TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ALEKSI HÄKLI

IMPLEMENTATION OF CONTINUOUS DELIVERY SYSTEMS

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Computer and Electrical Engineering
on 4 May 2016

ABSTRACT

ALEKSI HÄKLI: Implementation of Continuous Delivery Systems
Tampere University of Technology
Master of Science thesis, 46 pages, 2 Appendix pages
June 2016
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Kari Systä
Keywords: Continuous Integration, Continuous Delivery, Continuous Deployment

Continuous Integration, Delivery, and Deployment are subjects that have been on the table in the recent years. The books Continuous Integration by Duvall et. al (2007), and Continuous Delivery by Humble et. al (2010) are, however, the only extensive literature that has been published on the subject. In addition to the books there is information available based on miscellaneous conferences and scientific publishments, but this information is fairly scattered and hard to compile.

A lot of companies suffer from long delays between implementing and delivering software features. Multiple parties ranging from foreign developers to Finnish leader projects have researched the benefits that can be gained from automating and making software delivery continuous. Benefits include among other things rapid customer feedback, lowered production delivery costs and delays, and smaller number of errors in processes.

Companies do, however, have difficulties in implementing Continuous Integration, Delivery, and Deployment. This is mainly due to the arbitrarity of the work. There is information available on the subject, but collecting, studying, and distributing that information can be very costly.

This Master of Science thesis researches the implementation a Continuous Delivery system in a Finnish software company. The research work is done in an internal project, and the aim is to implement a generic software build, test, and delivery system. The process involves gathering business, technical, and user requirements, compiling a requirements definition, designing a phased project plan, and executing that plan to implement a Continuous Delivery system on top of the AWS cloud platform.

TIIVISTELMÄ

ALEKSI HÄKLI: Jatkuvien toimitusjärjestelmien toteutus
Tampereen teknillinen yliopisto
Diplomityö, 46 sivua, 2 liitesivua
Kesäkuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Prof. Kari Systä
Avainsanat: jatkuva integraatio, jatkuva toimitus, jatkuva tuotantoonvienti

Jatkuva integraation, toimitus ja tuotantoonvienti ovat viime vuosina paljon puhuttuja aiheita. Duvall et. alin kirja Continuous Integration (2007) sekä Humble et. alin kirja Continuous Delivery (2010) ovat kumminkin ainoat kattavat teokset, jotka on julkaistu aiheeseen liittyen. Näiden lisäksi aiheesta on saatavilla informaatiota erinäisten konferenssien ja tieteellisten julkaisujen muodossa, mutta tieto on melko hajanaista ja vaikeasti koottavaa.

Monet yritykset kärsivät pitkistä viiveistä ohjelmiston ominaisuuksien toteutuksen ja niiden asiakkaalle toimittamisen välillä. Monet tahot ulkomaisista kehittäjistä ja tutkijoista suomalaisiin kärkihankkeisiin ovat tutkineet etuja, joita saadaan toimituksen automaatiosta ja jatkuvaksi tekemisestä. Etuja ovat muun muassa nopea asiakaspalaute, alentuneet tuotantoonvientikustannukset ja -viiveet sekä vähentyneet virhemäärät prosesseissa.

Yritysten on kumminkin vielä vaikea toteuttaa jatkuvaa integraatiota, toimitusta ja tuotantoonvientiä, koska prosessien toteuttamiseen ei ole paljoakaan eväitä. Aiheesta on paljon hajanaista tietoa, mutta tämän tiedon kokoaminen, opiskelu ja välittäminen on erittäin kallista.

Tämä diplomityö tutkii ohjelmiston jatkuvan toimittamisen toteutusta suomalaisessa ohjelmistoyrityksessä. Tutkimustyö suoritetaan yrityksen sisäisessä projektissa, jonka tavoitteena on toteuttaa yleishyödyllinen ohjelmiston koonti-, testaus- ja toimitusjärjestelmä. Projektin aikana suoritetaan liiketoiminta-, teknologia- ja käyttäjätarpeiden selvitys. Näistä tarpeista kootaan vaatimusmäärittely, jonka pohjalta tehdään projektisuunnittelma sekä toteutus ohjelmiston jatkuvalle toimituspalvelulle AWS-pilvipalvelualustaa hyödyntäen.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

AMI	Amazon Machine Image
APT	Advanced Package Management Tool
AWS	Amazon Web Services
CapEx	Capital Expense
CD	Continuous Delivery
CI	Continuous Integration
CIDR	Classless Inter-Domain Routing
CM	Configuration Management
DB	Database
EC2	Elastic Compute Cloud (in AWS)
GoCD	Go Continuous Delivery
IaaS	Infrastructure-as-a-Service
IP, IPv4, IPv6	Internet Protocol, version 4 and 6
ISP	Internet Service Provider
LTS	Long-Term Support
OpEx	Operating Expense
OS	Operating System
PaaS	Platform-as-a-Service
RPM	RPM Package Manager
SaaS	Software-as-a-Service
SCM	Source Control Management
TCO	Total Cost of Ownership
TFS	Team Foundation Server
TUT	Tampere University of Technology
URI	Uniform Resource Identifier
URL	Uniform Resource Locator (special case of URI)
VCS	Version Control System (such as git or Subversion)
VPC	Virtual Private Cloud (in AWS)
YUM	Yellowdog Updater, Modified (package manager)

PREFACE

A modern software project is ideally developed, tested, and delivered to customers rapidly, keeping the time delta between the conception of a software feature and its activation by users as small as possible. Speed enables monetization and value delivery of features for companies, adjusting software functionality based on customer feedback for management, and increasing productivity for development teams. These all sum up to competitive edges.

This thesis discusses the research and development of an open-source, private cloud based Continuous Delivery system, which enables faster software delivery for Vincit, the Finnish software company, in the spring of 2016. This thesis is intended for people studying and implementing Continuous Integration, Deployment, and Delivery, and attempts to shed light to challenges and practicalities of software delivery.

I would like to thank Vincit for sponsoring the writing of this thesis, Tampere University of Technology and its staff for all the direction and knowledge that is available to Computer Engineering students, and all the individuals who have have helped in the course of this thesis project by generously sharing their experience, knowledge, and patience with me. I would like to especially thank professor Kari Systä for his insight and experience in directing of this thesis work.

I would also like to thank my family, especially my mother and father, for supporting and steering my ambitions in this life. The endeavour of this Master's degree and thesis would have been wholly another if it wasn't for you.

Always remember that an education is a privilege. A free education such as that which we have in Finland is a major one. We should all do our best to give back.

Little by little, one travels far. - John Ronald Reuel Tolkien

In Kuopio, Finland on May 24, 2016

Aleksi Häkli

1. INTRODUCTION

In 2015 a project at Vincit Oy, the Finnish software company, was started. The project's primary goal was to implement support for complex Continuous Delivery pipelines with multiple build, test, and deployment platforms. Our motivation was to make delivering software to our customers faster and less error-prone by the means of software system automation. Vincit kindly agreed to sponsor the writing of this Master of Science thesis as a part of the project in the hope of providing useful information and documentation to others undertaking such a task.

In this technically oriented thesis we largely focus our discussion on the implementation of a customized open-source Continuous Delivery system for a medium-sized software company. The different phases we look into include research, requirements definition, design, implementation, and refactoring of a whole Continuous Delivery software system on top of pure cloud infrastructure. This purely cloud based system architecture later mutates into a hybrid cloud architecture, and we briefly discuss transforming from pure cloud to hybrid cloud solutions.

1.1 Background

Contemporary software development teams and their customers might expect that their software can rapidly and automatically

1. be built and compiled to a deliverable format on source code changes;
2. be tested on unit, integration, and end-to-end levels;
3. be delivered to customer for evaluation;
4. be deployed to production at each and every stage of development, and;
5. be monitored in each of the aforementioned stages.

While this is indeed the state where most would like software development to be in, it is very rare for software projects to have mechanisms in place to support all of the aforementioned.

Many software development teams have a working *version control system* or *VCS* [1] such as *Git* [2] or *Subversion* [3] incorporated into their workflow. Many teams also have tests that test software functionality on unit, such as class or module level. These level of tests are called *unit tests* [4, 5]. Some teams have *integration tests* [6] implemented, which test how two or more units of software work together. For example, a generic sales company could benefit from testing how warehousing and sales systems might integrate with each other in isolated testing environments. Some modern companies have *end-to-end tests* [7] in use. End-to-end tests test how complete software systems function as whole when deployed into a staging or production environments, reducing the need for *manual testing* and reducing the amount of *regressions* that are introduced into systems over time in development processes. These are some types of tests that can be ran automatically to verify software functionality. [8, 9]

In practice many software projects are limited to a version control system containing the project source code and developers running an array of unit tests manually to test if software units function as intended with some input, and if the system seems to function as expected. Some contemporary projects might have an automated Continuous Integration server that periodically builds the software based on the current time or a trigger such as a VCS commit.

There are a number of books, articles, and blog posts describing how not testing software can cost time for the software development team and money for the customer. Not testing software units introduces logic errors; not testing software integrations introduces problems in building products from modular projects; not testing end-to-end functionality introduces problems in deploying the product when single deployments last for hours or days. These all lead to delays in receiving feedback on product features and functionality. [10]

Most companies would like the aforementioned build-test-deliver cycle to be automated, but lack the resources in personnel or infrastructure, or the will of implementing automation to the degree that developers could confidently say they can deliver a recent, working version of their software in, for example, one day's time period to production, or demonstrate the latest version of the software to a customer on the spot. There might not be enough knowledge about such systems to make it viable to implement them in-house. The costs might be too high, because the initial investments have not been made to set up baseline and infrastructure to train personnel. The technologies people are working on might not have any explicit support for automated testing built into them. Each and all of these reasons make it more difficult to set up Continuous Integration, Delivery, or Deployment for a project.

1.2 Scope of this Thesis

In this thesis we discuss what Continuous Integration, Delivery, and Deployment are and discuss and deepen our knowledge on Continuous Delivery. We research and gather requirements for a Continuous Delivery system. Further, we choose a way of implementing Continuous Delivery, and based on our choice we design a custom Continuous Delivery system architecture on top of the *Amazon Web Services* or *AWS* cloud [11], and implement a private cloud based Continuous Delivery system on top of open-source software called *Go Continuous Delivery* or *GoCD* [12].

This thesis acknowledges that there are multiple ways for implementing Continuous Delivery pipelines on top of *Software, Platform, and Infrastructure-as-a-Service* or *SaaS, PaaS, and IaaS* [13] solutions and platforms, as well as on-premise. These providers and solutions might fit your requirements or they might not. In this thesis, we have specific reasons for wanting to set up and administer a system ourselves. You might have differing needs, but we want this thesis to provide useful insight for you.

1.3 Timeline of the Project

In the initial kick-off meeting of the project we decided to split work into phases that were planning, research and surveys, prototyping and AWS setup, implementation, and refactoring. Design of the actual system would take place in between the planning and prototyping steps, and corrections to the design would be made along the way. Each step would interleave with the others, to make working in an agile manner possible, but time would be allocated for tasks separately according to our collective assessments of task difficulty and time requirements. Our project goal was to provide an useful suite of tools to our developers in the spring of 2016. The project timeline is visualized in figure 1.1.

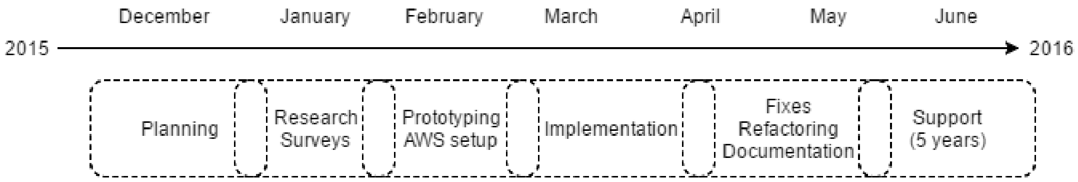


***Figure* 1.1** *Initial project timeline*

In the first planning and research phases we plotted the timeline and scope of the project and looked into different Continuous Integration, Delivery, and Deployment solutions. We also surveyed our developers and management for their requirements. After this we started prototyping the system on AWS to see if a cloud platform would be suitable for our use. After this prototyping we started the real implementation, and adjusted the solution based on feedback from developers and the problems we faced. After documenting the system we will educate users and introduce Continuous Delivery and the new tooling to our project teams and provide support for them.

2. WHAT IS CONTINUOUS DELIVERY?

What does everybody mean when speaking of different continuous practices in the software development world? How are Continuous Integration, Delivery, and Deployment related to each other, and can some exist without the others?

Continuous Integration was coined as a term and became a mainstream as a concept after the defining book *Continuous Integration* by Paul M. Duvall, Steve Matyas and Andrew Glover was published in 2007 [6]. Continuous Delivery is a more recent concept defined by Jeff Humble and David Farley in their book *Continuous Delivery*, published in 2010 [8]. Continuous Deployment has not received a titled book as of yet.

Continuous Integration means that building and unit testing software is automatic, reproducible and frequent. Frequency means that software is built on periodically or, for example, on every version control commit. The important thing to note about Continuous Integration is that the testing process, the so called software integration, is automated with tools that do not require manual intervention in building and unit and integration testing of software. [6]

Continuous Delivery means making sure that the software is always deployable. Software is built and tested as in Continuous Integration, and also deployed into testing environments for further testing. The main thing is that build versions are automatically proven to be deployable. [8]

Continuous Deployment includes always automatically deploying software to production when it is committed to version control branches corresponding production environments and qualified by the automatic tests to be production-ready. The build, test, and deployment pipeline is not touched manually after version control commits are made. [14]

The difference between Continuous Delivery and Deployment is that in Continuous

Delivery software is proven deployable and production deployments are *manually* triggered. In Continuous Deployment software is *automatically* deployed to production. Continuous Delivery and Deployment are often used in the same context and can be mistaken with each other. [14]

We can also visualize the relations and differences of Continuous Integration, Delivery, and Deployment.

