We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

5,300
Open access books available

130,000
International authors and editors

155M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter 3

# Continuous Anything for Distributed Research Projects

Simon Volpert, Frank Griesinger and
Jörg Domaschka

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.72045

**Abstract**

International research projects involve large, distributed teams made up from multiple institutions. These teams create research artefacts that need to work together in order to demonstrate and ship the project results. Yet, in these settings the project itself is almost never in the core interest of the partners in the consortium. This leads to a weak integration incentive and consequently to last minute efforts. This in turn results in Big Bang integration that imposes huge stress on the consortium and produces only non-sustainable results. In contrast, industry has been profiting from the introduction of agile development methods backed by Continuous Delivery, Continuous Integration, and Continuous Deployment. In this chapter, we identify shortcomings of this approach for research projects. We show how to overcome those in order to adopt all three methodologies regarding that scope. We also present a conceptual, as well as a tooling framework to realise the approach as Continuous Anything. As a result, integration becomes a core element of the project plan. It distributes and shares responsibility of integration work among all partners, while at the same time clearly holding individuals responsible for dedicated software components. Through a high degree of automation, it keeps the overall integration work low, but still provides immediate feedback on the quality of the software. Overall, we found this concept useful and beneficial in several EU-funded research projects, where it significantly lowered integration effort and improved quality of the software components while also enhancing collaboration as a whole.

**Keywords:** Continuous Delivery, Continuous Integration, Continuous Deployment, project management, software quality, DevOps, distributed software development

## 1. Introduction

The rise of agile software engineering strategies has leveraged the realisation of Continuous Integration proposed by Grady Booch as early as 1991 [18]. Continuous Integration propagates a constant integration of changes to code as opposed to Big Bang integration done at the end

IntechOpen

© 2018 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. (cc) BY

of a development cycle. It has, in turn, paved the path to Continuous Delivery and Continuous Deployment of software components and entire software platforms. The core idea of Continuous Delivery is to be able to roll out new releases at any time and not only at the end of larger development cycles. In order to reach this goal, Continuous Delivery demands the automation of all steps required to compile, bundle, test, and release the software. Testing ranges from unit tests targeting a single software component, over integration tests, acceptance test to user acceptance tests. While this approach is emergently successful in industry, it is barely used for scientific software, neither is it used in collaborative research environments.

In contrast to industrial projects and even open source software projects, distributed research projects (for instance, large(r) EU-funded ICT projects), the project itself is almost never in the core interest of the partners forming the project consortium. Instead, every partner is interested in the niche aspect that made him join the project and that makes the consortium look complete and gives the consortium a complementary appearance. In reality, though, the agendas of the project partners are often driven by their individual interests and particularly academic partners do have an obvious interest in the actual research aspect of the work and are less focussed on the provisioning of dependable, sustainable software artefacts. Neither are they preliminary interested in the common, integrated, stable software platform. In practise, this may mean that Partner A wants to improve on an algorithm they have, while Partner B would define a domain specific language (DSL) for a specific scope and Partner C will provide an improved kernel module for handling I/O on solid state disks. From a research point of view, this means that the main work for Partner A will be on the definition of the algorithm, its implementation, and evaluation in some limited, publishable scope. For Partner B, the main work will be on the definition of the DSL and on applying and realising it in two or three use case scenarios. Partner C's work will be on the definition of the new approach and on the realisation and evaluation of the kernel module, probably for one specific version of Linux.

While the behaviour of all three partners is fully legitimate and understandable, it is the nature of a distributed research project that interdependencies between parts of the software exist. Usually software integration is required at certain project milestones where prototypes should be released and new, emergent features be demonstrated to the public or at least the funders. Here, the lack of common interest in the project in combination with the described "research style" code quality makes the integration a painful, cumbersome, and frustrating task. Our experience shows that in many projects the task of integrating software from different partners is outsourced in an own project work package and then assigned to one or at most two partners that were not or only marginally involved in improving the algorithm from Partner A, developing the DSL from Partner B, and realising the kernel module of Partner C. Furthermore, in many projects the whole integration of all software components is done in a Big Bang style before a review or before an obligatory software release and even worse often performed by a single individual. This poor devil ends up integrating and fixing several dozen software components (s)he has not developed, is not owning, and has never been responsible for.

We argue that instead of putting all integration responsibility and work on the shoulders of a single individual, it is way better to spread out work among project partners and make it everybody's task. We further believe that the techniques and strategies offered by **Continuous Integration**, **Continuous Delivery**, and **Continuous Deployment** are beneficial for enforcing the distribution

of the task, automating the necessary steps, monitoring the status of, and gaining confidence in the produced software. The main contribution of this chapter is a concise technical and organisational framework for Continuous Integration, Continuous Delivery, and Continuous Deployment in distributed (research) projects. The framework is based on our experience in half a dozen mid-sized EC projects of 5–25 partners and several smaller sized national-funded research projects. It fairly distributes work among partners and improves overall code quality.

The rest of this chapter is structured as follows. Section 2 identifies the requirements in more detail and presents related work. Sections 3–5 introduce background on Continuous Integration, Continuous Delivery, and Continuous Deployment, respectively. Section 6 presents our framework on both conceptual and tooling level while Section 7 discusses the approach. Section 8 concludes and gives an outlook on future work.

## 2. Problem statement and related work

International ICT research projects involve large, distributed teams (consortia) made up from multiple companies or institutions, so called partners or beneficiaries. These teams create research software artefacts that need to work together in order to demonstrate and ship the project results. In the following, we analyse the challenges of such constellations, and why this requires a special integration strategy. Finally, we carve out the requirements towards such an integration strategy and discuss related work.

### 2.1. Challenges with distributed research teams

Development in distributed, that is, non co-located, teams is challenging, as the distribution aspect hinders communication. For instance, meetings and synchronisation actions barely can happen in a timely or even ad hoc manner, causing delays. From a technical point of view this may lead to diverging developments at different locations. From an organisational point of view, it causes overhead.

