Importance of software architecture in continuous deployment

Olli Rissanen
Department of Computer Science
University of Helsinki
Helsinki, Finland
Email: olli.rissanen@helsinki.fi

***Abstract*—Currently more and more software companies are moving to lean practices, which often include shorter delivery cycles and thus shorter feedback loops. However, to achieve continuous customer feedback and to eliminate work that doesn't generate value, even shorter cycles are required. In continuous deployment the software functionality is deployed continuously at customer environment. This process includes both automated builds and automated testing, but also automated deployment. This adds more elements to the development pipeline, which often in a lean team consists of a version control system and a continuous integration server. Automating the whole process minimizes the time required for implementing new features in software, and allows for faster customer feedback. In this paper we conduct a literature review to investigate the importance of architecture in development processes that implement continuous deployment. We investigate the challenges an architecture typically faces during continuous deployment and during the transition to continuous deployment. We also explore the main issues and attributes required for the architecture of the the pipeline to allow keeping the software in a deployable state at all times. We aim for a coherent view of essential features required to build a development pipeline and adopt continuous deployment in such manner. The results are a set of architectural styles and practices that benefit the process of continuous deployment.**

***Index Terms*—Continuous deployment, Software architecture, Deployment pipeline**

I. Introduction

Continuous deployment is an extension to continuous integration, where the software functionality is deployed frequently at customer environment. While continuous integration defines a process where the work is automatically built, tested and frequently integrated to mainline [1], often multiple times a day, continuous deployment adds automated acceptance testing and deployment. The purpose of continuous deployment is that as the deployment process is completely automated, it reduces human error, documents required for the build and increases confidence that the build works [2].

In an agile process software release is done in periodic intervals [3]. Compared to waterfall model it introduces multiple releases throughout the development. Continuous deployment, on the other hand, attemps to keep the software ready for release at all times during development process [2]. Instead of stopping the development process and creating a build as in an agile process, the software is continuously deployed to customer environment. This doesn't mean that the development cycles in continuous deployment are shorter, but that the development is done in a way that makes the software always ready for release.

It should also be made clear that continuous delivery differs from continous deployment. Refer to Fig. 1 for a visual representation of differences in continuous integration, delivery and deployment. Both include automated deployment to a staging environment. Continuous deployment includes deployment to a production environment, while in continuous delivery the deployment to a production environment is done manually. The purpose of continuous delivery is to prove that every build is proven deployable [2]. While it necessarily doesn't mean that the software is released more often, keeping the software in a state where a release can be made instantly is often seen beneficial.

In this paper we investigate the importance of software architecture, and especially architectural challenges required for the software to enable continuous deployment. In an agile process software architecture is often ignored, citing You Aren't Gonna Need It (YAGNI) and Big Design Up Front (BDUF) [4]. In YAGNI a team might deem architectural features too complicated and costly, and simply come up with the simples thing that could possibly work. In BDUF the software is completely designed before the implementation starts. In a process with continuous deployment, architecture is especially important due to the fact that the software is continuously integrated, deployments are made frequently and often features are developed to the mainline from the beginning. An important research question regarding continuous delivery, posed by Akerele et al., is to investigate the variables in software projects that have a significant impact on the frequent delivery [5]. According to Akerele et al., software architecture is one of the technological factors that affect the stability of the process.

Important architectural quality attributes considering continuous delivery are especially extensibility and flexibility, which are better defined in [6]. Quality attributes are, however, often subjective, and hold a different meaning for different stakeholders. Extensibility generally covers the future growth, which in an incremental development process such as continuous deployment is a primary concern. Flexibility, on the other hand, is used to define how easily a system adapts to varying situations and copes with changes in policies. Often due to an extensible architecture flexibility is also provided.

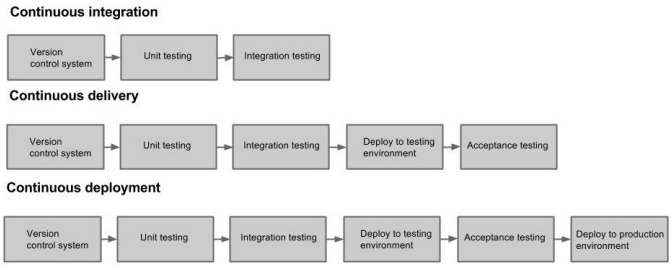

Fig. 1. Continuous integration, delivery and deployment.