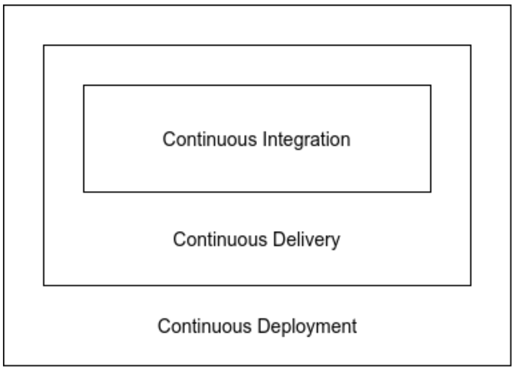


Figure 2.1 *Relations of Continuous Integration, Delivery and Deployment*

The relations of the different terms can be seen as subsets and supersets of each other as in figure 2.1. One cannot implement Continuous Deployment without first implementing functional Continuous Integration and Delivery systems.

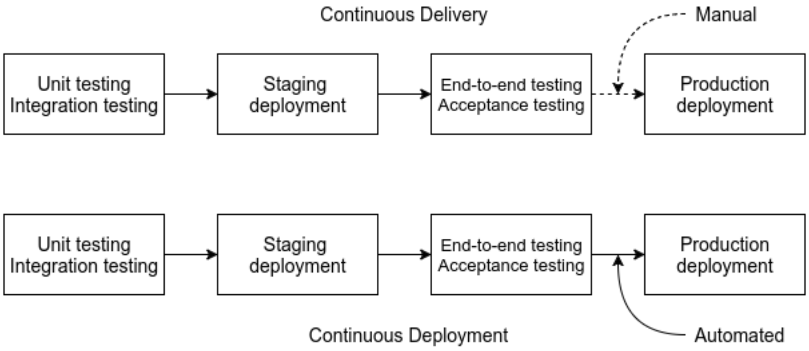


Figure 2.2 *Difference between Continuous Delivery and Continuous Deployment*

A visualization of the difference between Continuous Delivery and Deployment is represented in figure 2.2. Yassal Sundman defined the differences of Continuous Delivery and Deployment in her 2013 blog post [15]. Carl Caum from *Puppet* [16] later offered an expert view on the difference between Continuous Delivery and Deployment in a 2013 blog post [14]. Both definitions were based on the material of Jezz Humble, who in turn referred to Continuous Deployment first in 2010 [17], making a statement about Timothy Fitz' blog post from 2009 [18]. From this chain of online material we can date the Continuous Delivery and Deployment concepts to be at least 7 years old, although they have been used well before that as well [19, 20].

2.1 Terminology

For the sake of clarity, in this thesis we define different Continuous terms in the following way:

- *Continuous Integration* is a group of practices that aims to improve software development quality and speed with build and test automation in order to improve reproducibility and to remove the chance of errors from manual build step execution or environment state mutation in build processes. Continuous Integration is a subset of Continuous Delivery;
- *Continuous Delivery* is a group of practices that includes Continuous Integration and adds to them the automated end-to-end testing and delivery of software in such a way that software builds are stateless, reproducible, and proven deployable across target environments and platforms. Continuous Delivery makes delivering recent software iterations to production at any given time feasible and aims at guaranteeing deployability. Continuous Delivery is a superset of Continuous Integration, and a subset of Continuous Deployment. Continuous Delivery does not include automatically deploying software to production environments;
- *Continuous Deployment* is a group of practices that includes the aforementioned Continuous Integration and Delivery practices but adds to them the practice of automatically deploying software to production environments. Continuous Deployment ideally removes the need of manual production environment updates and aims to roll-forward only deployments. Continuous Deployment is a superset of Continuous Delivery.

We hope to avoid confusion with these definitions, which are based on contemporary literature [6, 8, 14, 15, 17].

3. WHY IS CONTINUOUS DELIVERY IMPORTANT?

The main point of Continuous Delivery is to deliver software to the end-user more repeatably, reliably, and predictably. This makes delivery faster, lowers costs, and reduces risk involved in deployments. Receiving feedback is faster and the amount of errors and wrong directions taken in development is reduced. [8, p. 17-22]

Continuous Delivery practices can greatly save time and decrease costs in projects. Development teams can be more certain that software has been tested and ready for production environment when there are successfully tested and deployed builds rolling from the build pipeline into staging environments. Managers and sales personnel can happily tell the customer that software is ready to be deployed when the customer asks to see the latest sprint result in action. Customers do not have to wait for a week or a month of deployment delays. [8]

Continuous Delivery can be implemented in stages, supporting processes where the need for automation and orchestration is greatest, or just implementing the parts which offer the most returns for invested money and time. The degrees of implementing automation have been defined in the Continuous Delivery Maturity Model which offers guidelines for what to implement in what order to be more efficient [21, 22, 23].

3.1 Adaptation of Continuous Delivery

Continuous Delivery search hits and adoption have been growing in popularity for the last few years [19, 24]. Since Humble and Farley published the self-titled book in 2010, which offered a firm basis for discussion and concrete implementation of Continuous Delivery practices [8], many company executives have begun understanding the need to shorten software delivery times and deltas. The Agile manifesto has been promoting shorter develop-build-test-deploy cycle times for a long time, but concrete steps aside from talking about process agility have been sparse in companies [25].

A research paper studying the degrees of Continuous Deployment implementation in companies in 2015 by Tampere University of Technology and Aalto University researchers in Finland was recently published [24] as part of the Need 4 Speed project [26], shedding light into how successful companies have been in adopting automation into their software release flows. The research team found that teams have overall enjoyed great success in the implementation of Continuous Delivery to a certain degree, but are yet to implement Continuous Deployment as part of their repertoire due to Continuous Deployment being a fairly new thing and customers not yet adopting the ideology. Amazon is one of the few public examples to employ Continuous Deployment as part of their workflow for scalability reasons [27].

3.2 Advantages of Continuous Delivery

For many years the process of delivering the software product to the customer has been a rather tedious process. Building software, testing it, and making a deliverable can require very specialized knowledge. Information is not required only for a specific software technologies and domain problems. Build machinery, target platforms, and end-user affect the process as well. All of the aforementioned combined make software delivery and deployment

- *time consuming and expensive*, because when software is built manually, the build processes usually take time and require someone to watch over them. Builds for web projects usually take 15 minutes to complete, while builds for large native C projects can take multiple hours;
- *error prone*, because when software is built manually, there is room for human error where build and configuration steps are repeated by hand, and;
- *mutating and evolving* because software requirements and properties and even target platforms constantly change.

All of the aforementioned make software delivery slow and expensive. In other words, either companies, customers, or both will end up paying if software is built and delivered manually. The costs also cumulate over long times when the number of builds grows and expertise gathers into knowledge silos between teams and personnel. [8]

In addition to the disadvantages of not using Continuous Delivery, reported benefits of Continuous Delivery are, among other things, shorter time-to-market, feedback benefits in feature steering, reliability of releases, quality and customer satisfaction improvements, and improved efficiency [28].

Large companies have researched and implemented Continuous Integration, Delivery and Deployment as part of their IT strategies for a multitude of different reasons [10, 22, 23, 24, 27, 28]. The only question that remains for us is *when and in what order should we start implementing Continuous Delivery as part of our processes*?

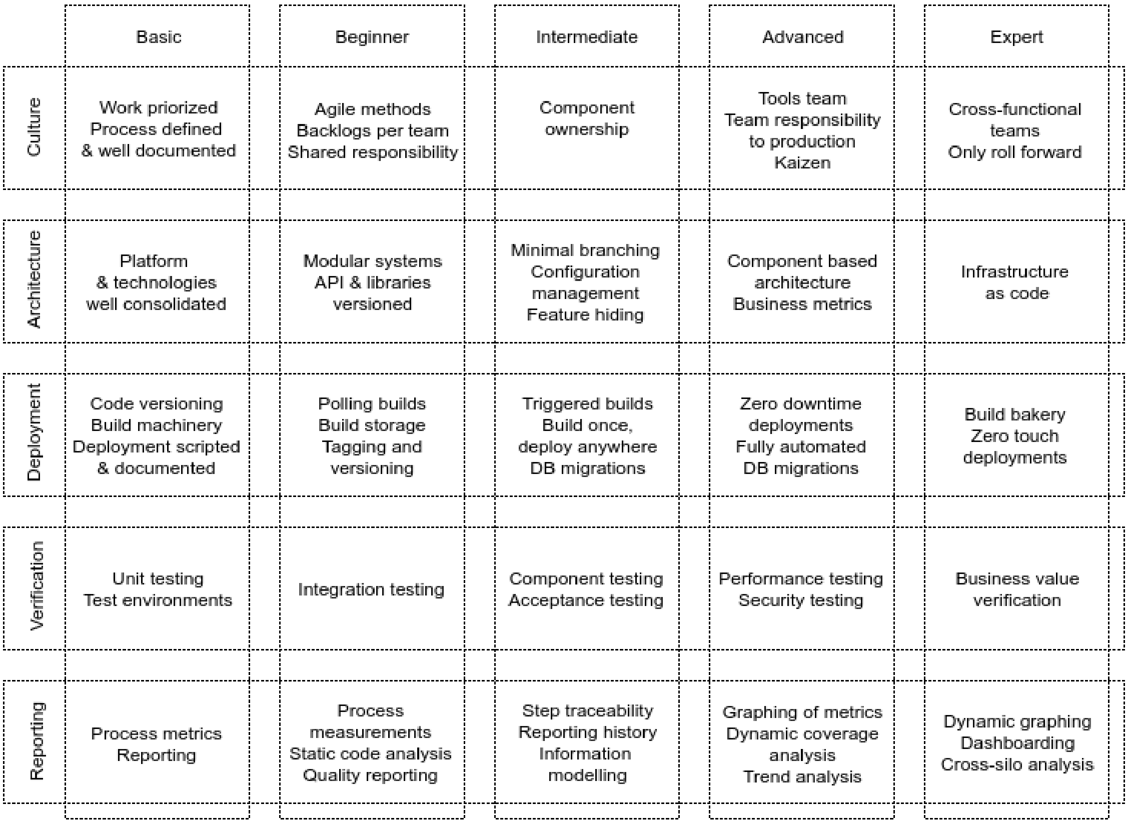
3.3 The Continuous Delivery Maturity Model

Continuous practices enveloping integration, delivery and deployment practices have gotten a serious look in the recent years when smaller and larger companies have retrospected their software development and delivery practices. In between 2013 and 2015 a number of papers emerged describing the benefits of defining a framework and a model for implementing Continuous Delivery.

IBM has published papers on Continuous Delivery maturity assessment in 2013 and 2014 proposing a basis for so called *maturity models* [22, 23]. The notable Agile software company ThoughtWorks commissioned a technical report from Forrester Consulting in 2013 to find out the current state of software automation in companies [29]. Rehn et. al described a simplified, common-sense approach to Continuous Delivery in their InfoQ post in 2013 [21]. Research has since taken place to explore the state of Continuous Delivery and Deployment in companies [24, 28].

IBM has mainly analyzed the effect of having a framework in place for keeping different parties of the development and deployment cycle synchronized and on a reasonable delta in terms of competence and sophistication to mitigate effects of some area of the software delivery chain falling behind or being too much ahead of others to reap balanced benefits. This seems to be advantageous to a large software company. The technical reports advice that a framework be established on the delivery time of software. [23]

ThoughtWorks has found that many business executives, departments and software delivery companies have different perspectives on how fast a software order-delivery cycle should be. Most (51%) surveyed executives in 2013 expected their ordered software to be delivery and deployment ready in less than 6 months time. Most IT executives had differing views. The study proposes that a model be established to evaluate how fast software can be delivered in working condition. [29]



	Basic	Beginner	Intermediate	Advanced	Expert
Culture	Work priorized Process defined & well documented	Agile methods Backlogs per team Shared responsibility	Component ownership	Tools team Team responsibility to production Kaizen	Cross-functional teams Only roll forward
Architecture	Platform & technologies well consolidated	Modular systems API & libraries versioned	Minimal branching Configuration management Feature hiding	Component based architecture Business metrics	Infrastructure as code
Deployment	Code versioning Build machinery Deployment scripted & documented	Polling builds Build storage Tagging and versioning	Triggered builds Build once, deploy anywhere DB migrations	Zero downtime deployments Fully automated DB migrations	Build bakery Zero touch deployments
Verification	Unit testing Test environments	Integration testing	Component testing Acceptance testing	Performance testing Security testing	Business value verification
Reporting	Process metrics Reporting	Process measurements Static code analysis Quality reporting	Step traceability Reporting history Information modelling	Graphing of metrics Dynamic coverage analysis Trend analysis	Dynamic graphing Dashboarding Cross-silo analysis

Figure 3.1 *Continuous Delivery Maturity Model*

The InfoQ article highlights the IBM findings and go over different aspects of software projects, proposing that each aspect and part of a development organization such as *culture* and *verification*, should be split into different levels, more specifically *Basic, Beginner, Intermediate, Advanced*, and *Expert*. These different facets of software projects and organizations should be developed in respect to each other, balancing the amount of automation and Continuous Delivery methodology that is introduced into software development and delivery. Figure 3.1 illustrates the model InfoQ proposes, which is largely in concert with the suggestions IBM makes. [21, 22, 23].

3.4 Discussing the Continuous Delivery Maturity Model

Large companies practice software business at a large scale where implementing the correct processes at the correct times can be vital. When software projects gain traction they start requiring support from DevOps and business processes. Business operations can similarly start requiring support from software development processes when scaling. Prime examples of scaling businesses are ecommerce systems, SaaS platforms, and game development.