Koetter et al. [10] identify major problems in distributed teams and particularly with respect to research projects. At the core of their analysis, they identify the team distribution and lacking stakeholder commitment as major problems. The former complicates team communication leading to a lack of internal communication. The latter, a consequence of diverging goals and different (research) interests, leads to a lack of incentives for prototype integration.[1] The impact of these causes is further increased by different cultural and technical background, etiquette, company policies, and high personal fluctuation in research projects.

In order to cope with the diversity and resulting centrifugal forces, it is common that project management applies rules: these range from a common toolset and document template to regular virtual and physical meetings; both intended to improve communication. Additionally,

[1]Please note that “integration” as in “Continuous Integration” has a different meaning than “prototype integration”. As defined later, the first tackles integration on code level, while the later addresses the integration of distributed software architecture. In order to avoid misunderstandings, we will always use the full terms.

most projects announce a central technical responsible whose role is to break ties in technical discussions. Finally, technical work is often separated such that local teams at partner sites work independently on certain sub topics producing isolated assets.

From our experience, these measures usually work fine and minimise the tension in the consortium. The lack of common goals usually gets masked by introducing a storyline every partner can agree on. Yet, we claim that the only aspect that cannot be handled by these measures is the work to be done for prototype integration, because it requires that components developed in isolation, work together smoothly despite the weak communication, and common goals. The often-practised Big Bang integration of artefacts causes a lot of work, troubles the consortium, and results in poor software quality.

### 2.2. The cause for Continuous Anything

Understanding and accepting that Big Bang integration causes pain and sub-optimal results, leads to the insight that a different prototype integration strategy is needed. Ironically, software development industry was facing similar issues decades ago [16] which led to abandoning of the waterfall model and the introduction of so called agile development methodologies. These were stated in the agile manifesto [17] and are being realised by methodologies such as extreme programming, Scrum, or Kanban.

All of these methodologies assume co-located teams with large common interests and a high intrinsic motivation to deliver high quality, usable software. In consequence, they cannot be applied directly to research projects that do not fulfil the necessary preconditions. Nonetheless, at the core of their prototype integration[2] methodology, agile methodologies rely on a highly automated, frequently executed, and constant process to reduce the possibility for human errors and to obtain continuously executable software artefacts.

While such an approach requires an upfront and constant invest in prototype integration, the overall amount of effort needed per partner and particularly per consortium is likely to be a lot less compared to Big Bang integration. This is due to the fact that changes are small and can be easily reviewed. Moreover, the use of automation allows dealing with the complexity of even larger and more diverse teams.

### 2.3. Constraints and requirements

We claim that automation can reduce the pain for prototype integration in (large) research projects. Yet, as with improving communication within the consortium, introducing an automated process, this improvement will not happen for free. Work from the project management is needed to establish and enforce such a process, which may cause resistance.

Therefore, our major aim is to minimise the upfront investment of project partners and the management effort needed to enforce the strategy. The overall goal is to develop an automated prototype integration schema that takes into account the specific needs of research consortia. This is broken down into particular requirements and challenges presented in the following sections.

[2]For industry it should rather be "product integration".

#### 2.3.1. General requirements

This section covers requirements towards automating prototype integration. We present them from the perspective of a research project, but they can be applied in different settings.

**R.1 (distribute work among partners):** As partners do not have common interest and no incentives for prototype integration, it is necessary to not burden one team or even one individual with this task, but to achieve a fair distribution of work among partners.

**R.2 (reduce manual burden):** Integration work distracts researchers from their work and so does testing. Due to that, as much as possible of the prototype integration work and testing should be automated. This includes reporting if code is currently working or not.

**R.3 (denote responsible persons):** With high automation degree and the capability to identify non-working portions of the code, denoting responsible persons for individual software components, libraries, and features, allows explicitly tasking those for fixing the non-working parts of the architecture.

**R.4 (make the product easy to start and use):** Having a project outcome that is easy to install, to start, and to demonstrate, tremendously reduces the burden when planning for a review, a demo, or a webinar.

**R.5 (clarify big picture and software dependencies):** In large software projects, it is often the case that the big picture is forgotten. In particular, in research projects, researchers lose themselves in details of research questions. Hence, it is important to keep an eye on the overall architecture and ensure that the interactions between components work as intended.

**R.6 (make the software status visible):** Making the product (including its sub-products) visible helps consortium members to understand what the others are doing. It also helps use case partners giving feedback on the project and the work currently done.

#### 2.3.2. Specific challenges of research projects

Besides the generic requirements that can be found in many distributed teams, the fact that distributed research projects are often executed by loosely coupled beneficiaries creates further technical challenges.

**R.A (cater for closed code or even unavailable source code and binaries):** Due to different commercial interests of partners, it may be the case that some of them release source code only in a restricted manner or not at all. In some cases, partners do not even release binaries to the rest of the consortium. The overall prototype integration strategy has to be able to deal with this.

**R.B (support fluctuation of team members):** Research projects have a very high fluctuation of team members. As the budget is fixed, it happens that more people come into the project towards the end, if budget is still available. On the other hand, if the project is short on budget, expensive, that is, senior researchers are moved away and juniors or even undergrads join. This forbids that there are hidden, that is, implicit, agreements between individuals.

**R.C (keep track of targeted outcomes):** The fluctuation of team members and the dynamic of IT research lead to often changing technical goals of the research project. As a consequence, the current goals need to be documented and be accessible by all project members in order to keep a common focus. Ideally, they are immediately visible to all contributors.

**R.D (support different configurations):** Often research projects do not build generic solutions, but only demonstrators for specific use cases. The prototype integration strategy has to be able to deal with this and provide the capability to set-up different environments.

**R.E (allow different programming languages and development methodologies):** Research projects barely start from scratch, as many partners continue earlier work. Further, the knowledge and suitability of languages is very specific to problem domains. Therefore, a prototype integration strategy has to be open to different programming languages.

