"*. The prestructural stage is the stage of ignorance, and this was also clearly stated by some of the students, e.g. *"I don't know"*. The overall evaluation of the questionnaire and the students' stage before the course is, therefore, *prestructural*.

Evaluation of Module 1: Course introduction, Cloud computing, frameworks
The questionnaire of Module 1 contained three questions with a total of 13 students completing the assignment. The questions asked focused on the main topics of the week, namely cloud computing and specifically microservices architecture. The evaluation of the answers supplied by the students show a transition from the *prestructural* stage to the *unistructural* stage. A common theme is small simple observations like *"Connections between devices. Being consistent with interfaces"* when asked about the challenges in cloud computing and microservice architecture. However, two answers show a higher degree of understanding and use of vocabulary to describe the challenges, and are closer to the *multistructural stage*. The unified evaluation shows a transition from the *prestructural* stage to the *unistructural* stage.

Evaluation of Module 2: Docker lightweight containers
In Module 2, the students have not progressed in terms of the SOLO taxonomy stages. The students show an individual understanding of the topic of the week, however, the bigger picture is still missing. The students demonstrate their understanding of Docker containers in conjunction with microservices. In the context of the entire course, the relations are not demonstrated. It is a natural stage to be in at this point in the course. One student actually demonstrate the ability to look forward, and makes the connection to the following week, *"Docker helps us deploy the microservices in an easy manner, especially when Kubernetes starts getting involved, which will hopefully grant a better overview of running services etc."*. Despite this observation the majority of the answers still shows small misconceptions and *unistructural* responses such as *"Its smart because its not hardware dependent"*. The overall evaluation of Module 2 results in the *unistructural* stage.

Evaluation of Module 3: Kubernetes cluster manangement
The evaluation of Module 3 results in a transition from the *unistructural* stage to the *multistructural* stage.

This transition is demonstrated with responses such as *"Docker is used to pack software into containers which afterwards can be pushed to the docker hub. Kubernetes responsibility is to run the containers and distribute them among one or multiple machines in a cluster."* The students in general demonstrate the ability to understand the topics in isolation. Simple connections are made between different introduced topics of the course. However, some are still providing *unistructural* answers such as *"Kubernetes manages the docker containers"*. The overall evaluation results in a transition from the *unistructural* stage to the *multistructural stage*.

Evaluation of Module 4: Resilience and load testing
No progression is made from Module 3 to Module 4 in terms of SOLO. The number of students handing in this assignment were only a total of 9. The answers are fairly short, but a few students demonstrate the *multistructural* understanding with answers such as *"You would unit test single services to check if the service is working as intended. Load testing to see if the system works as indended under normal or slightly above normal load. Stress testing to see how the system reacts under massive load and how it recovers from the problems that arise."* The overall evaluation of Module 2 results in the *multistructural* stage.

Evaluation of Module 5: Project work and external presentation
The evaluation of Module 5 results in a transition from the *multistructural* stage to the *relational* stage. This transition is demonstrated by multiple answers describing the relationship between many of the topics in the course, e.g. *"Microservices is the idea of having Small independant applications instead of Big monoliths. They are usually used together with docker. Docker containerize your app making them self contained in the Sense that they have their own enviroment Inside the containers. Kubernetes is a container management system that helps with stuff like resilience by replicating and restoring downed applications. Continous delivery in this context is about being able to upgrade these small containers individually instead of a monolith where you have to do the big bang deployment"*. In general, the students have submitted longer and more detailed descriptions and the overall evaluation of the week is a transition from the *multistructural* stage to the *relational* stage.

Discussion of Experiment

As seen from the results in Table 9.1 the students' progression is increasing in terms of the stages in SOLO taxonomy. The students started out before the course, as expected, knowing little about the concepts, and in Module 5 demonstrates the ability to analyze and compare different topics.

The number of students submitting answers has led to a smaller respondet pool than expected. For the majority of the assignments slightly over half of the 18 students are handing in the reflection assignments. The validity of the experiment can be questioned, because of the small number of respondents. However, the assignments have been an important tool in getting insights into the students' progression and knowledge from module to module. If it is clear from the assignments that a topic has been misunderstood, we have the possibility of correcting the mistake in the following module.

Another topic of discussion is the evaluation of answers provided by the students. Many answers are fairly short, and can be categorized as prestructural, or at best unistructural. A reason for these types of answers is properly related to the nature of the assignment. Even though the students are told the assignments are mandatory, they consider them optional. This may be because of the formative and qualitative nature of the assignments, and that it is purely based on their thoughts and not actual theory. The students are asked not to use any tools, and only answer to the best of their knowledge. Furthermore, they are told that they are not judged by their answers.

9.3 Weekly Evaluation of KubeCloud

Design of Experiment

To determine KubeCloud's usefulness as a learning object, an experiment covering the students' experience with it and perception of it is designed.

The questionnaires from the previous and this experiment are combined in the weekly reflection exercises. Questions about KubeCloud's role are asked to disclose the students' mental models during the course. The answers will be used to clarify and classify KubeCloud's role as a learning object according to existing theories for learning objects.

Test Protocol

The questions for the questionnaires are presented below. The questionnaires are sent out in the last hour of Module 2, Module 3, and Module 4. In Module 1 a question is asked immediately after a demonstration of parallel workloads distributed on a Raspberry Pi cluster. The Raspberry Pi cluster in Module 1 is not KubeCloud, and the other cluster only appears to visualize the benefit and drawbacks of distributing workloads across multiple Raspberry Pis. The answers are categorized and compared to the theories presented in Chapter 2.

Module 1

1. Did the cluster give you a better understanding of cluster computing?

Module 2 reflection

1. Describe how (if) the Raspberry Pi Cluster helped your learning

Module 3 reflection

1. Describe how (if) the Raspberry Pi Cluster helped your learning
2. Describe how the Kubernetes visualization tool helped your learning
3. How did the cluster help you understand the scheduling done by Kubernetes?
4. How did the workshop help you understand this week's topic?

Module 4 reflection

1. Describe how (if) the Raspberry Pi Cluster helped your learning
2. Did the cluster support your understanding of what resilience is? If yes, how?

Metrics

Since the questions are open-ended questions, the metrics will cover a qualitative, formative evaluation. Statements from the students are expected to confirm the value of the cluster as a learning object and help classify the type of learning object according to Churchill's classifications. The cluster's role as a mediating tool as described in activity theory is also expected to be seen from the answers. Lastly, the importance of an object to explore and experiment with, as described in constructivist learning theory, is expected to be observed in the students' answers.

Data Basis

The data basis is varying from week to week (Table 9.1).

Results of Experiment

The responses from each module show that the students' responses contain different characteristics of learning objects. The usefulness of the cluster in regards to learning is indicated in the responses, but generalized conclusions cannot be drawn from the answers. This section will present the main points from the questions. The raw data responses and an analysis can be found in Attachment 7.

Evaluation of Module 1: Course introduction, Cloud computing, frameworks
After demonstrating distributed workloads on a cluster (not KubeCloud), 15 of 17 students stated that it provided them with a better understanding of cluster computing.

Evaluation of Module 2: Docker lightweight containers
The students were asked whether KubeCloud helped their learning. Six of eight students expressed, either directly or indirectly, that the cluster was a help e.g. because it visualized or illustrated something for them. Two students expressed that it did not help them. They point out that the workshop could have been done without the cluster, and that the cluster will be more useful when Kubernetes is introduced. The largest other category found in the answers was a description of the cluster as a 'scale model' or similar description. The cluster is e.g. referred to as a "remote host" and "a small-scale model". Furthermore, enthusiasm was present in two of the responses since running containers on the cluster is called 'cool' and because 'motivation' is mentioned.

Evaluation of Module 3: Kubernetes cluster manangement
All seven students reported that KubeCloud helped their learning in this module. The help from the visualization was mentioned in all the responses. How it helped to understand the location of the pods was described in different ways. A student sums up: *"The cluster illustrates the purpose and setup visually in a nice way. It is very hands on to work with these compared to some cloud setup where the actual servers in the cluster are hidden away"*. The students did, however, not agree on whether the cluster helped them understand the scheduling done in Kubernetes. Four students either did not see it as a help or did not understand the scheduling happening in the cluster. Three students described how it helped them know which Raspberry Pi, pods were being scheduled on.
When asked about the workshop, three of the students mentioned that they liked the *"hands on"* or practical application of the theory. Statements such as *"hands on is the best learning tool"* and *"Hands on is all ways good."* supports this. The workshop format was both complimented and criticized. Sharing a cluster was pointed out as a drawback by a student. A student pointed out that the one at the pc *"got most out of the workshop"*.

Evaluation of Module 4: Resilience and load testing
Seven of eight students stated that KubeCloud helped their learning in this module. The last student referred to another question. Six of the students mentioned that they liked the physical aspect of the exercise or the possibility of pulling the network cable. A student describes: *"[...] This helped understading how a part of a system could be down, and other parts can take over the service"*. When asked if KubeCloud supported their understanding of resilience, seven of nine students said it did. Six students mentioned that their understanding of the mechanisms and concepts of Kubernetes were supported by the cluster. Lastly, four students used the word recover and two more described the concept.

Discussion of Experiment

In Module 2, similarities are seen between KubeCloud and Churchill's *practice object* which is a *"representation that allows practice and learning of certain procedures"*. The criticism of the exercises being possible without the cluster is fair. The inconvenience of Docker on ARM is unfortunate. However, Docker is necessary to use Kubernetes in Module 3. Motivation is seen from the results which is in agreement with Hein's emphasis on motivation and Piaget's emphasis on play.

In Module 3, the importance of the visualization is seen. This is in agreement with Felder and Silverman's research on engineering students being visual learners. The sensory preference is also seen in some of the students, who like the hands-on aspect. Furthermore, that the cluster is shared in groups during workshops is criticized. This is an interesting point of view compared to mediating tools from activity theory. The students are, according to one student, not able to attain the same knowledge during the workshops since the cluster (mediating tool) is not equally available to all members of the group. We agree that the workshop format leads to one person typing in the commands, however, this limitation is only due to the format of the workshop. KubeCloud allows students to connect and deploy services independently.

In Module 4, many of the students had positive feedback and highlighted the physical aspect and the ability to pull the network cable and see the reaction. Again, the sensory and visual preference is reflected.

9.4 Overall Evaluation of the Learning Activities

Design of Experiment

To combine the previous two experiments and to evaluate the students' overall outcome of the designed learning activity, a last experiment is conducted. The goal of this experiment is to quantify the students' overall thoughts and view on the learning experience they have been exposed to.

Test Protocol
The questions for the overall evaluation of the tangible cloud computing cluster and the course in general are joined. The questionnaire is sent out in the beginning of the last module of the course and the students are given 20 minutes to complete the questionnaire. The questionnaire is divided into two parts; one focusing on the effect of using a cluster, and one focusing on the course.
Statements relating to the course design:

1. **Course structure**
   a) The distribution of theory and practical work was suitable
   b) The order of the topics was well-planned
   c) The problem based project allowed me to apply the concepts and theories
   d) The workshops provided me with the necessary skills for the project work

2. **Materials**
   a) The reading and video materials provided me with the essentials for each module
   b) The slides covered the necessary theory
   c) The blog (rpi-cloud.com) helped me getting started with Spring Boot and Spring Cloud

3. **Comparision**
   a) In comparison with other courses my overall rating of the course is

Metrics
All questions are asked as statements that the student can rate their compliance in terms of a five-level Likert scale. (1: Strongly disagree; 2: Disagree; 3: Neutral; 4: Agree; 5: Strongly Agree). All questions are related to a top-level classification in order to provide a summarized result. The categories of interest are a rating of the *Course structure*, a rating of the supplied *Materials*, and a rating of the designed learning activity in *Comparison* with other courses.

Data Basis
The data basis consists of 16 students as described in the introduction of this chapter.

Results of Experiment

The overall evaluation of the designed learning activity is displayed in the tables below. The results of each of the three metrics is presented below. Further details can be found in Appendix G. Note: Statement (Comparison) is asked as a rating from very bad to very good.

Statements (Course structure)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The distribution of theory and practical work was suitable	0	6.3	12.5	43.8	37.5
The order of the topics was well-planned	0	12.5	12.5	50	25
The problem based project allowed me to apply the concepts and theories	0	0	18.8	43.8	37.5
The workshops provided me with the necessary skills for the project work	0	18.8	25	31.3	25
Total	**0**	**9.4**	**17.2**	**42.2**	**31.3**

Table 9.2: Overall Evaluation of the Designed Learning Activity: Course Structure

Statements (Materials)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The reading and video materials provided me with the essentials for each module	0	6.3	25	31.3	37.5
The slides covered the necessary theory	0	0	18.8	50	31.3
The blog (rpi-cloud.com) helped me getting started with Spring Boot and Spring Cloud	0	12.5	18.8	37.5	31.3
Total	**0**	**6.3**	**20.9**	**39.6**	**33.4**

Table 9.3: Overall Evaluation of the Designed Learning Activity: Materials

Statement (Comparison)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
In comparison with other courses my overall rating of the course is	0	6.3	0	50	43.8
Total	**0**	**6.3**	**0**	**50**	**43.8**

Table 9.4: Overall Evaluation of the Designed Learning Activity: Comparison

Discussion of Experiment

The overall rating of the designed learning activity is that 75.8% of the students either agree or strongly agree with the statements. The workshops scored lowest within the category *Course structure* with 56.3% agreeing or strongly agreeing. Nearly one out of five disagreed with *The workshops provided me with the necessary skills for the project work*. A student expressed that the workshops were too simple or linear and further adds: *"I think the workshops could be a little more free to experiments to increase the understanding of the concepts."*. These results indicate that the workshops should have allowed for more experimentation as described in constructivist learning theory. However, the overall choice of PBL seems suitable in relation to the project work. A student points out one of the core theories in constructivist learning theory: *"The real understanding of the concepts arrived when we worked on the project."*. Another key finding pointed out by a student is the order of the last two subjects DNS and service discovery. A better structure would have been placing the external presentation in one of the last modules. The initial point behind this was to make room for group work and give the students the ability to leverage some of the concepts presented, such as automation with Jenkins.

Lastly, in comparison with other courses 93.8% of the students rate the designed course as either good or very good. This result suggests that there is an interest for these topics among the students. The students point out the relevance of the topics: *"Much of this could very well be mandatory on the IKT engineering studies as the distributed systems are growing and the standardized monolithic architectures are disappearing from the more modern and scalable applications."*. One student mentions that the course introduced many new concepts which resulted in a "huge" workload for some of the students.

9.5 Overall Evaluation of KubeCloud

Design of Experiment

The design of the experiment of the overall evaluation of KubeCloud is as mentioned evaluated in conjunction with the overall evaluation of the learning activities.

Test Protocol
The test protocol is the same as described in the *Overall evaluation of the learning activities*.
Statements relating to the tangible cluster

1. **Physical appearance of the cluster:**
   a) The cluster provided me a better understanding of what a cloud consists of
   b) The physicality of the cluster provided a better understanding of errors in distributed systems (e.g. pulling the network cable)

c) The physical design of the cluster gave me associations to a real server rack

2. **Group activity**
   a) The cluster allowed me to experiment
   b) The cluster initiated group discussions
3. **Visualization**
   a) The cluster combined with the visualizer helped me understand the concepts of the cluster
   b) The cluster combined with the visualizer helped me understand the how errors are handled
4. **Movitation**
   a) The use of a cluster increased my motivation
   b) The cluster motivated me to do out of curriculum experimentation
   c) The cluster was a fun and playful way of learning the concepts
5. **Relation to the real world**
   a) Using a cloud provider (e.g. Google Container Engine) instead of a hands-on tool would have improved my learning
   b) The cluster represents a small-scale datacenter
   c) The cluster improved my skills in relevant technologies

Metrics
All questions are asked as statements that the student can rate their compliance in terms of a five-level Likert scale. (1: Strongly disagree; 2: Disagree; 3: Neutral; 4: Agree; 5: Strongly Agree). All questions are related to a top-level classification in order to provide a summarized result. The categories of interest are a rating of the *Physical appearance of the cluster*, a rating of KubeCloud's role during *Group Activity*, a rating of the *Visualization* aspects provided by KubeCloud, a rating of KubeCloud's ability to *Motivate*, and a rating of KubeCloud's *Relation to the real world*.

