A Framework for Evaluating Continuous Microservice Delivery Strategies

Martin Lehmann
Faculty of Technology
Westerdals Oslo School of Arts, Communication and Technology
Oslo, Norway
martin@lehmann.tech

Frode Eika Sandnes
Faculty of Technology
Westerdals Oslo School of Arts, Communication and Technology
Oslo, Norway
sanfro@westerdals.no

Faculty of Technology, Art and Design
Oslo and Akerhus University College of Applied Sciences
Oslo, Norway
frode-eika.sandnes@hioa.no

ABSTRACT

The emergence of service-oriented computing, and in particular microservice architecture, has introduced a new layer of complexity to the already challenging task of continuously delivering changes to the end users. Cloud computing has turned scalable hardware into a commodity, but also imposes some requirements on the software development process. Yet, the literature mainly focuses on quantifiable metrics such as number of manual steps and lines of code required to make a change. The industry, on the other hand, appears to focus more on qualitative metrics such as increasing the productivity of their developers. These are common goals, but must be measured using different approaches. Therefore, based on interviews of industry stakeholders a framework for evaluating and comparing approaches to continuous microservice delivery is proposed. We show that it is possible to efficiently evaluate and compare strategies for continuously delivering microservices.

CCS Concepts

•**Software and its engineering** → **Software version control; Programming teams;**

Keywords

Continuous deployment; microservices; deployment strategy; evaluation framework; cloud computing; microservice architectures

ICC '17, March 22 2017, Cambridge, United Kingdom

DOI: http://dx.doi.org/10.1145/3018896.3018961

1. INTRODUCTION

Continuous Delivery is a fairly simple task in a monolithic system [7]. Here, all required configurations are instructions that are used to build, test and deploy the monolithic application. With a low release frequency, such as a few times a day, it is feasible to maintain a continuous stream of releases manually. Automation can be achieved by simply introducing a single build server that builds, tests, and deploys the system whenever it detects that new code is pushed to the revision control system. The way software is developed and deployed, as well as server technology, has changed dramatically over the last decade. Focus has shifted from distributed task scheduling [19, 20] and the configuration management of servers [6, 17, 18] to microservices-based systems and deployment to cloud infrastructure [28].

A microservice-based system [7, 24], however, introduces a whole new set of complications. In a relatively large system with hundreds of microservices, the system may require hundreds, or even thousands, of deployments in a single day. Moreover, the deployment roles and responsibilities may be divided between different individuals. A single person, or even team, will not realistically be able to manually handle the influx of new versions and changes in a production system.

The literature addressing the quality the software deployment pipeline [15] mostly employs quantitative measures. Frequent metrics include number of steps to perform a deployment and lines of code to change the configuration. Although useful, such metrics do not consider the changes to projects over time. For example, the number of lines of code required to make a change does not directly give any indication of the value added by the same change. Furthermore, aspects such as testability [13] of the software system are crucial to a large-scale development project, yet testability has received limited attention in the research literature.

Individuals responsible for the development must usually decide which underlying technology stack to choose. This is a challenging task where poor choices are expensive. Moreover, there are few tools that can help navigate the com-

plex and rapidly changing landscape of available options and tools for packaging, testing, deploying, and scaling services. This work therefore proposes a framework to help evaluate such technologies, so that one may commit to the most suitable technology stack.

2. BACKGROUND

Talwar et al. [23] define a *service* as a piece of software that encapsulates and presents some useful functionality, and can be part of an overall system. In other words, a service is a specialized, autonomous, stand-alone server. The idea of splitting a large application's code base into multiple services is often referred to as Service-Oriented Architecture (SOA) [25, 3].

Evans and Fowler [11] used the term *domain* about the subject area to which a piece of software is applied. In this context, the entire set of services is a software expression of the domain: it both contains the data, and provides means of accessing and manipulating them. A domain has multiple contexts, especially in large projects. It is therefore important to define bounded contexts[1] in which a model applies [11]. A project should initially be a monolith, and be decomposed into microservices the domain is explored and the project grows in size and complexity [16].

Each client in a system is an expression of a bounded context. A client has traditionally meant either a desktop application or a website, rendered to static assets (HTML, CSS, and JavaScript) on the server before being sent to the end user's web browser. In a system with only microservices and clients, the client connects directly to one or more services. It is then the client's task to combine data in a meaningful way and display the result to the user.

With the emergence of front-end JavaScript frameworks such as Google's AngularJS[2] and Facebook's React[3], as well as mobile apps, it has become a common practice to build an API (Application Programming Interface) gateway[4] in addition to the clients. An API gateway is similar to the classic façade pattern [14] from object-oriented programming. The responsibility of the API gateway is to combine data from the various microservices within the bounded context, and expose the data to the client through a tailored web API. In this pattern, each API gateway represents a single bounded context.