### 2.4. Approach

In industrial contexts solving integration and communication issues is realised by introducing three kinds of orthogonal, but complementary approaches: *Continuous Integrati*on handles the integration on code level per component. It builds and tests the component whenever a new version of the code is available. *Continuous Delivery* is concerned with taking the new version of a component and packing it in a shippable box. In addition, it runs integration tests to ensure the interplay with other components. Finally, *Continuous Delivery* takes the component and installs it in a pre-defined environment.

In Sections 3–5, we show that an adapted process to Continuous Integration, Continuous Delivery, and Continuous Deployment can indeed overcome integration issues for distributed research projects. In addition, Section 6 presents a set-up that is able to deal with the requirements from Section 2.3.

### 2.5. Related work

While there is a lot of literature on DevOps [15], agile methods, Continuous Integration, Continuous Delivery [13], and Continuous Deployment not much can be found with respect to academia and academia/industry collaboration. Eckstein provides guidelines for distributed teams [11].

Rother [1] lays part of the foundation of what is today perceived as DevOps by presenting the production pipelines and methodologies of Toyota. Being more on the cultural side of DevOps and CI/CD spectrum, Davis and Daniels [14] and Sharma [12] give some insights on how to bring these ideas to industry.

Especially Continuous Integration was significantly influenced by Duvall et al. [2]. There, the authors describe most of the paradigms important for Continuous Integration. These are still valid today and are considered as de-facto standard. Fowler's influential articles on Continuous Integration, for example [5], and testing, for example, through micro-service scenarios [4], lay the foundation on what is being perceived as Continuous Integration along with best practices.

Regarding academia, there is ongoing effort in bringing Continuous Integration and Continuous Delivery to teaching. Eddy et al. [7] describe how they implement a pipeline for supporting their lecture on modern development practices. An academic view on Continuous Experimentation is brought up by Fagerholm et al. [9] by investigating multiple use-cases of industry partners. They analyse the demands and propose solutions to create experimentation-happy environments utilising Continuous Integration and Continuous Delivery.

On Academia/Industry collaboration Sandberg and Crnkovic [6] and Guillot et al. [8] investigate challenges between those parties and how to solve them with agile methods. Both analyse the adaption of the rather strict scrum methodology on said collaboration in multiple case studies with positive results. However, also that approach is highly dependent on team agreement.

Koetter et al. analyse the characteristics and problems of software development in distributed teams in research projects [10]. They give a literature review of common problems and typical solutions. With the focus on Software Architecture, the authors summarise the issues and sketch solution approaches on a methodological level.

## 3. Background: Continuous Integration

This section gives an introduction into the concepts of Continuous Integration. The next subsection defines the scope of the methodology and gives a definition. Later sub-sections introduce the general concept and the Continuous Integration loop in more detail and introduce basics to testing and best practises.

### 3.1. Definition and scope

Continuous Integration describes a methodology to always have the latest successfully built and tested version of a software component available [2]. At its core, it aims at removing diverging developments of different developers by enforcing that all the code changes of every developer are integrated with each other to a shared mainline “all the time” (hence, it focuses on the integration of code of a single build artefact). Integrating small changes at high frequency reduces the chance of diverging code and the pain of code integration.

In Continuous Integration, the process of building and testing the component is usually described by scripts and hence, easy to reproduce by any developer and easy to automate. In consequence, it overcomes the issue of hard-to-reproduce builds that is a reoccurring problem in traditional development environments where developers usually have their code being built and run inside their different IDE in terms of version or even brand.

### 3.2. The integration pipeline

A successful adoption of Continuous Integration in any environment has to rely on automation in order to achieve a permanent feedback loop. This is illustrated in **Figure 1**. At some point in

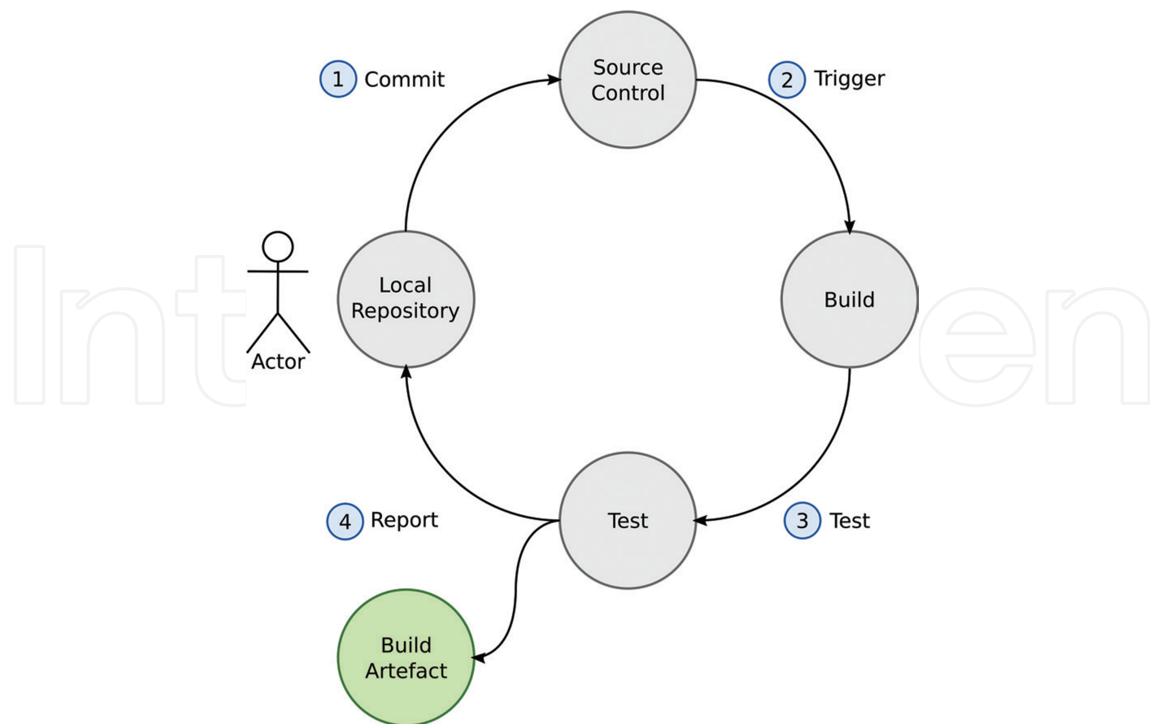


**Figure 1.** The Continuous Integration feedback loop realising the integration pipeline.

time, developers working on a local version of the code will be finished with their work, for example, a new feature or a bug fix. Then, they commit (step i) their changes to the version control system shared by all developers of that component. In addition to the code, the repository contains additional data and procedures to build and also test the software.