Data Basis
The data basis consists of 16 students as described in the introduction of this chapter.

Results of Experiment

The overall evaluation of KubeCloud is displayed in the tables below. The results of each of the three metrics is presented below. Further details can be found in Appendix G.

Statements (Physical appearance)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The cluster provided me a better understanding of what a cloud consists of	0	0	0	56.3	43.8
The physicality of the cluster provided a better understanding of errors in distributed systems (e.g. pulling the network cable)	6.3	0	6.3	56.3	31.3
The physical design of the cluster gave me associations to a real server rack	0	6.3	12.5	37.5	43.8
Total	**2.1**	**2.1**	**6.3**	**50**	**39.6**

Table 9.5: Overall Evaluation of KubeCloud's Physical Appearance

Statements (Group activity)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The cluster allowed me to experiment	0	6.3	18.8	37.5	37.5
The cluster initiated group discussions	12.5	25	18.8	12.5	31.3
Total	**6.3**	**15.7**	**18.8**	**25**	**34.4**

Table 9.6: Overall Evaluation of KubeCloud in Relation to Group Activity

Statements (Visualization)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The cluster combined with the visualizer helped me understand the concepts of the cluster	0	0	6.3	43.8	50
The cluster combined with the visualizer helped me understand the how errors are handled	0	6.3	31.3	12.5	50
Average	**0**	**3.2**	**18.8**	**28.2**	**50**

Table 9.7: Overall Evaluation of KubeCloud in Relation to Visualization

Statements (Motivation)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
The use of a cluster increased my motivation	0	0	43.8	43.8	12.5
The cluster motivated me to do out of curriculum experimentation	6.3	31.3	37.5	25	0
The cluster was a fun and playful way of learning the concepts	0	6.3	18.8	56.3	18.8
Total	**2.1**	**12.5**	**33.4**	**41.7**	**10.4**

Table 9.8: Overall Evaluation of KubeCloud in Relation to Motivation

Statements (Relation to real world)	**1** [%]	**2** [%]	**3** [%]	**4** [%]	**5** [%]
Using a cloud provider (e.g. Google Container Engine) instead of a hands-on tool would have improved my learning	0	25	31.3	43.8	0
The cluster represents a small-scale datacenter	0	0	37.5	50	12.5
The cluster improved my skills in relevant technologies	0	0	25	62.5	12.5
Total	**0**	**8.3**	**31.3**	**52.1**	**8.3**

Table 9.9: Overall Evaluation of KubeCloud in Relation to the Real World

Note: The result of the first question (Table 9.9) has been reversed due to the negated form of the statement.

Discussion of Experiment

The overall rating of KubeCloud is that 67.8% of the students agree or strongly agree with the statements. A key finding, within the physical appearance category of questions asked, is that the cluster supports the students' understanding of what cloud computing consists of. Furthermore, a surprising finding is that one student rates KubeCloud's ability to provide an understanding of errors in distributed systems to strongly disagree. However, 87.6% of the students either agree or strongly agree with the statement that KubeCloud provides a better understanding of errors in distributed systems.

KubeCloud was placed at a table in the front row of the classroom, and a physical distance between the groups and the clusters was present. One student points out: *"The cluster was for a long time this "wierd thing" which we were mostly scared of breaking - which led to only doing things to it described in workshops."*. The length of the network cables for each cluster introduced this limitation. Optimally, each group should have their cluster right in front of them instead of this physical distance. The benefit of having the physically located on the table in front of them would have minimized the mental gap. The students did not agree on KubeCloud's ability to initiate group discussions. 37.5% disagreed or strongly disagreed whereas 43.8% agreed or strongly agreed. A reason for this disagreement could be the students' delegation of tasks internally in the groups, which led to a parallel development of the project work. This way of decomposing the project was also observed from our facilitating role.

The visualization category is among the highest rated in the evaluation of KubeCloud with 78.2% of the students either agreeing or strongly agreeing with the statements. The visualizer provided the students with a better understanding of the concepts of the KubeCloud and how errors are handled in Kubernetes. However, as one student points out: *"It's unfortunate that the visualizer was so bugged."*. The visualization tool, even with our improvements, is still not performant enough.

The motivation category received the lowest agreement with the statements. A reason for this relatively low score is that 37.6% students disagreed or strongly disagreed with the statement: *The cluster motivated me to do out of curriculum experimentation*. This is a bold question since the students are asked whether or not they are motivated to do out of curriculum experimentation. 25% of the students answer that they agree with the statement, and have been doing extra activities. A key finding in the motivational category confirms our previously described limitation of Docker and ARM architectures: *"The cluster made the workshops and project more real compared to using a cloud cluster. Too bad the ARM architecture broke the seamless experience of using docker."*. This is unfortunate but to illustrate and improve the students' skills Docker on ARM is still a valid solution since the API is the same.

The most significant finding in the relation to the real world category is the trade-off between using the cluster opposite a cloud provider in relation to Docker. The limitation of Docker on ARM does not exist at the cloud provider, but the abstractions and understanding of the cluster are harder to grasp. A student points out: *"Cloud providers would have made the user of Docker more meaningful, but not the cluster understanding."*. Furthermore, according to the students, KubeCloud provided them with the ability to improve their skills with relevant real-world technologies.

10

Conclusions

A presentation of the lessons learned is needed in order to draw the overall conclusion of the effects of the designed learning activity and KubeCloud as a learning and research object.

The chapter will present the conclusions and final remarks of this present master's thesis. Many different topics and experiments have been presented. Until now we have presented the new world of rapidly changing software development and the importance of iterating fast, and the need for academia to keep up. We proposed the design of a learning activity involving a small-scale tangible cloud computing environment as a mediating object. The fundamentals of cloud computing infrastructure and architecture were presented to introduce the reader to these new concepts and provide an understanding of the important topics necessary to build and design a learning activity and a tangible cloud computing cluster. Among these topics are the movement towards service-oriented architectures and the introduction of inter-service communication. The complexity involved in building resilient, distributed systems were addressed and a definition of resilience made.

10.1 Lessons Learned

In this section we will describe some of the lessons learned from designing, building, and implementing KubeCloud and the corresponding learning activity.

The Good

A tangible, physical cloud computing cluster helps break down abstractions by making them visible in a physical cluster. In Module 3 all students expressed that KubeCloud supported their learning. A student mentions:*"The cluster illustrates the purpose and setup visually in a nice way. It is very hands on to work with these compared to some cloud setup where the actual servers in the cluster are hidden away"*. Furthermore, the overall evaluation of KubeCloud reported that 100% of the students agree or strongly agree with the statement *"the cluster provided me a better understanding of what cloud consists of"*. These results are in agreement with Felder and Silverman's research and our later confirming experiment on how engineering students learn. Especially in the category of active and sensing learning style preferences.

Visualization of the cluster's current state is important in order to understand the concepts in Kubernetes. In Module 3 the students expressed how the visualization helped their understanding of the cluster management. Furthermore, the overall evaluation of KubeCloud resulted in a that 93.8% agreed or strongly agreed with the statement: *"The cluster combined with the visualizer helped me understand the concepts*

of the cluster". Our experiment on the students' learning style preferences shows that the most dominant learning style preference is the visual.

Conveying skills used by industry is important in order to address the demand from industry. Containers and microservice architecture are entering mainstream according to a survey conducted by Nginx [9]. Furthermore, the external presenter in Module 5 discussed how they use Docker at Praqma. In addition, the presenter advertised a five-day course for students about, among other, containers and Docker to address their need for students with these skills.

A small-scale cloud computing environment is useful for research on real devices. Cloud computing simulations introduce a number of assumptions and, in many cases, fail to encapsule concepts of the application layers. The small-scale cloud computing cluster allows for the introduction of real network faults e.g. by pulling the network cable. This allows for experimentation with resilience at infrastructure and application level in a controlled test environment. Chapter 8 documented the effects of circuit breakers and the trade-off between consistency and availability. Furthermore, the effects of replication were documented in regards to recovery and fault tolerance.

Acquisition of small-scale cloud computing clusters is relatively inexpensive and enables universities to provide cloud computing education. The total cost of acquiring eight KubeClouds were DKK 12,716.96 (USD $1,915). Although the processing power is not comparable to real data centers, the Raspberry Pis provide sufficient processing power to convey multiple cloud computing concepts.

The Bad

The seamless portability of Docker is not possible on ARM-architecture. Throughout the course, the limitation of having to build Docker images on the Raspberry Pi (ARM architecture) has been a source of irritation among the students. Furthermore, the basic idea about Docker is blurred since the seamless portability of images cannot be accomplished. A student also points this out in the overall evaluation: *"The cluster made the workshops and project more real compared to using a cloud cluster. Too bad the ARM architecture broke the seamless experience using docker"*.

The performance of the Raspberry Pi is not impressive, and especially writing to the SD card is a hurdle. This reinforce the inconvenience with Docker because of the relatively large image sizes.

Practical issues were encountered during the setup of the clusters in the classroom for each module. Furthermore, the location of the clusters inside the classroom were distanced from the groups because of the limitations of the network cables. This contributed to a mental barrier between the groups and the cluster, explained by one of the students as: *"The cluster was for a long time "wierd thing" which we were mostly scared of braking - which led to only doing things to it described in workshops."*.

A larger data basis would have increased the validity of the evaluations of KubeCloud. The number of responses to the weekly evaluations vary from 9 to 17. The overall evaluation consisted of 16 students' participating.

10.2 Conclusion

In the thesis, we have described KubeCloud and presented how to design, build, and evaluate KubeCloud and an associated learning activity. KubeCloud is a small-scale tangible cloud computing cluster built of

low-cost Raspberry Pis, which allows for a practical hands-on teaching approach to cloud computing.

The benefit of a physical cloud computing cluster is the ability to inject real network faults without the limitations of simulations. This makes it possible to perform research on resilience in cloud computing e.g. microservice architectures. We have evaluated the effect of KubeCloud as a learning object in a designed learning activity lasting seven weeks. The results show that KubeCloud is a viable learning object for cloud computing educational strategies for engineering students. KubeCloud has acted as a mediating object to align the students' mental model with the real world. The students reported that visualization and hands-on experimentation improved their learning by breaking down the abstractions and making them visible in a physical cluster. The largest limitation of KubeCloud is that Docker images must be built for ARM architecture. This blurs and complicates the seamless experience of Docker. However, the benefits outweigh the limitations.

The evolution of the cloud computing paradigm and the transition towards highly distributed systems calls for new teaching strategies. KubeCloud can have a significant role in future engineering teaching strategies of cloud computing.

10.3 Future Work - Outlook and Perspectives

For future work, we see the possibilities of creating a step-by-step guide (assembly kit) for universities to download and start using KubeCloud as a learning object. We plan on open sourcing all produced materials and experiences for others to use. This makes it possible for universities to offer cloud computing courses targeting the students' preferred learning styles. The students will learn relevant skills preparing them with the required skill set demanded by the industry. This results in developers able to create solutions for the high expectations demanded by the consumers of the future.

Regarding the design of the cluster, further work on stability issues can be made. Some clusters required deletion of data because students directed too much load with a load testing tool. Another area of improvement is the visualization tool since it is slow and gets unresponsive over time.

To further strengthen the conclusion of this present master's thesis, a more detailed comparative study with more participants investigating the effect of using KubeCloud as a learning object would be useful.

We have now presented a low-cost platform for teaching and research. The possibilities of KubeCloud's usage are endless.

A

Designing the Learning Activity

This appendix will describe the different tools and materials used within the designed learning activity. This appendix will highlight the different resources created during this present master's thesis and all material can be found in the Attachment 1-4.

Slides

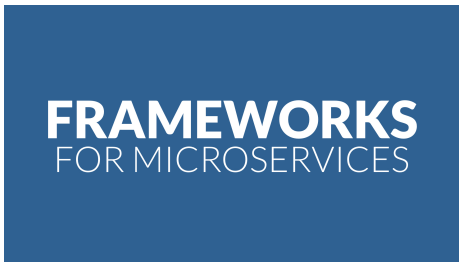
In order to present the topics during the various modules, slideshows have been created. These slideshows are used to present and give the students an introduction into the topic.

Module 1 - Course Introduction and Java Frameworks

Course presentation
(21 slides)
Course introduction and presentation of Martin Jensen and Kasper Nissen. Focus on the course and the topics to be covered.

Cluster Computing
(18 slides)
Sequential vs parallel programming. Anatomy of a Raspberry Pi cluster. Introduction to Message Passing Interface (MPI).

Cloud Computing & Microservices architecture
(60 slides)
Introduction to Cloud Computing, Monolithic and Microservices architecture, Continuous Delivery.



Frameworks for microservices
(21 slides)
Introduction to Spring Boot and Spring Cloud. Small code examples and an overview of how to apply these tools in the project.

Recap and demonstration
(15 slides)
Spring Boot and Spring Cloud recap and presentation of demo project to be demonstrated.

Module 2 - Docker Lightweight Containers

Containers & Docker
(21 slides)
A short introduction to Docker and the challenges Docker addresses. Anatomy of a Docker container and the architecture.

Module 3 - Kubernetes Cluster Manangement

Container Management systems & Kubernetes
(62 slides)
Introduction to Cluster Management and a more detailed introduction to Kubernetes, its concepts and how it works from a high-level perspective.

Module 4 - Resilience and Load Testing

Resilience & Load testing
(58 slides)
Introduction to resilience and load testing. Focusing on Michael Nygard antipatterns and patterns.

Module 7 - Service Discovery

Service Discovery
(20 slides)
Introduction to client-side service discovery, server-side service discovery (Kubernetes), and course evaluation.

Reading and Video Materials

The reading and video material selected to be part of the course curriculum can be seen below. All material has been carefully selected to optimize the students' learning outcome.

Module 1 - Course Introduction and Java Frameworks

Mandatory

- Microservices: a definition of this new architectural term
  - Link: `http://martinfowler.com/articles/microservices.html`
- Video: Josh Long: Getting started with Spring Cloud
  - Link: `https://youtu.be/SFDYdslOvu8`
- The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures
  - Link: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6679872`
- Getting Started with Spring Boot: Building a RESTful Web Service
  - Link: `https://spring.io/guides/gs/rest-service/`

Optional

- Book: Building Microservices, Sam Newman, 2015
- Video: Martin Fowler - Microservices
  - Link: `https://youtu.be/wgdBVIX9ifA`
- Spring Cloud:
  - Link: `http://projects.spring.io/spring-cloud/`

Module 2 - Docker Lightweight Containers

Mandatory

- G. Schulte "What is Docker?"
  - Link: `https://www.youtube.com/watch?v=aLipr7tTuA4`
- D. Bernstein "Containers and Cloud: From LXC to Docker to Kubernetes"
  - Link: `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7036275`
- Docker.com "What is Docker's architecture?" - Only the section "What is Docker's architecture?"
  - Link: `https://docs.docker.com/engine/understanding-docker/#what-is-docker-s-architecture`
- J. Turnbull "The Docker book" p. 1-4 with focus on "What is a Docker image?"
  - Link: `http://dockerbook.com/TheDockerBook_sample.pdf`

Optional

- C. Anderson "Docker"
  - Article: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7093032`

- Podcast: http://www.se-radio.net/2015/01/episode-217-james-turnbull-on-docker/

Module 3 - Kubernetes Cluster Manangement

Mandatory

- Video: B. Dorsey "Kubernetes: Changing the Way That we Think and Talk About Computing", Brian Dorsey
  - Link: https://www.youtube.com/watch?v=DGlQgNmobuc
- B. Burns (Google Research) "Borg, Omega and Kubernetes"
  - Link: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf
- A. Mouat "Swarm v. Fleet v. Kubernetes v. Mesos"
  - Link: https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos

Optional

- K. Hightower "Day 1, Opening Keynote: Kubernetes Update; KubeCon EU 2016"
  - Link: https://www.youtube.com/watch?v=Wyl4O3CHzV0
- J. Langemak "Kubernetes 101 – Networking"
  - Link: http://www.dasblinkenlichten.com/kubernetes-101-networking/
- B. Burns "GCP Next: Painless Container Management with GKE & Kubernetes"
  - Link: https://youtu.be/_xNFt7FsWaA?list=PLIivdWyY5sqKXJZfLHVaKidLsW9P949Zi

Module 4 - Resilience and Load Testing

Mandatory

- Video: Boner, J.: Without resilience nothing else matters
  - Link: https://youtu.be/beC49rexj7I
- Nygard, M - Release It!: Chapter 5 - Stability Patterns, page 89-116
- Mayada, O: Resilience Of Networked Infrastructure Systems: 2.1, 2.2, 2.3 (-2.3.2)
  - Link: http://site.ebrary.com.ez.statsbiblioteket.dk:2048/lib/stats/reader.action?ppg=31&docID=10775250&tm=1461092185230

Optional

- Friedricsen, U.: Patterns of Resilience
  - Link: https://youtu.be/T9MPDmw6MNI

Module 5 - Project Work and External Presentation

Mandatory

- "Understanding DevOps & Bridging the gap from Continuous Integration to Continuous Delivery", Virmani, M. 2015
  - Link: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7173368

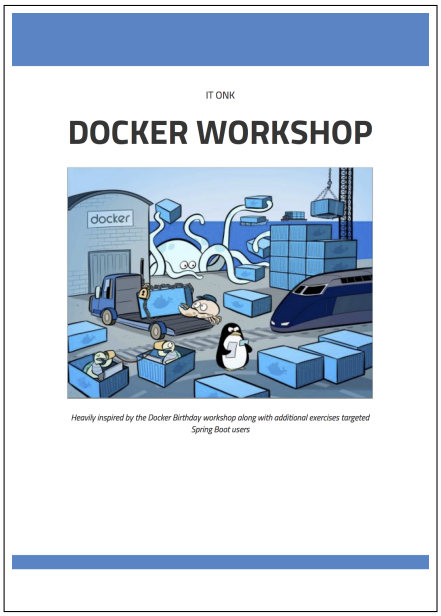
Module 7 - Service Discovery and Project Presentations

Mandatory

- "Service Discovery in a Microservices Architecture", Richardson, C., 2015
  - Link: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Workshops

Workshops have been a created to give the students direct hands-on learning experiences. This approach is selected since both Docker and Kubernetes uses a command-line tool, that has to be learned through use. Furthermore, the Resilience and Load testing workshop adds command-line tools that has to be used in order to run these kinds of tests.



