A Practical Approach to Software Continuous Delivery

Everton Gomede and Rodolfo M. Barros
Department of Computer Science
State University of Londrina
Londrina, Paraná, Brazil
e-mail: evertongomede@gmail.com, rodolfo@uel.br

***Abstract*—To deliver quality software continuously is a challenge for many organizations. It is due to factors such as configuration management, source code control, peer-review, delivery planning, audits, compliance, continuous integration, testing, deployments, dependency management, databases migration, creation and management of testing and production environments, among others. To overcome these challenges, this work-in-progress paper presents a continuous delivery process that promotes artefacts produced by developers, in a managed fashion, to production environment, allowing bidirectional traceability between requirements and executables. As a preliminary result, we obtained an ecosystem of tools and techniques evaluated, tested and put into production in order to support this process.**

Keywords–Continuous Delivery; Process Quality.

I. INTRODUCTION

Software Delivery Process (SDP) consists of several tasks in order to promote artifacts created in the production environment (server(s) where an executable is installed to delivery features to the users) [1]. These ones can occur in either environment, producer or consumer. Due to the unique characteristics of each software product, a general process to various contexts probably cannot be set. Therefore, we should interpret a SDP as a *framework* to be customized according to the requirements and characteristics of each product (Software Delivery Process, in this context, is a part of Software Development Process).

This customization usually causes a *manual* execution of SDP [2]. Production environment is configured in a manual way by the infrastructure team using terminals and/or third-party tools. Artifacts are copied from a continuous integration server to a production environment and possibly some data and/or metadata are adjusted before software is released for operation.

However, this process has some weaknesses. Predictability is the first one, because it increases risk and downtime whether any unexpected situation occurs [3]. Additionally, the repeatability factor may compromise the diagnosis of post-deployment problems [2]. Finally, this process is not auditable and it does not allow the recovery of information about all events that were held to deliver a version.

There is a growing interest in practices to overcome these problems [4]. Such practices are known as *Software Continuous Delivery* (SCD), defined as the ability to publish software whenever necessary. This publication may be weekly, daily or every change sent to the code repository. The frequency is not important, but the ability to deliver when it is necessary [2].

This approach has great importance in software development because it helps who is in charge of delivering to understand better their process and, consequently, improve it. Such improvements can be in terms of automation, decrease delivery time, rework reduction, risk reduction, or others. Among them, the main is the ability to have a version of software, ready for delivery, each new code added to the repository.

In this context, this paper presents a practical approach to address the problems of software continuous delivery. The main objective is to contribute with a setup of servers, process, techniques and tools that assist to deliver software continuously. In addition, some recommendations and further work are discussed. Architecture issues, project management and other dimensions related to software development were omitted. We collect these data through of a case study.

Thus, this article was divided into five sections, including this introduction. In Section II, we present fundamental concepts and related works. In Section III, we present an approach to Software Continuous Delivery. In Section IV, we present preliminary results. Finally, in Section V, we present conclusions, recommendations and suggestions for future work.

II. FUNDAMENTAL CONCEPTS AND RELATED WORKS

There is a relation between quality of software products and quality of the process used to build them. Implementation of a process aims to reduce rework, delivery time and increase product quality, productivity, traceability, predictability and accuracy of estimates [2]. In general, a software development process contains activities shown in Figure 1.

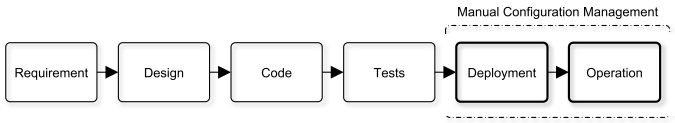

Figure 1. A simplified software development process [1] [2].

Configuration management tasks of deployment and operation activities, highlighted in Figure 1, are usually performed manually [2]. This practice, according to Humble and Farley [2], is accompanied by anti-patterns:

- Deploying software manually: there should be only two tasks to perform manually; (1) choose a version and (2) choose the environment. These are goals to be achieved in process like in [5].
- Deploying after development (requirement, design, code and tests) was complete: it is necessary to integrate all activities of the development process and

put stakeholders working together since the beginning of the project.

