Pipeline for continuous deployment and continuous experimentation

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. However, adopting continuous deployment doesn't necessarily mean that more value is created for the customer.**

Continuous deployment attempts to deliver an idea to users as fast as possible. Continuous experimentation instead attempts to validate that it is, in fact, a good idea.

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].

An important part of continuous deployment is the deployment pipeline. A deployment pipeline can be loosely defined as a consecutively executed set of validations that a software has to pass such before it can be released. The deployment pipeline is therefore an automated implementation of the application's build, deploy, test and release process [2]. Common components of the deployment pipeline are a version control system and an automated test suite.

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 teams release often, keeping the software in a state where a release can be made instantly is often seen beneficial.

Adopting a continuous deployment process doesn't necessarily mean that more value is delivered to a customer. Avinash Kaushik states in his Experimentation and Testing primer [4] that "80% of the time you/we are wrong about what a customer wants". Mike Moran also found similar results in his book Do It Wrong Quickly, stating that Netflix considers 90% of what they try to be wrong. A way to tackle this issue is to adopt a process of continuous experimentation, where the entire R&D system responds and acts based on instant customer feedback, and where actual deployment of software functionality is seen as a way of experimenting and testing what the customer needs [5].

Continuous deployment attempts to deliver an idea to users as fast as possible. Continuous experimentation instead attempts to validate that it is, in fact, a good idea. In continuous experimentation the organisation runs controlled experiments to guide the R&D process. As the experiments are run in a regular fashion, it is only natural to attempt to integrate experiments to the deployment pipeline, and to change the development process in such fashion that functionality is developed based on some actual data. The operational changes required for a deployment pipeline to support continuous experimentation include

In this paper we're investigating

This paper is organized as follows.

II. Methods

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/)

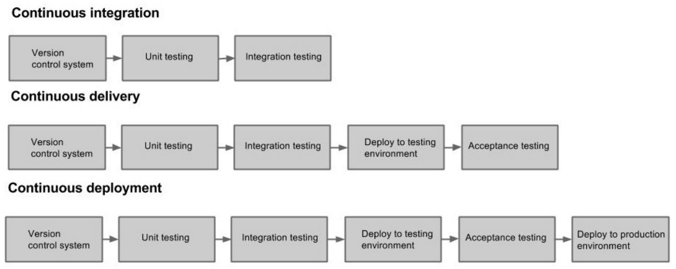


Fig. 1. Continuous integration, delivery and deployment

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

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.

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

A deployment pipeline for continuous deployment consists of the phases a build has to pass in order for it to be deployed. A pipeline for continuous experimentation, on the other hand, consists of the steps required to design, execute and analyse an experiment. The cycle of continuous experimentation resembles the build-measure-learn cycle of lean startup [6]. In continuous experimentation, the main process is to deploy new functionality, measure usage and other performance metrics and subsequently to use the collected data to drive development [7].

Lean startup [6]

Stairway to heaven Olsson [5]

Building products as innovative [7]

Practical guide [8]

Deployment production line [9]

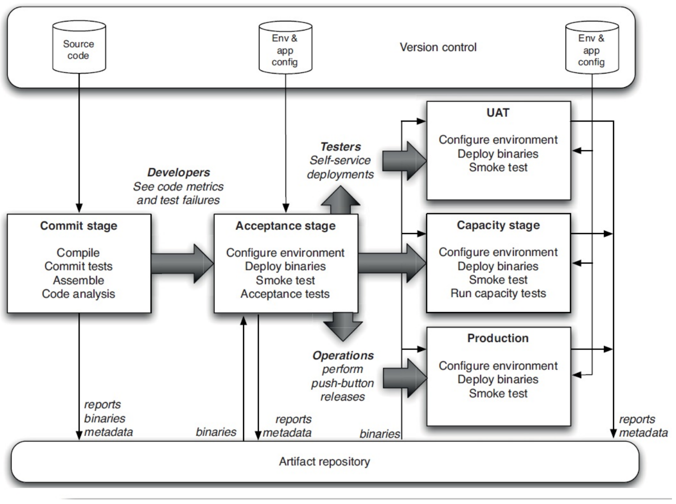


Fig. 2. A basic deployment pipeline [2].