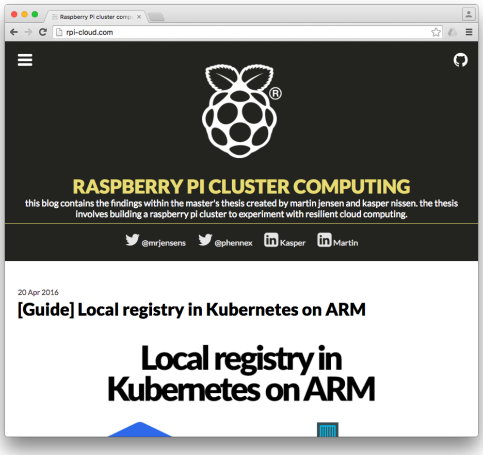
Docker workshop
(17 sider)
This workshop will give the student the ability to create docker images, run these images as containers, and lastly build and package a Java Spring Boot application into a container and running it on a Raspberry Pi. Furthermore this workshop will provide more detailed insights in how Docker works, along with developing the students confidence in using Docker thereby improving their skills.

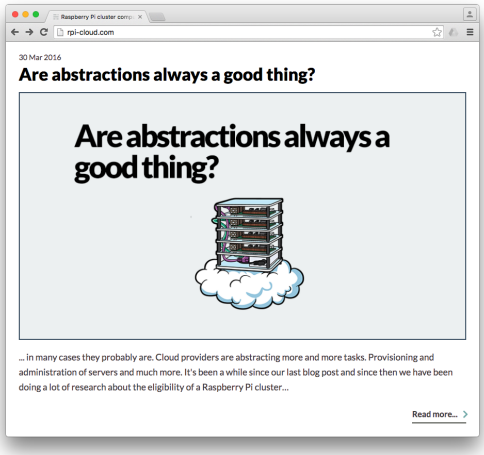
Kubernetes workshop
(13 sider)
This workshop will provide the student with the ability to deploy containerized applications into the Raspberry Pi Kubernetes cluster. It will provide hands-on experience with using the command-line tool kubectl and familiarize the student with the presented concepts and how they can be leverage with Kubernetes.

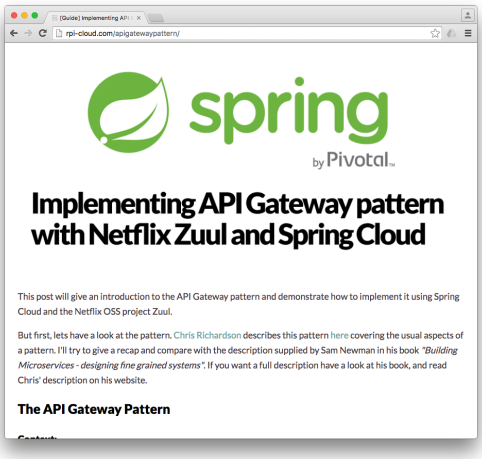
Resilience & Load Testing workshop
(19 sider)
This workshop will provide the student with insight into the possible faults and errors that can happen in a distributed architecture. This workshop features already built docker images, the students has to configure and run different load-testing scenarios. The focus is on the effect of applying the Circuit Breaker pattern and using replication in Kubernetes.

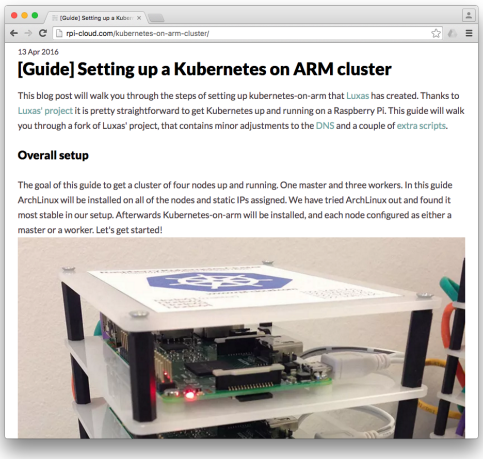
Blog

A blog has been created to share the progress of the project along with a source for the course in terms of guides explaining how to set up, e.g. Spring Cloud Zuul or similar. This guide contains a total of 18 blog posts about how to setup a Raspberry Pi Kubernetes Cluster, How to use Spring Boot, Configurations for various Kubernetes related topics, etc.









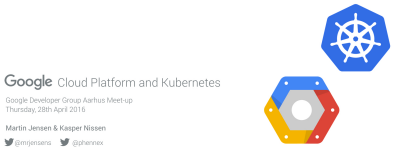
Additional Tools and Methods

In order to get the most possible experience with teaching and speaking publicly, some external presentations have been given. These events have helped us be better instructors and get feedback from the community about what we are trying to accomplish.

The following additional activities have been done during this present master thesis

- Docker mentors at 'Docker Meet-up group Aarhus: Docker Birthday 3 years'
- Presentation at IT Minds ApS to an internal event
- Presentation at Google Developer Group Aarhus meet-up event
- Workshop at Praqma about Kubernetes

Morgenbooster: Demystifying the cloud
(39 slides)
30 minute presentation and demonstration of Microservices, Docker, Kubernetes. Presentation was given at IT Minds ApS, Aarhus, Denmark.



Google Developer Group: Google Cloud Platform and Kubernetes
(93 slides)
60 minute presentation and demonstration of Cloud Computing, Google Cloud Platform, and Kubernetes. Presentation was given at Google Developer Group Aarhus Meet-up.

B

Additional Kubernetes Concepts

This appendix describes how a declarative definition of a production environment can be specified using YAML files. Furthermore, canary deployments, scaling, and databases in Kubernetes are described.

Declarative YAML Files

Kubernetes promotes the use declarative definition of production environments. These definitions are often stored as YAML files. Listing B.1 shows a definition of a deployment of the cat-service from the demo project described Chapter 7. The definition specifies the deployment's labels and names. Furthermore, a replica set is defined in the *spec* section. The replication set is specified to keep three replicas running of pods defined from the *template*. Each pod contains a single container constructed from a specified Docker *image*. The *labels* on under the *template* and *metadata* plays an important role in regards to services.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: cat-service
    visualize: "true"
  name: cat-service
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: cat-service
        tier: backend
        track: stable
        run: cat-service
        visualize: "true"
    spec:
      containers:
      - image: rpicloud/demo-cat-service:v1
        name: cat-service
  minReadySeconds: 120
```

Listing B.1: Cat-service v1 deployment

The definition is saved as a YAML file and used by the command-line tool. The deployment of cat-service v1 could e.g. be created in the following way:

```
kubectl create -f cat-service-v1-deployment.yaml
```

Listing B.2 shows a definition of a service. The important part is the *selector* that decides which pods to route traffic to. The cat-service service filters pods by their labels. The cat-service service routes traffic to pods having the labels: *app: cat-service* and *tier:backend*. The previous deployment (Listing B.1) contains these labels, and traffic will, therefore, be directed to the deployment's pods. Labels can be added and removed dynamically which leads to a loose and dynamic coupling between services and pods. Services stored in YAML files are also created using the command-line tool.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cat-service
    tier: backend
    run: cat-service
    visualize: "true"
  name: cat-service
spec:
  ports:
  - port: 9000
    protocol: TCP
    targetPort: 9000
  selector:
    app: cat-service
    tier: backend
```

Listing B.2: Cat-service service

Canary Deployments

One of the benefits of the loose coupling between services and pods is the ability to make canary deployments and the possibility of doing AB testing. A part of the incoming requests are routed to a new version while the majority of the requests will be directed to an old version. An example of this, and how it can be done using labels, is shown in Figure B.1.

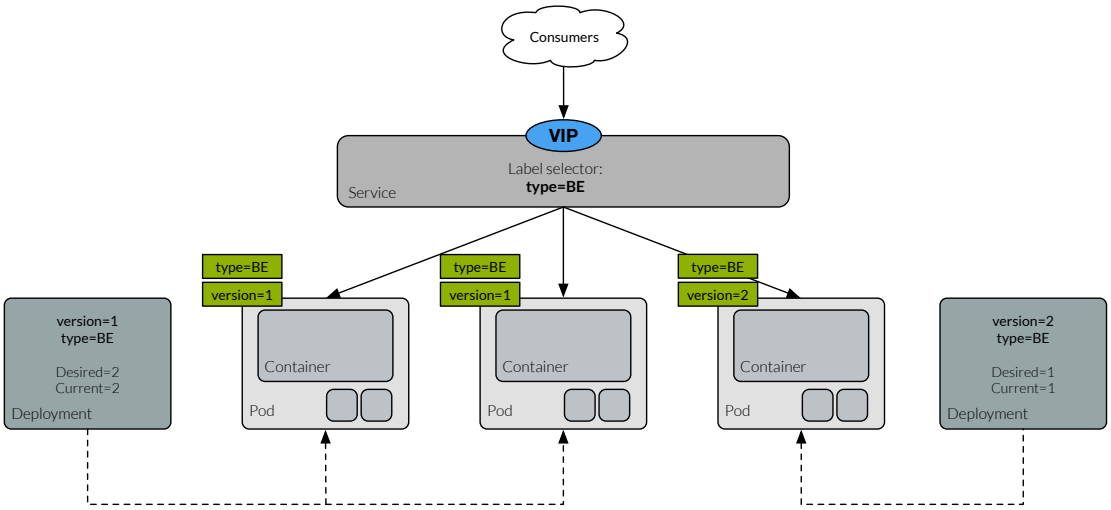


Figure B.1: Canary Deployment

The canary deployment is done using labels and two deployments. As it is seen in Figure B.1 the service has a label selector *type=BE*. There are two different deployments, one controlling version 1 of the application and one controlling and maintaining the desired state of version 2. Because both of the deployments' pods have the label *type=BE* traffic is directed to them. Listing B.3 shows the canary deployment containing version 2 of the cat-service. Notice that the *app* and *tier* labels are as the service expects, and that the image ends with the tag *:v2*.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: cat-service
    visualize: "true"
  name: cat-service-canary
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: cat-service
        tier: backend
        track: canary
        run: cat-service
        visualize: "true"
    spec:
      containers:
      - image: rpicloud/demo-cat-service:v2
        name: cat-service-canary
  minReadySeconds: 120
```

Listing B.3: Cat-service v2 canary deployment

Scaling

Kubernetes features manual scaling and autoscaling. A deployment can easily be scaled by using the *kubectl* command-line tool. The example below scales the number of running replicas to 10 for a deployment called NAME.