- Manual configuration management of production environments: All aspects of configured environments should be applied from a version control in an automated process.

In this context, some Software Continuous Delivery Practices arises. It is a developing discipline, which builds up software that can be released into production at any time, by minimizing lead-time [3].

To assist this type of software delivery approach, from construction to operation, Humble and Farley presents the *Deployment Pipeline* (DP), a standard to automate the process of SCD. Despite each organization may have an implementation of this standard, in general terms it consists of activities shown in Figure 2.

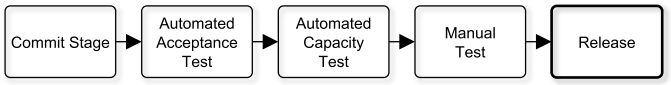

Figure 2. The deployment pipeline [2].

With each change, artifacts are promoted to next instance of pipeline through a series of automated tasks. The first step of the pipeline is to create executables and installers from the code repository, in a process known as Continuous Integration (CI). Other activities perform a series of tests to ensure that the executable can be published. If the release candidate passes all tests and criteria, then it can be published [2].

To implement this pipeline, some approaches were presented. Among them, Krusche and Alperowitz [5] described the implementation of a SCD process for multiple project. Their goal was to obtain the ability to publish software to their clients with just a few clicks. The main contribution of this work was to show that developers who have worked on projects with SCD, understood and applied concepts being convinced from the benefits of it.

Bellomo et al. [6] presented an architectural framework together with tactics to projects that address SCD. The main contribution of this work is a collection of SCD tactics in order to get software products performing with a higher level of reliability and monitoring into production environment.

Fitzgerald and Stol [4] published trends and challenges related to what the authors called ”Continuous *”, which is, all topics related to software delivery that can be classified as continuous. The authors addressed issues such as; Continuous Integration (CI), Continuous Publication (PC), Continuous Testing (CT), Continuous Compliance (CC), Continuous Security (SC), Continuous Delivery (EC), among others. An important point of this paper is the distinction between the Continuous Delivery and Continuous Publication. According to the authors, Continuous Publication is ability to put into production software products in an automated manner. This definition is complementary to the software continuous delivery definition given above.

Although all these works have a practical nature, none of them showed which tools were used, which recommendations for similar scenarios and which were the techniques used during deployment. Therefore, the work presented in this paper seeks to fill these gaps.

III. A Practical Approach

A. Main Proposal

To provide an infrastructure that allows the Software Continuous Delivery is the main goal of setup shown in Figure 3. It has 4 areas: Commit Stage (CS), Quality Assurance (QA), Staging (ST) and Production (PD).

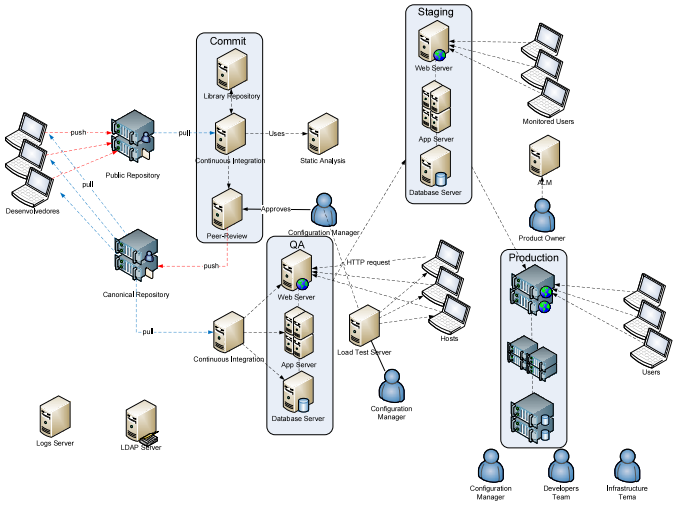

Figure 3. An overview of a setup of servers and areas.

