Towards Cloud Native Continuous Delivery: An Industrial Experience Report

Aleksi Häkli
Vincit
Helsinki, Finland
Email: aleksi.hakli@vincit.fi

Davide Taibi, Kari Systä
Tampere University of Technology
Tampere, Finland
Email: davide.taibi@tut.fi, kari.systa@tut.fi

Abstract— **Several companies suffer from long delays between implementing and delivering software features.**

Continuous Integration, Delivery, and Deployment tools and practices are increasing their popularity since they can help companies decrease delivery costs and delays, and reduce bugs and errors in the delivery processes thanks to automation.

Companies have difficulties in implementing Continuous Integration, Delivery, and Deployment. There is information available on the subject, but collecting, studying, and distributing that information can be very costly.

The CI and CD pipelines are examples of well founded Cloud Native applications that run in a variety of different configurations and provide flexibility and speed for companies, but require elastic execution platforms and knowledge of modern cloud technologies for optimal use.

In this experience report we present the lessons learned during the implementation of a cloud-based Continuous Delivery tool stack. Our experience includes the design and implementation of a pure cloud and hybrid cloud based Continuous Integration and Delivery solutions at Vincit.

I. Introduction

In 2015 Vincit, a Finnish software development company, started a Continuous Integration and Delivery project as part of the Finnish N4S project [2] with a the goal of providing software developers build and test automation tools that support complex Continuous Delivery pipelines which contain multiple build, test, and deployment targets with complex configurations.

Continuous Delivery practices can save time and reduce costs in projects. Development teams can ensure that software has been tested and ready for production when working and verified builds are rolled from integration pipelines 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 [4].

Continuous Delivery can be implemented in stages, supporting first the processes where the needs for automation and orchestration are the 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 [9], [10], [11].

We had previous experience in build and test automation solutions from running tools such as Hudson, Jenkins, and Travis CI, which we currently use in our repertoire as well. However, the creation of configuration pipelines and delivery was still complex in 2015. Since then, all these projects have seen tremendous improvement in terms of features and maturity, and nowadays, are much more viable for advanced Continuous Delivery usage.

We consider Continuous Delivery to be a Cloud Native application as it requires elastic computing capacity and the capacity requirements change rapidly. This has financial implications since it is hard to estimate the short- and long-term financial implications of implementing systems. Continuous Delivery pipelines also require advanced monitoring capabilities.

In this paper we report the lessons learned while implementing a customized open-source Continuous Delivery system. 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.

The paper is structured into distinct sections that discuss design, implementation and evaluation of a Continuous Delivery system. Section 2 reports Background and Related Works. Section 3 describes the different possibilities for implementing a CD platform. Section 4 analyzes the difference between self-hosted and cloud platforms. Section 5 discusses cost of systems while Section 6 discusses the choice of Continuous Delivery tools. Section 7 reports the case study with an analysis of how the different steps of the CD pipeline have been implemented. Section 8 evaluates the results of the case study, and finally, Section 9 draws conclusions.

II. Background

Continuous Integration [3] is related to the frequent automatic software integration, which commonly means building and testing changed source code. Frequency means that software is built and tested periodically or, for example, on every version control commit.

Continuous Delivery [4] includes Continuous Integration and making sure that the software is always configurable and deployable. This requirement is usually satisfied with an automated staging environment deployment.

Continuous Deployment includes always automatically deploying software to production when it is committed to version control system branches corresponding to production environments and qualified by the automatic tests to be production-ready.

Continuous Delivery and Deployment are often used in the same context and can be mistaken with each other, but in academic context differentiating the terminology is important [7]. Figure 1 shows the relations between Continuous Integration, Delivery, and Deployment.

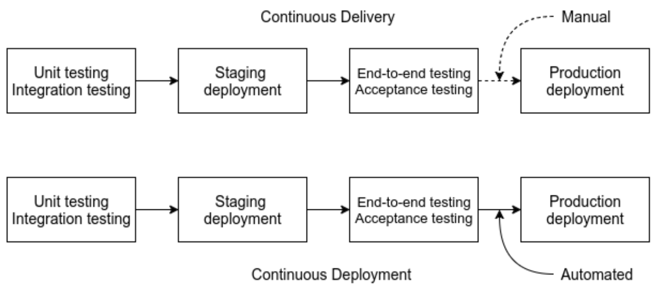
Continuous Delivery
Manual
Unit testing Integration testing
Staging deployment
End-to-end testing Acceptance testing
Production deployment
Unit testing Integration testing
Staging deployment
End-to-end testing Acceptance testing
Production deployment
Continuous Deployment
Automated

Fig. 1. Relations of Continuous Integration, Delivery and Deployment

Companies cannot hence implement Continuous Deployment without Continuous Integration and Delivery systems.