Maturity models for Continuous Delivery offer a ready-thought and research-backed framework for implementing Continuous Delivery practices for software development in stages. They aim to make lives of people deciding on the implementation of continuous processes simple, and are intended to offer a roadmap for executives, management, and developmen teams implementing continuous practices themselves. Their purpose is to guide teams to the right path.

At the same time it is important to remember, that the processes are implemented in large companies at scale, and it may not be the most effective option for a small or medium sized company to implement the same things. Sometimes companies should not enforce continuous methodologies at all, at least on the scale Maturity Models would like to illustrate. In each and every situation it is necessary assess the possible benefits of continuous practices in a company and analyze what is the problem that is being solved without looking blindly at the model. There does not seem to be considerable amounts of validation data available on the maturity models as of yet, so you should use the best judgement available when utilizing them.

4. HOW CAN CONTINUOUS DELIVERY BE IMPLEMENTED?

There are many tools for Continuous Integration and Delivery written in multiple languages. Java is a much used runtime powering services such as Hudson Continuous Integration [30] (an important project, but presently largely deprecated by its alternatively licenced fork Jenkins), Jenkins [31], and Go Continuous Delivery [12]. There are also alternatives for Java powered platforms such as Buildbot [32] written in Python, Travis [33] written in Ruby, and Strider [34] and Drone [35] written in Node.js. These are just few examples of the tools available for implementing Continuous Integration. Many of the listed alternatives are available as partly or fully open-source software.

Equally many architectural models exist for Continuous Integration and Delivery systems. Systems range from simple examples running bash or Python scripts to multi-tiered enterprise solutions that can be hosted in multiple data centres. For example, the simplest of build systems can be implemented in hours on top of Buildbot on a single computer, just requiring a Python runtime on any operating system. Some platforms such as Travis or Snap CI [36] require an enterprise licence and a multiple machine set up just to operate on-premise. Sometimes enterprise services such as Team Foundation Server [37] are needed to orchestrate building and deployment of complex enterprise scale C# solutions and sometimes simple shell scripts are enough to build and test whole software products.

A noteworthy thing about Continuous Delivery is that it does not necessarily require any tools that are specifically manufactured for Continuous Delivery. Platforms such as Buildbot can be perfectly viable for implementing Continuous Delivery for a software product, but the set up of the pipeline from development to production server deployments with configuration management and source code builds can be more difficult to master and scale. Some specifically tailored software platforms can be much easier because they support the scenarios that you can run into when setting up Continuous Delivery out of the box.

This said, developers have a myriad of options for adopting Continuous Delivery into their work and project flows. Some of the obvious options are

- using a free or licenced hosted SaaS solution such as Travis CI or Snap CI;
- buying an enterprise solution with support and hosting it on-premises, or;
- hosting an open-source solution, on-premise or in the cloud.

Each of the aforementioned options can be a valid one, depending on the current and future situation and conditions in the company. Smaller companies should prefer to use lightweight solutions and avoid over-committing to one path unless there is a clear need for a heavyweight system. Larger enterprises might need multiple different systems to support their operations. In each case, an understanding of the different alternatives and their service and cost models is necessary in making the right choice.

4.1 Analyzing the Different Paths

Service-driven SaaS solutions are offered by multiple different vendors. The idea of a SaaS solution is that you *buy the right to use a service*, most often a centrally hosted one [38]. Most SaaS solutions are pre-configured for most common technologies and offer good basic functionality for Continuous Integration in terms of build machinery and unit test support. Some modern SaaS software offers support for very complex Continuous Integration and Delivery pipelines. A good example is a Snap CI that offers both simple and complex build pipeline support for a rather reasonable price and also bundles in enterprise support plans [36]. Another contenders are the widely used Travis CI and Drone.io, both open-source alternatives that offer software that has integrations with the most widely used cloud-based version control providers such as GitHub [39] and Bitbucket [40]. SaaS solutions might be a good option for those who are developing open-source services, which have free hosting, or customer projects where maintaining your own build machinery is not feasible in terms of time or money spent. Many users find that SaaS services offer adequate capabilities in proportion to the price paid and are happy with them.

Enterprise licenced SaaS solutions are good for those who wish to buy a solution they can host on their own premises or, for example, cloud infrastructure, but wish to have enterprise support or exclusive access. Good examples are developers of large software products such as Microsoft or Oracle, who in fact develop the aforementioned Team Foundation Server and Hudson platforms, and use them to run parts of their own integration platforms. Different options exist for this: proprietary and non-proprietary, even open-source solutions with company backing. For example, Travis CI is primarily based on SaaS business model, but is partially open-source and can be hosted on your own private infrastructure as an enterprise licenced version. Enterprise licenced solutions can be good for companies that have a large infrastructure and project portfolio and wish to simply buy a solution that has the required features to build and test their software without the extra need for maintenance. Sometimes customization is a problem and the degree of vendor lock-in can be considerable in enterprise solutions.

Open-source solutions are an option for those who are willing to invest into the development, maintenance, and support of their own Continuous Service portfolio and have the necessary resources to upgrade and improve their systems continuously. This might require a team of some sort, working on the build, integration and delivery machinery part or full time. Usage of open-source solutions might be harder than of their commercial brethren, and documentation and training can be costly. Often a need for in-house support comes bundled in with open-source solutions, as one cannot simply install open-source software platforms and expect them to run unmaintained in perpetuity. Open-source solutions might be good for companies who need diverse platform capabilities and offer customizability at the price of expertise.

4.2 Researching Continuous Delivery Systems

There are a number of factors that should be looked into when making a decisions regarding Continuous Delivery tooling. Not all platforms offer enough features to make them feasible to use and support in the long run. Some might lack present or future support for a specific framework or language. Some might be overly complex to host, develop, and customize.

Management should, of course, be asked to specify their requirements as clearly as possible. Developer requirements should be researched. A good way to find out

Table 4.1 *Open-Source Continuous Integration and Delivery tools*

Software	Implementation	Published	Maintainer	Licence
Buildbot	Python	2003	Mitchell et. al	GPL 2.0
Go	Java	2007	ThoughtWorks, Inc.	Apache 2.0
Jenkins	Java	2011	Kawaguchi et. al	MIT
Travis CI	Ruby	2011	Travis CI, GmbH	MIT
Strider CD	Ruby	2012	Radchenko et. al	MIT
GitLab CI	Ruby	2012	GitLab, Inc.	Open-source
Drone	Go	2014	drone.io	Apache 2.0

technological metrics is to survey the developers. Such a survey can reveal language and tool usage metrics that would otherwise be hidden information and very useful in implementing systems.

At Vincit we first started researching different options and models for Continuous Delivery systems by looking into the open-source systems that other companies were using and exploring technological, architectural, and cost models of such systems. At the research phase we would take a look at existing Continuous Integration, Delivery, and Deployment platforms and hosted solutions from the following perspectives:

- Maturity and age: Is the platform stable? Will it exist in 5 years' time. Has it showed signs of evolution in its lifespan?
- Implementation technology: What language is the platform implemented in? Can it be expanded easily by us?
- Architecture: Does the architecture make it possible to host the solution ourselves? Does it scale vertically and horizontally?
- Licence: Is the platform licence permissive? Is it truly open-source, permissive, and modifiable?

We gathered the most prominent open-source Continuous Integration and Delivery systems into table 4.1. Out of these tools, we saw the earliest project, Go Continuous Delivery, which was started in 2007 as a project named Cruise, to be the most prominent choice that we would like to further explore. We had previously used Jenkins, and have a lot of Java expertise in-house.

4.3 Discovering Fan-in and Fan-out

We discovered the concept of fan-in and fan-out on build tool level when we researched the GoCD tool. Fan-in means that a *component can depend on* multiple upstream components. Fan-out means that a *component can be a dependency* for multiple downstream components in a build chain.

Fan-in and fan-out are relevant concepts in choosing tools when implementing Continuous Delivery and their meaning is illustrated in figure 4.1. In the diagram we see that a product version is dependant on UI and server versions, and that a product build provides a dependency for testing and the integration environment. In other words, a whole product will be built and tested as a whole when there is change anywhere in the dependency chain. Fanning in and out are important when one is abstracting a build pipeline and thinking of builds as streams of interdependent changes. Some complex projects need support for graph-like dependency models on the build tool level to correctly reflect software composition on the architectural level. Otherwise build tool configuration and management might not be feasible. [41]

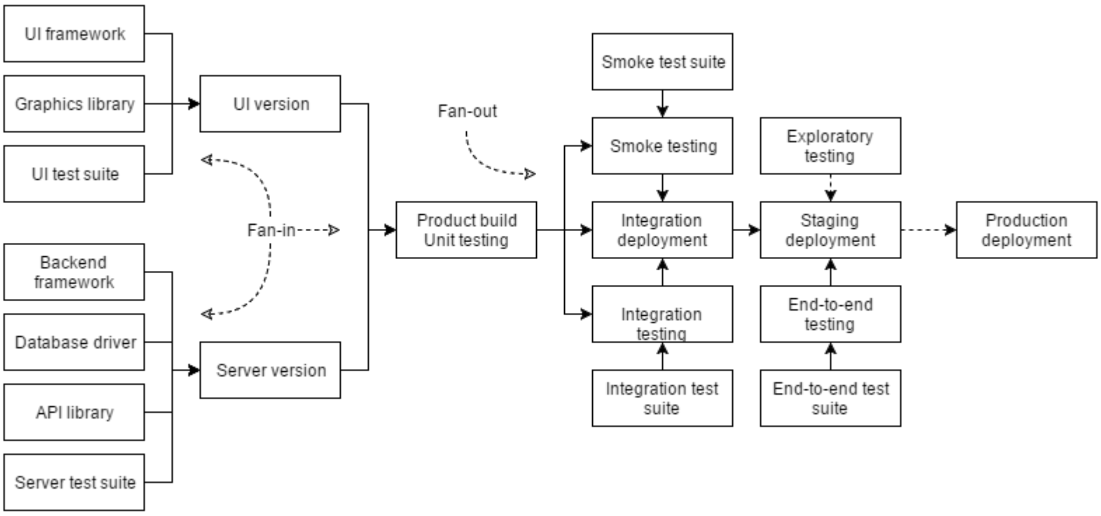


Figure 4.1 *Fan-in and Fan-out in build tools*

Fan-in and fan-out are rare features in Continuous Integration, Delivery, and Deployment tools, and we did not find them in any other tool than GoCD. For example, the very popular Jenkins doesn't support fan-in and fan-out dependencies, which we require in our complex projects, although they might be possible to support with plugins.

4.4 Cost Models for Continuous Delivery Options

Different kind of Continuous Delivery solutions have different cost models, which are essential knowledge when making management decisions regarding the implementation and lifespan of Continuous Delivery systems. Terms such as operating expenses and capital expenses become relevant [42, ch. 5]. In short, capital expenses are multiple-term costs that are tied to a system for a long time, such as data centre build and system vendor acquisition costs, network infrastructure acquisition and initial large licence purchases. Operating expenses are single-term costs that are tied to running the system at a certain load, for example network transfer and electricity costs and manual maintenance labor. Using an operation-ready SaaS platform has a fairly simple cost model: you can pay for what you use. Implementing a platform yourself with on-premise hardware or cloud hardware can have very different cost models which can include servers acquisition, management, power, cooling, backups, maintenance, licences, and an assortment of other things, which can be very hard to predict.

If you are acquiring a platform licence to, for example, Travis CI, Circle CI or Snap CD, the platform cost model is straightforward. You pay for a licence and get a certain amount of build capacity. The costs are predictable if project sizes and capacity needs are predictable.

If you are building your own platform and using the cloud, the expenses become more complicated to predict. Most cloud service providers bill you for networking, CPU, RAM, and storage capacity. For example, Amazon Web Services bills you for network components such as VPN connectivity and outbound traffic (in respect to the Amazon Web Services cloud platform), server usage and storage capacity for servers and storage units. In addition to this, if you are using proprietary operating systems, you will have to pay the operating systems licences, such as Windows or Red Hat Enterprise Linux, licences on per-machine basis. If you are using a proprietary software solution, you will most probably be required to pay for a usage licence to your Continuous Delivery tool. [43]

If you are building your own platform on-premise, you will, in addition to the cloud platform components, have to pay for power, cooling, and staff work. You will also have to have resources for handling with power outages, loss of data, et cetera. These different expenses can be found in detail from multiple data centre design and administration books and should be looked into if deciding to host a solution on-premise.

Making a difference between operating and capital expenses is vital. Capital expenses mean up-front investments to servers, racks, power supplies, networking equipment and the like. These investments have to be made before a system can even be built, and are tied to the system at once. Operating expenses on the other hand are exemplified by power, networking costs, and virtual server prices that are accumulated from operating the system over time.

From these concepts we arrive to fixed and floating costs. Fixed costs are baseline costs that are tied to the running of the system and rarely change unless scaling production: data centers and equipment are examples of fixed costs. Floating costs, on the other hand, are costs that change in the lifespan of the system.

The ability and willingness to pay large fixed costs and make purchases up-front affects the choice of service and hosting model. If a company can predict capacity needs in detail and has liquidity, then an upfront investment can be wise. If, on the other hand, capacity need evolve and can change drastically, it might be wise to build a Continuous Delivery system with more operating than capital expenses. This way, capital is not tied to fixed investments, and risks are reduced.

5. REQUIREMENTS SPECIFICATION

Defining and prioritizing requirements for a Continuous Delivery system is much the same as requirements definition in any IT project. The exception is that end users are software development professionals, which are a homogeneous group. This is the case with Vincit with all developers being experienced with a multitude of software systems.

After understanding the client or software target group, the requirements specification process typically involves three main phases [44].