As research on software architecture in continuous deployment is next to nonexistent, other areas of research have to be reviewed and applied to continuous deployment. Such areas are for example architectural challenges in agile development [4] and analysis and management of architectural dependencies in iterative release planning [7]. Especially useful are also the research in importance of software architecture during release planning [8] and software release management [9]. The research on these topics can then be applied to continuous deployment, and along with the principles for continuous deployment [10] combined into a coherent view of architectural challenges and features.

Continuous deployment process is concerned with all aspects of deployment, such as configuration management, release management, continuous integration and automated testing. As it has evolved from continuous improvement efforts within the Agile community, but also from prior knowledge on releasing software, it is only natural to investigate these subjects to gain understanding on continuous deployment.

This paper is organized as follows. Chapter II explains reasons behind choosing the used papers, and the used methods in finding the papers. Chapter III explores architectural challenges and methods in release management, release planning and iterative release planning. Chapter III also investigates the deployment production line and the known architectural constraints within. Chapter IV then attempts to apply these findings to continuous deployment, and produces an analysis of the importance of architecture in continuous deployment. Future research is briefly introduced in chapter V, and Chapter VI concludes the paper with the key points.

II. Methods

As continuous deployment is a relatively new area of research, all aspects of it have not yet been investigated. To better understand the architectural challenges in continuous deployment, it is important to research architectural challenges of processes with relations to continuous deployment.

Searches were performed using the keywords shown in Table I. The searches were performed during February and March 2014 using IEEE Xplore (http://ieeexplore.ieee.org/) and Google Scholar (http://scholar.google.com/) search engines. Final papers were downloaded from the publishers web sites, if available. Plenty of research concerning software release management, release planning and iterative release planning were found, but the focus on architectural qualities appeared sparsely.

TABLE I
KEYWORDS USED FOR SEARCHING RESEARCH MATERIALS.

Search string	Search engine	Article
software release management	Google Scholar	Software Release Management
release planning architecture	IEEE Xplore	Importance of Software Architecture during Release Planning
deployment production line	IEEE Xplore	The deployment production line
iterative release planning architecture	IEEE Xplore	Analysis and Management of Architectural Dependencies in Iterative Release Planning

As searches regarding architecture in continuous delivery and deployment returned no results, a decision was made to instead focus on research related to architectural features and issues in releases, and then attempting to apply those findings to continuous deployment. Articles regarding release management, release planning, iterative release planning and deployment pipeline were chosen for this purpose.

III. Results

Release planning primarily plans releases that often occur months out at the start of a project. Iterative release planning on the other hand plans releases for the current iteration, which depending on the team could last from a week to four weeks. Release management is a process of managing often complex and distributed releases throughout development stages to release. In continuous deployment the challenge is to continuously keep the software in a state where it can be released.

This section investigates the architectural issues and features in release management, release planning and iterative release planning, along with some known architectural issues and features in continuous deployment. The knowledge from previous research on release planning proves useful considering continuous deployment, as it is effectively continuously managing to maintain the software in a releasable state.

A. Software release management

Software release management defines the process of managing software releases throughout development stages to release. As in continuous deployment, release managements purpose is to build a bridge between development process and deployment process [9], but it in contrast doesn't define how often releases should be made. Hoek et al. provide a set of requirements in their article Software Release Management [9], which can be used to define architectural qualities in software systems required for releases. While the article mainly touches releases of systems of systems, the research can also be applied to releases of any systems with internal or external dependencies.

TABLE II
REQUIREMENTS IN SOFTWARE RELEASE MANAGEMENT PROCESS [9].

Requirements	
1	Dependencies should be explicit and easily recorded
2	A system should be available through multiple channels
3	The release process should involve minimal effort on the part of the developer
4	The scope of the release should be controllable
5	A history of retrievals should be kept
6	Sufficient descriptive information should be available

Requirement 1 in Table II states that "Dependencies should be explicit and easily recorded". Hoek et al. define the requirement such that it should be possible for a developer to document dependencies as part of the release process, even if the dependencies cross organizational boundaries. Requirement 2 means that once a release is made, it is made available to all environments where it is used. Hoek et al. also elaborate that the release management tool should automatically make it available via such mechanisms as FTP sites and Web pages. A requirement also states that "Sufficient descriptive information should be available". This makes it possible for an user to determine the current version of system in use.