A. Terminology

For the sake of clarity, we define different terms adopted in this work:

- *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 super-set 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 super-set of Continuous Delivery.

III. How can Continuous Delivery be implemented?

There are many tools for Continuous Integration and Delivery written in many languages illustrated in Table I.

TABLE I
OPEN-SOURCE CONTINUOUS INTEGRATION AND DELIVERY TOOLS

Software	Implementation	Published	Maintainer
Buildbot	Python	2003	Mitchell et. al
GoCD	Java	2007	ThoughtWorks, Inc.
Jenkins	Java	2011	Kawaguchi et. al
Travis CI	Ruby	2011	Travis CI, GmbH
Strider CD	Ruby	2012	Radchenko et. al
GitLab CI	Ruby	2012	GitLab, Inc.
Drone	Go	2014	drone.io

Java is one of the most commonly used languages for CD tools, powering services such as Jenkins [12] and GoCD [6]. There are also alternatives for Java powered platforms such as Buildbot [13] written in Python, Travis CI [14] written in Ruby, and Strider [15] and Drone [16] 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. The 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 a few hours on top of Buildbot on a single computer. Some platforms such as Travis CI or Snap CI [17] require a multiple machine set up just to operate on-premise.

Platforms and tools such as Make and 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 intended for building a specific technology solution such as Azure Pipelines for Microsoft products can be much easier to utilize, because they might support the scenarios that one can commonly run into when setting up Continuous Delivery pipelines.

Developers have a myriad of options for adopting Continuous Delivery into their work and project flows which can be categorized as:

- managed SaaS solutions such as Travis CI;
- hostable enterprise solutions such as Azure DevOps, or;
- self-hosted open-source solutions such as GoCD.

Each of the aforementioned options can be valid, depending on the current and future situation and conditions in the company. Smaller companies should prefer to use lightweight managed 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.

IV. Deciding on Cloud Platforms versus Self-Hosted Solutions

One important factor in choosing the right alternative is the solution's extensibility. If in 5 years time we need a feature that is not implemented, what would we do? Many of the current 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 cannot be modified at all. We already knew some requirements for the Continuous Delivery system we wanted to implement in our Minimum Viable Product (MVP)[5].

The first of our requirements was extensibility. Another one was the ability to support cross-platform builds. We wanted the same tool to be usable on macOS, Windows, and Linux environments. The last important requirement was that we could host the Continuous Delivery service in private or public data centers. 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 tool-sets to each other. Narrowing the search down, we found ourselves facing yet another decision: choosing the right open-source option for our company.

V. Cost Models for Continuous Delivery

Different Continuous Delivery solutions have different cost models, which are of essential knowledge when making management decisions regarding the implementation and lifespan of Continuous Delivery systems. Understanding of the cost implications of the different implementation choices is an important decision criterion.

Operating expenses and capital expenses are two main elements of the cost models [21], [33]. In short, capital expenses are multiple-term costs that are tied to a system for a long time, such as data centre investments 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 pay-per-use cost model.

Implementing a platform 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.

Most cloud service providers bill for networking, CPU, RAM, and storage capacity. For example, Amazon Web Services bills for network components such as Internet and VPN connectivity and outbound traffic, server resource usage such as CPU cores and memory, and storage capacity. In addition to this, for the users of proprietary operating systems, such as Windows or Red Hat Enterprise Linux, licence fees apply on per-machine basis. In case of proprietary Continuous Delivery tools, a licence is also applicable. [22], [32]

On-premise computing capacity, in addition to the cloud platform components, adds the cost for power, cooling, and staff work. Moreover, on-premise systems also require resources for handling with power outages, loss of data and other similar issues.

From these concepts we defined to *fixed* and *floating* costs.

- **Fixed costs** are the baseline costs that are tied to the running of the system and rarely change: data centers and equipment are examples of fixed costs.
- **Floating costs** 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 capacity needs are not static and change over time, making optimal investment choices can be hard. With cloud computing platforms capital is not tied to fixed investments, and risks are reduced. [33]

VI. Choosing the Continuous Delivery Tools

One of our requirements that was realized and refined during the project was the ability to support and specify dependencies for build steps and different projects [20]. Complex dependency management is important when building, for example, a microservice architecture or complex multiple tier software where one wants to define the build, test, and deployment pipeline as a dependency graph. For example, it might be necessary to build the backend and frontend software first, and, then test their component integration, and finally test the end-to-end functionality of the system.