The first step is typically the gathering of *business requirements*. This usually entails asking the right questions and finding out what the business needs and outcome goals are for the project. Are we trying to solve a new problem or improve some existing solution? Are we trying to create a new product or system? Is the system we will be working on purely for internal use or offered to external audiences? How much time and money do we have to use?

After determining what is the high-level task that we are trying to perform, we can start working on *user requirements* and finding out what the end-users are trying to achieve with such a system. Is the Continuous Delivery system used for a single, repetitive task? Are we performing a lot of differing tasks? What kind of integrations do we need to provide for users? From where and how must the system be usable?

From business and user requirements we can transition to defining *functional* and *non-functional* requirements for the system and gathering *constraints* that will be set to the system. What programming languages and platforms do we need to support? How many concurrent builds will we be executing? Where do we need to store build results?

5.1 Gathering Requirements

An important part of the design and development of a software system is finding out what our end-users would like in such a system. Our end users are our developers and software engineers. Hence, we were motivated to find out what exactly a group consisting of our own employees with a very specific demography was expecting from such a system. We designed a survey and ran it for all our personnel to provide information about our current way of developing software.

In structuring surveys it is important to take great care in designing the questions and deciding what kind of data the survey is intended to provide. The target audience and the timing of the survey are equally as important as the questions and the format it is conducted in. [45]

At Vincit we firstly decided that anyone involved in software projects was eligible to answer the survey. This was because we wanted the largest possible coverage from the results. We have a very flat organizational hierarchy where everyone is involved in everything in our software projects. A developer can be the lead developer, the mobile programmer, and the release engineer, if he or she is the best person to handle the job.

We also decided that we would like the survey to be an online one. This was easiest to organize with our large target audience, and would give our developers the opportunity to answer the questions when they felt comfortable doing so.

5.1.1 Surveying the the End-Users

We designed a survey that focused on the following topics to find out our end-user requirements:

- developer profile;
- technological profile;
- internal service usage, and;
- current and future needs.

These topics were discussed beforehand and based on our prior questionnaires that we conduct on about a yearly basis. Their purpose was to provide information on our current state of developer's personal position in projects, their technological responsibilities, utilization of tooling, and the needs from our internal support teams offering DevOps [46] and IT services.

In the *developer profile* section we wanted to find out what our developers are doing in our software projects. Are they just programming? Are they designing the software architecture? Are they designing and possibly implementing user interfaces? And how many of those developers are currently testing their software, delivering it to the customer and possibly deploying it?

In the *technological profile* section of the questionnaire our motivation was to find out what languages technologies our developers are using and in what proportions. Are they programming Java, Python, or PHP? What operating systems are still in use? This is important in deciding which technologies will be supported first and which will receive official support down the road. Do we need Ruby build infrastructure and package management if only one of our developers is currently programming Ruby?

In the *service profile* portion we wanted to find out how are developers currently using the services we offer to them. Do they utilize Dokku infrastructure for deployments, deploy into AWS, or build their staging and production environments? It is relevant to know from what is being used from where and which tools should be able to interact with each other.

In the *current and future needs* portion of the survey we focused on finding out what developers were currently missing and would like to be implemented.

Analyzing the different sections of the survey we wanted to implement, we quickly saw, that a fairly simple tool supporting *scalar*, *text*, and *multiple selection* answering options would be suitable. Since at Vincit we use Google Apps for Work [47], we took a look at Google Forms. It had the features we required from a survey:

- support for all our question types;
- authentication for users;
- access control for people who can answer with only single answer per person,

and;

- easy export for survey answers and built-in visualization tools.

We decided to use Google Forms for implementing our survey as it would offer an easy-to-use survey that could be easily sent to our whole group of employees and be answered online, when developers had time to take a look at it. This enabled very lean survey implementation without the need to introduce any new processes or tools for gathering data.

We sent the survey out to 178 people and got back 21 responses in two weeks' time period. Hence our sample size for the survey was 178 and our answering rate was 11.8%. Some of the answer highlights are illustrated in Appendix A in the hope that they will be useful to our readers.

5.2 Compiling the Requirements Specification

When we started gathering our technical and non-technical requirements we separated our findings into different relevance classes as *business, user, and functional and non-functional* requirements as described before.

Our initial business requirement for the project was to implement a scalable system that could offer additional value to our developers and customers. From business point of view we are not hugely interested in the technical details, but instead on the end result of the project: it has to be competitive in comparison to other similar systems, it must be scalable and support, at the very least, our most represented technologies in respect to our company's order base.

In respect to our user requirements we wanted to implement support for our most used technologies and tools first. Agile software companies, our company included, are very technology driven. A lot of competitiveness stems from having the right expertise and tooling for the technologies customers wish to invest into. The technological and functional requirements largely span from the user or developer profiles we wish to support.

This meant that based on our technology survey, we wanted to support Java versions 7 and 8 and select Node.js versions, namely the current *long term support* or *LTS*

Table 5.1 *Initial technology support requirements*

Technology	Importance	Phase of implementation
Ubuntu 14.04	High	Beta
Ubuntu 16.04	High	First revision
Mac OS X	High	First revision
Windows 10	Low	Future revisions
Java	High	Beta
Node.js	High	Beta
Python	Medium	First revision
Clojure	Medium	First revision
Objective-C	Medium	Future revisions
Swift	Medium	Future revisions
C/C++	Medium	Future revisions
Selenium	High	First revision
iOS / XCode	Medium	Future revisions
Android SDK	Medium	Future revisions

and the more recent development versions. In addition to programming languages we wanted to support their ecosystems that include build tools such as Ant, Maven and Gradle for Java and npm, bower, grunt and gulp for Node.js. These became our initial web platform targets.

Mobile technologies such as Objective-C and Swift for the iOS and Java for the Android are largely represented in the survey, and they would be supported after prototyping the system with web technologies.

Vincit has a large customerships in the so called native programming side which include platform specific binary software that is programmed in C family languages, primarily C++. Native projects are usually developed for a specific platform such as embedded Linux and tightly coupled to the static and dynamic libraries that are either operating system specific or distributed with the software. These are very specific projects, and most of our current customers have their own Continuous Integration platforms implemented to support them. They did not become our first priority.

We also identified that a lot of our developers are using Python and Clojure for software development, but projects in those languages are much less common than Java, JavaScript, Android, and iOS projects. They are, however, fairly easy support, and would be implemented in the first revisions of the Continuous Delivery system.

As the nature of the project was largely exploratory and we did not want to set up a requirements definition that was too rigid at this point, we wished to stick to a lean and simple list of things that we needed (primary requirements), that we would need (future requirements), and that would be nice to have. This way of defining requirements is quite useful if one is not writing a comprehensive definition, as it can usually be fit into a single whiteboard or paper sheet, creating a rather compact representation of the project needs. Hence we gathered our initial support requirements into table 5.1.

We decided to implement support for some large stand-outs representing our web technologies in the first beta phase, namely Java and Node.js support. After testing the system with those technologies, we would implement support for other technologies into our first revision, and later on add support for mobile platform tooling. This would allow us to implement the Continuous Delivery system in reasonably sized chunks without spending too much time working on support for technologies that we might never use in case the system wasn't up to the task on some facet.

6. CHOOSING THE RIGHT TECHNOLOGY FOR CONTINUOUS DELIVERY

In the beginning of the project we wanted to research existing solutions and see how those supported the different functionalities we required. If those systems would fit our way of doing things we could leverage them as the whole Continuous Delivery solution, part of the solution, or maybe as an architectural example for our own solution.

6.1 Deciding on Cloud Platforms versus Self-Hosted Solutions

One important factor in choosing the right alternative for us was the solution's extensibility. If in 5 years time we need a feature that is not implemented, what would we do? Taking a look at the solutions at hand, a lot of systems do not offer extensibility. Travis CI offers access to its deployment tools, but a lot of platforms offer no access to their inner workings or source code, and can't be modified at all.

We already knew some requirements for the software system we wanted to implement at this stage. Important factors were that the platform was extensible so that we could alter its behaviour or implement features ourselves. Another one of the required features was also the ability to support cross-platform builds. We wanted the same tool to be usable on Mac OS X, Windows, and Linux environments. An important requirement was also that we could host the Continuous Delivery service ourselves in the place we wanted to. Having someone else host the service was simply too rigid of an option. Our customers have a need for flexibility, so we wish to offer them as many options as possible.

Hence we decided that we wanted to invest in an open-source solution that we could extend and program ourselves, and hopefully host ourselves, if needed. In the open-source front there are a few options that have a community around them offering support and toolsets to each other. Narrowing the search down, we found ourselves facing yet another decision: choosing the right open-source option for our company.

6.2 Choosing the Right Open-Source Alternative for Vincit

One of our requirements that was born during the project was the ability to support and specify fan-in and fan-out dependencies for build steps and different projects [41]. Complex dependency management is important when building, for example, a microservice architecture or complex multi-tier software where one wants to define the build, test and delivery chain as a graph. For example, first build the backend and frontend software such as Java server and JavaScript UI, then test their units and statically analyze or lint [48] their code bases, then test component integration, and finally test the system's end-to-end functionality as described in chapter 4.

Considering the different requirements regarding tooling support for multiple platforms, languages and tools we decided to look further into options that offered script based and non-opinionated architectures. The most prominent of these systems was GoCD. GoCD is a Continuous Delivery tool written in Java and backed by a company named ThoughtWorks.

GoCD has most of the things we wanted our tool to have. It is:

- open-source and has a permissive licensing;
- platform agnostic and runs anywhere where Java is supported;
- non-opinionated and runs anything you can script to run via system shell;
- scalable, both horizontally and vertically, and lastly;
- has a stable user community and good documentation.

All these factors combined, the only issue we had with the project was its lack of an established plugin ecosystem, such as the one in Jenkins. Jenkins CI has a myriad of different extensions and supports most common tools because of its age and community. GoCD is, at the time of writing, in middle of implementation of some very central features such as dynamic build agent provisioning. Small delays, however, are things that we are willing to deal with when investing into long-term tooling.

7. DESIGNING A CONTINUOUS DELIVERY SYSTEM

System design is an important part of each and every software project. Because the project was both multi-platform and had multiple different technologies associated to it, we decided that the system would have its initial architecture designed and reviewed by all stakeholders before starting implementation.

Because at Vincit we do not like to invest heavily in something we have not yet prototyped, we anticipated that it would be wise to set up a lightweight cloud prototype before committing to a large scale implementation or purchasing any fixed resources such as servers or test machines to our office. This way we would be paying periodically for testing the project setup instead of investing heavily in something that we didn't have any experience in. Due to the author's previous experience in the AWS cloud we decided that we would implement and host the initial version of the system in AWS EC2, and evaluate how the system performed on that platform.

AWS has a lot of very specific terminology and building blocks that are well out of the scope of this master's thesis, but we will briefly discuss the architectural and technological terminology that is relevant. Further, up-to-date documentation can always be found in the AWS documentation portal, and should be referred to when reading this thesis due to the possibility of it having outdated information. This is due to the fast evolution of the platform. [49]

7.1 Network Architecture

We firstly decided that the system should be in its own private network segment, but still available in terms of network addressing to our office network, offering loose but usable coupling between our current and the new infrastructure. Network design is an important factor that defines if it is even possible to connect private

Table 7.1 *Network segments available in an AWS VPC*

Network segment address	Prefix length	Netmask	Addresses
10.0.0.0	/8	255.0.0.0	16777214
172.16.0.0	/12	255.240.0.0	1048574
192.168.0.0	/16	255.255.0.0	65534

Table 7.2 *Network latencies to AWS service centers from Kuopio, Finland*

AWS datacenter location	Network latency
US-East (Virginia)	150 ms
US-West (California)	215 ms
US-West (Oregon)	210 ms
Europe (Ireland)	80 ms
Europe (Frankfurt)	65 ms
Asia Pacific (Singapore)	395 ms
Asia Pacific (Sydney)	350 ms
Asia Pacific (Japan)	330 ms
South America (Brazil)	300 ms

networks to each other transparently. AWS offers top level network segment called the *Virtual Private Cloud* or *VPC* that is configured as a private subnetwork in the standardized, private IPv4 CIDR blocks as defined in IETF RFC 1918 [50]. VPCs can thus be internally configured to the IPv4 blocks illustrated in table 7.1. It is worth noting that AWS does not support IPv6 at the time of writing [51].

Looking at our office network which contains addresses in blocks 10.170.0.0/16 through 10.176.0.0/16, we decided that we would freeze our AWS network range into the 10.177.0.0/16 block. This would allow us to freely connect our AWS private cloud network segment to our office network by simply configuring a Virtual Private Network or VPN tunnel between the two network and establishing IP routes.

After deciding that we would be setting up a remotely connected network segment, we quickly realized that we should try and locate our private cloud as close to our offices as possible. This namely meant that we would be hosting our Continuous Delivery servers in Europe. The nearest locations to our offices in Tampere and Helsinki, Finland are AWS data centres in Dublin, Ireland and Frankfurt, Germany. Out of these two locations Frankfurt is the closest one to us, and thus offers the smallest networks latencies. We further tested this theory by pinging different AWS data centres, and got the average responses illustrated in table 7.2 back, rounded to the nearest 5 millisecond interval. The ping testing was done from Kuopio, Finland.

We decided that we would like to pursue a high-availability network topology at some point. High-availability schemas require that networks are designed for fault tolerance. Even if one segment of the network becomes unavailable, services are still available to users. Luckily, AWS makes this rather easy. When defining VPCs in an AWS regions, VPCs being regional services, one can define a VPCs subnetworks to be placed into different *Availability Zones* or *AZs*. An AZ is a physically separated segment of a network that is hosted in a different physical building with dedicated power supply that is guaranteed by AWS to be unaffected by networking, power and cooling problems in different availability zones. A good example would be a power outage or a fire. In case of such an event happened in one of the data centers, our Continuous Delivery machines would still be available to our users.