A. Pipeline for continuous deployment

The primary purpose of continuous deployment is to improve the process of software delivary. This is done by automating the whole deployment process, since unless deployments are automated, errors will occur every time they're performed [2]. The components of a deployment pipeline typically are a version control system and an automated test suite consisting of unit tests, integration tests and acceptance tests.

Humble and Farley define the deployment pipeline as a set of stages, which cover the path from a committed change to a build [2]. Refer to Fig. 2 for a graphical representation of a basic deployment pipeline. The commit stage compiles the build and runs code analysis, while acceptance stage runs an automated test suite that asserts the build works at both functional and nonfunctional level. From there on, builds to different environments can be deployed either automatically or by a push of a button.

Humble et al. define four principles that should be followed when attempting to automate the deployment process [9]. 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

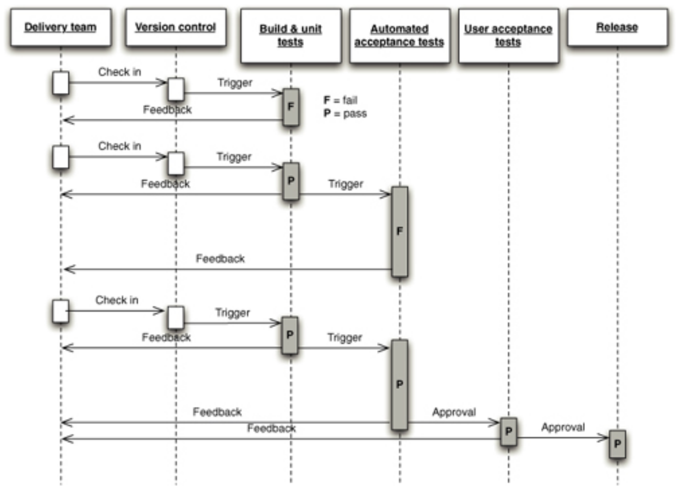
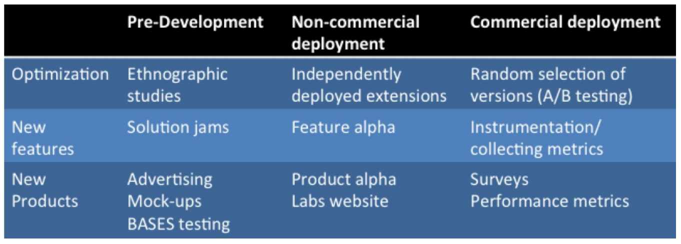


Fig. 3. Components of the development process [2].



	Pre-Development	Non-commercial deployment	Commercial deployment
Optimization	Ethnographic studies	Independently deployed extensions	Random selection of versions (A/B testing)
New features	Solution jams	Feature alpha	Instrumentation/ collecting metrics
New Products	Advertising Mock-ups BASES testing	Product alpha Labs website	Surveys Performance metrics

Fig. 4. Scopes for experimentation [7].

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.

”A deployment pipeline is an automated implementation of your application’s build, deploy, test and release process” [2].

B. *Experimentation*

An experiment is essentially a procedure to confirm the validity of a hypothesis. In software engineering context, experiments attempt to answer questions such as which features are necessary for a product to succeed, what should be done next and which customer opinions should be listened to. According to Jan Bosch, ”The faster the organization learns about the customer and the real world operation of the system, the more value it will provide” [7]. Most organizations have many ideas, but the return-on-investment for many may be unclear and the evaluation itself may be expensive [8]. I

In Lean startup methodology [6] experiments consist of Build-Measure-Learn cycles, and are tightly connected to visions and the business strategy. The purpose of a Build-Measure-Learn cycle is to turn ideas into products, measure how customers respond to the product and then to either pivot or persevere the chosen strategy. The cycle starts with forming a hypothesis and building a minimum viable product (MVP) with tools for data collection. Once the MVP has been created, the data is analyzed and measured in order to validate the hypothesis. To persevere with a chosen strategy means that the experiment proved the hypothesis correct, and the full product or feature can is implemented. However, if the experiment proved the hypothesis wrong, the strategy is changed based on the implications of a false hypothesis.