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 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 was, in 2016, in middle of implementation of some very central features such as dynamic build agent provisioning. Small delays, however, are things that we were willing to deal with when investing into long-term tooling.

VII. Implementing Continuous Delivery

We implemented the CD system in AWS cloud based computer system. Our implementation work for the private cloud at Vincit began by creating a network layout.

We created a VPC (Virtual Private Cloud) in Frankfurt with /16 CIDR block that was compatible with our existing network layout, and created three different subnets in that network segment. Our subnets consisted of a /24 management subnet, a /24 GoCD server subnet, and a /24 GoCD agent subnet. Once we had our network layout defined, we set up a VPN gateway to it and opened a ticket to our ISP (Internet Service Provider) 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 via private connection 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 (Elastic Compute Cloud) virtual server configuration by searching for the Ubuntu LTS AMI (Amazon Machine Image) from the AWS Marketplace [23], which houses software that can be run on the AWS. Most Linux distributions can be found on the Marketplace free of charge as they have permissive licensing schemes.

After finding and launching our Ubuntu instances, we continued by configuring them with SSH keys and setting up secure connectivity with them. After successfully 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 2.

The system seemed to work in the beta testing environment, and we had everything running smoothly. Builds were executed on the GoCD agents and we were managing nodes with an integrated Salt solution. Our Salt scripts would install packages and whole programming environments required in builds, and fetch and configure the SSH keys and configurations needed to interact with source code repositories.

We were also tracking the raw node statistics with Sensu, which was running on every agent node.

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 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: 1) CPU usage; 2) disk read and write statistics, and; 3) 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. [24]

After running the service for a while we realized that we were running workers that were not used during the night time since all our developers were out-of-office. This meant that we were running idle computing capacity but paid for the full capacity. Because AWS supports capacity scaling with ASGs (Auto Scaling Groups) and Autolaunch Configurations, we created an automatically scaling cluster that could increase capacity in the morning and decrease capacity in the night time. The system would run zero instances in the night and 2-4 instances between 6AM and 8PM, local time. This would total to 40% less running time for worker instances, which reduced the total costs of four worker instances and two management and 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 [26] to a node each time a cluster machine is brought up, and Salt, our orchestration tool, would configure the node as a GoCD agent after that. All-in-all, our whole bootstrapping for the instance constituted to a simple shell 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 macOS or Windows for Salt configuration. This removes the need for manually configuring computers. More details on the script can be found in the associated thesis work [1].

VIII. Evaluation results

The systems implemented and discussed in this paper offered data and subjective experiences that we would like to further discuss and evaluate. Some interesting technical aspects of the system were cost factors, the technical evolution of the tooling, improvements of processes and other tooling we made with implementing Continuous Delivery tools, and measurements of the systems.

A. Cost and labor factors

Work wise, we have invested about two weeks of time in different stages of design and meetings throughout the project. One person has also worked on the project and on a Master thesis describing it for about three months [1]. In grand total, we have about four months of work invested in research and development and the 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.

In early 2016 we were paying for a few medium sized nodes in AWS. We had a master build node, an orchestration node and a monitoring node for GoCD in AWS. These amounted to three fixed cloud computing nodes. Rest of the fleet was flexible build capacity that we were running as needed. In addition to the computation capacity we are also