B. Areas

Commit Stage (CS) has primary responsibility to implement continuous integration of all code reviews sent to the repository. This area consists of the following services:

- Public Code Repository
  - Purpose: to get code reviews that have not been approved.
  - Tool: Git (git-scm.com).
  - Technique: it has a single branch, called master, which receives revisions of all developers.
- Continuous Integration
  - Purpose: to integrate all code reviews sent to the server.
  - Tool: Jenkins (jenkins-ci.org) and Maven (maven.apache.org)
  - Technique: it does integration performing unit testing and adding first acceptance step in peer-review server.
- Static Analysis
  - Purpose: to make code analysis generating quality reports.
  - Tool: SonarQube (sonarqube.org).
  - Technique: each integration performs a series of tests, such as size metrics, complexity, test coverage, dependency calculation, among others. Creates a baseline quality of the project.
- Peer-Review
  - Purpose: to enable promotion/rejection of codes from public to canonical repository.

- Tool: Gerrit (code.google.com/p/gerrit).
- Technique: approval of two steps, the first being carried out by continuous integration server and the second by the configuration manager. If the review through both sides, code is promoted to canonical repository.

- Canonical Repository
  - Purpose: to receive approved code reviews.
  - Tool: Git (git-scm.com).
  - Technique: it has a single branch, called master, which receives revisions of peer-review server.
- Repository Libraries.
  - Purpose: to store libraries and components used in integration.
  - Tool: Nexus (sonatype.org/nexus).
  - Technique: libraries and components are installed automatically or manually on the server being available for use at the time of integration.

Layout and operation of Commit Area are shown in Figure 4.

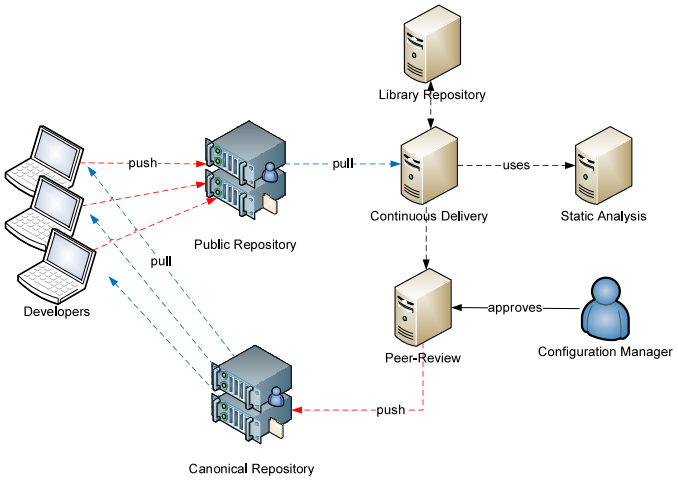

Figure 4. Commit Stage (CS).

Quality Assurance Area (QA) has the main purpose of performing all automated tests and allow Quality Manager perform manual tests, such as exploratory testing [2]. This area consists of the following services:

- Continuous Integration
  - Purpose: to obtain a copy of the code and perform integration, functional and automated load tests.
  - Tool: Jenkins and Maven (maven.apache.org).
  - Technique: get a copy of canonical repository to generate executable, install them into library server, application servers and database server. After that, execute integration, functional and load tests.
- Page Servers, Application and Database
  - Purpose: to host application to test
  - Tools: may vary according to the technology used; Wildfly and MSSQL are some examples.
  - Technique: can vary depending on the technology used (to install and configure, basically).
- Load Test
  - Purpose: to perform a load test against the page servers, application and database.
  - Tool: Jmeter (jmeter.apache.org) and Vagrant (vagrantup.com).
  - Technique: it performs script created by quality manager allocating hosts as required to test. It generates a supported load from baseline.

Operation of Quality Assurance area is shown in Figure 5.