```
kubectl scale deployment NAME --replicas=10
```

Scaling replicas can also be achieved using YAML definitions. Instead of using the *kubectl create* the *kubectl apply* command is used.

Kubernetes has an autoscaling mechanism is called *Horizontal Pod Autoscaler (HPA)*. HPA allows to automatic scaling of the number of pods in a replica set or a deployment based on observed CPU utilization. This mechanism is implemented as a Kubernetes API resource and a controller. The resource describes the behavior of the controller, whereas the controller periodically adjusts the number of replicas to match the average CPU utilization to the target specified by the user. An example of this can be seen in Figure B.2.

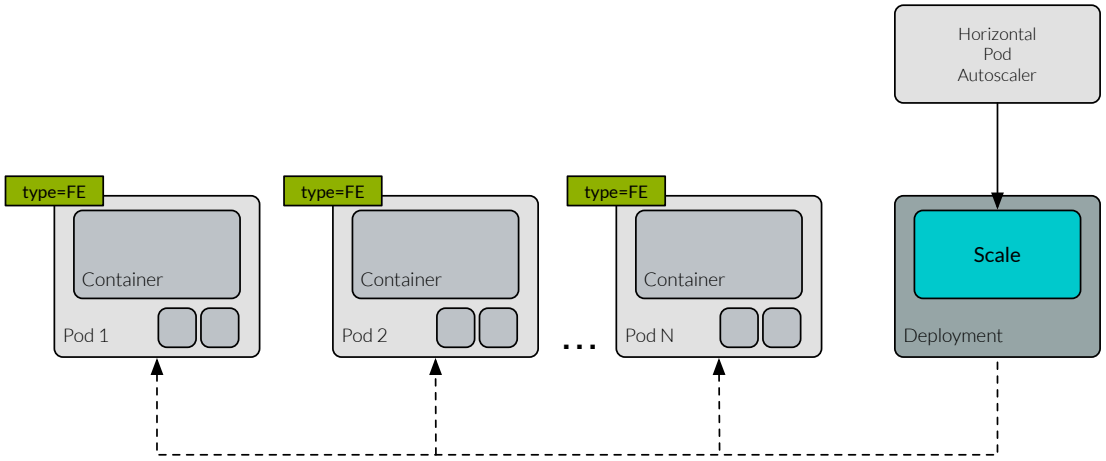


Figure B.2: Horizontal Pod Autoscaler

Databases in a Dynamic Environment

The role of databases has not been the focus of this master's thesis, but because it is an important part of applications the options in Kubernetes will be described in this section. Databases in dynamic and stateless environments such a Kubernetes is a challenge. Packaging a database in a Kubernetes pod and saving data in a container is not a stable solution because pods are ephemeral instead of long-lasting and durable. Three available database solutions described by Dorsey[1] are described in this section: outside of the cluster, adapting to the cluster, and cluster native.

[1] https://www.youtube.com/watch?v=DG1QgNmobuc

Outside of the Cluster

The first approach is to keep the database out of the cluster and have applications connect to a database outside the Kubernetes cluster. The database can be an already running database. Moving the data away might, though, lead to higher latencies depending on the database's location.

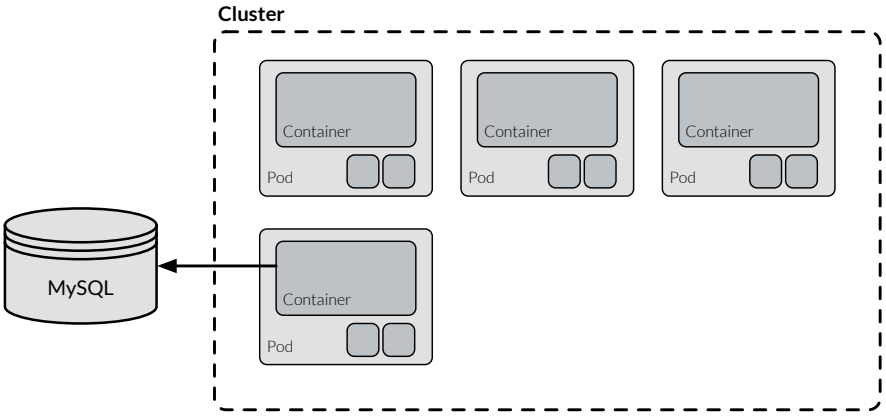


Figure B.3: Database outside Cluster

Adapt to the Cluster

In Kubernetes, volumes can be mounted in pods. Thereby databases can be started in containers without storing data in an ephemeral container but instead stored on the external volume. The volume can e.g. be a networked file share (NFS) or storage from a cloud provider.

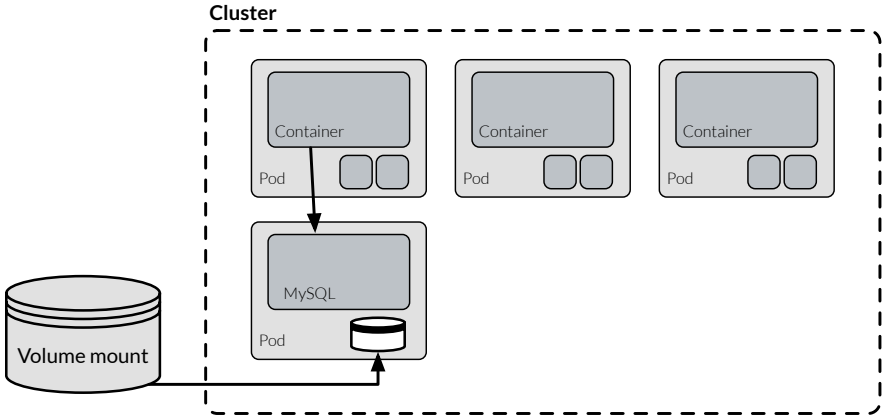


Figure B.4: Database Adapting to Cluster

Cluster Native

Databases such as Cassandra and Riak are designed to run in a distributed environment which is more cluster native approach. Availability, scalability, and fault tolerance are some of the benefits they provide. As described in Chapter 6, a trade-off between consistency and availability can be made e.g. in Cassandra depending. Sharding and replicating databases are factors used to achieve these benefits.

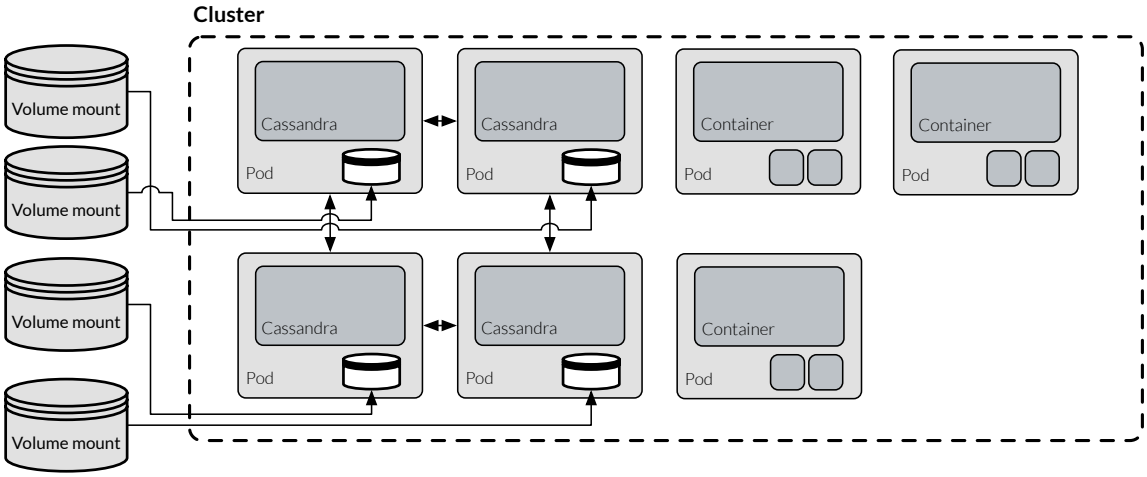


Figure B.5: Database Cluster Native

C

Characteristics and Principles of Microservice Architecture

Fowler and Lewis [53] describe nine characteristics of microservice architecture.

Nine Characteristics of a Microservice Architecture
Fowler and Lewis describe nine characteristics of a microservice architecture in their article *Microservices, a definition of this new architectural term, 2014* [53]

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dump pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

Newman [51] also describes seven principles of microservice architecture.

Seven Principles of Microservices
Newman describes seven principles of microservice architecture in his book *"Building Microservices - Designing Fine-grained systems"* [51, p. 245-249]

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

The following will provide a detailed walkthrough of the nine characteristics of Microservice architecture defined by Fowler and Lewis. Furthermore, Newman's seven principles will be related to these characteristics.

Componentization via Services

Fowler and Lewis define a component to be *"a unit of software that is independently replaceable and upgradeable"* [53, p. 4]. Componentization via Services refers to breaking down the architecture into services. Services then become inter-process components which communicate with other services through what Fowler and Lewis refer to as *dumb pipes*. This decomposition allows for services to be independently deployable. However, one of the downsides is that communication between services is replaced with remote calls, instead of intra-process calls, which introduces the non-deterministic behavior of a network. Newman also describes this as one of his seven principles, *Hide Implementation Details*, and states that *"to maximize the ability of one service to evolve independently of any others, it is vital that we hide implementation details"* [51, p. 247].

Organized around Business Capabilities

As described in Chapter 5, one of the main inspirations in microservice architecture is the concept of Domain Driven Design (DDD). DDD focusses on identifying entities from the domain and applying them to the implementation. Furthermore, the concept of bounded context focuses on separating these entities and grouping them in separate context among high cohesive services. This allows for high decoupling. Eric Evans argues in DDD [71, p. 217] that a unification of the domain model is not feasible or cost-effective. A unification of the domain model results in one view of each entity, and this is usually not the case in an enterprise system. An example of this are the entities *Customer* and *Product* in a e-commerce system (Figure C.1). If two teams are developing components using the same view of the model but with different understandings or interpretations of how e.g. an attribute of the model is to be used. Figure C.1 depicts the situation of different interpretations of what a *Customer* or a *Product* is in different contexts. This is where the concept of a bounded context comes to the rescue and splits the domain model into two separate contexts with each context having their own internal representation of the shared model.

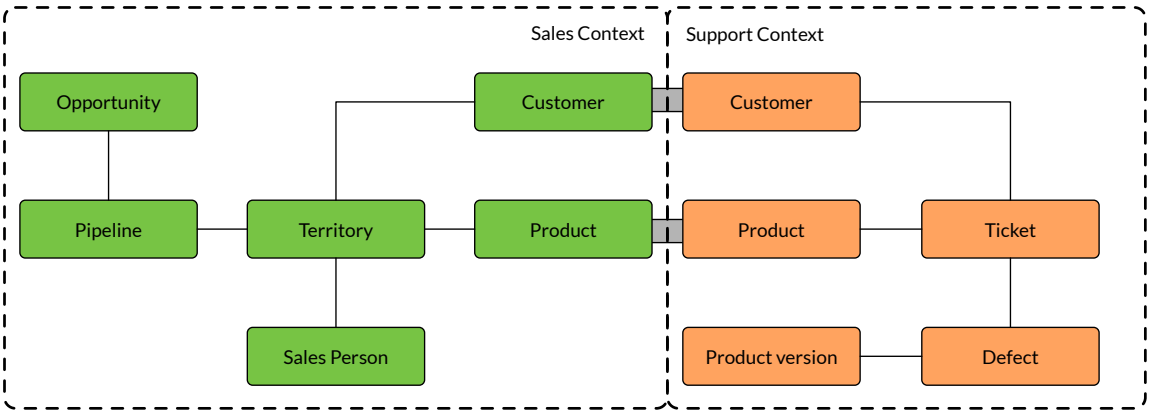


Figure C.1: Bounded Context [72]

The traditional way of managing large application is often, for management, to focus on separating into technology layers instead of business capabilities, leading to UI teams, server-side logic teams, and database teams. The problems that may arise when teams are separated along technology lines is that even simple changes can lead to cross team projects, resulting in longer implementation times. Melwyn Conway described in 1968 that an organization's communication structure will be reflected in the way teams design their systems.

> *"[...] organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations"* **- *Melwyn Conway, 1968*** *[54, p. 31]*

Figure C.2 shows Conway's law in action. Siloed functional teams lead to siloed application architectures.

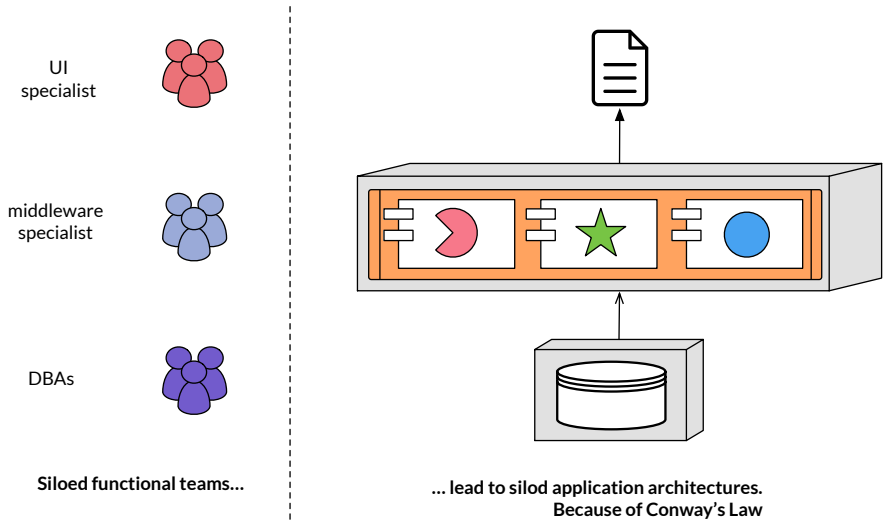


Figure C.2: Conway's Law in Action [53, p. 5]

Instead, the microservice architectural approach is to organize around business capabilities in cross-functional and autonomous teams that clearly reinforce the boundaries between services and teams. Teams are responsible for the full stack, which is clearly reflected in Amazon's motto: *"you build it, you run it"*. Newman describes this under his principle, *Model Around Business Concepts*, and states that *"interfaces structured around business-bounded contexts are more stable than those structured around technical concepts"* [51, p. 246]. The microservice approach to organization is depicted in Figure C.3.

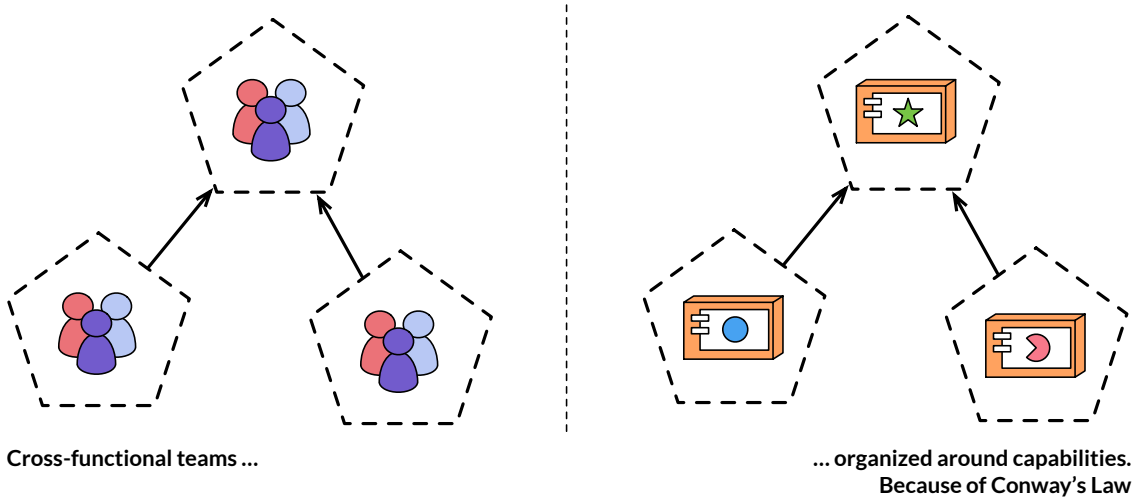


Figure C.3: Organized around Business Capabilities [53, p. 6]

Products not Projects
Many application development efforts use a project model where the goal is to deliver software. When the software is developed and delivered, the project is considered complete. In many cases the software is then handed over to a maintenance or operations team and the project team, that build the software, is disbanded. Microservice architecture style tries to avoid this model, by preferring the notion that a team should own a product over its full lifetime. Rather than looking at software as a set of functionality to be completed, there is an on-going relationship where the question is: *"how can software assist its users to enhance the business capability"* as Fowler and Lewis describe it [53, p. 7].

Smart Endpoints and Dumb Pipes
There exist multiple ways of communicating between different processes. One approach is the Enterprise Service Bus (ESB) which puts a lot of *smarts* into the middleware layer. The Enterprise Service Bus is as mentioned in Chapter 5 an architecture used for designing and implementing communication between software applications in a service-oriented architecture (SOA). The ESB is responsible for e.g. routing of message exchanges between services and a lot other *smarts*. The microservice community favors putting the smarts into the services instead of in the middleware. They build decoupled and highly cohesive services that own their own domain logic. The most commonly used protocols are HTTP and lightweight messaging, such as RabbitMQ or ZeroMQ.

Decentralized Governance
Fowler and Lewis state that *"one of the consequences with centralized governance is the tendency to standardize on single technology platforms"* [53, p. 7-8]. This approach is constricting - *"not every problem is a nail and not every solution is a hammer"* [53, p. 8]. When splitting applications into services, architects have the choice of using the right tool for the particular responsibility, that a specific service needs to fulfil. The team is not restricted to use a specific technology, since they are working on an isolated service. However, *"just because you can do something, doesn't mean you should"* [53, p. 8]. The technology stack is not the only decentralized part in a microservice architecture. Centralized standards are not preferred and teams usually build helpful tools instead that can be shared among teams and even open sourced. Newman describes decentralization in his seven principles in the principle *Decentralize all the things*, where he describes the importance of *"embracing self-service wherever possible, allowing people to deploy software on demand, making development and testing as easy as possible, and avoiding the need for separate teams to perform these activities"*. [51, p. 247]

Decentralized Data Management
The monolithic approach is having one central database. This method can introduce lots of difficulties, since we allow different services to view and bind to the internal implementation details of the database schema. All services using this shared database have access and basically, the database is shared in its entirety. If a service wants to change the schema, represent the data better, or make the service easier to maintain, all services using the database will break. Furthermore, this approach ties consumers of the database to a specific technology choice. As Newman mentions, *"perhaps right now it makes sense to store customers in a relational database [...] what if over time we realize we would be better off storing data in a nonrelational database?"* [51, p. 41]. This approach has high coupling between services that consume the database and the database itself, which is not desirable.

Instead of having one central database implementing the entire domain model, microservice architectures embrace the concept of decentralized data management. Again referring to the principles of Domain Driven Design and the notion of bounded contexts. This results in bounded contexts owning their own data and domain model and only exposing necessary data through its external interfaces. Fowler and Lewis describe that the microservice community *"prefers letting each service manage its own database,*