Hoek et al. introduce a prototype software release management tool, Software Release Manager (SRM), which is used to satisfy previous requirements. SRM is composed through a four-part architecture: a logically centralized but physically distributed release database, interfaces for inserting and retrieving components from the database and a retrieve database that records information about the components users have retrieved. The types of artifacts stored in the release database are the released components, metadata describing each component, dependency information for the component and web pages for each component.

B. Release planning

Lindgren et al. investigate the importance of software architecture in release planning in their article entitled Importance of Software Architecture during Release Planning [8]. Bass et al. state that within the research community it is known that early design decisions, which are manifested by software architecture, "...are the most difficult to get correct and the hardest to change later in the development process, and they have the most far-reaching effects" [11].

Lindgren et al. define relevant areas that address software architecture in release planning as identifying possible architectural constraints, maintaining the architecture of evolving systems and identifying if proposed needs introduce architectural conflicts.

The major findings of the articles are that product management generally has low architectural awareness, there is no method for how to balance investments in quality improvements vs. feature growth and that in companies with a software architecture present architectural decisions are made with less "gut-feeling". The presence of a software architect when release planning decisions are made improves the chances that important quality issues are taken into consideration.

C. Iterative release planning

Iterative release planning essentially attemps to plan releases for the current iteration. Brown et al. investigate dependencies between capabilities, including both functional and non-functional requirements, and architectural elements [7]. Brown et al. state that understanding the dependencies between capabilities ensures that a useful feature set is released to the end user. The purpose in investigating the dependencies between capabilities and architectural elements is to implement the architecture in a staged manner. Analysing the dependencies of architectural elements provides information on the refactoring costs that might incur if the architecture is developed incrementally.

Brown et al. state that the visibility of architectural quality can be improved by providing quantifiable quality models of the architecture module structure during system development. This derives from the assumption that using measurable criteria to quantify architecture quality provides guidance for iterative release planning. Brown et al. state that metrics associated with dependency provides data for change propagation, especially in dependencies between architectural elements. The propagation cost can then be used to provide insight into degrading architectural quality to understand when to invest in improving architecture.

Brown et al. calculate end-user value and total implementation cost of each release as $Tc_n = Ic_n + Rc_n$, where Tc_n is the cost of release n, Ic_n is the cost of implementation and is a sum of all architectural elements implemented in release n, where each architectural elements are given an individual cost. Rc_n is the rework cost for release n, and it is calculated by computing the rework cost associated with each new architectural element implemented in n, and multiplying for each existing element with dependencies to that element times the implementation cost of the referencing element, times the propagation cost of release $n-1$. The cost of each element is then added up to form Rc_n.

D. Deployment production line

Humble et al. define four principles that should be followed when attempting to automate the deployment process [10]. Some of these principles transform into architectural constraints. The first principle states that "Each build stage should deliver working software". As software often consists of different modules with dependencies to other modules, a change to a module could trigger builds of the related modules as well. Humble et al. argue that it is better to keep builds separate so that each discrete module could be built individually. The reason is that triggering other builds can be inefficient, and information can be lost in the process. The information loss is due to the fact that connection between the initial build and later builds is lost, or at least causes a lot of unnecessary work spent in tracing the triggering build.

The second principle states that ”Deploy the same artifacts in every environment”. This creates a constraint that the configuration files must be kept separate, as different environments often have different configurations. Humble et al. state that a common anti-pattern is to aim for 'ultimate configurability', and instead the simplest configuration system that handles the cases should be implemented.

Another principle, which is the main element of continuous deployment, is to ”Automate testing and deployment”. Humble et al. argue that the application testing should be separated out, such that stages are formed out of different types of tests. This means that the process can be aborted if a single stage fails. They also state that all states of deployment should be automated, including deploying binaries, configuring message queues, loading databases and related deployment tasks. Humble et al. mention that it might be necessary to split the application environment into $slices$, where each slice contains a single instance of the application with predetermined set of resources, such as ports and directories. $Slices$ make it possible to replicate an application multiple times in an environment, to keep distinct version running simultaniously. Finally, the environment can be smoke tested to test the environments capabilities and status.