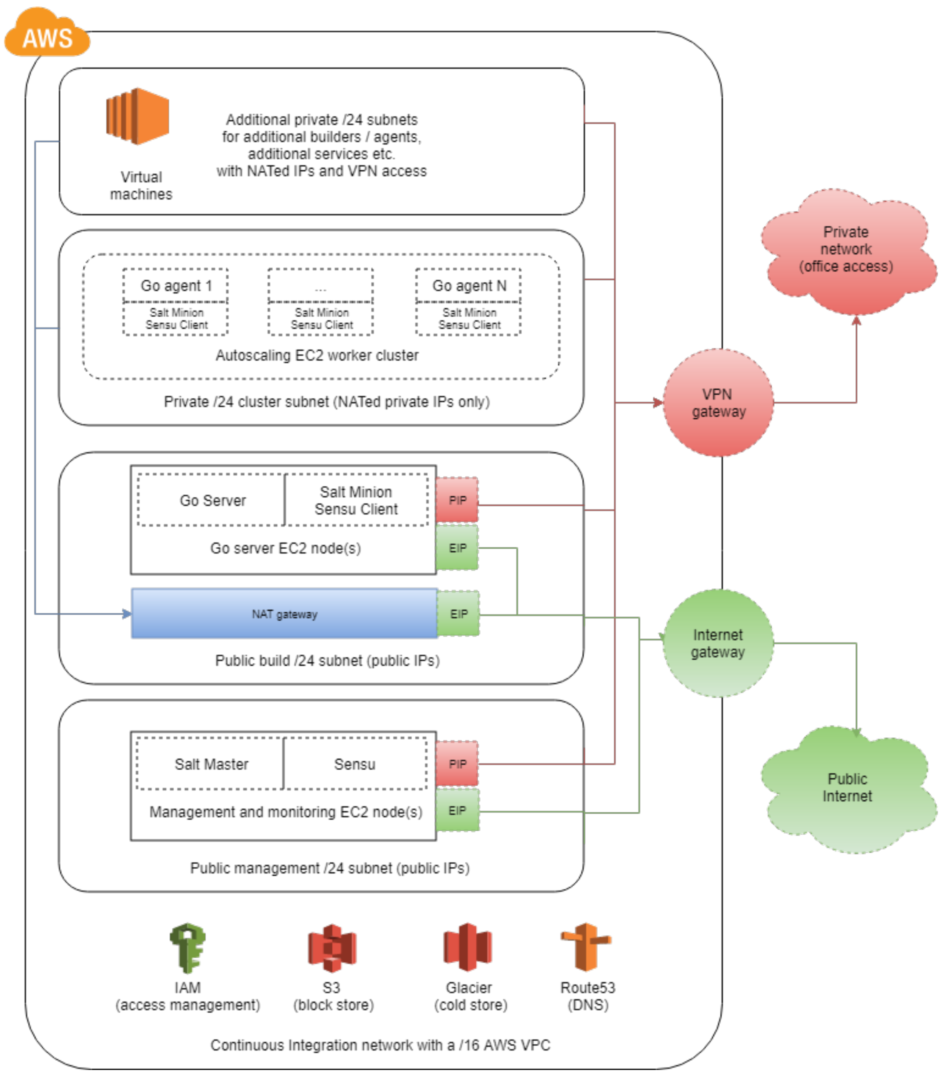
AWS
Virtual machines
Additional private /24 subnets for additional builders / agents, additional services etc. with NATed IPs and VPN access
Go agent 1
Salt Minion Sensu Client
...
Salt Minion Sensu Client
Go agent N
Salt Minion Sensu Client
Autoscaling EC2 worker cluster
Private /24 cluster subnet (NATed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

Fig. 2. Pure Cloud Continuous Integration and Delivery System Architecture

paying for approximately 200GB of fixed storage capacity and low network transfer costs. We did not have fixed dedicated capacity tied to the build system, but acquisition and utilization of extra capacity was fairly easy. This setup is illustrated in Figure 2.

The total cost of the system was in the few hundreds of dollars per month range for the pure cloud solution. Most notable individual cost of the system was an AWS VPN gateway for a hybrid connection that cost USD 100 per month, with the computing capacity costing between USD 100 to USD 300 per month, totaling to a well less than USD 500 per month in all cases for a small build fleet.

Maintenance and administration work has required an average of one to two day investment of work per month on the tools, resulting in notable continuous running costs. These costs, are however, hard to define and attribute, as some could be associated to project setup, and at least some of the work used on maintenance of the tools would be required in a managed tool such as Travis CI because projects need to be kept up-to-date and build failures investigated.

B. *Evolution of the technology stack*

Later in 2016 and 2017 we moved into a hybrid and then onto a on-premise hosting model due to hosting structure changes, and moved the GoCD master node into a fixed virtual machine and inside a Docker container. The build agents are run as Docker containers, too. This seems to offer excellent price-to-performance ratio due to lowered overheads. Containerization also corrects many of the problems of running multiple build agents on the same physical or virtual server. All agent workspaces are isolated because the GoCD agent Java processes run inside containers, having very limited and closely controlled access to the host system resources aside from ephemeral file system and network access. This has worked well for us for well over a two years.

At the time of writing in late 2018 we are running Jenkins as our primary and GoCD as our secondary internal build system with both doing Continuous Delivery for projects. Jenkins has a larger amount of projects, slaves, and users due to its more complete feature set. GoCD also has a distinct group of projects, but most new projects are set up in Jenkins.