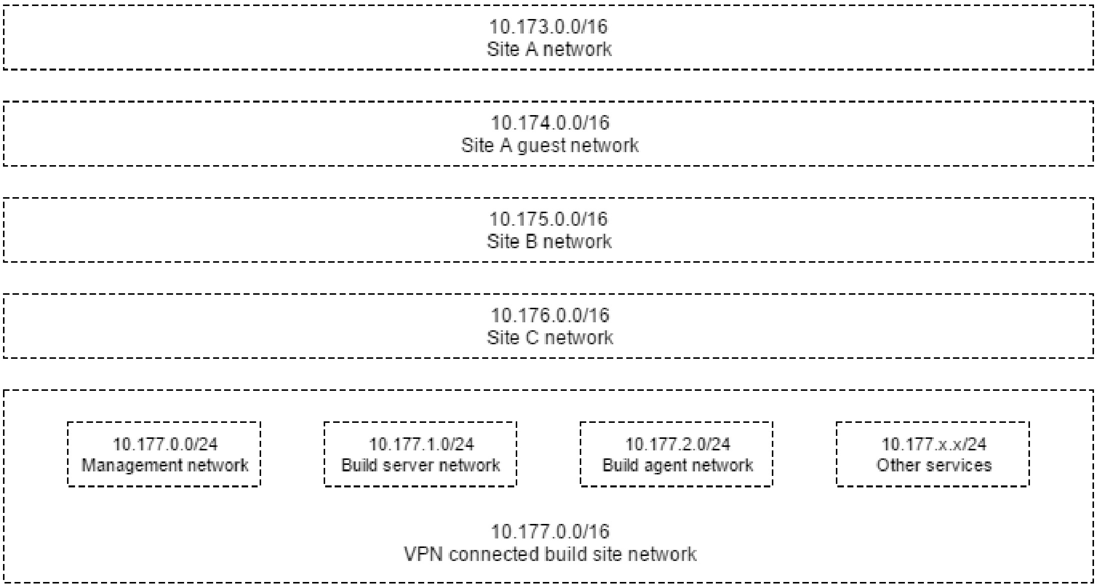


Figure 7.1 *Network design for Go build agents and servers*

In the end, we ended up on further splitting the VPC up to /24 sized subnets that can each contain 254 hosts (two addresses in AWS subnetworks are reserved, one for subnetwork gateway and one for subnetwork broadcasting). These subnets would separately contain our network management nodes and worker nodes. We would configure the topology to multiple AZs in production phase to avoid downtime from AZ outages. Our current AWS network layout is illustrated in figure 7.1.

7.2 Software Architecture

System wise, a large scale networked computer system consists of many machines that must be manageable on node and network level. System nodes consist mostly of their software setup in virtualized environments. They have an operating system and software which interacts with the network. Virtualized computers must be provisioned, have their security updates promptly applied, et cetera.

We started the designing of our system topology in the previous chapter by firstly deciding on the network topology. It is also important to decide on the operating system, its monitoring and management, and the interaction mechanisms with individual computers from the perspective of system administration. Provisioning and managing servers manually one by one is error prone, and automation is recommended where feasible to implement.

7.2.1 Operating System

Since we wish to use open-source software as much as possible due to its customizability and transparency on the tool level, we immediately were interested in the prospect of hosting the whole system on top of Linux operating system. We already host large parts of our infrastructure on top of Ubuntu Linux, and the AWS cloud offers, among other choices, Red Hat Enterprise, Ubuntu, Debian, and Amazon (RHEL derivative) Linux virtual machine images, specifically *Amazon Machine Images* or *AMIs* in AWS terminology. We briefly compared *RPM* and *DEB* based distributions, since their main difference is the packaging mechanism. RPM based distributions are based on the Linux kernel and *RPM Package Manager* as well as Yum which offers dependency based RPM package management with central repositories. Debian is based on Linux kernel as well, but uses *dpkg* and *Advanced Package Manager* or *APT* and its derivatives to manage system packages.

We had a discussion on which distribution would be the best option for us, and since we are already running Ubuntu *Long-Term Support* or *LTS* version on our infrastructure, we decided to prototype the system on Ubuntu 14.04 LTS version, and to upgrade to Ubuntu 16.04 LTS rolling out in May 2016, before our production phase launch.

Table 7.3 *Software properties for select IT orchestration and automation tools*

Tool	Appeared	Language	Syntax	Architecture	Licence
Ansible	2012	Python	YAML, Python	Push	GPL 3.0
Chef	2009	Ruby	Ruby	Pull	Apache 2.0
Fabric	2008	Python	Python	Push	Open-source
Puppet	2005	Ruby	Ruby	Pull	Apache 2.0
Salt	2011	Python	YAML, Python	Pull	Apache 2.0

7.2.2 Orchestration and Monitoring Tools

It is important to have IT orchestration and automation as well as system monitoring tools available in case the system grows and needs to scale horizontally, which was a basic requirement for us all the way from the start. Managing multiple computer systems by hand is simply too error prone and starts accumulating technical debt rapidly when the amount of connected nodes grows. We therefore decided to incorporate automation and monitoring tools into our AWS tool stack from the beginning.

Taking a look into different tools there exist many options for system automation. The brief contenders at the moment are Puppet [16] and Chef [52] written in Ruby, and Ansible [53] and Salt [54] written in Python. All tools offer the same basic functionality of programmable environment and large-scale infrastructure management, but differ in architecture in terms of machine communication and remote command execution policies. Since we were evaluating management tools at the time, we took a look at all the options. The different tools we researched are summarized in table 7.3.

In the end we ended up trying out Salt management tool since it is open-source, has a pull based command execution architecture, and is a lighter option to heavyweight competitors that require heavy investment in assorted tools before making use. Management of inventories requires extra components to be installed for Chef, Puppet and Ansible, but Salt has computer inventories, file servers, secure connections and high performance all built in. It is also smaller in size, and is very customizable both via included YAML based command execution with Salt Recipes (platform specific term for state description, which is called Recipe in Chef as well, and Playbook in Ansible) and via Python modules.

Monitoring tools offer a few options for system monitoring as well. Nagios [55] is a well established option with a decent sized user base. Sensu [56] is an API compatible modern option for Nagios. We didn't need to consider monitoring options, since we were happily using Nagios and Sensu in our infrastructure, and didn't feel a need for additional tools. We eneded up using Sensu for monitoring the whole system.

The system architecture regarding the management nodes and worker nodes began to look like a decoupled, remotely manageable environment, where components can be ported across different operating systems and platforms. GoCD, Salt, and Sensu are all platform agnostic and can be run on Windows, Mac OS X, and a variety of Linux distributions. We have a kernel and operating system tools, on top of which we install GoCD, Salt, and Sensu. As a whole and have a working build node, as illustrated in figure 7.2.

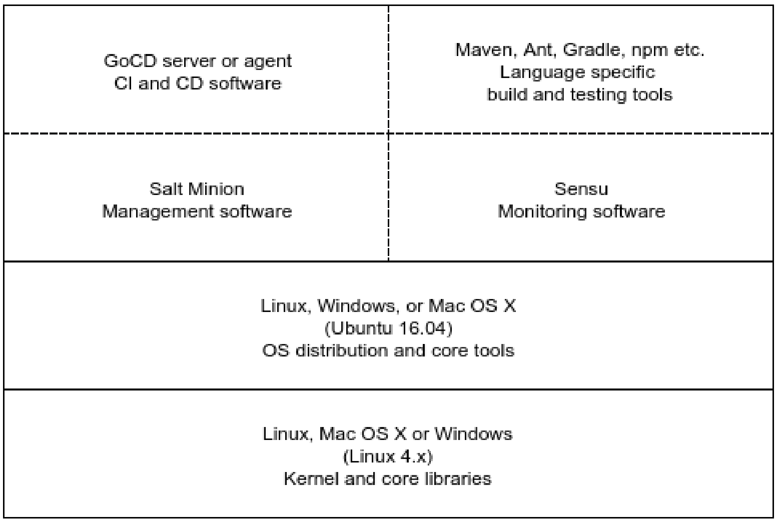


Figure 7.2 *Software design for Go build agents and servers*

7.3 System Cost and Scalability

When we take a look at the system costs based on the current designs regarding the networking and worker design, we can calculate some of the cost factors on a monthly and yearly basis for the system. This is useful to do before implementing the first version of the system because we can save ourselves from unwanted surprises in terms of high operating expenses. Some system architectures are not very suitable for running in the cloud and can be rather expensive due to large storage or data transfer costs.

Since we already know that we are implementing the system in the AWS *Elastic Compute Cloud* or *EC2*, we know that we need a specific network setup in the cloud. We need some management nodes, a central GoCD server, some worker nodes and a basic networking setup that can be connected to a private network if needed.

If we built a small-scale system with, for example, a single management server, a single central GoCD server and some worker nodes, we could try and specify the following computing requirements for our components.

- CPU: Since we are running management and computation tasks on the network, we will wish to have at least dual core CPU virtual machines to avoid hangups introduced by running multiple heavy tasks on a single core.
- RAM: Amazon EC2 offers the smallest possible amount for dual core computers at 4GB. We will initially try this size and resize if necessary: changing virtual machine instance size simply requires stopping and restarting a computer, so we can increase memory later.
- Storage: Amazon EC2 offers storage from a minimum of 8GB per virtual computer. Since we are running a fairly large Linux server instance on each node, including kernels and storage needs, we will wish to start at a minimum of 16GB of storage, which is a decent amount of storage for a contemporary Linux worker node. For the GoCD server, we will wish to have a decent sized storage disk locally for storing build results and artifacts. We will start with a 128GB disk. Disks can be migrated to larger volumes, so we can increase disk sizes later if needed.
- Networking: A VPC network segment with internet access and publicly addressable IPs for at the very least the management server and the GoCD server.

We arrive to the overall requirements specified in table 7.4

Cost-wise, if we decided to initially run, for example, 4 servers, we would have 6 Amazon EC2 nodes of *t2.medium* size. We would in addition to this have 208GB of Elastic Block Storage or EBS storage capacity. This would, at the time of the writing, sum up to the costs illustrated in table 7.5 without any discounts [57].

Table 7.4 *Computing capacities for modelled AWS EC2 system nodes*

Purpose	EC2 instance size	CPU cores	RAM	Storage
GoCD server	t2.medium	2	4GB	128GB
GoCD worker	t2.medium	2	4GB	16GB
Maintenance server	t2.medium	2	4GB	16GB

Table 7.5 *AWS capacity pricing in Frankfurt*

Component	Service	Item	Units	Unit price	Monthly price
Computing	EC2	t2.medium	4320h	$0.06	$259.20
Storage	EBS	GP2 SSD	208GB	$0.149	$30.99
Networking	VPC	VPN GW	720h	$0.052	$36.00
Networking	VPC	NAT GW	720h	$0.052	$37.44
Total price per month					$363.63
Total price per year					$4363.56

From these fairly simple calculations we can see that running a 6 instance setup with about 200GB of storage per month and VPN and internet connectivity will cost us about $4400 annually, data transfer costs excluded. Transferring data out of Amazon VPC costs $0.090 per GB for the first 10TB transferred, but data transfer costs vary so wildly that you should simulate the load you are expecting. We calculated that with 10GB of data transferred per day, or 300GB per month, we would be paying about $27 for transfer costs.

The over $4000 price is, however, the maximum price for such a setup. Amazon Web Services offers discounts if instances are bought up-front for example a year, and will greatly reduce computing instance prices for the so called *reserved instances*. We can also calculate the pricing for a single reserved *t2.medium* EC2 instance for a 1-year and 3-year reservation period, which we have done in table 7.6.

For all instances this would mean that pre-purchasing computing capacity from AWS for a single year's period could bring the total system price down to $2112 for EC2 computing nodes and $3357 for the whole system per year. Pre-purchasing for

Table 7.6 *AWS EC2 t2.medium instance pricing in Frankfurt*

EC2 instance type	Term	Payment	Monthly price	Yearly price
On-demand	1-year	Monthly	$43.2	$518.40
Reserved	1-year	Upfront	$29.22	$352.00
Reserved	3-year	Upfront	$19.92	$239.00

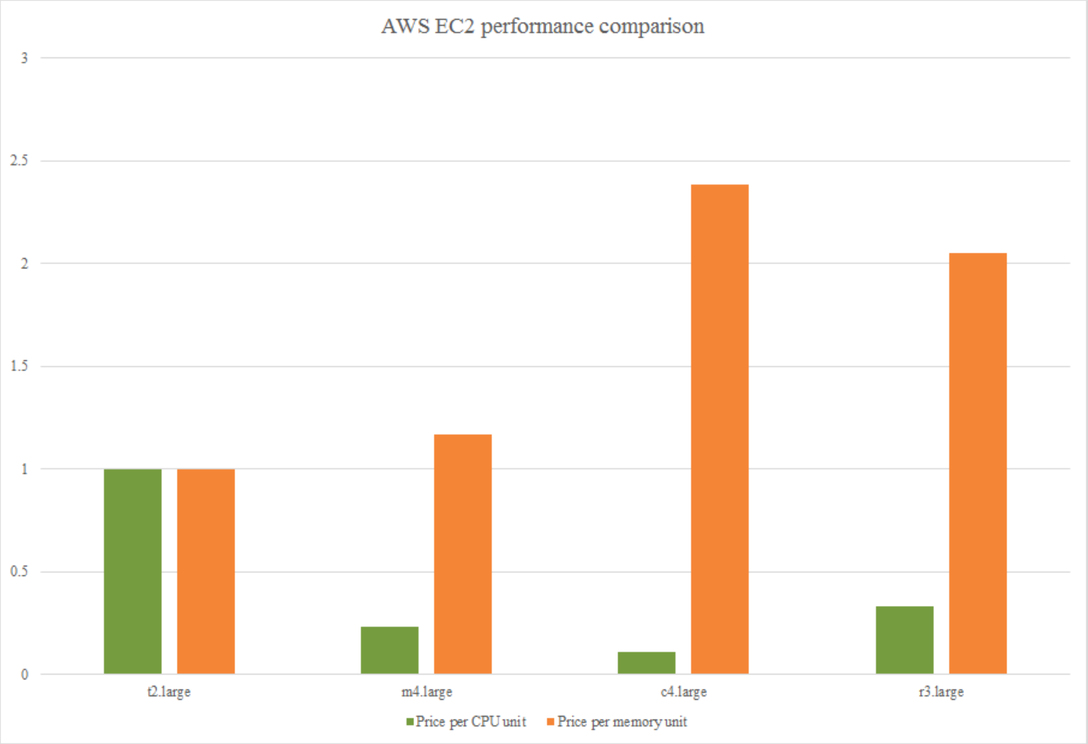


Figure 7.3 *AWS performance comparison per VM instance class*

a period of three years could bring the total cost of EC2 computing nodes down to $1434 and the total system price down to $2687, totalling for almost a 40% reduction in system price.