The last principle states ”Evolve your production line along with the application it assembles”. Humble et al. state that attempting to build a full production line before writing any code doesn't deliver any value, so the production line should be built and modified as the application evolves.

IV. Discussion

In continuous deployment the software is developed in small, functional releases which are continuously integrated. The architecture and requirements are validated continuously, and an unsuitable architecture is possible to be caught during early prototyping. This eases the burden of the importance of early design decisions mentioned by Lindgren et al. [11], and it doesn't take until release to realize that the software doesn't meet it's non-functional requirements. Often it's the architectural issues that are the hardest to undo, change and refactor [4].

The goal of the development process is to maximize business value. Using the cost model [7] by Brown et al., it is clear that architectural dependencies multiply the rework cost very quickly. Brown et al. also extracted metrics from purely the architecture of an application, further elaborating the importance an architecture has on the projects success.

As software release management is primarily concerned in building a bridge between the development process and deployment process [9], it shares a common goal with continuous deployment. The requirements given by Hoek et al. pose many similarities to the continuous deployment practices, and some of them can be addresses with architectural decisions. The requirement ”A system should be available through multiple channels” and the role of release management tool in automatically deploying the application after a release has been made requires the software management tool to have access to multiple channels. In a continuous deployment process, the deployed software also has to be configured according to the environment. With some packaging applications, such as the J2EE, configurations have to be packaged in the same war or ear file with the rest of the application [2]. The software architecture must in such cases make it possible to package distinct modules with their required configurations in simple file formats.

In release planning it is known that early design decisions play an important role in the development process [8]. As an example in a system with a high performance requirement, requests shouldn't traverse through several tiers. In a continuous deployment process, enough analysis on non-functional requirements has to be made up front to make an informed decision on what architecture to choose for the system [2]. Often an architecture also involves trade-offs between non-functional requirements, and analysis methods such as the Architectural Tradeoff Analysis Method (ATAM) [12] can be used to decide a suitable architecture. The team can then build a set of automated tests to ensure the non-functional requirements are met, and that empirical evidence to refactor and rearchitect a project is provided [2].

The requirement ”Each build stage should deliver working software” by Humble et al. [10] creates an architectural requirement, in which each component should be possible to be upgraded individually. The same principles apply to individual services that form a service-oriented architecture [2]. A component-based architecture with loose coupling makes it easier for teams to develop and collaborate, especially in cases where the team sizes are large and codebases big. A component-based architecture is also effectively flexible, as components can be added and removed with often only having to modify related components. A decision to build the software in components should be made early on, as the time it takes to rearchitecture a large application into discrete components increases as the application grows.

As continuous deployment keeps the software in a state where it can be deployed at all times, incomplete features often pose architectural challenges in keeping the build stable. Shipping semicompleted functionality along with the rest of the application is a good practice, because the entire application is tested and integrated at any time [2]. To implement the application in such manner that features can be included even if they are incomplete requires an architecture that supports it. A common tradition in an agile process is to use branching in version control to develop features, but long lived feature branches are not inline with continuous integrations goal to integrate code to mainline daily. Instead, merging daily with the mainline allows the possibility to build the application with incomplete features allows releases even during the development of major features, and the features will be integrated more often that ”integration hell” is prevented.

Martin Fowler even argues against long feature branches, stating that developing in feature branches without merging with the mainline daily is not continuous integration but continuous building [13]. Fowler quotes Dan Bodart stating

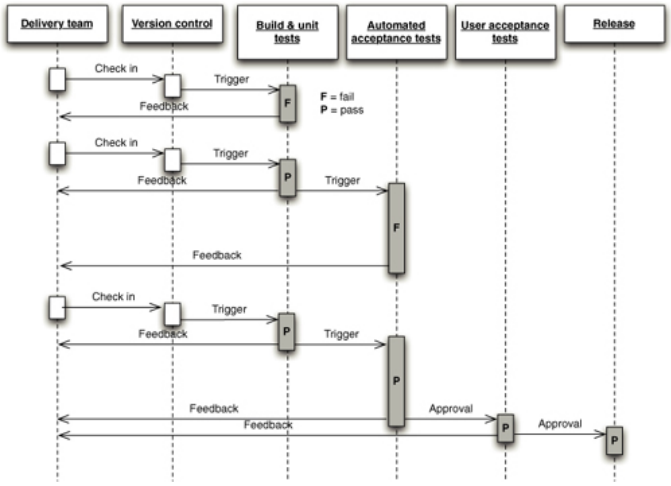

Fig. 2. Architecture of the development process [2].