either as different instances of the same database technology or as entirely different database systems. This approach is called Polyglot Persistence" [53, p. 9]. Newman contributes to this view: *"Services should hide their database to avoid falling into one of the most common sorts of coupling that can appear in traditional service-oriented architectures, and use data pumps or event data pumps to consolidate data across multiple services."* [51, p. 247]. Figure C.4 shows the difference between the centralized and decentralized data management approach.

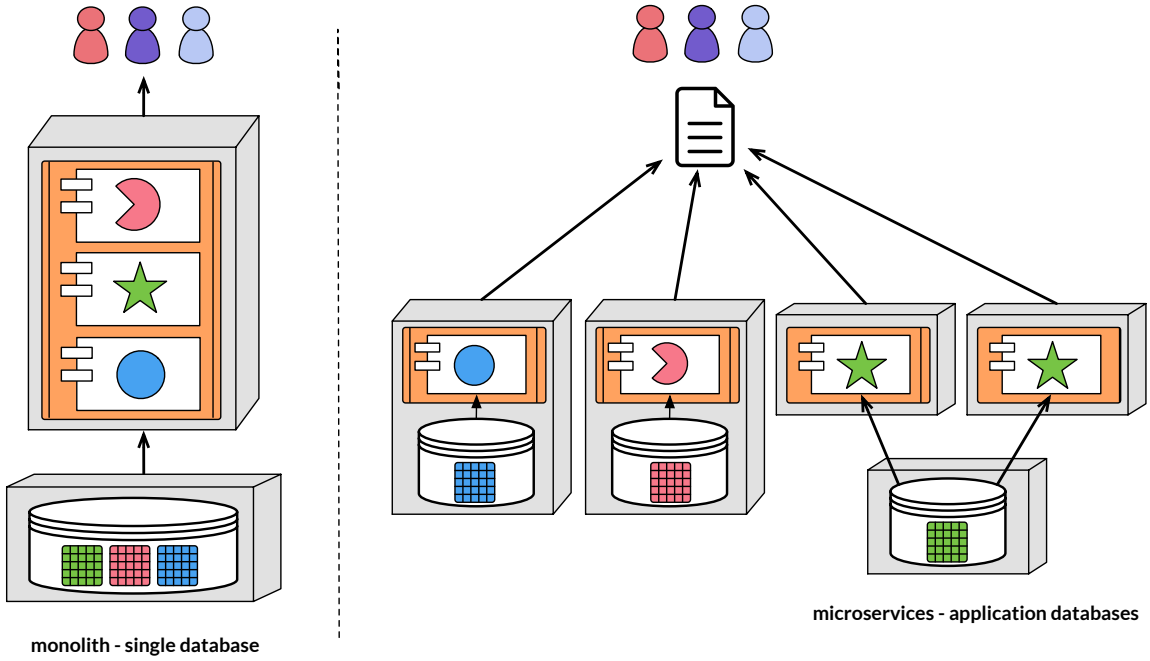


Figure C.4: Decentralized Data Management [53, p. 10]

Centralized data management is usually used within monolithic applications to ensure consistency. By using transactions in a centralized approach, it is easy to roll-back if something goes wrong. In the decentralized approach, distributed transactions are hard to implement, and working with microservices usually settles with eventual consistency. However the argument, as Fowler and Lewis state, *"is that the trade-off of having to deal with a degree of inconsistency to respond quickly to demand is worth it if fixing mistakes is less than the cost of lost business under greater consistency."* [53, p. 10]

Infrastructure Automation

One of the key enabling technologies for microservice architecture has been the evolution of infrastructure automation techniques for the past five years. This evolution has reduced the operational complexity of building, deploying and operating applications. Fowler and Lewis explain that *"many of the products or systems being build with microservices are being built by teams with extensive knowledge of Continuous Delivery and its precursor Continuous Integration"*[53, p. 10]. Two key concepts of Continuous Delivery worth mentioning are the extensive use of automated tests and the promotion of working software pipelines to automate deployments. Newman's *Culture of Automation* principle describes this characteristic and states: *"microservices add a lot of complexity, a key par of which comes from the sheer number of moving parts we have to deal with. Embracing a culture of automation is one key way to address this"* [51, p. 246]. He further states that *"Automated testing is essential"*. When deploying monolithic applications the entire application usually has to be redeployed. Microservices allow teams to deploy services independently, making the deployment landscape strikingly different compared to monolithic applications. This is depicted in

Figure C.5

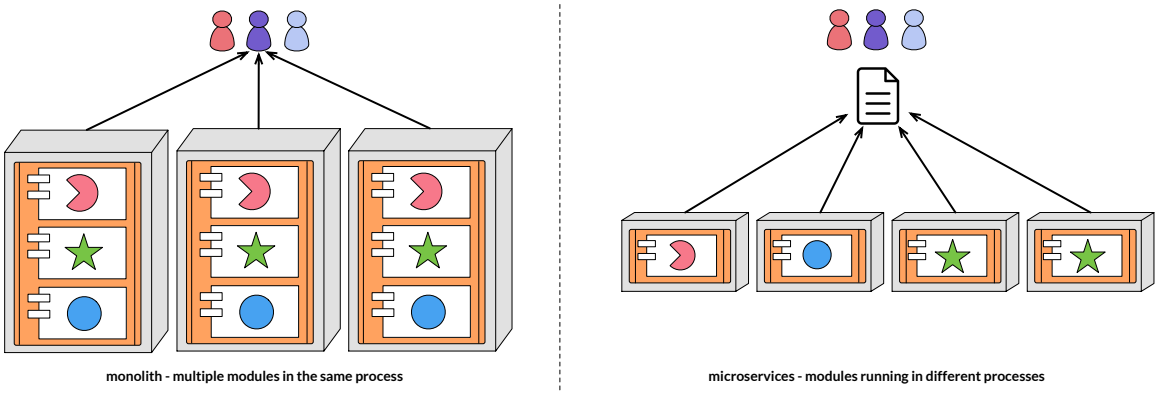


Figure C.5: Infrastructure Automation [53, p. 11]

Design for Failure

One of the big consequences of transitioning from a monolithic architecture to a microservice architecture is the movement from an intra-process integrated application to a distributed modularized system. Any call between services can and will at some point fail, due to the non-deterministic behaviour of the underlying network. Microservice architecture therefore needs to be designed for failure, and to respond as gracefully as possible in case of failure. To cope with these potential failures, Netflix has introduced the Simian Army which is discussed in Chapter 6. Basically they introduce failures of services and even data centers during working days to test both applications' resilience and monitoring. Newman presents two principles that revolve around this characteristic, namely the principles Isolate failure and Highly observable. He states that *"a microservice architecture can be more resilient than a monolithic system, but only if we understand and plan for failures in part of our system"* [51, p. 248]. Newman further emphasizes the importance of aggregation of logs and stats. *We cannot rely on observing the behavior of a single service instance or the status of a single machine to see if the system is functioning correctly* [51, p. 249].

Evolutionary Design

Fowler and Lewis state that *"microservice practitioners come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change"* [53, p. 12]. Controlling change does not necessarily mean a reduction in how frequent changes are implemented. When decomposing applications into services we are faced with the decision of how to slice up the application. Fowler and Lewis emphasize that the key property here *"is the notion of independent replacement and upgradability"* [53, p. 12].

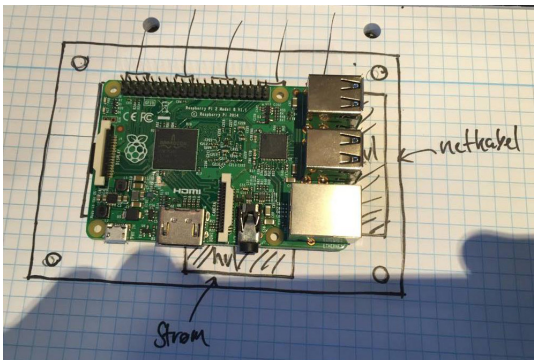
D

Physical Design

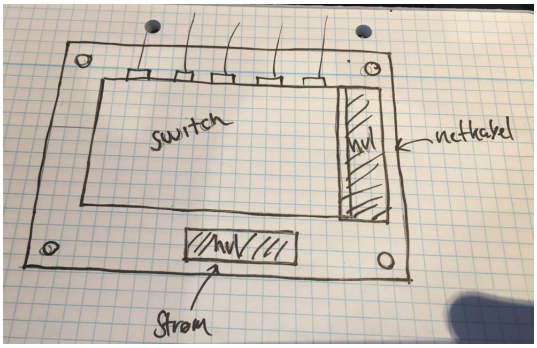
The cluster has been designed to look like a scale-model of a server rack in a datacenter. This appendix will present the sketches, 3D models, technical drawings and final physical layouts.

Sketches

In order to design the cluster to fit the size of a Raspberry Pi several sketches and drafts were made. Figure D.1 (a) shows how the the cable holes were designed for the layers with Raspberry Pis. Two different layers were made: a layer serving as the bottom/top layers and a layer with holes to mount a Raspberry Pi and holes for power and network cables. Figure D.1 (b) shows the first version of the Raspberry Pi layer with the location of the switch shown underneath. The switch's cable plugs were later moved to the back because we found a smaller switch than expected.



(a) Sketch of Raspberry Pi on layer



(b) Sketch of Raspberry Pi layer with switch

Figure D.1: Sketches

3D Models

Figure D.2 shows the designed 3D model before components were bought. The switch was replaced with another model, but the rest of the model is identical to the clusters we have build. More information and 3D models can be found at `http://rpi-cloud.com/guide-how-to-build-a-raspberry-pi-cluster/`.

Figure D.2: Cluster 3D Model

Technical Drawings

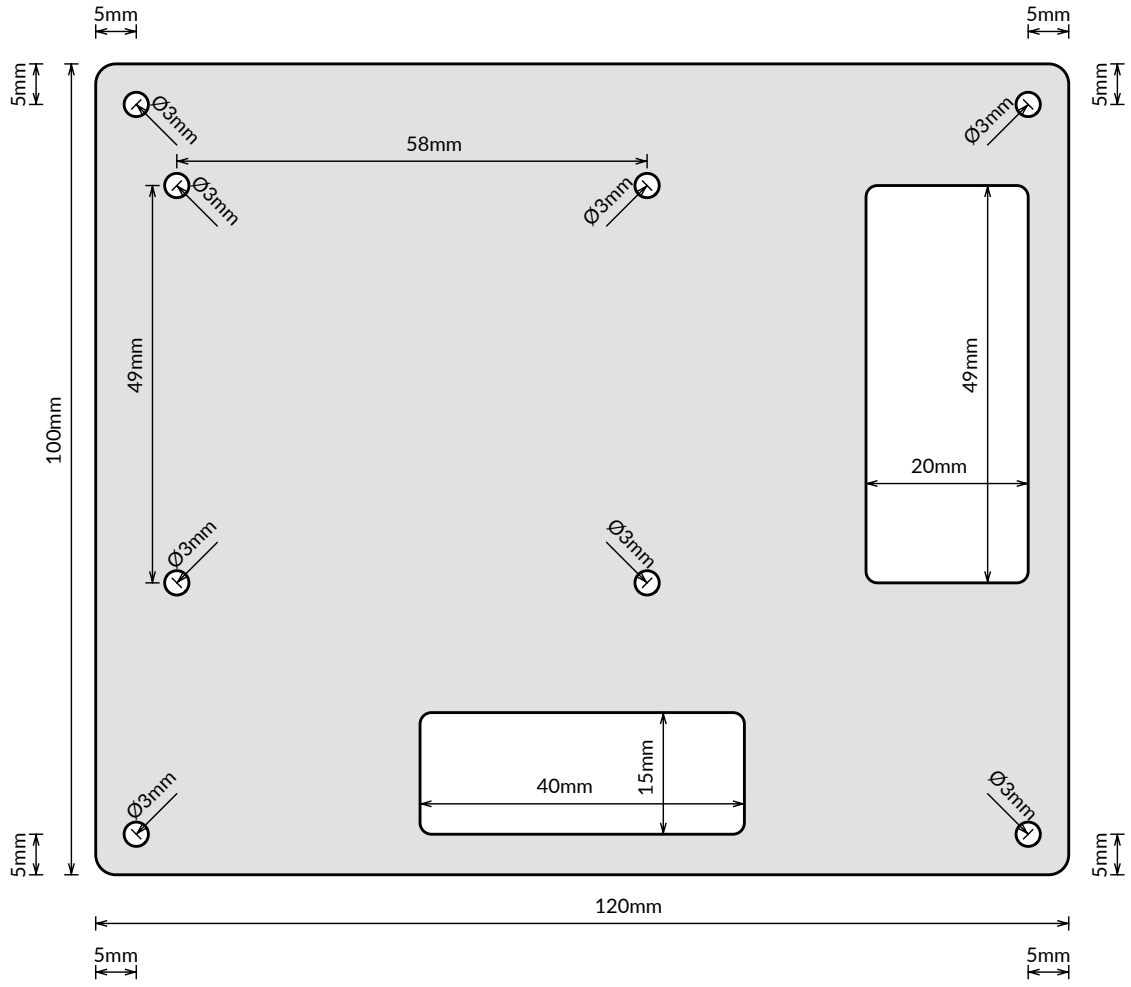


Figure D.3: Raspberry Pi Layer 1:1

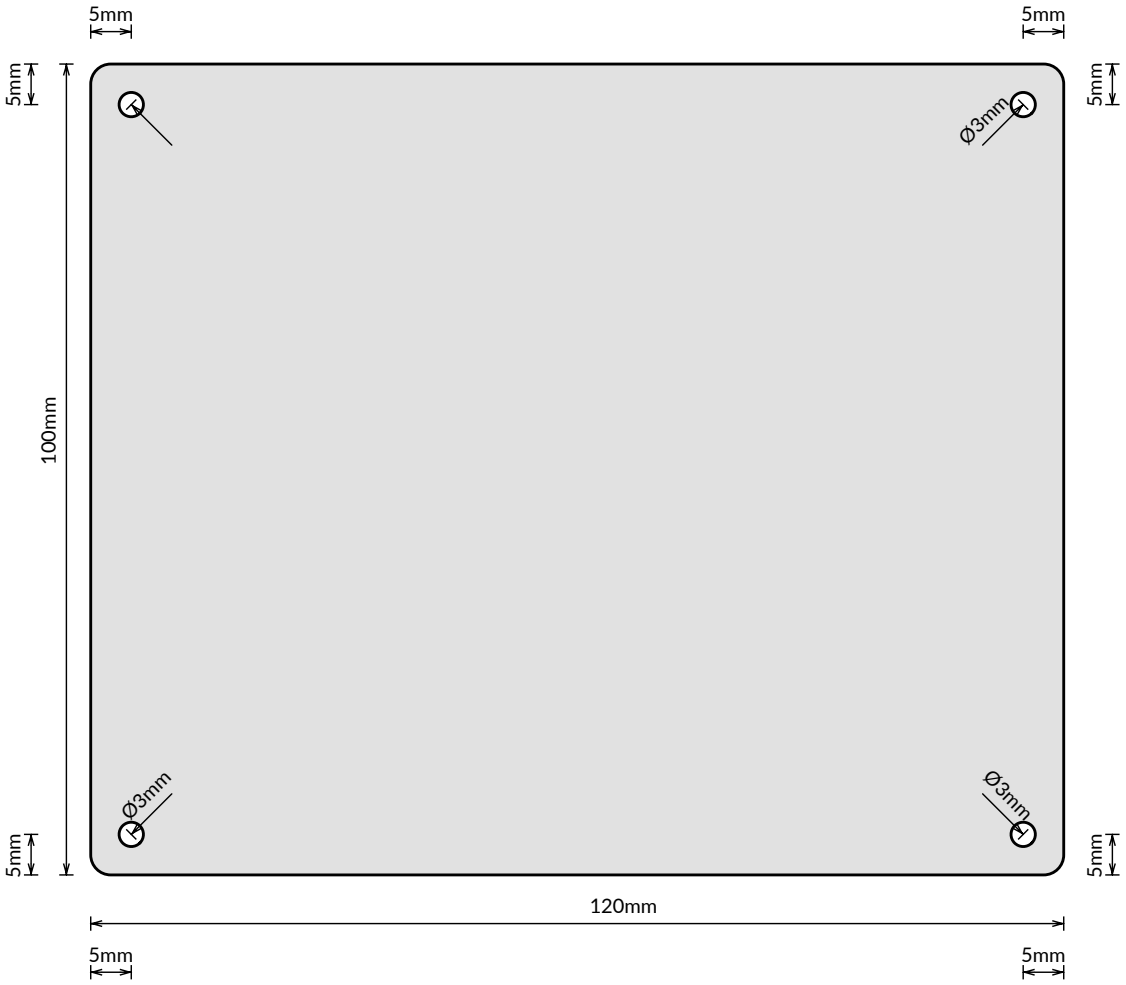
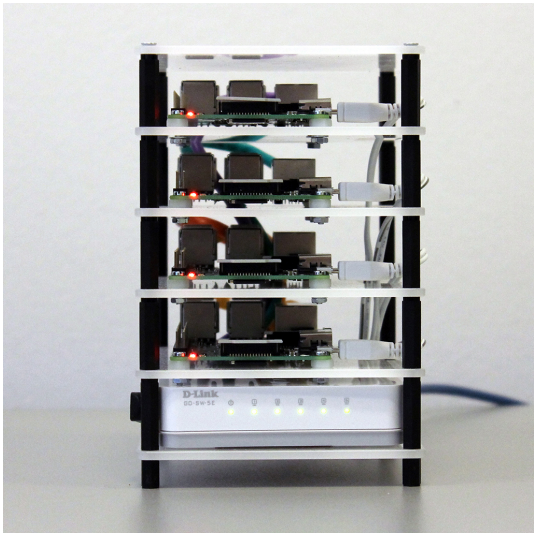


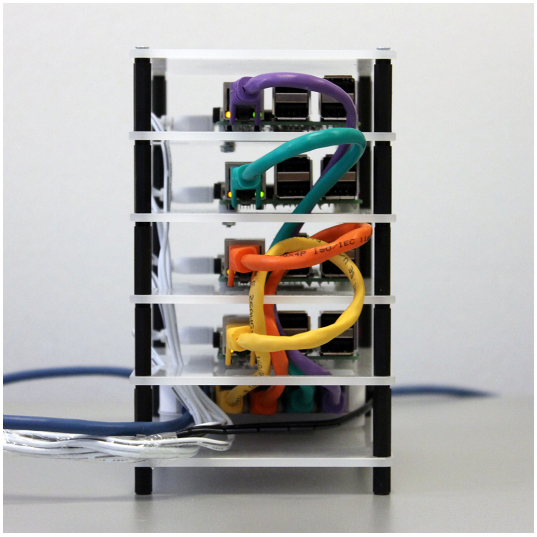
Figure D.4: Top/bottom layer 1:1

Final Physical Layouts

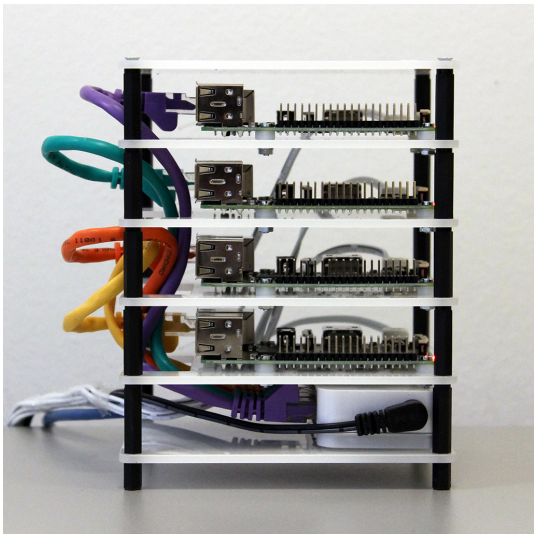
The fabricated cluster is shown in the following images.



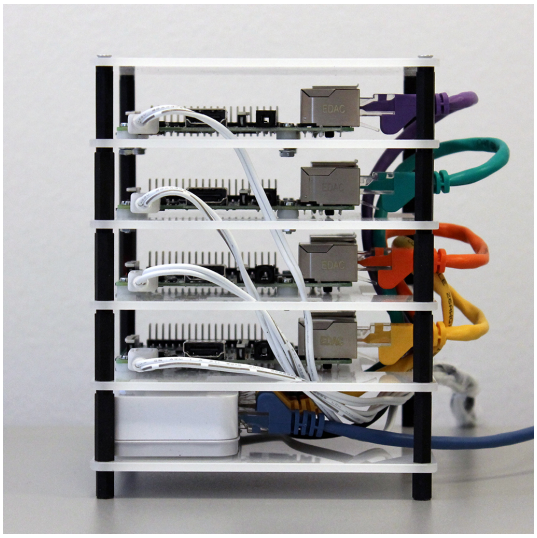
(a) Front



(b) Back



(c) Left



(d) Right

Figure D.5: KubeCloud

Application Layer - Spring Boot & Spring Cloud

This Appendix will provide an overview of the software used in the application layer within this present master thesis.

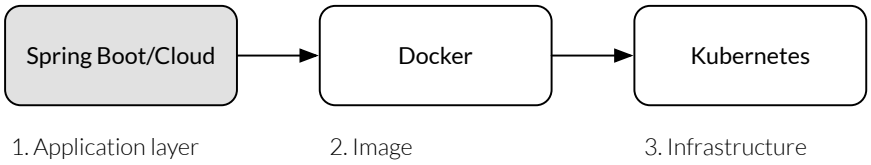


Figure E.1: Application flow

There exist a lot of different frameworks for building microservices. A popular framework that lets you get started fairly quickly is Spring Boot and Spring Cloud. These two frameworks are chosen because of the rapidness of getting started. Another important reason is that Spring Boot runs great with Java. The 10 students of the 17 students answered Yes to the Votiee question about whether or not they have used Java.

What is Spring Boot?

Spring Boot is a framework developed by Pivotal. It is designed to simplify bootstrapping and development of new Spring applications. Spring Boot takes an opinionated view of building production-ready Spring applications. It favors convention over configuration and is designed to get you up and running as quickly as possible. Spring Boot has evolved into what is defined as a chassis framework for microservices by Chris Richardson.[1] A microservice chassis framework helps you develop microservice applications fast. Spring Boot projects can easily be started via the Spring Initializr[2], which allows you to configure your project and set up dependencies. The Spring Boot framework can be used on the JVM with languages such as Java, Groovy, or Kotlin.

[1] http://microservices.io/patterns/index.html
[2] http://start.spring.io

What is Spring Cloud?

Spring Cloud builds on Spring Boot and provides developers a way to quickly build common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).[3]

Spring Cloud Netflix OSS

Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment. Netflix Open Source Software Center (Netflix OSS) is the source for Netflix services. Netflix has been a huge contributor to the open source environment, and many to the tools used at Netflix are open sourced and can easily be integrated into your applications. Spring Cloud Netflix enables you to quickly enable and configure Netflix OSS software by applying a few simple annotations. Some of the patterns used within this present master thesis includes, Circuit Breaker (Hystrix), Intelligent Routing (Zuul), HTTP Rest client (Feign).

Zuul - API Gateway

The first Netflix OSS component we will discuss is the Netflix's implementation of the API Gateway pattern, namely Zuul. Before going into the details of Zuul, we will first describe what the API Gateway pattern is. The pattern is described by Chris Richardson.[4]

The API Gateway Pattern

Context

Building a microservice architecture, you have multiple clients communicating with your backend. The clients communicate with individual services and thereby have to communicate with many different services to aggregate the data internally upon reception of responses from these services

Problem

How do clients minimize the number of request to the backend, e.g. reduce these chatty interfaces? and how to allow clients to communicate with individual services in a microservice based application?

Forces

Chris Richardsen describes some of the many forces as follows:

- Granularity of APIs provided by microservices is often different than what a client needs
- Different clients need different data
- Network performance is different for different types of clients
- The number of service instances and their locations (host+port) changes dynamically

[3] http://projects.spring.io/spring-cloud/
[4] http://microservices.io/patterns/apigateway.html

- Partitioning into services can change over time and should be hidden from clients

Solution

The solution is to implement a server-side aggregation endpoint or API gateway. The API Gateway is responsible for aggregating data or in some cases acts as a simple routing layer for appropriate services. The API gateway can be seen as a single point of failure, or as Sam Newman describes it as a *single monolithic gateway*. A better approach may be to create multiple API gateways for the different platform or frontends that your application needs to support. This is also referred to as *backends for frontends*.

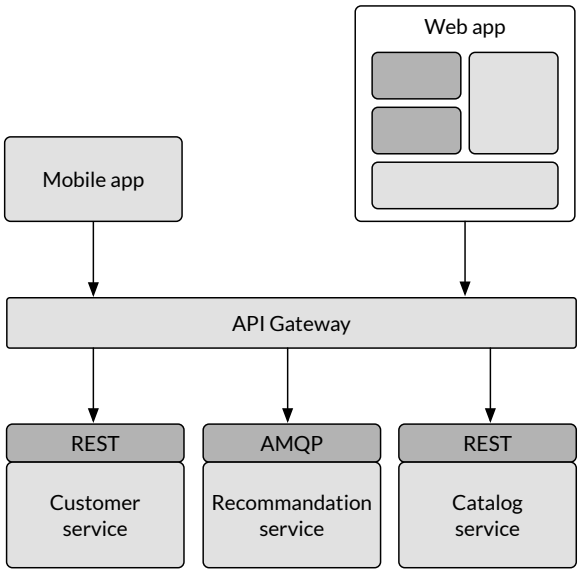


Figure E.2: Pattern: API Gateway

Resulting Context

Benefits:

- Clients are isolated from the partitioning of the microservice architecture behind the gateway
- Clients do not have to worry about locations of specific services
- Reduces the number of requests/roundtrips
- Simplifies the clients by moving the aggregation logic into the API gateway

Drawbacks:

- Increased complexity - API gateway is one more added to the list of moving parts within you microservices architecture
- Increased response time compared to direct calls, because of the additional network hop through the API gateway
- Danger of implementing to much logic in the aggregation layer

Zuul

To implement the API Gateway pattern in Spring Cloud we use the Netflix OSS component, Zuul. Zuul makes the process of implementing this pattern easy. In your main application file, a simple annotation @EnableZuulProxy has to be added