Accordingly, a new commit triggers (step ii) a new build of the software component. In case the build is successful (step iii), tests of the code get executed. Here, build automation enables that both building and testing can run automatically and do not require any human integration.

On a technical level, both build step and test step are executed on one or multiple build servers which is tightly integrated with the code repository and gets triggered through changes to the codebase. At the end of the build and test process, it will (step iv) report the status back to the users. Such a report includes information about failed builds or failed test cases. On success the build server issues a versioned and downloadable build artefact.

For closing the Continuous Integration loop, other developers react on reports issued by the build server. In case of successful build and test steps, they are supposed to immediately integrate the changes in their own code base. This core concept behind Continuous Integration ensures that the code bases of different developers evolve compatibly.

In order to successfully implement this feedback loop, it is important for every developer to commit very often (commonly interpreted as at least once a day). This ensures that merge conflicts stay minor and are easier to resolve.

### 3.3. Testing

Testing is necessary, as a successful build process does not give any hints whether the code is actually working. Hence, testing increases confidence on the codebase which creates an

experiment-happy environment and reduces the risk introduced by possible ambiguity of requirements. Further, testing may yield information about code quality and runtime behaviour. Consequently, testing is the main vehicle to ensure reliability of and trust in the code. Obviously, this trust is higher, the higher the test coverage. Due to the many builds per day, testing can only be realised in an automated manner. In consequence, high automated test coverage is a core demand for Continuous Integration [2].

While there is no general agreement on a fixed number for code coverage percentage, there are suggestions and guidelines [3] about that metric. In practise, however, the desired coverage degree is dependent on the project and the criticality of the code.

As with the whole Continuous Integration methodology, it is important that all team members have understood the importance of testing and practise it. Unit and Integration Tests are the minimum amount of tests necessary to achieve that. Consequently, they are our main concern in this chapter. Further details on testing of distributed applications are available elsewhere [1].

Unit tests target small portions of code in the codebase and usually operate on a class- or routine-level. They are built alongside the application and are executed on a successful built. Implementing them makes sure that individual parts of the component are working as expected and intended. In contrast, integration tests are run against a fully built and unit-tested component. An integration test executes the software component as a whole and runs tests against APIs and if necessary utilises mocks.

### 3.4. Best practices

While the Continuous Integration loop as detailed earlier is simple, a true realisation of the approach requires flanking measures on the management and organisational side.

In order to decrease the change of a broken build and failing tests, developers have to build and test the application locally before committing their changes to the shared code repository. This practise leads to the desire that the automated test environment used on the build server and the test environment provided by the developer's IDE be compatible. Only then can the same tests be run in both environments and only then is the effort for the developer minimal to follow the principle of Continuous Integration.

Having such a set-up, a developer will commit more often to the shared code repository when experiencing short feedback cycles from the Continuous Integration loop. Ideally, the time from committing code changes to a tested software artefact is as short as running the tests on the development machine or even shorter.

However, even performing local tests will not avoid that at some point a build or a test will fail leading to a broken build. While the build is broken no developer should commit to the repository. Instead, everyone in the team is encouraged to contribute to fixing whatever caused the build to break. Only then further commits to the code repository are allowed.

In order to enable developers to witness that a build breaks and to trigger the process of handling this broken build, visibility of the current build and test status is a major concern. This can be as simple as a red or green badge being shown in a dashboard, an e-mail, but could also include bots that report on it on a messaging platform.

### 3.5. Summary

Summarising, Continuous Integration gives reproducible builds, versioned downloadable artefacts, and quick feedback on broken builds. Hence, it addresses many of the requirements brought up in Section 2: breaking down development into small units that are independently built and tested distributes work among partners (**R.1**) and at the same time, identifies responsible persons (**R.3**). Automating the build and testing process tremendously reduces the manual burden (**R.2**). It also checks dependencies on build level (**R.5**) and makes the software status visible (**R.6**). The availability of ready-to-use binaries is a first step towards an easy-to-use prototype (**R.4**). The definition of unit tests and integration tests allow people joining the project late to confidently make changes to the source code (**R.B**). If used properly, tests also serve as a testimonial of the currently defined requirements of the project (**R.C**). The separation of build and test phase enables some support for closed source code (**R.A**).

On the downside, Continuous Integration does not address dependencies on service level (**R.5**) and neither allows for a full easy-to-use set-up (**R.4**). In consequence, it also does not make the full software status available (**R.6**). With respect to closed and unavailable source code (**R.A**), further means have to be established.

Yet, the use of Continuous Integration also introduces new requirements:

**R.CI.1 (additional project infrastructure):** The use of Continuous Integration requires more infrastructure to be brought into the project. These include a revision control system, a build server, and a test server. All of them have to be maintained and explained to the consortium, for example, through tutorials. The build server in addition has to support all programming languages used in the project (**R.E**).

**R.CI.2 (support for private code):** For those partners in the project that want to disclose their source code to the public, the project infrastructure needs to support a private repository.

**R.CI.3 (support for closed code):** For those partners in the project that want to disclose the source code of their components even to the project, additional mechanisms have to be established in order to connect these components to the overall application.

**R.CI.4 (team agreement):** For Continuous Integration to work properly, all project partners have to agree on its use and be willing to take their share of the load. This is a management issue that can be supported when lean technology and good documentation is applied.