Villamizar et al. [25] conclude that there are several benefits to being able to publish a system as a set of smaller services that can be managed independently. Specifically, they point to independence in development, deployment, scaling, operation, and monitoring as a key enabler for companies to manage large applications with a more practical methodology, and scale individual development teams more easily.

Relevant to this discussion is also the *CAP theorem*. Also known as *Brewer's theorem*, it states that it is impossible for a system to simultaneously guarantee consistency, availability and partition tolerance [5], where consistency refers to all nodes seeing the same data at the same time, availability refers to all requests receiving a response indicating success or failure and partition tolerance means that a system continues to function under network failures.

2.1 Software Deployment

Wahaballa et al. [27] define software deployment as "all of the activities that make a software system available for use". Following this definition, deployment is something every single provider of an online service must handle in some way. Deployment strategies can be simple, such as logging onto a server and manually editing some code in an interpreted language running in production. On the other end of the spectrum, they can involve comprehensive code review processes followed by running a pre-built artifact through several suites of automated tests, multiple testing environments for manual quality assurance involving layers of bureaucracy, and finally deploying the new changes to a controlled subset of the production servers, known as blue/green deployment[5], and a fraction of the end users of the service, known as canary release[6].

Software deployment dominates system administration cost, and configuration is a major error source [23]. As the popularity of service-based computing rises, so does the importance of answering which deployment approach is the best fit for the context [22].

Talwar et al. [23] define and compare four different approaches to deployment of services: manual, script-based, language-based, and model-based as a function of scale, complexity, and susceptibility to change. They also define several evaluation metrics, which they call Quality of Manageability for the deployment configuration, namely:

1. Lines of code (LOC) for deployment configuration.
2. Number of steps involved in deployment.
3. LOC to express configuration changes.
4. Time to develop and deploy a change.

Based on these criteria, they conclude that for systems with few deployed services and configuration changes, a manual solution is the most reasonable approach. Few deployed services with comprehensive configuration changes call for a script-based approach. Larger environments where changes involve dependencies prefer language-based solutions. Finally, a model-based approach is ideal for large systems where the underlying service design is affected by the deployment. This is mirrored in terms of configuration [23].

In summary, there are multiple areas to look at in future research related to cloud computing and microservices. Perhaps most prevalent is the need for an evaluation of various strategies and tools for deployment automation. One way to approach this can be development of tools for quantitative comparison of deployment automation strategies. Looking at deployment configuration as source code allows using techniques from software engineering for this evaluation.

Armour [2] stated that a software system is not in itself a product, but a container for knowledge; code is, indeed, executable knowledge. While his focus is on domain knowledge as it lives within software systems, it is possible to apply this idea to any code: as Spinellis [21] points out, code is an executable specification, so expressive and concise code is self-documenting in a way that never rots.

[1] http://martinfowler.com/bliki/BoundedContext.html
[2] https://angularjs.org/
[3] https://facebook.github.io/react/
[4] https://www.nginx.com/blog/building-microservices-using-an-api-gateway
[5] http://martinfowler.com/bliki/BlueGreenDeployment.html
[6] http://martinfowler.com/bliki/CanaryRelease.html

Another of Talwar et al.'s [23] key findings is that maintainability and documentability are proportional to the number of lines of code, and that the number of steps and lines of code are both reduced with the introduction of more sophisticated deployment tools.

2.2 Continuous Delivery and DevOps

Continuous delivery (CD) can be said to be a combination of two ideas [26], namely continuous integration (CI) and continuous deployment. Continuous integration is the practice of integrating changes into the mainline early (e.g., master branch if the team uses Git for version control). Continuous deployment is the practice of deploying changes to the end users as soon as they make it into the mainline.

Combined, these render a development workflow where developers frequently merge their changes into the production-ready version of the code base, and those changes are immediately deployed to the end users. This way of developing may introduce a need for feature management such as blue/green deployment and canary release.

Continuous Delivery is only a part of a deployment strategy, but deserves specific attention because of its potential organizational impact. With a monolithic architecture, it may be feasible to have a dedicated operations (ops) team and still deploy new features and fixes to the end users somewhat continuously, as the team will only have to deal with a single artifact. Even if they need to deploy it multiple times per day, tools can be developed to quickly verify and deploy the artifact. In a microservice context, however, with multiple services and teams each selecting their own technology stacks and deployment habits, it quickly becomes infeasible for a dedicated operations team to manually verify and deploy each service as it receives changes. Furthermore, having to provide the ops team with a production-ready package for them to verify and deploy whenever a change is made to the code base introduces unnecessary overhead for the development teams. This requires an organizational shift of deployment responsibility to the development teams, known as DevOps [10].