This is because we have previous history with Jenkins, and its development for both core and and plugins have moved forward with a faster pace than in GoCD. Most reasonable technical requirements are also met by the existing plugins. Jenkins has also introduced a large feature called pipelines in 2016 and 2017. Pipelines enable writing build definitions in either declarative or scripted manner, and enable shareable libraries for defining common build operations. Pipelines also enable easy parallel build steps and allow granular build dependency definition in free-form graphs.

A major note we would like to make is that both Jenkins and GoCD have access control and identity management limitations. In our use they are currently only offered to our developers internally in company intranet due to security considerations. Both could be opened up to the Internet with some restrictions, but little is to be gained in terms of usability, and system security will need to regularly audited.

In terms of user management and authentication Jenkins and GoCD both do feature LDAP based login backends, allowing easy directory integration. Both also offer varying quality SSO solutions based on SAML 2.0, OAuth 2.0 and OpenID Connect 1.0. These plugins, however, are community driven and rarely if ever audited security-wise, and the responsibility of ensuring their security often falls to the administrators of the tools.

C. *Perceived and measured improvements*

Our build duration dropped by approximately 25% due to moving our builders to the AWS 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. This is also possible with our on-premise build system that is based on virtualization.

Failure rates due to system errors have reduced, because we are only executing a single build on a single system or container, and are not introducing conflicts, caching problems, or computing resource exhaustion into the build processes. These are all things that are fairly expensive to debug, because a person has to go and investigate the build and system 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 improved our build tooling on many parts during the project. GoCD supports resource tagging, build environment specification, heterogeneous builders, management of project dependencies, 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 push-button deployments on GoCD, but we have implemented similar features using Travis CI for Continuous Delivery with the AWS Elastic Beanstalk platform using the dpl tool by Travis CI [27]. 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, because 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.

In addition to us implementing continuous improvements to the build tooling since 2016, we have gained a considerable amount of knowledge in the Continuous Integration, Delivery, and Deployment domain. The increased knowledge has steadily improved the processes and tools in use, and will hopefully enable us to move forward with software development practices as well.

D. *Metrics, data, and information*

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 [4]. 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 [29].

At this time the collection and evaluation of metrics and data is hard due to the use of multiple different build systems that build different kinds of software projects. The effort of closely measuring and analyzing the systems and their differences has not been yet undertook, but is an essential step in developing more robust and sophisticated tooling.

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 the said metrics into usable formats. The implementation of a metrics and measurement service that integrates into different projects, their services, hosting platforms, and other tools is therefore a task that will require more research efforts.

IX. Conclusion

In this paper we report on our experience in implementing a Continuous Delivery system based on open-source technologies. We first described the different possibilities to implement a Continuous Delivery platform, highlighting pros and cons and we compared cost between hosted and cloud solutions. In our case study we then implemented a complete cloud-based Continuous Delivery pipeline, and then we migrated into a hybrid solution. It is interesting to notice how and when we maximized the benefits between cloud and hosted solutions, and how we integrated them.

The same benefits could be transferred to any Cloud Native application, not only Continuous Delivery. Companies could learn from our experience when and why it could be beneficial to migrate parts of their systems into the cloud.

We believe that a pure cloud hosting and containerization offers the most flexibility for implementing Continuous Integration, Delivery and Deployment systems. The current trends in open-source build systems seem to moving towards so called Cloud Native Continuous Integration. Quite recently CloudBees and Kohsuke Kawaguchi have been driving the Jenkins development into a more containerized direction, even opting to redesign existing solutions and dropping backwards compatibility to move development faster going forwards [30].

Future work include the definition of a cost model to support companies in understanding the most suitable alternatives between different cloud providers and a in-premise solution. We will also analyze the reasons why companies are migrating to Cloud Native pipelines [31], of patterns and anti-patterns of the Cloud Native pipelines, following the previous approaches applied in [8], [19], [31]. Moreover, we are also planning to support companies in using data collected from the CD platform to predict different software characteristics. As an example, it we are planning to extend data-driven models for software reliability [34], maintenance [35] also considering dynamic measures [28] and suitability of Cloud Native patterns [25].

References

[1] A. Häkli, "Implementation of Continuous Delivery Systems" Master Thesis. Tampere University of Technology, 2016. http://dspace.cc.tut.fi/dpub/handle/123456789/24113

[2] Need4Speed Project. DIMECC N4S-Program: Finnish Software Companies Speeding Digital Economy.