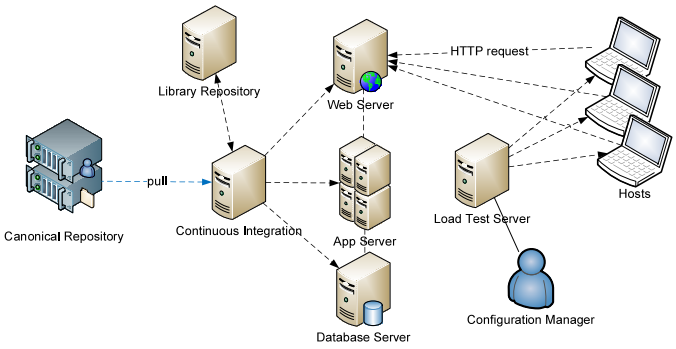

Figure 5. Quality Assurance (QA).

Staging Area aims to provide for monitoring users and product owners an environment as close as possible to production environment, so they perform approval tests. These ones are related to user experience and their perception regarding how software product meets specified requirements. This area has a copy of operating environments, both in terms of operating systems, tools and settings, and in terms of data. Monitored users are the ones chosen to perform approval tests in a monitoring way. Occasionally, they are in the product owner role. Figure 6 shows this area.

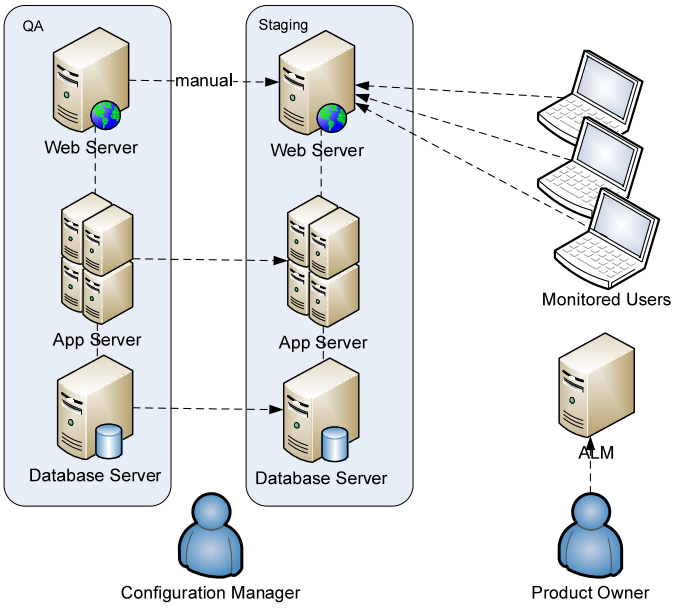

Figure 6. Staging (ST).

Finally, configuration manager makes promotion from Staging Area artifacts to Production Area manually by Configuration Manager. However, developers and infrastructure staff are present to perform this task. Figure 7 shows this area.

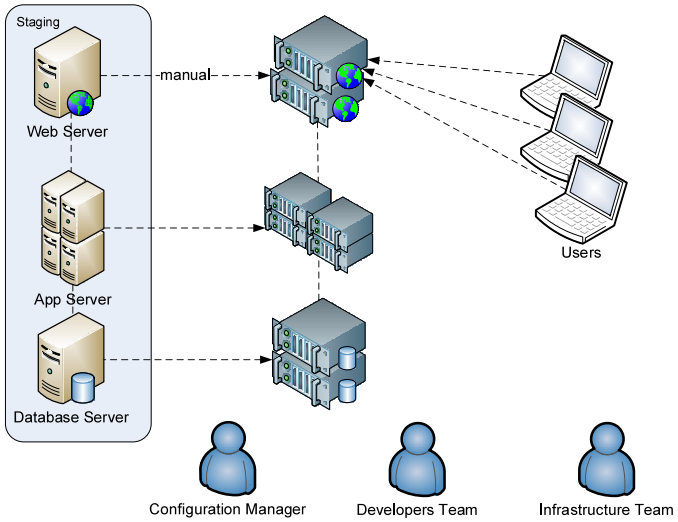

Figure 7. Production (PD).

Also, the following servers were used: (1) Log Server and (2) LDAP Server. The first has a very important function in the setup; to get all events occurred by indexing logs. This assists the diagnosis, providing information for reporting, alerts and dashboard. The tool used in this case is Splunk. The second server has a function to allow authentication and authorization for all setup servers. This is necessary because it is costly to maintain users across all the servers involved in an individualized way, in addition this increase security flaws. The tool used in this case is OpenLDAP (openldap.org).