Jan Bosch has widely studied continuous experimentation, or innovation experiment systems, as a basis for development. The primary issue he found is that ”experimentation in online software is often limited to optimizing narrow aspects of the front-end of the website through A/B testing and inconnected, software-intensive systems experimentation, if applied at all, is ad-hoc and not systematically applied” [7].

Fig. 4 introduces different stages and scopes for experimentation. For each stage and scope combination, an example technique to collect product performance data is shown. As startups often start new products and older companies instead develop new features, experiments must be applied in the correct context. Bosch states that for a new product deployment, putting a minimal viable product as rapidly as possible in the hands of customers is essential [7]. After the customers can use the product, it is often not yet monetizable but is still of value to the customer. Crook et al. investigated the common pitfalls encountered when running controlled experiments on the web [10], which are listed in Table II. The Overall Evaluation Criteria (OEC) used in the first pitfall is a quantitative measure of the experiment’s objective. In experimental design, Control is a term used for the existing feature or a process, while a Treatment is used to describe a modified feature or a process. As a motivator for the first pitfall, Crook et al. introduce an experiment where the OEC was set to the time spent on a page which contains articles. The OEC increased in the experiment implementation, satisfying the objective. However, it was soon realized that the longer time spent on a page might have been caused by confusion of the users, as a newly introduced widget was used less often than a previous version of it.

Aside from picking a correct OEC, common pitfalls deal with the correct use of statistical analysis, robot users such as search engine crawlers, and the importance of audits and control.

Experimentation doesn’t necessarily require development of software.

TABLE II. PITFALL TO AVOID WHEN RUNNING CONTROLLED EXPERIMENTS ON THE WEB [10].

Pitfall 1	Picking an OEC for which it is easy to beat the control by doing something clearly wrong from a business perspective
Pitfall 2	Incorrectly computing confidence intervals for percent change and for OECs that involve a nonlinear combination of metrics
Pitfall 3	Using standard statistical formulas for computations of variance and power
Pitfall 4	Combining metrics over periods where the proportions assigned to Control and Treatment vary, or over subpopulations sampled at different rates
Pitfall 5	Neglecting to filter robots
Pitfall 6	Failing to validate each step of the analysis pipeline and the OEC components
Pitfall 7	Forgetting to control for all differences, and assuming that humans can keep the variants in sync

C. Pipeline for continous experimentation

A system with continuous experimentation must be able to release minimum viable products with integrated data collection instruments in a rapid manner.

Kohavi et al. investigate the practical implementations of controlled experiments on the web [8], and state that the implementation of an experiment involves two components. The first component is a randomization algorithm, which is used to map users to different variants of the product in question. The second component is an assignment method which, based on the output of the randomization algorithm, determines the contents that each user are shown. The observations then need to be collected, aggregated and analyzed to validate a hypothesis. Kohavi et al. also state that most existing data collection systems are not designed for the statistical analyses that are required to correctly analyze the results of a controlled experiment.

The components introduced by Kohavi et al. are aimed primarily for A/B testing on websites.

IV. DISCUSSION

The quantitative measure must be correct (pitfall)

What kind of elements need to be added to the deployment pipeline based on these papers?

The pipeline for continuous deployment

The pipeline for continuous experimentation

Based on the features required by .., a set of restrictions can be ..

Restrictions 1 Running an experiment should be optional, and deploying without experimenting must be made possible

V. FUTURE RESEARCH

VI. CONCLUSION

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] [Online]. Available: http://www.kaushik.net/avinash/experimentation-and-testing-a-primer/

[5] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, pp. 392–399.

[6] E. Ries, *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Random House LLC, 2011.

[7] J. Bosch, "Building products as innovation experiment systems," in *Software Business*. Springer, 2012, pp. 27–39.

[8] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007, pp. 959–967.

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

[10] T. Crook, B. Frasca, R. Kohavi, and R. Longbotham, "Seven pitfalls to avoid when running controlled experiments on the web," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 1105–1114.