## 4. Background: Continuous Delivery

This section details background on Continuous Delivery. It starts with a definition and the usage scope, then presents the delivery pipeline and further testing steps. In contrast to Continuous Integration that has many challenges on social level, but a clearly defined build artefact at the end of a pipeline, the exact result of a run of the delivery pipeline is a design choice; the only demand is that it packages the binaries into something executable. Section 4.3 is concerned with

the various possible approaches to packaging. Finally, when executing a component, various parameters may need to be configured, depending on the context the component is used in. We sketch possible design choices in Section 4.4.

### 4.1. Definition and scope

Continuous Delivery takes an executable binary (e.g. a build artefact) and packages it in a ready-to-run execution environment that resolves all internal and external dependencies, for example, to the operation system kernel, third-party libraries, and remote services. At this end, this automatic process creates packaged runtime environments for binaries and other artefacts. The rational is that pre-configured and tested self-contained packages are easy to roll out in different environments increasing the reliability of the roll-out process. In addition, abandoning manual actions strengthens maintainability and trust in the process.

When combined with Continuous Integration, Continuous Delivery provides a methodology that ensures that at any time a packaged, tested, and reliably deployable artefact is available based on the latest successful run of the integration pipeline.

### 4.2. The delivery pipeline

The delivery pipeline starts where Continuous Integration ends. It introduces the packaging step plus further automated and manual acceptance tests. A visual example of such a pipeline is shown in **Figure 2**.

Continuous Delivery starts with **build artefact(s)** that could be the outcome of Continuous Integration. The packaging step integrates one or more of them with any external dependencies and bundles them into packed artefacts (or simply artefacts). The pipeline is not necessarily linear and hence, can general more than one package. Depending on the process, packages for multiple architectures or use cases can be generated. While Continuous Integration contains



**Figure 2.** Example delivery pipeline.

a set of basic tests, Continuous Delivery introduces more sophisticated acceptance test. These are a crucial part of any useful delivery strategy and often contain both manual and automatic steps that validate if the software component behaves as expected. It will almost always include mocking of remote services.

### 4.3. Possible packing formats

The outcome of a run of the delivery pipeline is at least one deployable artefact packing an application component. While the delivery concept *per se* does not foresee a specific format and basically an arbitrary number of formats are possible, the following four approaches have found wide-spread acceptance and are commonly used. They differ in the size of the package, the coupling between component and host, and possible interferences with other components on the same host.

Virtual machine images package the component together with a suited operating system and all required third-party libraries. Executing the image in a virtual machine on a hypervisor introduces very strong runtime isolation between component instance (inside the virtual machine), the host installation (outside the virtual machine), and the host's hardware. It also creates barely any external dependencies and no direct interferences with other components on the same host. On the downside, virtual machine images are heavy weight in terms of size and resource usage. Container images are a lightweight alternative that also bundle the component with all third-party libraries. Yet, the container's operating system strongly depends on the hosting environment in terms of operating system version and kernel configuration. Still isolation between co-located components is available.

Both virtual machine and container images create an isolated and fully self-contained environment for the component. A conceptual different approach is followed by configuration management tools and distribution packages. Both of them install software directly on the host platform and barely create any isolation between different components. Software distribution packages are special archives that wrap the binary and files it ships with, but also contains hints to packages this binary depends on. Obviously, they integrate deeply with the dependency management of the host platform and utilise shared system resources and libraries directly. Configuration management tools provide a layer of abstraction, as they attempt to (re-)configure and change the hosts environment to reach a state which ensures that the application can run. They may do so by using software distribution packages. Both approaches are rather lightweight in terms of storage size.

### 4.4. Package configuration

The major goal of Continuous Delivery is to always have the latest deployable package of a component available. In consequence, this means that when creating the package, it is not known in which environment it will run. For instance, IP addresses, port numbers, paths to files may not have been defined yet, or can change over time. Consequently, when preparing a component for Continuous Deployment, it is important to foresee a configuration interface. Several approaches exist ranging from environment variables as suggested by the

12FactorApp[3] over configuration files as commonly used for components provided as Linux packages, to key/value stores or a database.

The choice of a configuration approach has influence on the overall implementation of a component. In addition, there is a mutual influence between configuration and packaging format.

### 4.5. Summary

Summarising, Continuous Delivery gives tested, versioned, and downloadable artefacts that are shippable, installable, and configurable out-of-the-box. As with Continuous Integration, the high degree of automation reduces manual burden (**R.2**). The use of Continuous Deployment helps the installation and management of the project outcomes (**R.4**) and at the same time helps remembering the big picture due to acceptance tests (**R.5**). The latter also contributes to the visibility of the software status (**R.6**). Similarly, acceptance tests help keeping track of desired outcomes (**R.C**) and support the fluctuation of team members (**R.B**).

On the downside, creating a larger deployable component from smaller parts, may counteract the equal distribution of load over partners (**R.1**) and makes the naming of responsible persons harder (**R.3**). Yet, when Continuous Integration is used in addition, logical bugs should have been filtered out and only those produced by acceptance test remain.

Continuous Delivery introduces the following additional requirements towards prototype integration and management.

**R.CDel.1 (packaging format):** Continuous Delivery requires to decide on one or more packaging formats per delivery pipeline.

**R.CDel.2 (configuration options):** Continuous Delivery requires to decide on the approach taken towards configuration per pipeline. It has to be consistent with the packaging format.

**R.CDel.3 (support for closed artefacts):** As with Continuous Integration additional mechanisms have to be established for any kind of closed code or closed binaries.

## 5. Background: Continuous Deployment

This section gives background on Continuous Deployment. As before, we start with a definition and set the scope for this methodology. Then, we describe the deployment pipeline. Finally, we consider deployment environments and application state.

### 5.1. Definition and scope

*Deployment* as such describes the process of enacting an application or application component. In general, it covers the steps from acquiring the necessary and possibly distributed hardware resources over installing as well as configuring the component(s) on these resources

[3]https://12factor.net/de/config

and starting the necessary deliveries. The task of deciding in what order components should be started is referred to as *orchestration*, the task of enacting components to find each other is called *discovery* or *wiring*.