2.3 Measuring Continuous Delivery

Several studies have specifically attempted to measure the quality of a deployment pipeline [23, 25, 12], and others have focused on only a specific part of the pipeline [21, 8]. Given that deployment configuration is indeed code, it becomes relevant to look to metrics designed to measure code quality in general with regard to architecture [4].

Chen [9] introduces the concept of architecturally significant requirements (ASRs), referring to the architectural requirements imposed upon the service itself by the deployment strategy.

One important factor described by Chen [9] is which architecturally significant requirements the deployment strategy imposes on the project. While Chen considers architecting for continuous delivery in a broad sense, different strategies will impose fewer or more architecturally significant requirements of differing types on the code base of the actual service. More architecturally significant requirements can make the transition to a continuous delivery set-up more difficult, more time consuming, and thus more expensive.

One particular architecturally significant requirement is considered by Addo et al. [1], who describe an architecture for automatically routing traffic to other cloud providers if one fails. This points back to the CAP theorem, and raises the question of how to replicate data and ensure consistency across multiple cloud providers.

2.4 Summary

First, provisioning and maintaining the microservice runtime is a key concern. It ties in with the CAP theorem, stating that consistency, availability, and partition tolerance is an unattainable goal.

Second, cloud computing is an important factor when discussing microservices, as many of the key advantages of a microservice architecture come into play when computing resources are virtually unlimited. A system of microservices naturally requires more resources such as CPU cycles and RAM than a monolithic system, as the microservice-based system comprises multiple runtimes and databases. If computing resources are not limited, however, this isolation allows both scaling and deployment of independent services. This has become a realistic scenario due to the rise of cloud providers.

Third, the deployment regime in the organization plays a large role in the selection of a deployment strategy. It is essential to contrast how often the organization *wishes* to deploy changes to the end users with how often the infrastructure *allows* deployment. Automating the process takes work and introduces a learning curve. The deployment pipeline should therefore be tailored to suit the needs of the organization.

Fourth, the expressiveness and readability of the code that specifies the deployment strategy is a key factor in cutting the learning curve for the deployment strategy. As the configuration code grows, its quality can be measured using tools from the software engineering field—not just deployment.

Fifth, selecting a deployment strategy may have a significant organizational impact. For example, introducing DevOps to enable continuous delivery in an organization that previously had a centralized team responsible for deployments requires the organization to distribute these responsibilities to the developers, and possibly introduce new roles. In a sizeable organization, this distribution of responsibility can be challenging, because it affects the workflow of everyone responsible for the development, ranging from developers to managers. DevOps also affects processes for identifying and resolving problems with the software.

Last, there are multiple ways of measuring quality of deployment approaches. In particular, architecturally significant requirements of the deployment strategy may require so many changes to the existing code base that the strategy is unfeasible for the organization.

3. EVALUATION FRAMEWORK

The evaluation framework is designed to assist system architects in selecting a strategy and technology stack for implementing continuous delivery in a microservice-based software system. We define a *strategy* as the combination of technological components and organizational measures taken to achieve continuous delivery.

3.1 Overview

We have based the framework on a combination of experiences documented in the existing literature as well as two interviews of industry professionals. We interviewed two

technical team leaders at FINN.no, Norway's leading actor in the online classified ads market and a frontrunner in Norway for adopting modern technological trends. Their system comprises hundreds of microservices, and new changes are delivered to the end users tens of times per day by multiple autonomous teams. As a result, they have faced and countered many of the problems discussed here. We focused on their efforts towards automating continuous microservice delivery by interviewing the infrastructure team lead, as well as the lead developer on a team that has completely automated their software delivery process.

The framework itself is a set of criteria uncovered as important to the development and delivery process through the existing literature and the interviews. Each criterion is accompanied by a unit. These units can be used both to give a general overview of the strategy, and compare it to other strategies.

3.2 Testability

The framework measures testability broadly, as one of the following three options.

1. Trivial: single set-up per project, mirrors the production environment.

2. Time-consuming: some set-up per machine, but mirrors the production environment.

3. Uncertain: requires set-up per machine and cannot closely mirror the production environment.

Testability is perhaps the most prominent requirement for a system architect selecting a deployment strategy. All informants strongly agreed that automated tests are crucial to continuously delivering changes to the end users with certainty. These tests can be grouped into three levels: unit testing, for individual code fragments; module testing, for individual microservices; and feature testing, for cross-service integrations.