"Feature Branching is a poor man's modular architecture". In continuous integration an important principle is that commits are made to the mainline daily - this is especially important in continuous deployment as well. Building a modular architecture with loose coupling makes it easier to program in small improvements.

An architectural aspect regarding continuous deployment is also the architecture of the whole development process. A picture of the typical development process in continuous deployment is shown in Fig. 2. After the team pushes a change to the version control system, the project is automatically built and tests are triggered stage by stage. If a test stage fails, feedback is given and the deployment process effectively cancelled. In a continuous delivery process, the last stages are approved and activated manually, but in a continuous deployment process the last stages are triggered automatically as well.

V. Future research

Interesting research question, posed by Lindgren at al., is when and how are architectural issues addressed during release planning. Lindgren et al. are investigating the balance between feature growth and investments in architectural quality [8]. An important question is also who participates in making important architectural questions, and it has been mention by multiple sources [4], [8].

Kruchten presents an issue regarding identifying and resolving architectural issues, stating that the methods to identify architecturally significant requirements, to perform incremental architectural design and to validate architectural features are not well known.

Brown et al. state that the relative weighting and relationship between architectural elements and it's dependencies is still a subject of future research [7]. This affects continuous deployment in the sense that as the impacts of architectural dependencies are better understood, different architectural styles can be analyzed by using dependencies between architectural elements as a quantifiable metric.

The author of this paper will conduct a case study in applying continuous deployment and continuos experimentation practices to B2B context.

VI. Conclusion

Continuous deployment automates the whole deployment process of a software application. This effectively reduces human error, documentation required for deploying and increases the confidence that build works. It is concerned with all aspects of deployment, such as configuration management, release management, continuous integration and automated testing.

Continuous deployment is a process where the software is developed in an incremential manner, and as the whole software is validated continuously, architectural requirements are also validated from an early stage. This makes it possible to validate requirements, and to issue rearchitecturing from an early stage. On the other hand, enough analysis on non-functional requirements has to be made up front to make an informed decision on what architecture to choose for the system.

Architectural challenges posed by continuous deployment deal with keeping the software in a state where it is releasable at all times. As running long-lived feature branches are not essentially a part of a continuous deployment process, the architecture must allow incomplete features to be kept in the mainline. Architectural styles such as loosely coupled modular architecture make this easier.

The deployment pipeline provides architectural challenges in the sense that the application must be deployable and configurable to different environments. The deployment architecture must also automatically validate the stability of the build via a set of staged testing, and provide feedback to the development team in case the deploy has to be rolled back.

References

[1] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www.thoughtworks.com/Continuous Integration. pdf*, 2006.

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[3] A. Cockburn, *Agile software development*. Addison-Wesley Boston, 2002, vol. 2006.

[4] P. Kruchten, "Software architecture and agile software development: a clash of two cultures?" in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2. IEEE, 2010, pp. 497–498.

[5] O. Akerele, M. Ramachandran, and M. Dixon, "System dynamics modeling of agile continuous delivery process," in *Agile Conference (AGILE), 2013*, Aug 2013, pp. 60–63.

[6] S. H. Kaisler, *Software paradigms*. John Wiley & Sons, 2005.

[7] N. Brown, R. L. Nord, I. Ozkaya, and M. Pais, "Analysis and management of architectural dependencies in iterative release planning," in *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*. IEEE, 2011, pp. 103–112.

[8] M. Lindgren, C. Norström, A. Wall, and R. Land, "Importance of software architecture during release planning." in *WICSA*, 2008, pp. 253–256.

[9] A. Van Der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf, *Software release management*. Springer-Verlag New York, Inc., 1997, vol. 22, no. 6.

[10] J. Humble, C. Read, and D. North, "The deployment production line," in *Agile Conference, 2006*. IEEE, 2006, pp. 6–pp.

[11] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[12] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*. IEEE, 1998, pp. 68–78.

[13] M. Fowler. (2014) Featurebranch. [Online]. Available: http://martinfowler.com/bliki/FeatureBranch.html