Continuous Deployment describes a methodology to always have the latest version of all artefacts of an application deployed; and that updates to the application are visible in the deployment shortly after changes to the codebase. As with integration and delivery pipelines, the deployment pipeline is supposed to run automatically.

As all artefacts have gone through unit, integration, and acceptance tests, there is trust that individual artefacts work as expected. What is less reliable is the interplay of the components on an application-wide level. For that reason, in practise, multiple isolated environments are used and Continuous Deployment usually tackles the least critical environment, which is not linked with production systems. Yet, some companies like Amazon and Netflix demonstrate Continuous Deployment can go directly to production.

### 5.2. The deployment pipeline

The safety net of having multiple environments caters for incompatible version and interface changes of individual components. **Figure 3** shows an example of three traditionally used different environments as well as transitions between them.

The development environment contains the very latest version of the components' code and is automatically updated on every commit. In contrast, the production environment contains

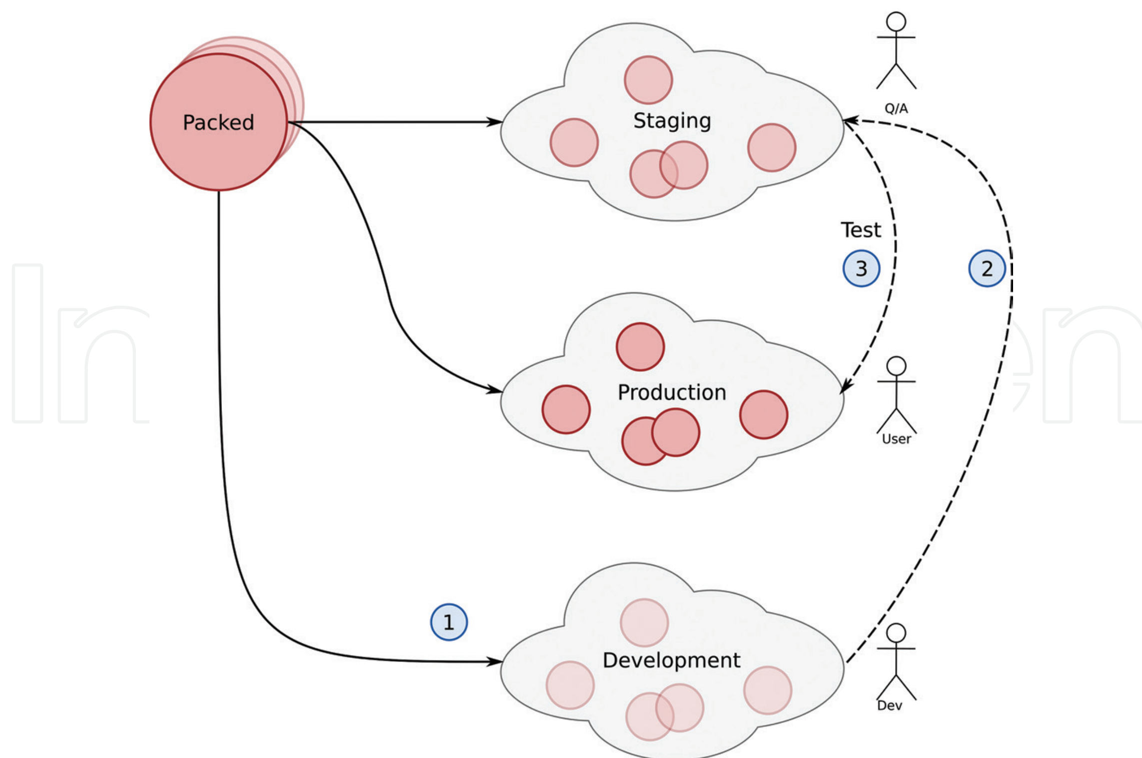


**Figure 3.** Deployment pipeline.

the actual live and fully functional environment facing users and customers. The staging environment is applied to validate updating the production environment to newer version. Therefore, staging uses a snapshot of the production data.

It is important to note that besides their build versions, the packaged components do not differ from environment to environment. What differs is their configuration in the respective environment (cf. Section 4.4) and the process of updating them. The development environment is automatically installed from scratch with each new deployable artefact from Continuous Integration. This environment is then used by developers in order to test and validate the common application. It is also used for reviews by Q/A. If these are successful, the version of the development environment is instantiated in the staging environment by updating the previous installation. This serves as a blueprint for updating the production environment. In case it succeeds, Q/A will enact more tests and finally decide to upgrade the production environment.

### 5.3. Application state

Usually at least one of the application components makes use of persistent state such as data stored in a database and on the file system. In order to support automatic re-deployment in case of failures and a seamlessly upgrade from one version to another, this state has to be separated from the artefact produced by the delivery pipeline. Otherwise, software and state cannot not be upgraded separately.

In consequence, state needs to remain available, even if the application environment is torn down. In IaaS clouds or containers, this can be achieved through the use of block storage/volumes; in more traditional approaches, a remote file system, a NAS, or a SAN could be used. In consequence, the location of the state has to be configurable.

### 5.4. Summary

Summarising, Continuous Deployment realises support for a constantly deployed instance of the project outcome. In addition to that, it enables the realisation of use-case specific or demo-specific environment (**R.D**). Similar to Continuous Integration and Continuous Delivery, it helps distributing work among partners (**R.1**), reduces manual burden (**R.2**), and makes the software status visible (**R.6**). Its orchestration is the missing link to make available an easy to start and use environment (**R.4**) and clarifies the big picture (**R.5**). There are no immediate downsides to Continuous Deployment, but further requirements emerge:

**R.CDep.1 (environment planning):** The consortium has to agree on the number of environments and the desired flexibility in creating environments. In the most extreme cases the creation of a new environment is fully automatized and developers can flexibly create new environments.