The interviews showed that the strategy must allow test suites on all three of these levels. Unit tests only require the language runtime to be executed, so the deployment strategy will typically have little effect on the unit testability. Testing on the module level, on the other hand, requires starting the actual application with a production-like context. In this case, the strategy may drastically impact the simplicity of running the actual service for testing. Feature level tests take this problem even further, as multiple services must both run and communicate with each other. The technical implementation of the deployment strategy is certain to affect the difficulty of feature level testing.

Testability, and especially feature level testing, is a complex matter regardless of deployment strategy. A measurement of a system's testability is most useful when compared to that of other strategies: systems often differ too much to allow easily measuring the time to develop a running test suite. However, it is possible to quantitatively measure things like the number of significant differences from production environment (e.g., operating system or available computing resources) and the number of manual steps required of each developer to run the test suite on their local machines.

3.3 Deployment Abstraction

Deployment abstraction concerns the learning curve for developers in order to efficiently deploy their services. Deployment abstraction can be evaluated as a group of criteria. First, the *number of manual steps* a developer must take between finishing their changes and deploying a verified artifact to the end users is a key indicator of the strategy's quality in this regard.

Second, given that code is executable knowledge [2] [21], the expressiveness of the deployment configuration and learning required to use it must be considered relatively to that of other strategies.

We define the following three levels of deployment configuration expressiveness.

1. Highly expressive: no highly error-prone manual steps, little learning required.

2. Somewhat expressive: some manual steps.

3. Manual: largely manual deployment, error-prone.

Clearly, deployment abstraction ties in closely with the number of manual steps required to perform a single deployment. A greater number of manual steps makes the deployment process much more error-prone [23]. In other words, a good abstraction allows the deployer to follow the artifact as it automatically passes through each quality assurance step before it is released to the end users.

As Talwar et al. [23] found, maintainability and documentability are proportional to the number of lines of code (LOC), and LOC are reduced by introducing more sophisticated deployment tools. However, in the DevOps context, introducing tools comes at the expense that developers in each team must learn how to configure and use them. On the other hand, using no tools requires the developers to be intimately familiar with the infrastructure in addition to their own services.

3.4 Environment Parity

We measure the parity between environments as one of the following three options.

1. Equal: any software bug is present in all environments.

2. Distinguishable: some differences that can be easily mitigated.

3. Disparate: parity must be ensured manually on all development, testing, and production machines.

The parity[7] of runtime environments for development, testing, and production is crucial to quality assurance. Operating systems differ in many ways, for example the implementation of the file system. Furthermore, runtimes implemented for multiple platforms inevitably differ in some ways.

The goal for maintainability must be to achieve completely equal environments. However, this may not always be feasible: testing and production servers typically sport headless[8] operating systems, which is unfeasible for development.

[7]Dev/prod parity is described in detail in The Twelve-Factor App, a collection of factors that increase the maintainability of a system: https://12factor.net/

[8]Without a graphical user interface.

Some measures can be taken to circumvent this issue, such as running the code inside a virtual machine with the same operating system as the production environment. On the other hand, this can be impractical, as virtual machines are expensive in terms of computing resources. Some differences, such as in available processing power, are mostly unavoidable.

In many development contexts, including at FINN.no, most developers develop and run applications locally on macOS. The production servers, on the other hand, usually run on Linux-based distributions. This disparity between the local and production environments is compensated for with production-like build servers. Each code change requires the artifact to be rebuilt and tested on a build server.

3.5 Time to Deploy

The framework measures the average overall time to deploy a service in minutes.

The time taken to deploy an artifact is a major concern in continuous delivery. Traditionally, completing the automated test suites of a service could take several hours. In a microservice environment, and particularly if the build servers are shared between projects, building and verifying a single artifact cannot be allowed to occupy resources for long. Furthermore, the developer is often required to watch over the deployment to ensure it went well, and an extensive deployment time directly hinders productivity for the developer.

The total time taken to deploy is affected by multiple aspects of the deployment strategy. On the strictly technical side is the number of long-running automated tests. On the organizational side, there is the number of staging environments and the amount of manual quality assurance required before delivering the changes to the end users.

3.6 Availability Adequacy

High Availability is a well-known quality attribute for software systems. The framework measures the deployment strategy's impact on system availability as one of the following three options.

1. Adequate: zero-downtime deployments and automatic scaling of computing nodes in response to both increased and decreased server load.

2. Excessive: zero-downtime deployments and easy manual scaling of computing nodes.

3. Error-prone: significant error-prone manual work involved in deployment or resource scaling.

For FINN.no, high availability means staying online during both deployments and peak load. This is obviously important to Software as a Service (SaaS) providers[9] to make continuous delivery feasible. Zero-downtime deployments can be easily accomplished for stateless services by running more than one instance of the same service behind a load balancer.