```
@SpringBootApplication
@EnableZuulProxy
public class ApiGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

Besides this simple annotation, we need to apply some configuration on what Zuul has to do when it receives a request. This can be configured in application.properties file by adding the following:

```
zuul.routes.book.path=/movies/**
zuul.routes.book.url=http://localhost:9000
ribbon.eureka.enabled=false
server.port=8080
```

This tells Zuul, that whenever a request is received at localhost:8080/movies, Zuul will route this request to a microservices running at localhost:9000/movies.

This was a high-level overview of how Zuul can be used, Zuul can be configured in many ways. In a production environment Zuul can act as the gatekeeper to the rest of your microservices backend by handling authentication, etc.
Our blog contain a more detailed description of how to setup Zuul, and example respository is also provided: `http://rpi-cloud.com/apigatewaypattern/`

Spring Cloud Config

Changing configurations on running services can be cumbersome, especially if the application does not have a way to be configured remotely. Spring Cloud Config provides a way to make this easier by introducing a Config Server from which clients fetch their configuration. The figure below illustrates how Spring Cloud Config works.

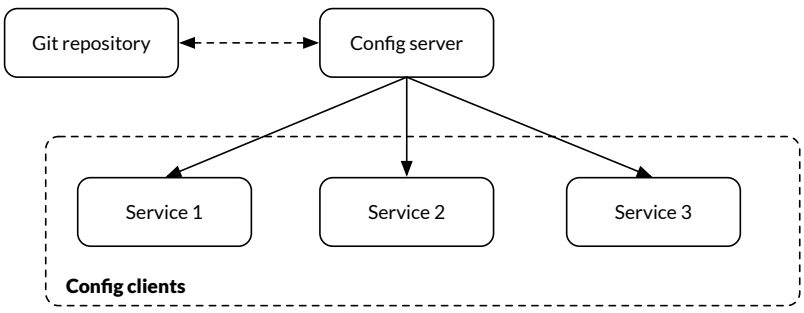


Figure E.3: Pattern: API Gateway

- The Git repository is used to store the configuration since Git is pretty good at tracking and storing changes.
- The Config server keeps itself up to date with the Git repository, and serves HTTP requests with configurations for the clients.
- The Config clients request the configuration from the config server, which sets the properties in the application.

Spring Cloud Config provides server and client-support for externalized configurations in distributed systems. An integrating this pattern into your applications is fairly simple.

Config Server

First we need to setup a Config server as a separate service. This can easily be done using the before mentioned Spring Initializr[5]. The Config server has to point to the Git repository where the configurations are stored. This is configured in the bootstrap.yml file as shown below:

```
spring:
  application:
    name: config-service
  cloud:
    config:
      server:
        git:
          uri: https://github.com/rpicloud/guide-cloud-config
server:
  port: 8888
```

Config Clients

The config clients has to be configured to point to the Config Server. This is easily done in the bootstrap.yml file of the client as follows:

```
spring:
  application:
    name: configclient
  cloud:
    config:
      uri: http://localhost:8888
```

This is all that is needed to setup the Spring Cloud Config pattern. This pattern allows us to have a central place to manage external properties of our applications across all environments.
A blog post explaining the setup in more detail can be found on our blog: `http://rpi-cloud.com/guide-spring-cloud-config/`.

[5] http://start.spring.io

Hystrix - Circuit Breaker

Hystrix is Netflix's implementation of the Circuit Breaker pattern. Hystrix makes it easy to apply timeouts, add the circuit breaker pattern with fallback methods, and further it provides a dashboard for monitoring these integration points as well. To enable Hystrix a few simple annotations is needed, @EnableCircuitBreaker, and @EnableHystrixDashboard if the dashboard is needed. This is shown below.

```
@SpringBootApplication
@EnableCircuitBreaker
@EnableHystrixDashboard
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }

}
```

Listing E.1: Enabling Hystrix

To implement a fallback method for when the circuit is open, i.e. the integrating service is down, a simple annotation is added to the method as shown below:

```
@HystrixCommand(fallbackMethod = "open", commandKey = "resources")
@RequestMapping(value = "/resources")
public ResponseEntity<List<Resource>> resources() {

    IService2 service2 = Feign.builder()
            .decoder(new JacksonDecoder())
            .target(IService2.class, "http://"+service2host+":"+service2port)
    ;

    List<Resource> resources = service2.resources();

    return new ResponseEntity<List<Resource>>(resources, HttpStatus.OK);
}

public ResponseEntity<List<Resource>> open(){
    return new ResponseEntity<>(state.getResources().subList(0, state.
    getAmount()), HttpStatus.PARTIAL_CONTENT);
}
```

Listing E.2: Implementing Hystrix

As shown in Listing E.2 the @HystrixCommand takes a fallback method as an argument. In the example we provide a method called open, to simply return an array of backup data which in this case is a list of strings.
In the code example shown in Listing E.2, Feign is used as a Rest Client builder. Feign is a declarative web service client that makes writing web service clients easier. Further it integrates easily with other Netflix tools, and also with already available components like the JacksonDecoder.

Experiment: Application Level and Infrastructure Level Resilience

This appendix will present additional results from the experiments, present the students' experiments, and cover how scripts were used during the experiments. The source code and results can be found in Attachment 5 and 6. All tests were performed in a KubeCloud consisting of four Raspberry Pis as seen in Figure 7.5.

Application Level Resilience Extra Results

In addition to the scenarios presented in Chapter 8 called *Integration points in a synchronous architecture*, three test runs of the setup without delay and circuit breakers were performed to act as a reference point. The measurements were performed to investigate the latencies and success rates at the 100 requests per second as in the experiment. Figure F.1 shows a normalized histogram of the three test runs.

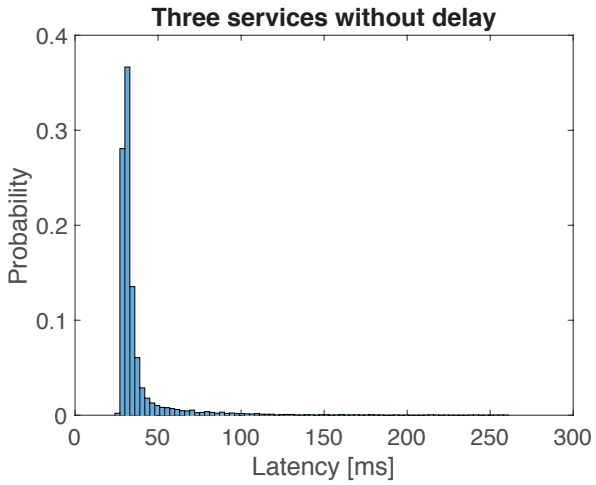


Figure F.1: Test Setup without Delay

All test runs had 100% success ratio and on average a response time of 36.6 ms.