**R.CDep.2 (handling of state):** Continuous Deployment requires to decide on the approach taken towards handling application state. In addition, stateful components need to be able to find their state, to validate it exists, and to initialise the storage location, in case it does not exists. In case state representation was changed, this has to be tolerated.

**R.CDep.3 (support for closed artefacts):** As before, additional mechanisms have to be established for any kind of closed binaries.

**R.CDep.4 (additional infrastructure):** In order to achieve the deployment and wiring of individual components, but also the whole project software, an orchestrator is necessary.

## 6. A research-oriented solution to software releases

In the following, we take the requirements put up in Section 2 and describe how we apply Continuous Integration, Continuous Delivery, and Continuous Deployment to support prototype integration as well as software releases for large-scale, widely distributed research projects. We also present how we address the additional requirements put up throughout Sections 3-5.

Section 6.1 presents the overall concepts and strategy we apply. The subsequent sections deal with the realisation of the individual pipelines. In each of these, we present our approach from a conceptual as well as a technical point of view and discuss the tools used. In addition, we present similar tools available on the market that could be used to provide the same or similar functionality.

### 6.1. Overview and concept

Sections 3–5 detail that the combined use of Continuous Integration, Continuous Delivery, and Continuous Deployment addresses almost all of the requirements established in Section 2. **Table 1** presents the coverage of requirements and methodology taken. Only the need to cater

| | Continuous Integration | Continuous Delivery | Continuous Deployment |
|---|---|---|---|
| R.1 | X | | X |
| R.2 | X | X | X |
| R.3 | X | | |
| R.4 | | X | X |
| R.5 | | X | X |
| R.6 | | X | X |
| R.A | (X) | (X) | (X) |
| R.B | X | X | |
| R.C | X | X | |
| R.D | | | X |
| R.E | X | | |

**Table 1.** Requirement mapping.

for closed code and binaries (**R.A**) is not naturally taken into account by any of the methodologies. It is, however, represented by the follow-up requirements R.CI.2, R.CI.3, R.CDel.3, R.CDep.3, and R.CDep.4 and has to be addressed by all three methodologies.

The following paragraphs sketch our approach on a high technical and management level.

#### *6.1.1. General software set-up*

Our approach centres around a project-wide code hosting platform that supports private repositories for code that shall not go public (**R.CI.3**). This platform is enhanced with a project-wide build and test server (**R.CI.1**) and further with an orchestration service linked to these two (**R.Dep.4**). Whenever possible, we rely on private hardware to host the needed infrastructures as well as the various environments of the deployment pipeline. In case this cannot be achieved, we fall back to a public cloud provider such as Amazon EC2 or Microsoft Azure.

#### *6.1.2. Management process*

In order to be able to apply Continuous Anything, decisions on the management level are required. These include first and foremost, the decision of the consortium to enact the methodology (**R.CI.4**). Once this decision is taken, the next step is to break down the overall project software into smaller components. This is a manual process that requires discussion and communication.

For each of the components an individual software repository is created and a responsible gets assigned. In an ideal case, exclusively members from one local team are responsible for one of these components. For each of the components then test cases are defined that detail how the component is supposed to interact with other components and more importantly that reflect requirements and goals of the project. Finally, an early integration pipeline for each component is realised that runs the tests. Only at this point, it is necessary that the developers of a component agree on a common technology including the programming language. Different components can agree on different languages.

In a next step, components are composed to deployable artefacts and for each of them a delivery pipeline is established. Acceptance tests are created and function as both a validation of the artefact's functionality as well as a representation of the project's requirements. Furthermore, a strategy towards packaging (**R.Del.1**) and configuration (**R.Del.2**) is decided upon. While it is principally possible to use different formats and approaches for different artefacts, the delivery pipeline benefits from unifying these.

In a last step, the consortium agrees on the number of environments to be used as well as the transition paths between environments (**R.CDep.1**). This is also the step where services with closed binaries get integrated in the whole system (**R.CDep.3**).

### 6.2. Continuous Integration

As clarified in Section 3, Continuous Integration requires additional infrastructure to be provided by the project. In particular, it demands for the operation of a code repository, a build server, and a test server.

*6.2.1. Concepts*

Due to the openness towards programming languages, the build server needs to cater for any reasonable programming language (**R.E**). We achieve that through specialised build environments. These build environments are used for the automated compiling and testing the components code (**R.2**) and are easily configurable and versioned **(R.B**) by the researchers themselves (**R.1, R.3**). The downloadable build artefacts are stored in the repository (**R.C**) with the appropriate access rights.

Access rights to each repository can be specifically set with permissions ranging from private (**R.CI.2**) over internal to public. Sometimes one (e.g. an industry partner) is not able to share their code or configuration parameters with the whole consortium or even publicly. Therefore, we limit access to code and encrypt certain configuration variables so only the actual owner has access to them (**R.CI.2**).

Regarding completely closed and private source code, which must not reside on the shared infrastructure (**R.CI.3, R.CDel.3**), we make the assumption that the build artefacts of these component are tested by their owners and their binaries available for use in the project. For closed binaries, we establish a customised deployment process (**R.CDep.3**).

*6.2.2. Selected tooling*

For our approach, we selected Git enhanced with GitLab (**R.CI.1**) as a source code repository. The primary reason for selecting this combination is due to state-of-the-art version control provided by Git as well as the user interface and eco-system provided by GitLab. Each software component is stored in an own Git repository.

As a build server we use GitLab Runner. On the one hand side this is due to its deep integration with GitLab, but on the other hand side, this is also due to its openness and flexibility also in supporting exotic demands (e.g. **R.E**). It achieves this, by enabling the use of custom build environments, giving the research teams a maximum amount of control.

Technically, each repository defines the build environment of the component stored in that repository. The environment also defines the integration pipeline and contains at least the two mandatory steps compiling and testing. When triggered, builds, and tests get executed in an instance of the defined build environment.