Automatic scaling of available computing power—usually the number of computing nodes—to handle requests is required to handle peak load times in a non-wasteful way. We call this *availability adequacy*: offering just enough computing power to solve the task. Using an Infrastructure as a Service (IaaS)[10] solution such as Amazon AWS[11] solves this problem at monetary cost. However, in a manual, self-hosted deployment regime, resource scaling requires manually adding or removing physical computing nodes, and updating the load balancer configuration.

3.7 Summary

We have identified *testability*, *deployment abstraction*, *environment parity*, *time to deploy*, and *availability adequacy* as the five most key elements of strategies for continuous microservice deployment. These do not necessarily come at the expense of each other. However, logically, any abstraction must be *learned* by the developers, and computing resources carry a cost. This must be considered by each system architect in the context of their organization and system as they evaluate various strategies.

All criteria aim to decrease the overall cost of deployment by way of reducing both time to build, verify, and deploy an artifact after it has been changed, and the number of errors resulting from manual work.

4. APPLYING THE FRAMEWORK

A simple web service system is used to demonstrate how the framework is used. Two contrasting strategies for continuously deploying the system are explored using the framework: manual, and automated with containers.

4.1 Test application

Figure 4.1 presents *BeerFave*, a small sample web application system comprising three different microservices and one web client. The system is intentionally designed to be simple, but its microservices are general enough to be worked on by individual and autonomous teams.

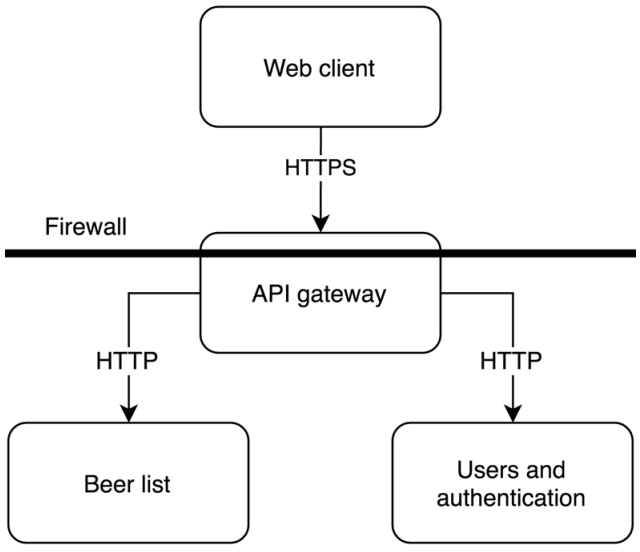

Figure 1: BeerFave architecture

The web client lives outside the firewall, and runs in the end user's web browser. It communicates with an API gate-

[9]Suppliers of applications meant to be used directly by end users, such as an online social network.

[10]Cloud-based, managed hardware and networking configuration on which to set up and configure applications

[11]https://aws.amazon.com/autoscaling/

way[12], a microservice designed to collect data from other microservices behind the firewall, combine the data, and expose them to the web client. This pattern lessens the load on the client, and restricts it from fully accessing the underlying microservices behind the firewall.

The last two microservices are simple. The users and authentication service handles the registration of users and verification of passwords for the system. The beer list service contains all beers available in the application. Another service could be added to allow users to "save" a subset of the beers to their profile.

The system thus contains three different microservices, two of which are dependent on some type of persistent storage such as a database system, as well as one web client. The following paragraphs explore how the framework can be used to evaluate two possible strategies for continuously deploying the system.

To simulate a real-world case, we require the system to remain available during deployment, and to run a production environment as well as a testing environment on self-hosted servers. We make a modest estimate that each service is modified and delivered 10 times per day.

4.2 Deployment strategy evaluation

An evaluation of a completely manual deployment strategy is first described, followed by deployment with Google's Kubernetes[13]. Each criterion in the framework is discussed and summarized.

4.2.1 Manual deployment

Testability is not affected in any particular way in a manual strategy: the microservices are all manually built and deployed, and so the task of running them locally is also a manual one.

Deployment abstraction is not introduced by manual deployment. In this system, it means that the deployer of each service much understand the infrastructure used for all environments. Rolling out a new artifact to the production environment requires uploading the pre-built artifact to each host server and somehow load it into a runtime environment. This does not scale well with an increasing number of services and server hosts. Each runtime environment must be pre-installed on the production and testing servers. However, most of the build and deployment process can be abstracted by using a continuous integration server[14] such as Travis CI[15] or Jenkins[16].