The source code of three services is available on Github as: lab-service-0, lab-service-1, lab-service-2 on `https://github.com/rpicloud/`. Docker images are available on Docker Hub as: exp-circuitbreaker-service-0, exp-circuitbreaker-service-1, and exp-circuitbreaker-service-2 tagged with version *0.0.7*: `https://hub.docker.com/r/rpicloud/`.

Application Level Resilience Scripts

The configuration of the circuit breakers in the three services (Service 0, Service 1, Service 2) are different in the scenarios. Vegeta was used to perform HTTP requests towards Service 0. Scripts made the process simpler and replicable. The scripts expect the host and rate as parameters, and different configurations such as delays and timeouts are configured. When the configuration is set up, a folder for output is created, the attack starts, and attack reports are generated. The line below shows how to call the script.

```
sh two_circuit_breakers.sh --host=192.168.1.71 --rate=100
```

Listing F.1 shows how the configuration, test run, and report generation for the scenario with two circuit breakers. The scripts for the three circuit breaker scenarios in Chapter 8 follow the same structure. Vegeta attacks produce .bin files. Afterward, Vegeta creates reports from the .bin files. These reports have been used to output e.g. graphs and CSV files.

```
#!/bin/bash
for i in "$@"
do
case $i in
    -h=*|--host=*)
    HOST="${i#*=}"
    ;;
    -r=*|--rate=*)
    RATE="${i#*=}"
    ;;
    *)
            # unknown option
    ;;
esac
done
if [ -z "$HOST" ]
  then
    echo "Host and rate required."
  else
  HOST0=$HOST:9000
  HOST1=$HOST:9001
  HOST2=$HOST:9002
  DELAYHOST1=0
  DELAYHOST2=15000
  RESULTS=10
  DURATION=30s
  TIMEOUTHOST0=1000
  TIMEOUTHOST1=1000
  curl $HOST0/circuitbreaker/enabled/true
  curl $HOST0/circuitbreaker/timeout/${TIMEOUTHOST0}
  curl $HOST0/fallbackenabled/true
  curl $HOST0/resultset/$RESULTS
  curl $HOST1/circuitbreaker/enabled/true
  curl $HOST1/circuitbreaker/timeout/${TIMEOUTHOST1}
  curl $HOST1/fallbackenabled/true
  curl $HOST1/resultset/$RESULTS
  curl $HOST1/delay/$DELAYHOST1
  curl $HOST2/resultset/$RESULTS
  curl $HOST2/delay/$DELAYHOST2

  echo SETUP DONE - Circuitbreakers, fallbacks, $DELAYHOST2 ms delayhost2,
    $RESULTS results, $RATE rate, $DURATION duration
```

```
  echo STARTING TEST
  NOW=$(date "+%m-%d_%H-%M-%S")
  DIR=4_cb_3svc_delay_${NOW}_d${DELAYHOST2}_r${RESULTS}_h${HOST}_rate${RATE}
  mkdir $DIR
  echo "GET http://"$HOST0"/resources" | vegeta attack -duration=${DURATION}
    -rate=$RATE | tee $DIR/results.bin | vegeta report
  echo "New directory with output: " $DIR
  vegeta report -inputs=$DIR/results.bin -reporter=json > $DIR/metrics.json
  vegeta report -inputs=$DIR/results.bin -reporter=plot > $DIR/plot.html
  vegeta report -inputs=$DIR/results.bin -reporter=text > $DIR/text.txt
  vegeta report -inputs=$DIR/results.bin -reporter="hist[0,100ms,200ms,300ms]
    " > $DIR/hist.txt
  vegeta dump -inputs=$DIR/results.bin -dumper=json > $DIR/dump.json
  vegeta dump -inputs=$DIR/results.bin -dumper=csv > $DIR/dump.csv
fi
```

Listing F.1: Two Circuit Breakers

Application Level Resilience Student Experiments

The scripts made it easy to reproduce the experiments for the students. Table F.1 shows the results of the students' results of the *Integration points in a synchronous architecture* experiment. Tests were performed on a shared wireless router whereas our experiments were performed with an ethernet cable. By inspecting the results, it is seen that errors such as 'too many open files' occur. This probably caused extra errors and lower success rate in some of the scenarios. Group 5 had a success rate of 49.1% whereas Group 2's success rate was 100% with one circuit breaker. Concluding exact values from these results does not make sense due to the variation, but the success rate was improved using circuit breakers in all except one test case.

Success rate	No circuit breaker	One circuit breaker	Two circuit breakers
Group 2	6.8%	100%	100%
(Extra)	6.9%	100%	75.77%
Group 3	6.67%	70.17%	79.27%
(Extra)	-	-	95.67%
Group 4	11.73%	74.9%	100%
(Extra)	-	-	98.77%
Group 5	12.07%	49.1%	99.53%
(Extra)	-	%	61.71%
Average	8.83%	78.83%	92.72%

Table F.1: Comparison of Success Rates from Student Exercises

Infrastructure Level Resilience Extra Results

Docker images for ARM architecture can be found on Docker Hub: `https://hub.docker.com/r/rpicloud/greeting-rpi/`.

Effects of Replication

In experiment *Recovery time* presented in Chapter 8 Table 8.3 presented the success rates for rate 100 requests/second and 200 requests/second. Table F.2 and Table F.3 shows the measurement from each iteration that Table 8.3 is based on.

Sucess rate	Iteration 1	Iteration 2	Iteration 3	Average
replicas=1	73.93% (13,307/4,693)	73.93% (13,307/4,693)	76.09% (13,696/4,304)	**74.65% (13,437/4,563)**
replicas=2	99.89% (17,980/20)	99.86% (17,975/25)	99.87% (17,977/23)	**99.87% (17,977/23)**
replicas=5	99.99% (17,998/2)	99.97% (17,994/6)	99.98% (17,996/4)	**99.98% (17,996/4)**

Table F.2: Rate=100 (Total of 18,000 Requests)

Success rate	Iteration 1	Iteration 2	Iteration 3	Average
replicas=1	74.26% (26,735/9,265)	71.03% (25,569/10,431)	72,95% (26,262/9,738)	**72.75% (26,189/9,811)**
replicas=2	99.86% (35,949/51)	99.90% (35,963/37)	99.88% (35,958/42)	**99.88% (35,957/43)**
replicas=5	99.99% (35,996/4)	99.99% (35,998/2)	99.97% (35,988/12)	**99,98% (35,994/6)**

Table F.3: Rate=200 (total of 36,000 requests)

Infrastructure Level Resilience Scripts

The scripts make the test process of the experiments simple and replicable. This ensures that the same steps are always taken during the execution of the experiments. Listing F.2 shows an example of the test script for running the *The effects of replication* experiment. Running the script takes the host as a parameter and can easily be directed against other hosts or clusters.

```
#!/bin/bash

for i in "$@"
do
case $i in
    -h=*|--host=*)
    HOST="${i#*=}"
    ;;
    *)
            # unknown option
    ;;
esac
done


if [ -z "$HOST" ]
  then
    echo "One argument required. Examples:"
  else
    rates=(100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500
    1600 1700 1800 1900 2000)
    requests=10000
    duration=5s
    filename="replicas_1"

     echo
  echo
  echo SETUP DONE

  echo STARTING TEST

  NOW=$(date "+%y-%m-%d_%H-%M-%S")
  DIR=${filename}_${NOW}
  mkdir $DIR

  for j in $(seq 0 2); do
    echo ITERATION ${j}
    for i in $(seq 0 `expr ${#rates[@]} - 1`); do

        current_rate=${rates[$i]}
        current_duration=`echo "($requests + $current_rate -1)/$current_rate"
    | bc`

        echo
        echo Starting test with duration: $current_duration seconds and rate:
     $current_rate req/sec
        echo "GET http://"$HOST"" | vegeta attack -rate="$current_rate" -
    duration="${current_duration}s" | tee $DIR/rate_${rates[$i]}-${j}.bin |
    vegeta report
    done
    echo ITERATION ${j}: Done - sleeping 1 minute
    sleep 30s
  done
fi
```

Listing F.2: Replica Test Ramp-Up

Infrastructure Level Resilience Student Experiments

Because the scripts made it easy to reproduce the experiments, it was possible to use the tests as an exercise for the students. Table F.4 shows the results of the five groups' test results of the *Recovery time* experiment. Tests were performed on a shared wireless router whereas our experiments were performed with an ethernet cable. Most of the results are very close to our experiments, but the scenario with one replica at a request rate of 100 requests/second has a success rate of 58.65% whereas our success rate is 74.65%. By inspecting the results, it is seen that errors such as 'too many open files' occur. This probably caused extra errors leading to a lower success rate.

Success rate	1 rep. (100)	2 rep. (100)	5 rep. (100)	1 rep. (200)	2 rep. (200)	5 rep. (200)
Group 1	60.48 %	99.95%	99.69%	74.93%	99.97%	99.92%
Group 2	52.42%	100%	100%	73.97%	99.98%	100%
Group 3	55.68 %	71.76%	99.99%	73.66%	99.91%	99.98%
Group 4	54.97%	99.88%	99.88%	74.92%	99.92%	99.98%
Group 5	69.68%	99.96%	99.97%	70.71%	99.85%	99.97%
Average	58.65%	94.31%	99.91%	73.64%	99.93%	99.97%

Table F.4: Comparison of Success Rates from Student Exercises

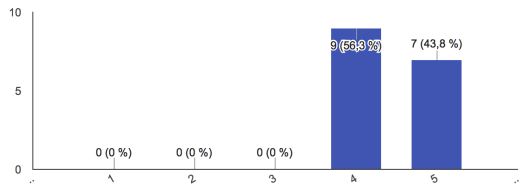
G

Experiment: Overall Evaluation of KubeCloud and the learning activity

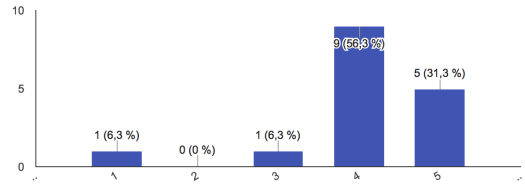
Evaluation of KubeCloud

The first part of the questionnaire consisted of questions about the cluster and the way it supported the students learning.

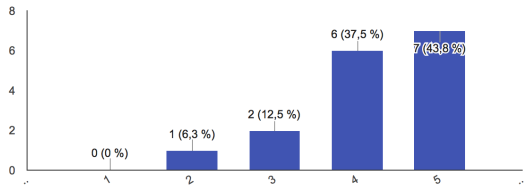
Physical appearance of the cluster



(a) The cluster provided me a better understanding of what a cloud consists of



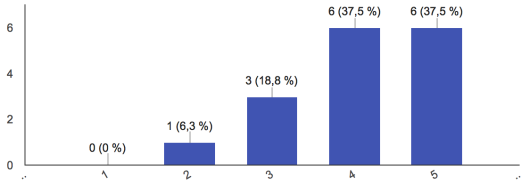
(b) The physicality of the cluster provided a better understanding of errors in distributed systems (e.g. pulling the network cable)



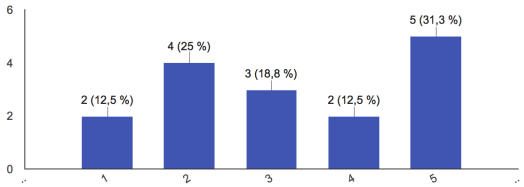
(c) The physical design of the cluster gave me associations to a real server rack

I don't think it is necessary to create the association with a real rack. The knowledge of the structure is enough.

Group activity



(a) The cluster allowed me to experiment

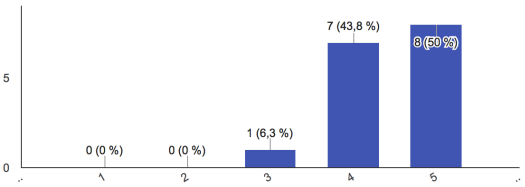


(b) The cluster initiated group discussions

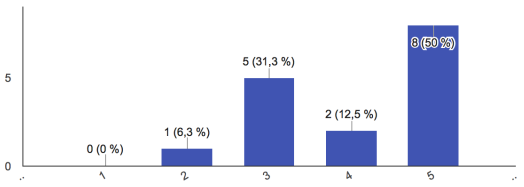
When we began the final setup of the project we got a better understanding of the cluster and could see the benefits of the cluster. This was not clear in the workshops.

The cluster was for a long time this "wierd thing" which we were mostly scared of breaking - which led to only doing things to it described in workshops. Only in the finishing hours of the course where we had to deploy and test our own application we dared to interact with the cluster in such a way that we gained the things above

Visualization



(a) The cluster combined with the visualizer helped me understand the concepts of the cluster

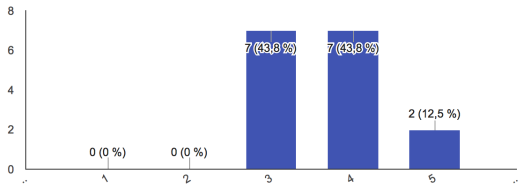


(b) The cluster combined with the visualizer helped me understand the how errors are handled

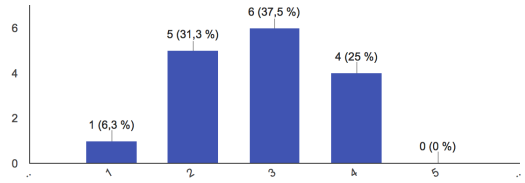
It's unfortunate that the visualizer was so bugged. It calls for a PR on the repository. ;)

The cluster + visualizer + WORKSHOP helped me understand these things. The resilience workshop where you could see how the node(s) went down and the corrosponding request failiures helped

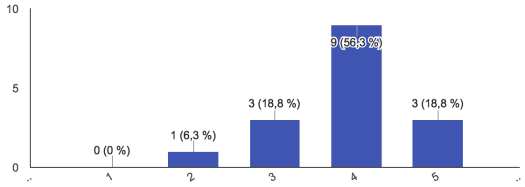
Motivation



(a) The use of a cluster increased my motivation



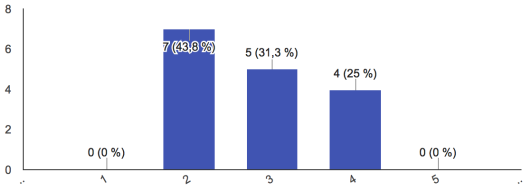
(b) The cluster motivated me to do out of curriculum experimentation



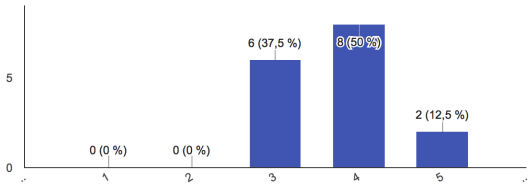
(c) The cluster was a fun and playful way of learning the concepts

The cluster seemed to be really nice in the beginning. But since learning that all the nice things about docker is getting shot to the ground with the cluster was demotivating (mostly with our web page docker build)
The cluster made the workshops and project more real compared to using a cloud cluster. Too bad the ARM architecture broke the seamless experience of using docker.
It's hard to say what could have been without a cluster. We were on day 1 introduced to the cluster and what it was, but not what the alternatives could have been if no cluster had been used. That said, the teachers understanding of the cluster was also crucial, due to the fact that such a small part of the course was error solving on the cluster, time could be destributed to learning instead.

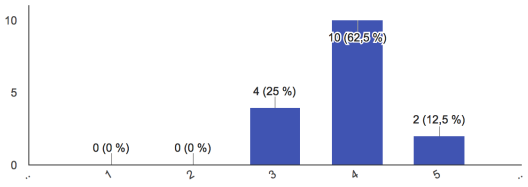
Relation to the real world



(a) Using a cloud provider (e.g. Google Container Engine) instead of a hands-on tool would have improved my learning



(b) The cluster represents a small-scale datacenter



(c) The cluster improved my skills in relevant technologies

I have never used Google Container Engine, so I don't know if it would have been better to use. There might be benefits and drawbacks which is unforseen.

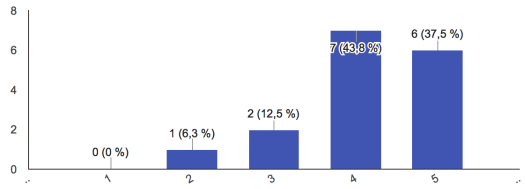
Cloud providers would have made the user of Docker more meaningful, but not the cluster understanding.