If we expected to be using for example 10 decent sized builder nodes in our Continuous Delivery system, we would be paying about $5000 per year plus storage and data transfer costs, which are about 10-20% cost increase in our modelled solution. If we were to buy capacity up front, we could reduce this to about $2500 plus storage and data transfer.

To further model price scaling in respect to system computing power scaling, we could graph price per instance in our cloud system and use the competitively priced *t2.large* type instances as our basic unit of computing resources. In figure 7.3 illustrating performance comparison and figure 7.4 illustrating pricing comparison we can see that the very competitively priced *t2* instance class has quite a good balance of CPU and memory capacity: here we are looking at instances from *m4* (general purpose), *r4* (memory optimized) and *c4* (computing optimized) classes, and can see that *t2* offers a good balance in all respects [57, 58].

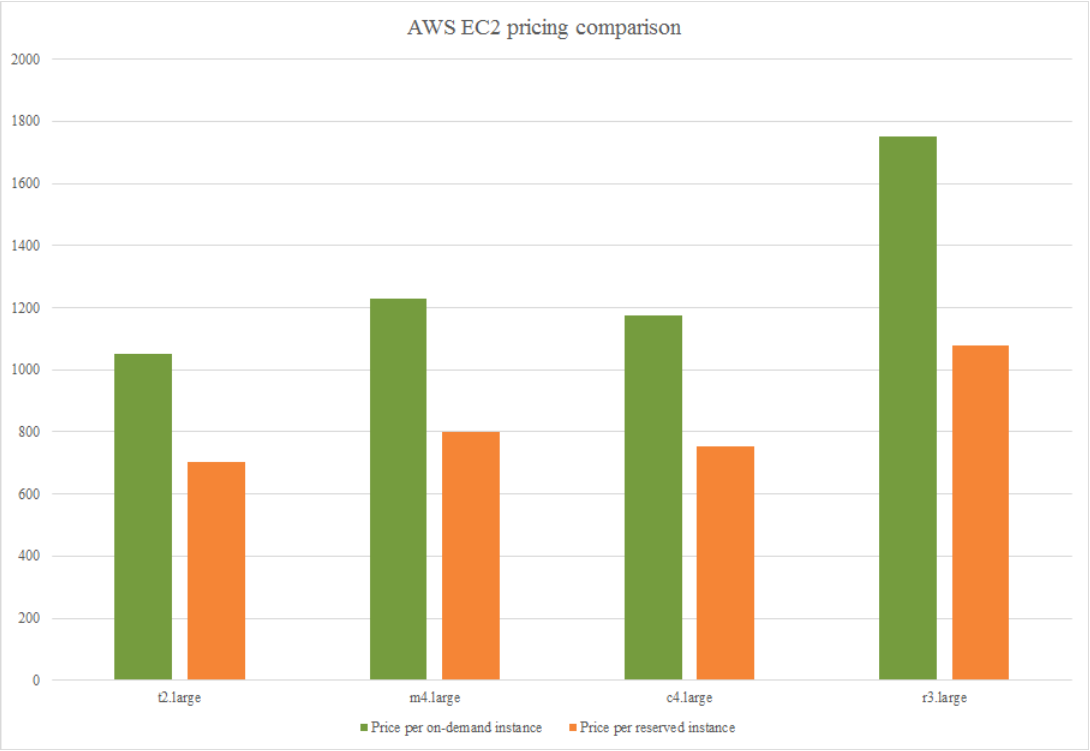


Figure 7.4 *AWS pricing comparison per VM instance class*

This discussion attempts to illustrate that running a private cloud of a dozen small servers with decent specifications can be more affordable than getting a single decent on-premise server, and does not necessarily have fixed costs, although in the case of converting floating costs to fixed costs we can reduce the system price further. Some might even argue that buying on-premise capacity is not sensible if we can not be certain of full utilization of resources, because cloud systems can be scaled to the exact capacity requirements in a very flexible and exact manner.

8. IMPLEMENTING A CONTINUOUS DELIVERY SYSTEM

8.1 Implementing the Continuous Delivery System

The implementation of an AWS cloud based computer system is rather easy once you have a proper design and have factored in needs for capital, capacity, system architecture, administration, and orchestration. An avid computer engineer experienced with Linux should easily enough grasp the basic concepts of the AWS cloud regarding networking, computing and storage capacity. The system illustrated in this thesis does not have much need for all the complex building blocks AWS offers, and can be built on the very basic concepts.

Our implementation work for the private cloud at Vincit began by creating a network layout illustrated in the previous chapters; We created a VPC in Frankfurt with CIDR block 10.177.0.0/16, and created three different subnets in that network segment. Our subnets consisted of a management subnet 10.177.0.0/24, a Go server subnet 10.177.1.0/24, and a Go worker subnet 10.177.2.0/24. Once we had our network layout defined, we set up a VPN gateway to it and opened a ticket to our *Internet Service Provider* or *ISP* requesting that our office network be connected via our router with VPN to the AWS network and routing policies be configured. This took about two weeks and a few failed configuration attempts from our ISP, but after the wait we had our networks defined and were able to connect to the AWS cloud from our office. During this waiting period we started setting up our virtual server infrastructure and software components into AWS to avoid downtime in the whole process.

We started our EC2 node configuration by searching for the Ubuntu 14.04 LTS Amazon Machine Image from the AWS Marketplace [59], which houses software that can be run on the AWS. Most free Linux distributions can be found on the Marketplace free of charge as they have permissive licencing schemes.

After finding and launching our Ubuntu instances, we continued by configuring them with SSH keys and setting up secure connectivity with them. After succesfully connecting to the instances we installed updates and provisioned the instances with Salt. Salt then proceeded to automatically install Sensu and GoCD software to the nodes. At this point we had the architecture illustrated in figure 8.1 implemented.

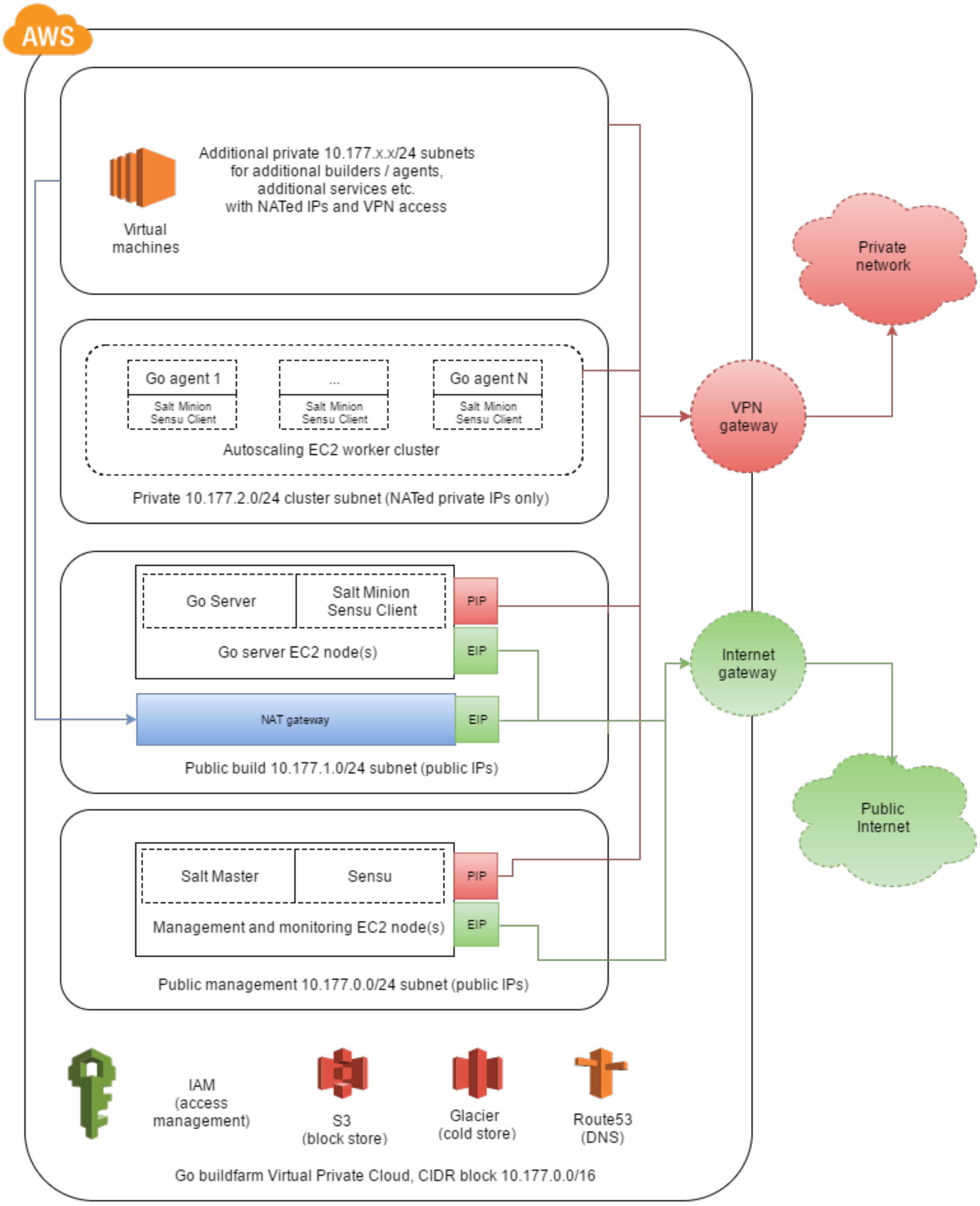

AWS
Virtual machines
Additional private 10.177.x.x/24 subnets for additional builders / agents, additional services etc. with NATed IPs and VPN access
Private network
Go agent 1
Salt Minion Sensu Client
...
Salt Minion Sensu Client
Go agent N
Salt Minion Sensu Client
Autoscaling EC2 worker cluster
Private 10.177.2.0/24 cluster subnet (NATed private IPs only)
VPN gateway
Go Server
Salt Minion Sensu Client
PIP
Go server EC2 node(s)
EIP
NAT gateway
EIP
Public build 10.177.1.0/24 subnet (public IPs)
Internet gateway
Public Internet
Salt Master
Sensu
PIP
Management and monitoring EC2 node(s)
EIP
Public management 10.177.0.0/24 subnet (public IPs)
IAM (access management)
S3 (block store)
Glacier (cold store)
Route53 (DNS)
Go buildfarm Virtual Private Cloud, CIDR block 10.177.0.0/16

***Figure* 8.1** *Initial pure AWS cloud architecture for GoCD*

The system seemed to work in the beta testing environment, and we had everything running smoothly. Builds were executing on the workers and we were managing node state and the server state with Salt, and following node statistics with Sensu. Our Salt scripts would install packages and whole programming environments required in builds, fetch SSH keys and configurations needed to interact with source code repositories. We had network connectivity to our office intranet and the internet from the private cloud system.

8.2 Analyzing, Managing, and Optimizing System Cost and Performance

Analyzing the degree of system usage is easy on most IaaS cloud platforms. Most IaaS platforms are virtualized and offer access to the virtualization system's CPU usage statistics. AWS is not too different in this regard. AWS offers numerous statistics of an instance that can be gathered and stored for an arbitrary period of time.

Some of the statistics that AWS offers via its proprietary CloudWatch system for an EC2 virtual machine instance are:

- CPU usage;
- disk read and write statistics, and;
- network device usage.

Memory usage statistics are not provided by the virtualization platform, but can be additionally monitored with reporting scripts running in the virtualized guest operating system. [60]

After running the service for a while we realized that we were running workers in AWS that weren't being used during the night time when all our developers were out-of-office. This meant that we were running computing capacity idle and paying for the full capacity.

Because AWS supports capacity scaling with Scaling Groups and Autolaunch Configurations, we created an automatically scaling cluster that would scale capacity up in the morning and only run a minimal amount of capacity in the night time. Namely the system would run zero instances in the night and 2-4 instances between

6AM and 8PM, local time. This would total to a 40% less running time for worker instances, which would reduce our EC2 instance costs for 4 worker instances and 2 management and server instances by over 25%. Since we earlier saw that the EC2 running costs constitute for about 80% of our overall costs, we could reduce our overall AWS costs by about 20%.