Environment parity is difficult to achieve perfectly with a fully manual approach to deployment. As Talwar et al. [23] found, manual configuration is a major error source in deployment work. Upgrading any given piece of software, or the operating system itself, must be repeated manually on every development machine, as well as every testing and production server. For example, upgrading the Java runtime for one application means installing the new Java runtime on each server in the stack, as well as any continuous integration servers in use.

Time to build, verify, and deploy any given new artifact is high in a multi-host environment, and even more so with multiple environments. For each microservice, for each delivery, for each environment, for each host, the same procedure must be repeated to complete a deployment. For *BeerFave* with two environments and two server hosts, redeploying a single application once would require deplying to 2 environments $\times$ 2 hosts, yielding 4 manual actions, which can be feasible. However, in a more realistic scenario, all four services are worked on and delivered simultaneously. A modest estimate of 10 deployments per application in a single day results in 4 microservices $\times$ 10 deliveries $\times$ 2 environments $\times$ 2 hosts, yielding 80 manual actions every day.

However, it should be noted that a fully manual approach with a scripting language and limited regard for quality assurance would make manual deployment extremely efficient. For example, one might log onto the single production host and change code in a PHP runtime that does not require reloading the environment to reflect changes. In this case, one could achieve both zero-downtime deployments and an extremely low time to deploy the artifact. This is not a healthy strategy in the long run, though, as major changes to the code base are reflected immediately after the files are changed.

Availability adequacy is time consuming to achieve with a manual approach to deployment. Merely ensuring availability during deployments requires at least two separate running instances of the microservice. This, in turn, typically requires the deployer to sign into at least two different servers and complete the task of deployment on each server. This can perhaps be feasible with two instances, but it logically does not scale well to the tens or hundreds, which might be required in a high-load environment.

Scaling the number of computing nodes handling requests to the system requires manual monitoring of the traffic. When traffic increases, new hosts must be manually provisioned and configured to receive traffic. As traffic decreases, traffic must be directed away from hosts before they are taken down. This would be extremely time consuming, so continually running enough nodes to handle the peak traffic is a more feasible, if wasteful, choice.

This brief evaluation of a fully manual approach to microservice deployment can be represented in table form as shown in table 1.

Table 1: Evaluation of a manual deployment strategy

Criterion	Evaluation
Testability ease	Uncertain
Abstraction expressiveness	Manual
Environment parity	Disparate
Number of manual steps	80 per day
Minutes to build, verify, deploy	*Unknown*
Availability adequacy	Error-prone

4.2.2 Deployment with Kubernetes

Kubernetes leverages Docker[17], a highly popular container-

[12]See for example https://www.nginx.com/blog/building-microservices-using-an-api-gateway
[13]http://kubernetes.io
[14]Also known as "build server"
[15]https://travis-ci.org
[16]https://jenkins.io
[17]https://www.docker.com

ization platform that allows isolating services to their own operating systems without the loss of computing power that comes with traditional virtual machines. The Kubernetes platform handles everything related to the actual deployment based on a per-project configuration file.

Testability is strongly affected by the introduction of Docker. Because all artifacts are packaged and published as Docker images, downloading and running an artifact (image) is a trivial task. This affects automated testing by making it much easier to simply start the services on which the service under test depends. For example, in the BeerFave system, the API gateway service depends on both the *users and authentication* service and the *beer list* service. Feature level testing of the API gateway, then, requires starting both of those services with dummy datasets. Without containers, they would typically be started locally in a manual manner along with databases and any other external services. However, a larger network of dependencies makes this approach infeasible. The interviews indicated a trend that it was easier to simply avoid writing feature level tests. Packaging and publishing services as container images greatly helps simplify this process.

Deployment abstraction is the entire role of Kubernetes itself. The application artifact, for example a WAR[18] file in a Java context, is installed into a Docker image. Docker introduces an additional step of building the image. The image is then typically uploaded to a server, and Kubernetes is able to download the image and create a container in which to run the service. In this case, the entire deployment is abstracted into the building and publishing of a Docker image. In other words, the developers do not need to understand the actual infrastructure, or worry about the underlying operating system.

On the other hand, Kubernetes requires each project to be configured through configuration files residing in the repository. In a way, the manual steps are moved to the beginning of the development instead of recurring with every deployment. These configuration files can be extremely short, but the configuration reflects the complexity of the image build process and the deployment itself.

The configuration is expressed as simple values, not code. This increases the expressiveness of the configuration, but decreases its flexibility.

Environment parity is still challenging, but Kubernetes makes it easy in comparison to manual deployment. As the developers configure and build their own Docker images, the testing and production servers are not required to have any runtimes pre-installed. Thus, upgrading runtime versions on a per-microservice basis becomes trivial. However, containerizing the application does not avoid the variance in available computing resources, or any differences between operating systems.