It is hard to seperate what was gained from the cluster alone, and what was gained from the course overall. The cluster only seemed like a mean to achive the goal, in the way that Docker and Kubernetes seemed like the driving forces, and not the cluster in this case.

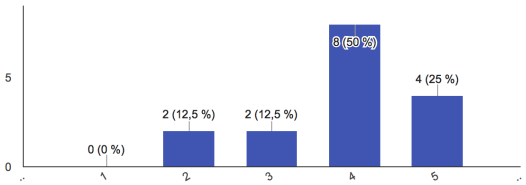
Evaluation of the learning activity

The second part of the questionnaire consisted of questions about the designed learning activity.

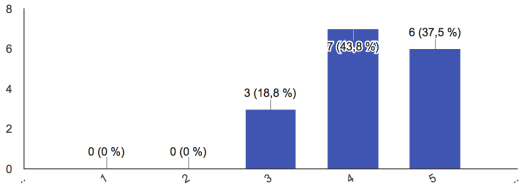
Course structure



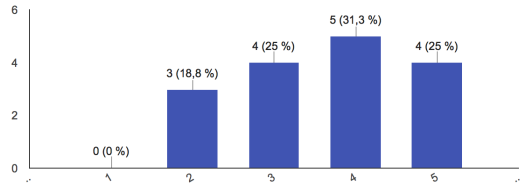
(a) The distribution of theory and practical work was suitable



(b) The order of the topics was well-planned



(c) The problem based project allowed me to apply the concepts and theories



(d) The workshops provided me with the necessary skills for the project work

A bit more theory, and maybe more in the start of the course

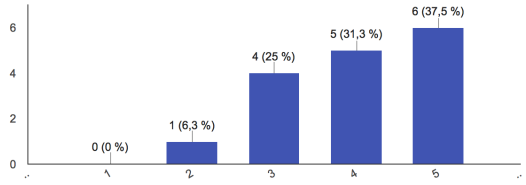
I think the continuous delivery lecture should be the last one since it wasn't directly relevant for our project but were still very interesting.

There has been some issues with the project. The workshops are very linear and specific for the concrete issue it wants to solve. When things differ (project work) it's hard to use the workshops for finding out what to change, and where things go wrong. Mostly because things are coupled so loose, makes it difficult without further explanation like "The name you type here must be the same as the one in the config".

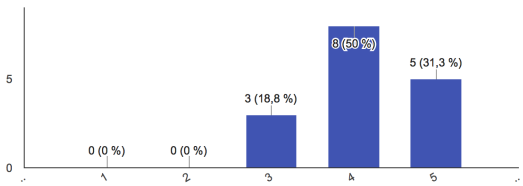
The workshops were often to simple or stepped through. The real understanding of the concepts arrived when we worked on the project.
If time allows, I think the workshops could be a little more free to experiments to increase the understanding of the concepts.

Alot of mixed questions to give answers to here. First of all I think the distribution of theory and practical work was good, and most of all needed, due to the massive work that had to be put into reaching the goal of a running system. Concerning the order of the topics, alot of the early topics was well planed, but the later topics felt like a rush (here thinking of DNS and Service Discovery). They fell so late in the course that there was no time to actually use these things in the project, and they therefore felt meaningless. Colaboration on this is the problem itself. Being near the end (now) it feels like the initial work with springboot and the REST endpoints felt like such a tiny part of the overall project, that you rather would have learned how to use the strong parts of Docker and Kubernetes. How to fit a suiteable problem to this would probably involve some sort of "finished" design that the groups "only" had to implement USING docker and kubernetes and such.

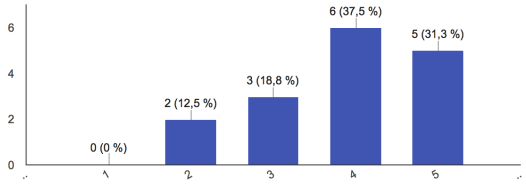
Materials



(a) The reading and video materials provided me with the essentials for each module



(b) The slides covered the necessary theory

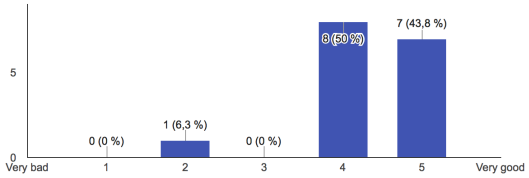


(c) The blog (rpi-cloud.com) helped me getting started with Spring Boot and Spring Cloud

The blog posts were sometimes to unexplained. It provides fine tutorials, but do not help in learning what the frameworks provide.

The slides were great and well thought through, which also relates to the fact that the theory lessons were great. The blog itself provided mostly "bonus" information to me, due to the coverage of the workshops which got me started. I do know that some of the other group participants used the blog a lot.

Comparison



(a) In comparison with other courses my overall rating of the course is

It's not the best, but it's far from the worst course!
Det var rart at lære noget der stadig er frisk
The course has been inspiring and very relevant with the global movements in computer engineering. Motivated presenters and well structured presentations are key to keeping the course interesting and motivational. Much of this could very well be mandatory on the IKT engineering studies as the distributed systems are growing and the standardized monolithic architectures are disappearing from the more modern and scalable applications.
I like the course, although i felt like the course was (is) very hard. There are many "new" concepts comming from a background NOT involving internet, ports, distributed systems, cloud architechture and such. It therefore felt like a HUGE load to get to the end, and learn all the topics due to "lacking" basic knowledge. This is not the course fault, but the education in itselv.

List of Figures

List of Tables

Attachments

The content of the DVD is denoted attachments. Each attachment is given an ID, and an overview is presented below.

1. Course slides
2. Workshops
3. Additional slides (external presentations)
4. Course Reading Materials
5. Experiment - Application level resilience
6. Experiment - Infrastructure level resilience
7. Experiment - Weekly evaluation
8. Experiment - Student experiments
9. Articles
10. Student reports

Several implementation examples can be found on our Github organization: `https://github.com/rpicloud`.

The department brought an article about the project: `http://eng.au.dk/aktuelt/nyheder/vis/artikel/aarhus-universitet-faar-sin-egen-sky/`.

Bibliography

[1] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: service-oriented architecture and web services*.

[2] i scoop.eu, "Customer loyalty: The (changing) reality of loyalty beyond retention." `http://www.i-scoop.eu/customer-loyalty/`, 2015.

[3] K. Schwab, "Are you ready for the technological revolution?." `https://www.weforum.org/agenda/2015/02/are-you-ready-for-the-technological-revolution/`, 2015.

[4] A. Mohamed, "A history of cloud computing." March 2009, 03 09. http://www.computerweekly.com/feature/A-history-of-cloud-computing - Revision at 05/02-2016.

[5] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, (Bordeaux, France), 2015.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[7] Q. Hardy, "Tech companies, new and old, clamor to entice cloud computing experts." `http://www.nytimes.com/2016/03/07/technology/tech-companies-new-and-old-clamor-to-entice-cloud-computing-experts.html?_r=0`, 2016.

[8] H. P. Breivold and I. Crnkovic, "Cloud computing education strategies," in *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE T)*, pp. 29–38, April 2014.

[9] Nginx, "The future of application development and delivery is now." `https://www.nginx.com/resources/library/app-dev-survey/`, 2016.

[10] M. Nygard, *Release it! : design and deploy production-ready software*. Raleigh, N.C: Pragmatic Bookshelf, 2007.

[11] M. Andreessen, "Why software is eating the world." `http://www.wsj.com/articles/SB10001424053111903480904576512250915629460`, 2011.

[12] Y. Jararweh, Z. Alshara, M. Jarrah, M. Kharbutli, and M. N. Alsaleh, "Teachcloud: a cloud computing educational toolkit," *International Journal of Cloud Computing 1*, vol. 2, no. 2-3, pp. 237–257, 2013.

[13] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, "The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures," in *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, July 2013.

[14] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkila, X. Wang, K. Hamily, and S. Bugoloni, "Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 2, pp. 170–175, Dec 2013.

[15] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. O'Brien, "Iridis-pi: a low-cost, compact demonstration cluster," *Cluster Computing*, vol. 17, no. 2, pp. 349–358, 2014.

[16] M. K. Potter and E. Kustra, "A primer on learning outcomes and the solo taxonomy." http:/www1.uwindsor.ca/ctl/system/files/PRIMER-on-Learning-Outcomes.pdf, 2012.

[17] D. Clark, "Bloom's taxonomy of learning domains." http://www.nwlink.com/~donclark/hrd/bloom.html, 2015.

[18] D. R. Krathwohl, "A revision of bloom's taxonomy: An overview," *Theory into practice*, vol. 41, no. 4, pp. 212–218, 2002.

[19] J. Biggs, *Evaluating the quality of learning : the SOLO taxonomy (structure of the observed learning outcome)*. New York: Academic Press, 1982.

[20] A. J. S, "Learning and teaching; solo taxonomy." http://www.learningandteaching.info/learning/solo.htm, 2013.

[21] R. M. Felder and N. Carolina, "K.: Learning and teaching styles in engineering education," *Engineering Education*, 1988.

[22] Teach.com, "*Teaching Methods*." http://teach.com/what/teachers-teach/teaching-methods, November 2015.

[23] A. Walker, *Essential readings in problem-based learning : exploring and extending the legacy of Howard S. Barrows*. West Lafayette, Indiana: Purdue University Press, 2015.

[24] C. E. Hmelo-Silver, "Problem-based learning: What and how do students learn?," *Educational Psychology Review*, vol. 16, no. 3, pp. 235–266, 2004.

[25] S. C. dos Santos, A. C. Monte, and A. Rodrigues, "A pbl approach to process management applied to software engineering education," in *2013 IEEE Frontiers in Education Conference (FIE)*, pp. 741–747, Oct 2013.

[26] D. Churchill, "Effective design principles for activity-based learning: The crucial role of'learning objects'in science and engineering education," *Paper presented at the Ngee Ann Polytechnic*, vol. 2, 2003.

[27] E. Fallon, S. Walsh, and T. Prendergast, "An activity-based approach to the learning and teaching of research methods: Measuring student engagement and learning," *Irish Journal of Academic Practice*, vol. 2, no. 1, p. 2, 2013.

[28] D. H. Jonassen and L. Rohrer-Murphy, "Activity theory as a framework for designing constructivist learning environments," *Educational Technology Research and Development*, vol. 47, no. 1, pp. 61–79.

[29] M. B. Postholm, "Cultural historical activity theory and dewey's idea-based social constructivism: Consequences for educational research," *Outlines. Critical Practice Studies*, vol. 10, no. 1, pp. 37–48, 2008.

[30] G. E. Hein, "Constructivist learning theory." http://beta.edtechpolicy.org/AAASGW/Session2/const_inquiry_paper.pdf, 1991.

[31] M. Nino and M. A. Evans, "Fostering 21st-century skills in constructivist engineering classrooms with digital game-based learning," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 10, pp. 143–149, Aug 2015.

[32] A. F. McKenna, J. Nocedal, R. Freeman, and S. H. Carr, "Introducing a constructivist approach to applying programming skills in engineering analysis," in *Proceedings Frontiers in Education 35th Annual Conference*, pp. T3H–20, Oct 2005.

[33] D. Churchill, "Towards a useful classification of learning objects," *Educational Technology Research and Development*, vol. 55, no. 5, pp. 479–497, 2006.

[34] D. Satterthwait, "Why are 'hands-on' science activities so effective for student learning?," *Teaching Science: The Journal of the Australian Science Teachers Association*, vol. 56, no. 2, 2010.

[35] B. Sosinsky, *Cloud computing bible*. Indianapolis, IN Chichester: Wiley John Wiley distributor, 2011.

[36] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," tech. rep., Gaithersburg, MD, United States, 2011.

[37] K. Subramanian, "Cloud computing's electricity metaphor has outlived its usefulness?." `https://www.cloudave.com/946/cloud-computings-electricity-metaphor-has-outlived-its-usefulness/`, 2010.

[38] Gartner, "Multitenancy." `http://www.gartner.com/it-glossary/multitenancy`, 2016.

[39] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, pp. 81–84, Sept 2014.

[40] C. Sanchez, "Everything at google runs in containers." `https://www.infoq.com/news/2014/06/everything-google-containers`, 2014.

[41] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pp. 342–346, March 2015.

[42] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, pp. 102–c3, May 2015.

[43] J. Turnbull, "Dockerbook sample." `https://www.dockerbook.com/TheDockerBook_sample.pdf`, 2016.

[44] A. Mouat, "Swarm v. fleet v. kubernetes v. mesos." `https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos`, 2015.

[45] Kubernetes, "Kubernetes.io." `http://kubernetes.io/`, 2016.

[46] J. Ellingwood, "An introduction to kubernetes." `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`, 2014.

[47] E. Raymond, *The art of Unix programming*. Boston: Addison-Wesley, 2004.

[48] J. Lewis and M. Fowler, "Microservices in a nutshell." `https://www.thoughtworks.com/insights/blog/microservices-nutshell`, 2014.

[49] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*, pp. 583–590, Sept 2015.

[50] A. Gupta, "Microservices, monoliths, and noops." `http://blog.arungupta.me/microservices-monoliths-noops/`, 2015.

[51] S. Newman, *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015.

[52] T. Mauro, "Microservices at netflix: Lessons for architectural design." https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015.

[53] J. Lewis and M. Fowler, "Microservices." http://martinfowler.com/articles/microservices.html, 2014.

[54] M. E. Conway, "How do committees invent?," *Datamation*, April 1968.

[55] J. Boner, *Reactive Microservices Architecture - Design Principles for Distributed Systems*. O'Reilly, 2016.

[56] C. Richardson, "Service discovery in a microservices architecture." https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/, 2015.

[57] P. Nommensen, "Service discovery in a microservices architecture." https://dzone.com/articles/service-discovery-in-a-microservices-architecture, 2015.

[58] B. Beyer, *Site reliability engineering : How Google runs production systems*. Sebastopol, CA: O'Reilly Media, 2016.

[59] M. Omer, *The resilience of networked infrastructure systems analysis and measurement*. Singapore Hackensack, N.J: World Scientific Pub. Co, 2013.

[60] R. R. for Survivability in IST, "Deliverable d9: First open workshop report." http://www.resist-noe.org/Publications/Deliverables/D9-1st_Open_Workshop_report.pdf, 2007.

[61] L. Strigini, "Fault tolerance and resilience: meanings, measures and assessment," in *Resilience Assessment and Evaluation of Computing Systems* (K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, eds.), Berlin, Germany: Springer, 2012.

[62] M. Monperrus, "Principles of antifragile software," *CoRR*, vol. abs/1404.3056, 2014.

[63] M. Bruneau and A. Reinhorn, "Exploring the concept of seismic resilience for acute care facilities," *Earthquake Spectra*, vol. 23, no. 1, pp. 41–62, 2007.

[64] J.-C. Laprie, *Predictably Dependable Computing Systems*, ch. Dependability — Its Attributes, Impairments and Means, pp. 3–18. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.

[65] A. Rotem-Gal-Oz, "Fallacies of distributed computing explained," *URL http://www. rgoarchitects. com/Files/fallacies. pdf*, vol. 20, 2006.

[66] M. Fowler, "Frequencyreducesdifficulty." http://martinfowler.com/bliki/FrequencyReducesDifficulty.html, 2011.

[67] G. Candea and A. Fox, "Crash-only software." https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf, 2003.

[68] C. Bennett and A. Tseitlin, "Chaos monkey released into the wild." http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2012.

[69] A. Basiri, L. Hochstein, A. Thosar, and C. Rosenthal, "Chaos engineering upgraded." http://techblog.netflix.com/2015/09/chaos-engineering-upgraded.html, 2015.

[70] searchsoftwarequality.techtarget.com, "Understanding performance, load and stress testing." http://searchsoftwarequality.techtarget.com/answer/Understanding-performance-load-and-stress-testing, 2016.

[71] E. Evans, *Domain-driven design : tackling complexity in the heart of software*. Boston: Addison-Wesley, 2004.

[72] M. Fowler, "Boundedcontext." `http://martinfowler.com/bliki/BoundedContext.html`, 2014.