Automatic scaling requires that each time an instance is started, all necessary software and configuration is installed to it. We configured our Linux instances to run a bootstrapping script that installs a Salt Minion [61] to a node each time a cluster machine is brought up, and Salt, our orchestration tool, would configure the node as a Go Agent after that. All-in-all, our whole bootstrapping for the instance constituted to program 8.1. The script is just a basic *Bourne shell* or *bash* script that can easily be modified to install Salt on any Linux distribution. It also can be reconfigured largely on the same principles to bootstrap a node that is running OS X or Windows for Salt configuration. This removes the need for manually configuring computers.

```
#!/bin/bash

# Set shell flags
set -x # verbose mode
set -e # exit on error
set -u # do not allow unset shell variables

# Fetch Salt Apt keys and add the salt repository
wget -O - \
    https://repo.saltstack.com/apt/ubuntu/14.04/amd64/latest/SALTSTACK-GPG-KEY.pub \
    | sudo apt-key add -
echo \
    "deb␣http://repo.saltstack.com/apt/ubuntu/14.04/amd64/latest␣trusty␣main" \
    > /etc/apt/sources.list.d/saltstack.list

# Install the Salt Minion program
apt-get update
apt-get install -y salt-minion

# Configure the Salt master node address
sed -i "s/^#master:.*/master:␣10.177.0.42/" /etc/salt/minion
sed -i "s/^#id:.*/id:␣go-agent-$HOSTNAME/" /etc/salt/minion

# Restart the minion program
# to initiate connection to management node
service salt-minion restart
```

Program 8.1 *Bootstrapping script for Go Agents*

8.3 Refactoring the System for New Requirements

After about two months of testing and running the system and adjusting its parameters such as instance and disk sizes, network settings, pre-installed packages, and software configurations, we were pretty happy with the overall health and performance of all components. We had about ten software projects building in the new system and had proven the concept of running a small-scale private cloud build farm that was connected to our office network as a transparent part of our logical intranet.

The only pain points of the software were that sometimes Salt was failing with message queue communications and the orchestration software was crashing, which might have been because we were not using a stable release, but a development release, or because we were running maintenance scripts that possibly affected the program's execution. Either way, everything but Salt worked quite smoothly.

A need for restructuring the system arose from an outward requirement. We had decided to move our production services such as LDAP, CI, and CD master nodes to a server that ran Docker Engine, an infrastructure service for running Linux containers [62]. At Vincit we advocate Docker because of the simplicity it provides in easily providing isolated services anywhere where Linux is installed. The basic idea is that a Docker container is an environment that is separated from the host operating system in which it runs, and programs can be bundled into such separate environments with necessary libraries and hosted on any server easily. This makes distributing and setting up programs easy, as no modification is usually required in a host system for running a container once container hosting service is set up. Downloading an image and running it is enough to set up software infrastructure.

We wished to move the Go Server into a our internal network, and enable agents from our own and connected networks to be plugged into it. This would make setting up network topology easier than running on AWS, because we have off-site capacity and off-site offices that need to be connected to the system, and Tampere acts as the central node for many of those systems. The system topology would mutate from the pure AWS setup slightly, as illustrated in figure 8.2.

The refactoring of the system is currently done and the Go Server nodes are transferred to our infrastructure without problems. A simple copy of the Go Server configuration and recreation of a few select projects were necessary when we upgraded

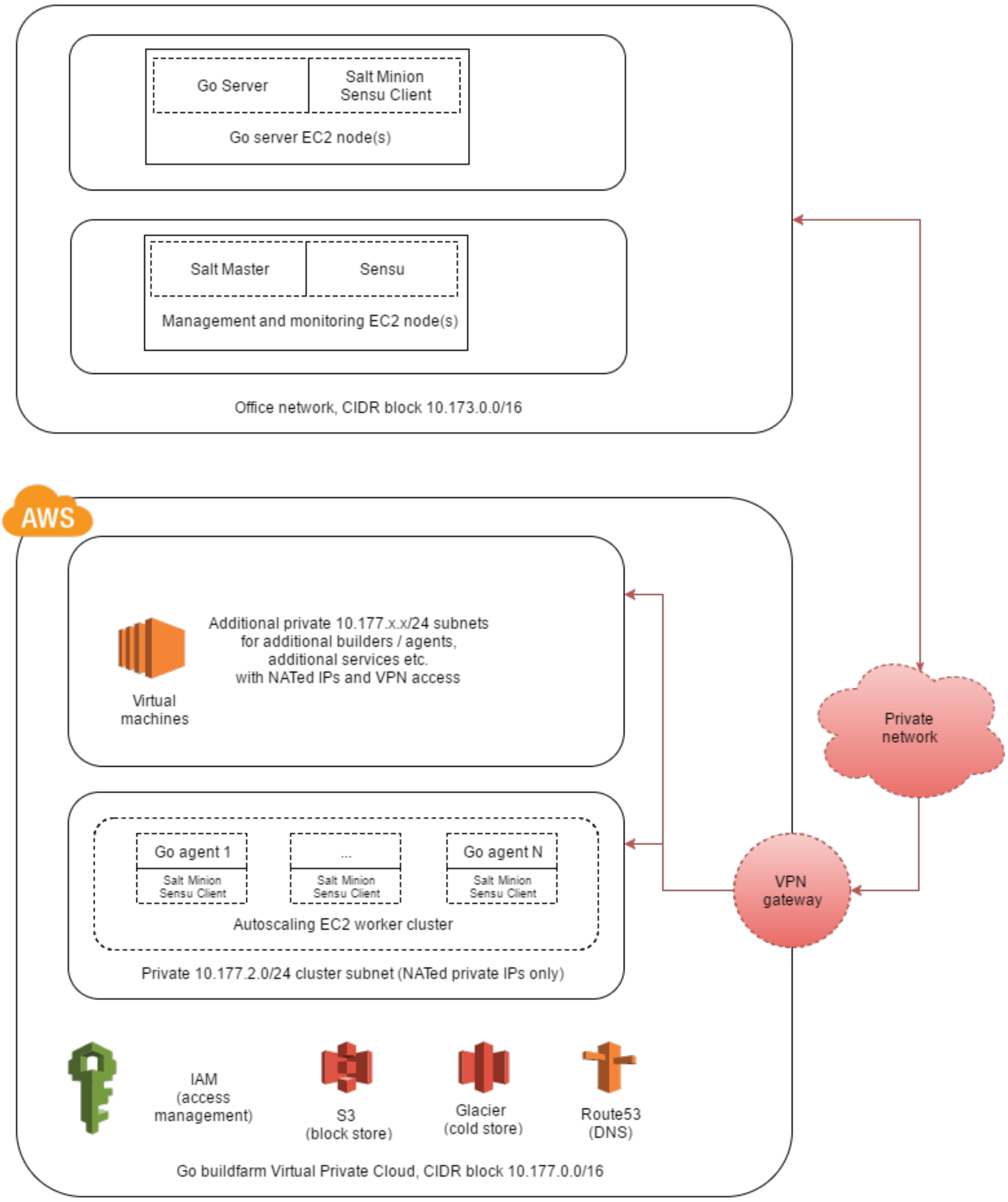

Go Server
Salt Minion
Sensu Client
Go server EC2 node(s)
Salt Master
Sensu
Management and monitoring EC2 node(s)
Office network, CIDR block 10.173.0.0/16
AWS
Virtual
machines
Additional private 10.177.x.x/24 subnets
for additional builders / agents,
additional services etc.
with NATed IPs and VPN access
Private
network
Go agent 1
Salt Minion
Sensu Client
...
Salt Minion
Sensu Client
Go agent N
Salt Minion
Sensu Client
Autoscaling EC2 worker cluster
VPN
gateway
Private 10.177.2.0/24 cluster subnet (NATed private IPs only)
IAM
(access
management)
S3
(block store)
Glacier
(cold store)
Route53
(DNS)
Go buildfarm Virtual Private Cloud, CIDR block 10.177.0.0/16

***Figure** 8.2 Refactored hybrid AWS cloud architecture for GoCD*

our server software after downloading and starting a Docker image that contains the Go Server software.

All-in-all, with our system architecture where software components and their responsibilities are clearly split, refactoring and physically moving parts of the system in and out of the cloud was not perceived hard.

9. PROJECT EVALUATION AND REFLECTION

Determining the success of a project before the product has been in production is hard. This can be even more true for a thesis project, where large amount of the work done is research, writing and documentation work. We do not have comparable system metrics such as build and deployment rates and durations as of yet.

From our former build systems built on top of Jenkins and GoCD we had data such as:

- build durations for multiple thousand different builds;
- failure rates due to configuration, build tool, and system errors;
- rate of deployments to development and production, and;
- end-user feedback.

From the new system we primarily have cost models, a small amount of build duration and rate data, and some early feedback.

9.1 System Costs

The Amazon costs are feasible to calculate. We are, at the moment, paying for a couple medium sized build nodes. In our initial pure cloud architecture we also had an orchestration node and a master server node. These amount to half a dozen cloud computing nodes. In addition to the computation capacity we are also paying for approximately 200GB of storage capacity and low network transfer costs. We do not have fixed dedicated capacity tied to the build system yet, but acquisition and utilization of extra capacity is fairly easy.

Work wise, we have spent about two weeks of time in design stages and meetings throughout the project. One person has also worked on the project for about three months. In grand total, we have about 4 months of work invested in research, implementation of the system. It might be possible to implement a Continuous Delivery system from scratch for a small amount of projects much faster, but overall, the effort of studying the tooling and theory associated to software automation, orchestration, metrics and data gathering, and different details such as cloud platform specifics is quite time consuming.

9.2 Benefits achieved

Our build durations have dropped by approximately 25% due to moving our builders to the cloud and having them less loaded. This is largely connected to a single builder node only processing a single project and not taking any additional load. We were able to select the correct build machine sizes for various projects and select the optimal amount of resources to host our systems, making it possible to finely tune the offered build capacity to the needs of our developers.

Relative failure rates due to system errors have reduced, because we are only executing a single build on a single system, and are not introducing conflicts, caching problems, or computing resource exhaustion into the build process. These are all things that are fairly expensive to debug, because a person has to go and look at the build logs to determine an indeterministic reason for a build failure. The exact reduction in errors is not transparent, but early data suggests we have solved some of our concurrency, virtualization, and container based problems, moving from platform problems to build node or job configuration errors. The latter are much easier to locate and fix.

It also seems that we have improved our build tooling on many parts. GoCD supports resource tagging, build environment specification, heterogeneous builders, fan-in and fan-out capabilities, and other features that are hard to find in traditional build tools. We have not faced any performance issues or instability from the tool.

In addition to improving our systems technically we have introduced the concept of push-button deliveries and high deployability. Only some projects are using this delivery scheme on GoCD, but we have implemented similar features using Travis CI for Continuous Delivery with the AWS Elastic Beanstalk platform using the Travis' *dpl* tool [63]. We are currently introducing push-button delivery to new projects, which has reduced the need for manual deployments and saved work time in projects.

All-in-all, the perceived improvements are considerable. The concrete measurable improvements which will save our customers money will hopefully come apparent in the upcoming months and years. Quantifying the project results is hard at this stage, when we do not have extensive data available yet. Many of the benefits we have achieved were not expected to be immediately available though, and will accumulate in time when an increasing number of projects adopt the Continuous Delivery methodology and gain confidence in rapidly available customer deliverables and increased deployment rates.

9.3 Measuring Progress for Continuous Delivery

Getting features implemented and delivered to the customer with less work, fewer errors, shorter development cycle, and less downtime is the main thing that automation enables [8]. To improve the rate of delivery we hope to implement a comprehensive measuring system that could give us insight on deltas between development, deployment, and activation times. We want to measure features and releases done per month in addition to other system statistics such as build durations and frequencies. We also want to make this information transparent to software development teams and customers [64]. Metrics and data enable improved decision making processes which are based on scientific methods.

Most build and deployment tools offer some built-in data visualization and metrics, but few offer simple APIs for exporting their internal metrics into usable formats. The implementation of a metrics service that integrates into our GoCD service, hosting platforms, and other tools is therefore a task that will require more research. Research is most likely suited for another thesis work.

REFERENCES

[1] Wikipedia, "Version Control System," https://en.wikipedia.org/wiki/Version_control, retrieved January 20, 2016.

[2] L. Torvals, "Git," https://git-scm.com/, retrieved April 29, 2016.

[3] Apache Software Foundation, "Subversion," https://subversion.apache.org/, retrieved April 29, 2016.

[4] R. Osherove, *The Art of Unit Testing.* Manning Publications, 2013.

[5] M. Fowler, "UnitTest," http://martinfowler.com/bliki/UnitTest.html, retrieved January 20, 2016.

[6] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007.

[7] M. Fowler, "TestPyramid," http://martinfowler.com/bliki/TestPyramid.html, retrieved January 20, 2016.

[8] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley, 2010.

[9] J. Mukherjee, *Continuous Delivery Pipeline - Where Does It Choke?: Release Quality Products Frequently And Predictably (Volume 1).* CreateSpace Independent Publishing Platform, 2015.

[10] R. Black, "Investing in Software Testing: The Cost of Software Quality," http://www.compaid.com/caiinternet/ezine/cost_of_quality_1.pdf, Tech. Rep., 2000, retrieved April 29, 2016.

[11] Amazon Web Services, Inc., "Amazon Web Services," https://aws.amazon.com/, retrieved May 13, 2016.

[12] ThoughtWorks *et al.*, "Go Continuous Delivery," https://www.go.cd/, retrieved May 13, 2016.

[13] National Institute of Standards and Technology, "The NIST Definition of Cloud Computing," http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf, retrieved May 13, 2016.

[14] C. Caum, "Continuous Delivery Vs. Continuous Deployment: What's the Diff?" https://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff, retrieved January 20, 2016.

[15] Y. Sundman, "Continuous Delivery vs Continuous Deployment," http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment, retrieved May 13, 2016.

[16] PuppetLabs, Inc., "Puppet," https://puppet.com/, retrieved May 14, 2016.

[17] J. Humble, "Continuous Delivery vs Continuous Deployment," http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, retrieved May 13, 2016.

[18] T. Fitz, "Continuous Deployment," http://timothyfitz.com/2009/02/08/continuous-deployment/, retrieved May 13, 2016.