Number of manual steps with a Kubernetes-based setup per delivery is the same as the number of steps required to publish the Docker image and notify Kubernetes to pull it. If this process is automated, the number of manual steps is 0. If not, the image can be built and published with two simple commands. This is negligible in comparison to publishing, uploading, and loading each artifact onto every host in every environment.

Time to build, verify, and deploy is comparatively low, as the entire deployment step is automated by Kubernetes. The build time may go up for simple applications because of the added steps of building and publishing Docker images. This is insignificant, however, because both the verification process and the deployment itself should become much faster and safer when automated.

Availability adequacy is handled in a fully automated manner by Kubernetes, and is activated through configuration parameters on a per-service basis in the Kubernetes configuration file. It supports both zero-downtime deployments through "rolling updates", and automatic scaling of resources. In other words, Kubernetes will try to utilize all available hardware resources in as efficient a manner as possible.

This brief evaluation of a container-based approach using Kubernetes is represented in table form as shown in table 2.

Table 2: Evaluation of a container-based deployment strategy

Criterion	Evaluation
Testability ease	Trivial
Abstraction expressiveness	High
Environment parity	Distinguishable
Number of manual steps	0
Minutes to build, verify, deploy	*Unknown*
Availability adequacy	Adequate

4.3 Summary

We have showed how to apply the framework to two very different strategies: one fully manual, and one completely automated with a complicated stack of technology. The results are discussed in the next section.

5. DISCUSSION

The framework solves two related problems for the system architect in selecting a strategy and technology stack for continuous delivery of microservices.

5.1 Predefined criteria

The framework consists of a set of predefined criteria that are useful in evaluating a deployment strategy. This set of criteria is designed to make it easy for architects to evaluate potential strategies in a structured way, even if they are not intimately familiar with the landscape of software deployment. In other words, the predefined criteria support an initial implementation of each strategy by defining what to look for in the strategies.

The effect of having these predefined criteria lowers the barrier to implementing continuous delivery: the technology and possible strategies are becoming increasingly diverse. It follows that it is difficult to establish confidence in that the most relevant candidate strategies have been evaluated, and the most suitable strategy for the project was selected.

5.2 Transparent comparison

A second effect of this structure for comparing strategies is that each comparison is valuable in itself. Using this framework for evaluating options results in data that are easy to

[18]Web ARchive

revisit and comprehend. The concise and structured nature of the framework allows architects to easily reuse their results in new projects, or even share the results with other teams.

Evaluation results for multiple strategies and technologies help make informed choices, as it becomes simple to understand what formed the basis of selecting a single strategy and technology stack. In a DevOps context, sharing the results in this manner is particularly helpful: the transparency of the evaluation results allows the architect to share the knowledge with the developers, who will be able to understand the strengths and weaknesses of the strategy.

The units presented as part of each criterion in the framework are particularly important in the process of sharing knowledge with others within the organization. Some units are intentionally relative to other evaluations, and thus require some evaluations to have been completed in order to be meaningful. Because of this, the value of the framework improves as it is applied to various strategies. On the other hand, some units are absolute: they allow the architect to present hard facts, such as the number of minutes a deployment can be expected to take.

5.3 Mitigating weaknesses

As each organization and system is different, the most common use case for the framework is to implement a prototype of a deployment strategy and test it for a small subset—possibly just one—of the services in the system. This allows quick evaluation of multiple strategies to get a broad overview of the possibilities, while simultaneously maintaining focus on the key quality indicators. If the organization has more specific requirements than the proposed framework offers, new criteria can easily be added. Similarly, if a criterion is not interesting to the organization, it can simply be removed.

After evaluating several strategies, the implementation of these strategies can be compared and contrasted systematically in a way that highlights important differences. For example, packaging artifacts as virtual machine images yields high parity between testing and production environments, at the cost of significantly increasing its computing resource requirements and the time taken to build, verify, and deploy the artifact.

An organization may not want to introduce a major abstraction. The cost of training developers, who will inevitably make mistakes, is an expected expense of abstracting deployment. This means that even though the framework shows one strategy as "better" than another overall, a conscious decision must be made to accept these expenses. FINN.no observed an immediate need to mitigate the problems with their current deployment regime, and thus found it necessary to train their developers in using a new abstraction.

As mentioned previously, the usefulness of the framework increases incrementally as it is used to evaluate more strategies. In particular, the relative units such as *testability* become more meaningful if multiple evaluations are presented together. Thus, it is in the interest of architects and developers alike to apply the framework and share the results to build a common knowledge base. In some organizations, results can be shared internally for use across multiple teams. In other organizations, results can be made public and thus be of common benefit.