C. Tools

Table I summarizes all tools used with its URL. These tools are used to Configuration Management (Git, Gerrit, Nexus, Flywaydb and Vagrant), Continuous Integration (Jenkins and Maven), Quality Assurance (SonarQube and Jmeter), Application Lifecycle Management (Redmine) and infrastructure (Splunk and OpenLDAP).

TABLE I. TOOLS USED.

Goal	*Name*	*URL*
Continuous Integration	Jenkins	jenkins-ci.org
Source Repository	Git	git-scm.com
Build	Maven	maven.apache.org
Gathering Logs	Splunk	splunk.com
Peer-Review	Gerrit	code.google.com/p/gerrit
Static Analysis	SonarQube	sonarqube.org
Load Test	Jmeter	jmeter.apache.org
Library Repository	Nexus	sonatype.org/nexus
Application Lifecycle Management	Redmine	redmine.org
Database Migration	Flywaydb	flywaydb.org
Automated Installation	Vagrant	vagrantup.com
Authentication and authorization	OpenLDAP	openldap.org

These tools were used because they are open/free software.

IV. RESULTS

Preliminary results about this approach are related to automation of many delivery tasks, resulting in a more predictable and managed process. Another aspect, related to collaboration, is due to communication between developers and infrastructure team was increased in all aspects of the process, since planning of a feature until its publication. These results are classified in a process maturity level [2], as shown in Figure 8.

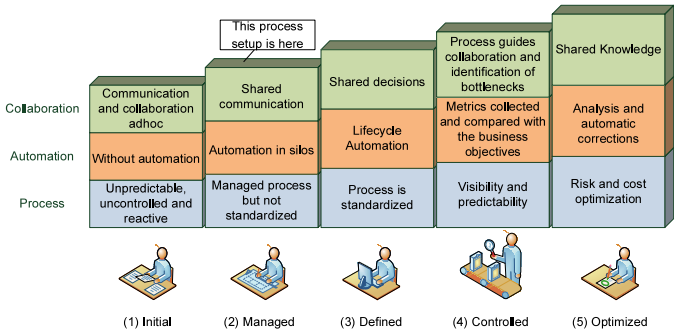

Figure 8. Process maturity level [2].

V. CONCLUSION

This work presents a practical approach that can be used in similar processes. Additionally, among the contributions can be mentioned (1) a set of tools evaluated and (2) a set of techniques, that can be used for organizations that do not use this type of approach, as for those which already have a higher level of maturity.

Moreover, some further work may be developed to improve setup provided in this article. The first one aims to get a strategy for publication with less impact in terms of unavailability of software products, including deployment across different timezones. The second one is linked with multiple projects scenarios. We can analyze how the artifacts, from several projects, are promoted to production by the same team.

Finally, this article has a practical purpose. However, to implement continuous delivery involves more than installing some tools and automate some tasks. It depends on effective collaboration among all of those involved in the delivery process, senior management support and especially the desire of stakeholders in become the changes a reality.

REFERENCES

[1] M. V. Mantyla and J. Vanhanen, "Software Deployment Activities and Challenges - A Case Study of Four Software Product Companies," 2011 15th European Conference on Software Maintenance and Reengineering, Mar. 2011, pp. 131–140.

[2] J. Humble and F. David, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. ser. Addison-Wesley Signature Series. Pearson Education, 2010.

[3] T. Ernawati and D. R. Nugroho, "IT Risk Management Framework Based on ISO 31000:2009," International Conference on System Engineering and Technology, vol. 11, 2012, pp. 1–8.

[4] B. Fitzgerald, "Continuous Software Engineering and Beyond : Trends and Challenges Categories and Subject Descriptors," RCoSE 14, 2014, pp. 1–9.

[5] S. Krusche and L. Alperowitz, "Introduction of Continuous Delivery in Multi-Customer Project Courses Categories and Subject Descriptors," ICSE Companion 14, 2014, pp. 335–343.

[6] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, "Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun. 2014, pp. 702–707.