[19] Google, Inc., "Google Trends: Continuous Delivery," https://www.google.com/trends/explore#q=continuous%20delivery, retrieved January 20, 2016.

[20] ——, "Google Trends: Continuous Deployment," https://www.google.com/trends/explore#q=continuous%20deployment, retrieved May 13, 2016.

[21] A. Rehn, T. Palmborg, and P.Böstrom, "The Continuous Delivery Maturity Model," http://www.infoq.com/articles/Continuous-Delivery-Maturity-Model, Tech. Rep., 2013, retrieved January 18, 2016.

[22] E. Minick, "Continuous Delivery Maturity Model," https://developer.ibm.com/urbancode/docs/continuous-delivery-maturity-model/, Tech. Rep., 2014, retrieved January 18, 2016.

[23] P. Bahrs, "Adopting the IBM DevOps approach for continuous software delivery: Adoption paths and the DevOps maturity model," https://www.ibm.com/developerworks/library/d-adoption-paths/, Tech. Rep., 2013, retrieved January 18, 2016.

[24] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The Highways and Country Roads to Continuous Deployment," *IEEE Software*, vol. 32, pp. 64–72, 2015.

[25] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices.* Pearson, 2002.

[26] Digile, "N4S-Program: Finnish Software Companies Speeding Digital Economy - Digile N4S," http://www.n4s.fi/en/, retrieved May 13, 2016.

[27] J. Seiden, "Amazon Deploys to Production Every 11.6 Seconds," http://joshuaseiden.com/blog/2013/12/amazon-deploys-to-production-every-11-6-seconds/, retrieved May 13, 2016.

[28] C. Lianping and P. Power, "Continuous Delivery: Huge Benefits, but Challenges Too," *IEEE Software*, vol. 32, pp. 50–54, 2015.

[29] Forrester Consulting, "Continuous Delivery: A Maturity Assessment Model: Building Competitive Advantage With Software Through A Continuous Delivery Process," https://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html, Tech. Rep., 2013, retrieved January 20, 2016.

[30] Eclipse Foundation, "Hudson Continuous Integration," http://hudson-ci.org/, retrieved May 15, 2016.

[31] K. Kawaguchi *et al.*, "Jenkins," https://jenkins.io/, retrieved May 15, 2016.

[32] D. J. Mitchell *et al.*, "Buildbot," http://buildbot.net/, retrieved May 15, 2016.

[33] Travis CI, GmbH, "Travis CI," https://travis-ci.com, retrieved January 21, 2016.

[34] I. Radchenko *et al.*, "Strider Continuous Delivery," http://stridercd.com/, retrieved May 15, 2016.

[35] drone.io, "drone.io," https://drone.io/, retrieved May 15, 2016.

[36] ThoughtWorks, Inc., "Snap CI," https://snap-ci.com/, retrieved January 21, 2016.

[37] Microsoft, Inc., "Team Foudation Server," https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx, retrieved May 15, 2016.

[38] PC Magazine, "SaaS Definition from PC Magazine Encyclopedia," http://www.pcmag.com/encyclopedia/term/56112/saas, retrieved May 15, 2016.

[39] Github, Inc., "GitHub," https://github.com/, retrieved January 21, 2016.

[40] Atlassian, Inc., "Bitbucket," https://bitbucket.org/, retrieved May 24, 2016.

[41] ThoughtWorks, Inc., "Fan-out & Fan-in," https://www.go.cd/videos/go-fan-out-fan.html, retrieved April 29, 2016.

[42] A. Damodaran, *Applied Corporate Finance: A User's Manual*, 4th ed. John Wiley and Sons, 1999. [Online]. Available: http://people.stern.nyu.edu/adamodar/New_Home_Page/ACF4E/appldCF4E.htm

[43] Amazon, Inc., "How AWS Pricing Works," https://d0.awsstatic.com/whitepapers/aws_pricing_overview.pdf, Tech. Rep., 2016, retrieved May 15, 2016.

[44] K. Wiegers and J. Beatty, *Software Requirements (3rd Edition) (Developer Best Practices)*, 3rd ed. Microsoft Press, 8 2013. [Online]. Available: http://amazon.com/o/ASIN/0735679665/

[45] N. Thayer-Hart *et al.*, "Survey Fundamentals: A Guide to Designing and Implementing Surveys, version 2.0," https://oqi.wisc.edu/resourcelibrary/uploads/resources/Survey_Guide.pdf, Tech. Rep., 2010, retrieved April 10, 2016.

[46] M. Loudikes, "What is DevOps?" http://radar.oreilly.com/2012/06/what-is-devops.html, retrieved May 4, 2016.

[47] Google, Inc., "Google Apps for Work," https://apps.google.com/, retrieved April 18, 2016.

[48] A. Pennebaker, "Linters - an introduction to static code analysis," https://github.com/mcandre/linters, retrieved May 15, 2016.

[49] Amazon Web Services, Inc., "AWS Documentation," https://aws.amazon.com/documentation/, retrieved April 20, 2016.

[50] Y. Rekhter, R. G. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "IETF RFC 1918," https://tools.ietf.org/html/rfc1918, retrieved April 20, 2016.

[51] Amazon Web Services, Inc., "AWS Virtual Private Cloud Documentation," http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/vpc-ip-addressing.html#vpc-public-ip-addresses, retrieved April 20, 2016.

[52] Chef Software, Inc., “Chef,” https://www.chef.io/chef/, retrieved May 15, 2016.

[53] Red Hat, Inc., “Ansible,” https://www.ansible.com/, retrieved May 15, 2016.

[54] SaltStack, Inc., “Salt,” https://saltstack.com/, retrieved May 15, 2016.

[55] Nagios, “Nagios Enterprises LLC.” https://www.nagios.org/, retrieved May 15, 2016.

[56] Heavy Water, LLC., “Sensu,” https://sensuapp.org/, retrieved May 15, 2016.

[57] Amazon Web Services, Inc., “Amazon EC2 Pricing,” https://aws.amazon.com/ec2/pricing/, retrieved April 21, 2016.

[58] ——, “Amazon EC2 Instance Types,” https://aws.amazon.com/ec2/instance-types/, retrieved April 21, 2016.

[59] ——, “AWS Marketplace,” https://aws.amazon.com/marketplace/, retrieved April 21, 2016.

[60] ——, “AWS CloudWatch,” https://aws.amazon.com/cloudwatch/, retrieved April 22, 2016.

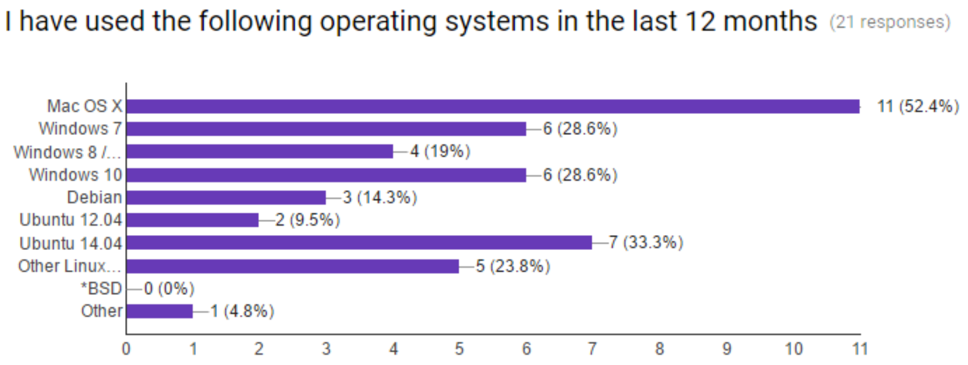
[61] SaltStack, Inc., “SaltStack architecture for system command and control,” https://saltstack.com/saltstack-architecture/, retrieved May 15, 2016.

[62] Docker, Inc., “Docker,” https://www.docker.com/, retrieved April 22, 2016.

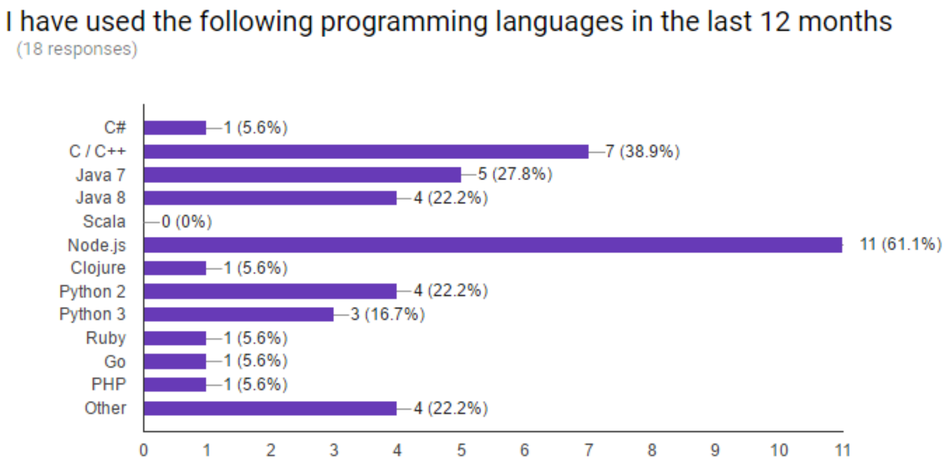
[63] K. Haase, “Dpl deployment tool,” https://github.com/travis-ci/dpl, retrieved May 4, 2016.

[64] T. Lehtonen, S. Suonsyrjä, T. Kilamo, and T. Mikkonen, “Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST’15) Tampere, Finland, Oct 9-10, 2015.: Defining Metrics for Continuous Delivery and Deployment,” *CEUR Workshop Proceedings*, vol. 1525, pp. 16–30, 2015. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:0074-1525-1

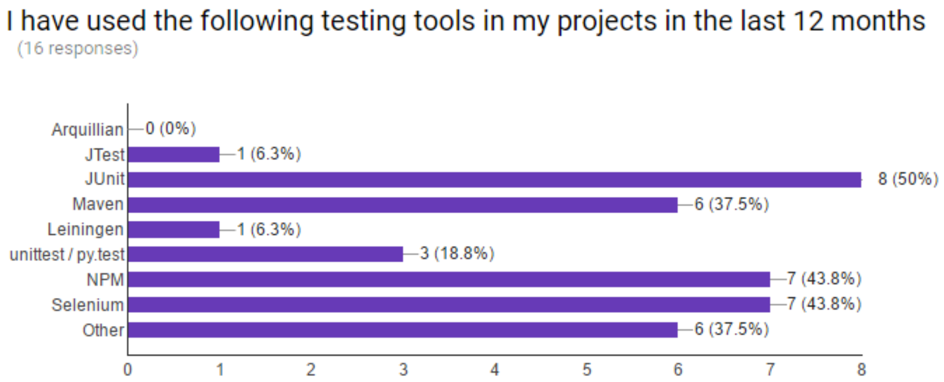
APPENDIX A. VINCIT TOOLING AND SERVICE SURVEY HIGHLIGHTS



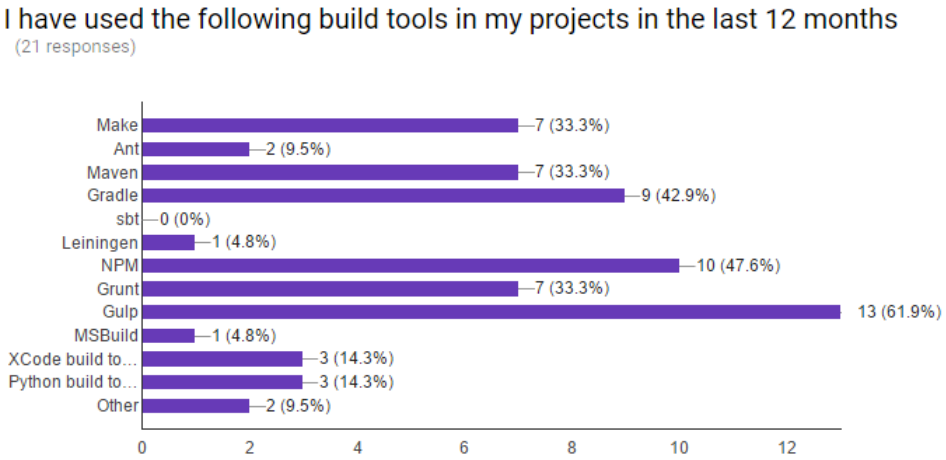
Operating system usage



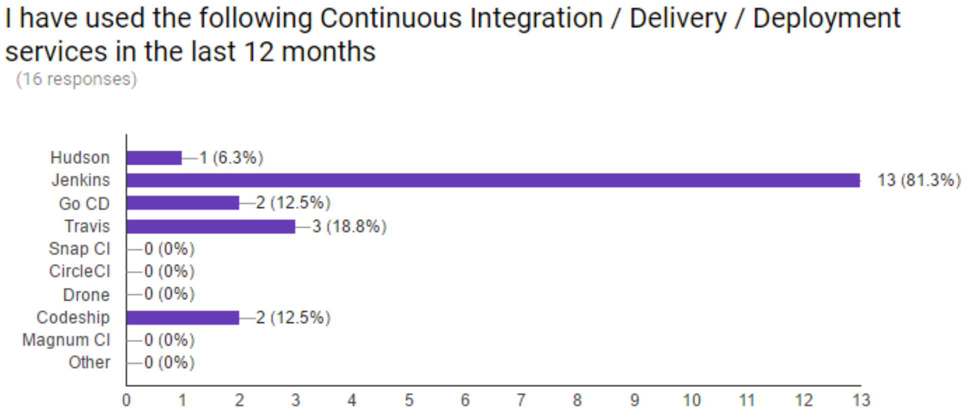
Programming language usage



Testing tool usage



Build tool usage



Continuous Integration and Delivery tool usage