The criteria themselves are intentionally broad, and can of course be tailored to the particular needs of an organization. For example, if a particular type of *testability*, such as feature level testing, is especially important to the industry, it may be simply added as a new criterion—or grouped together with other criteria. In other words, the proposed framework is intended to be a flexible guide that may be modified.

In the evaluations presented in the previous section, the framework seems to indicate that using Kubernetes is the obvious choice over a manual approach. A reliable measurement of developer productivity that respects factors such as the size of the team; the cost of implementing a new strategy; the number of projects any developer is working on simultaneously; and the number of deployments per project per day would be a useful addition to the framework. However, this would likely result in almost disparate frameworks for different types of organizations. In this early generalization of deployment strategy criteria, our focus has intentionally been on the strictly technical aspects of deployment.

6. CONCLUSIONS AND FURTHER WORK

This study addressed continuous delivery in the context of microservices. A framework for evaluating strategies for continuous delivery of microservices was proposed. The framework is based on issues that practitioners from the industry report as important. The work with the framework has revealed large discrepancies between the research literature and practice field. In particular, the literature mostly focuses on quantitative measurements of strategies such as the number of manual steps required to perform a deployment or the number of lines of code required to change the configuration, while the practice field values more general qualities, such as whether the strategy will increase the testability of the system or not.

The evaluation framework comprises a set of characteristics mapped to units. The framework is modifiable. Characteristics can be added, changed, or removed to make the framework better suit a particular context.

Several aspects of continuously deploying microservices were not considered, such as the organizational impact of introducing continuous delivery in a microservice and container-based deployments.

7. ACKNOWLEDGEMENTS

The authors are grateful to FINN.no for their help. The first author is also grateful for financial support towards the project.

8. REFERENCES

[1] I. D. Addo, S. I. Ahamed, and W. C. Chu. A reference architecture for high-availability automatic failover between paas cloud providers. In *Trustworthy Systems and their Applications (TSA), 2014 International Conference on*, pages 14–21, June 2014.

[2] P. G. Armour. *The Laws of Software Process: A New Model for the Production and Management of Software*. CRC Press, 2003.

[3] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, and A. A. Totok. *Service-Oriented Computing – ICSOC 2007: Fifth International Conference, Vienna, Austria, September 17-20, 2007.*

Proceedings, chapter Pattern Based SOA Deployment, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. Pearson Education, 3 edition, 2012.

[5] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.

[6] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software: practice and experience*, 27(9):1083–1101, 1997.

[7] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[8] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, Mar 2015.

[9] L. Chen. Towards architecting for continuous delivery. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 131–134, May 2015.

[10] C. A. Cois, J. Yankel, and A. Connell. Modern devops: Optimizing software development through effective system interactions. In *2014 IEEE International Professional Communication Conference (IPCC)*, pages 1–7, Oct 2014.

[11] E. Evans and M. Fowler. *Domain-Driven Design*. Prentice Hall, 2003.

[12] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013.

[13] R. S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6):553–564, 1991.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[16] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[17] F. E. Sandnes. Scheduling partially ordered events in a randomised framework: Empirical results and implications for automatic configuration management. In *Proceedings of the 15th USENIX Conference on System Administration*, LISA '01, pages 47–62, Berkeley, CA, USA, 2001. USENIX Association.

[18] F. E. Sandnes. Secure distributed configuration management with randomised scheduling of system-administration tasks. *IEICE TRANSACTIONS on Information and Systems*, 86(9):1601–1610, 2003.

[19] F. E. Sandnes and G. M. Megson. A hybrid genetic algorithm applied to automatic parallel controller code generation. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 70–75. IEEE, 1996.

[20] F. E. Sandnes and G. M. Megson. Improved static multiprocessor scheduling using cyclic task graphs: a genetic approach. *Advances in Parallel Computing*, 12:703–710, 1998.

[21] D. Spinellis. Don't install software by hand. *IEEE Software*, 29(4):86–87, July 2012.

[22] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, and G. P. Jung. Approaches for service deployment. *IEEE Internet Computing*, 9(2):70–80, March 2005.

[23] V. Talwar, Q. Wu, C. Pu, W. Yan, G. Jung, and D. Milojicic. Comparison of approaches to service deployment. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 543–552, June 2005.

[24] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[25] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590, Sept 2015.

[26] M. Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. pages 78–82, 2015.

[27] A. Wahaballa, O. Wahballa, M. Abdellatief, H. Xiong, and Z. Qin. Toward unified devops model. In *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on*, pages 211–214, Sept 2015.

[28] G. Zhao, C. Rong, M. G. Jaatun, and F. E. Sandnes. Reference deployment models for eliminating user concerns on cloud security. *The Journal of Supercomputing*, 61(2):337–352, 2012.