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

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

[5] V. Lenarduzzi, D. Taibi, "MVP explained: A systematic mapping study on the definitions of minimal viable product." 42nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2016, 112-119. doi:10.1109/SEAA.2016.56

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

[7] C. Caum, "Continuous Delivery Vs. Continuous Deployment. What's the diff?" https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff, retrieved Sept 2018.

[8] D. Taibi, V. Lenarduzzi, "On the definition of microservice bad smells." IEEE Software, 35(3), 56-62. doi:10.1109/MS.2018.2141031

[9] 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 Oct 2018.

[10] E. Minick, "Continuous Delivery Maturity Model," https://developer.ibm.com/ urbancode/docs/continuous-delivery-maturity-model/, Tech. Rep., 2014, retrieved Oct 2018.

[11] 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 Oct 2018.

[12] K. Kawaguchi et al., "Jenkins," https://jenkins.io/, retrieved Oct 2018.

[13] D. J. Mitchell et al., "Buildbot," http://buildbot.net/, retrieved Oct 2018.

[14] Travis CI, GmbH, "Travis CI," https://travis-ci.com, retrieved Oct 2018.

[15] I. Radchenko et al., "Strider Continuous Delivery," http://stridercd.com/, retrieved Oct 2018.

[16] drone.io, "drone.io," https://drone.io/, retrieved Oct 2018.

[17] ThoughtWorks, Inc., "Snap CI," https://snap-ci.com/, retrieved Oct 2018.

[18] Microsoft, Inc., "Team Foudation Server," https://www.visualstudio.com/ en-us/products/tfs-overview-vs.aspx, retrieved Oct 2018.

[19] D. Taibi, A. Janes, V. Lenarduzzi, "How developers perceive smells in source code: A replicated study." Information and Software Technology, 92. 2017. doi:10.1016/j.infsof.2017.08.008

[20] ThoughtWorks, Inc., "Fan-out & Fan-in," https://www.go.cd/videos/go-fan-out-fan.html, retrieved Oct 2018.

[21] 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

[22] Amazon, Inc., "How AWS Pricing Works," https://d0.awsstatic.com/whitepapers/aws pricing overview.pdf, Tech. Rep., 2016, retrieved Oct 2018.

[23] ——, "AWS Marketplace," https://aws.amazon.com/marketplace/, retrieved Oct 2018.

[24] ——, "AWS CloudWatch," https://aws.amazon.com/cloudwatch/, retrieved Oct 2018.

[25] D. Taibi, V. Lenarduzzi, C. Pahl, "Architectural patterns for microservices: A systematic mapping study." 8th International Conference on Cloud Computing and Services Science, CLOSER, 2018.

[26] SaltStack, Inc., "SaltStack architecture for system command and control," https://saltstack.com/saltstack-architecture/, retrieved Oct 2018.

[27] Docker, Inc., "Docker," https://www.docker.com/, retrieved Oct 2018.

[28] L. Lavazza, S. Morasca, D. Taibi, D. Tosi, "On the definition of dynamic software measures." International Symposium on Empirical Software Engineering and Measurement (ESEM) 2012. doi:10.1145/2372251.2372259

[29] K. Haase, "Dpl deployment tool," https://github.com/travis-ci/dpl, retrieved Oct 2018.

[30] K. Kawaguchi, "Jenkins: Shifting Gears," https://jenkins.io/blog/2018/08/31/shifting-gears/, retrieved Oct 2018.

[31] D. Taibi, V. Lenarduzzi, C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation." IEEE Cloud Computing, 4(5), 22-32. 2017 doi:10.1109/MCC.2017.4250931

[32] P. Rosati, F. Fowley, C. Pahl, D. Taibi, T. Lynn, "Making the cloud work for software producers: Linking architecture, operating cost and revenue." 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), 2018-January 364-375

[33] S. Aldawood, F. Fowley, C. Pahl, D. Taibi, X. Liu, "A coordination-based brokerage architecture for multi-cloud resource markets." 4th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2016, 7-14. doi:10.1109/W-FiCloud.2016.19

[34] L. Lavazza, S. Morasca, D. Taibi, D. Tosi "An empirical investigation of perceived reliability of open-source Java programs." ACM Symposium on Applied Computing (SAC), 1109-1114. doi:10.1145/2245276.2231951

[35] V. Lenarduzzi, A.C. Stan, D. Taibi, D. Tosi, G. Venters, "A dynamical quality model to continuously monitor software maintenance." 11th European Conference on Information Systems Management, ECISM 2017.