Due to the fact that we do not impose any programming languages, we do not rely on any specific build and dependency management frameworks. The same is true for testing frameworks. Here, the only requirement is that it can be included in the pipeline.

*6.2.3. Tooling alternatives*

The functionality we achieve through our set-up can also be realised through the use of other tools. For instance, Mercurial or SVN could be used as source code repository. Jenkins and Travis are alternatives for build servers.

With respect to build automation Maven is the de-facto standard for Java, while C programmers rely on make and pip could be used for Python. Testing can be implemented by JUnit or one of its derivatives for other languages.

### 6.3. Continuous Delivery

From Section 4, it is clear that the main challenge with Continuous Delivery are to decide on the packaging format and the configuration strategy.

#### *6.3.1. Concept*

For being able to easily start and use a component (**R.4**), we are packaging the artefact from Continuous Integration to make it executable. This process is automated by the build server (**R.2**). Once all the components from every partner have been packaged getting an instance of the application as a whole is comparatively low effort (**R.5**).

In contrast to the integration pipeline, not all repositories will have a delivery pipeline. Instead, multiple build artefacts can be combined to one packages artefact. For each packaged artefact, a root repository is selected that defines the delivery pipeline.

Similar to the integration pipeline, the process of packaging the component is specific to each repository. While the packaging format should usually be consistent for every component, it might be necessary to integrate with other build artefacts and external components, which is the task of the delivery pipeline.

While Continuous Anything does not demand for a specific packaging format on the concept level, the format should be (i) lightweight, to keep the delivery feedback cycle short, (ii) self-contained, to make acceptance testing easier, and (iii) configurable to cater for usage in different scenarios.

In order to support closed artefacts (**R.CDel.3**), we enhance the build server with a custom API that maintainers of closed artefacts are supposed to invoke (either automatically or manually) when a new version of their binary is available. This will then trigger the delivery pipeline for that artefact, if available or the delivery pipeline of artefacts that make use of it.

#### *6.3.2. Selected tooling*

Our approach does not impose any specific packaging format. Yet, for artefacts with standard demands, we encourage the use of Docker images, as they offer a good trade-off between isolation and ease of use. Containers are not as heavy weight as virtual machines, but still the software runs isolated. The resulting Docker images are pushed to an image repository, which is internal to GitLab for private artefacts or the public Docker hub for public ones.

For configuration, we suggest the use of environment variables for Docker containers as encouraged by good practises. For acceptance testing, we use the Selenium framework that enables record and playback of user interaction on interfaces.

#### *6.3.3. Tooling alternatives*

For packaging scenarios that demand higher isolation, virtual machines are the best choice. Here, Packer is a tool for the automated generation of virtual machine images. For configuration management, Puppet is an option, whereas for instance the Debian Package Manager can be used for creating distribution packages.

For configuration through key values stores, a myriad of different tools exists, ranging from Consul to classic databases, for example, MySQL, or even NoSQL databases, for example, MongoDB.

### 6.4. Continuous Deployment

Building on the decision we made in Section 6.3 by choosing Docker as packaging format, we need to align that to the additional requirements we set in Section 5. The following shows how we implement the Continuous Deployment of said containers.

#### *6.4.1. Concept*

We usually use the three basic environments **development**, **stating,** and **production** unless the project has special demands (**R.CDep.1**). Each repository with a delivery pipeline also has a deployment pipeline that automatically updates the development environment once a new packed artefact is available. Based on the development environment, the transitions between the other stages are handled as follows (**R.CDep.2**): (i) Upon manual decision, the artefacts deployed to development get redeployed in staging by overwriting the previous version. In addition, state from production is copied to staging for testing purposes. (ii) Going from staging to production is similar, except that no data are copied.

All environments are handled by an orchestrator and operated on a project-hosted infrastructure (**R.CDep.4**). For enabling state transition (**R.CDep.2**), this infrastructure comes with volumes to persist state and support mapping of state. For dealing with disclosed artefacts (**R.CDep.3**), we allow that callbacks are registered for each of them. These callbacks are used in order to trigger a new deployment or reset of the respective deployed artefact, as well as a transition of these deployed artefacts between their environments. The realisation of the callback is dependent on the responsible for the artefact. In addition, we introduce another API at the build server that owners of the closed artefact shall use to notify the environment about changes in their environment.

#### *6.4.2. Selected tooling*

The deployment in our system is done by the Rancher orchestration tool. For artefacts realised as containers, Rancher applies rancher-compose and docker-compose.yml files. These describe the actual configuration and a representation of the artefact to be deployed. Here, we can define the container (or virtual machine) image to use, the location of the state, and the desired configuration. Rancher also enables integrating external components.

#### *6.4.3. Tooling alternatives*

For orchestration of containers and virtual machines a plethora of different tools exist. These are either cloud-provider specific such as Amazon CloudFormation and OpenStack Heat, or reside outside the platform. In earlier work, we compare the features of these tools [19].

## 7. Discussions

Koetter et al. [10] are arguing that due to the tight schedule and different commitment of partners, a prototype integration is hard to achieve as it is too costly. This leads to only partially integrated systems that do not fully support all features. We argue that with our system, we have a clear and easy integration workflow that can be adopted by almost any (distributed) team with modern software development lifecycles and be adapted to existing ones. Therefore, we believe that our approach tackles the reported requirements and issues. Yet, our approach described in Section 6 is just one solution, from an overwhelming number of choices to make regarding the selection of tools and methods to realise Continuous Anything. The best possible technical realisation depends on what is currently used at the sides of the consortium members and familiarity of tools.

Nevertheless, team agreement as well as a clearly communicated and implemented methodology is more important than tool selection. The latter should always follow actual needs.

### 7.1. Project management culture

While Continuous Anything comes natural with the application of agile software development strategies, these are less an issue in distributed research projects. In this environment, it is hard to impossible to organise for instance daily stand up meetings or even weekly or bi-weekly sprints. As elaborated in Section 2 this is due to different schedules of the various stakeholders, the fact that barely anyone in the distributed team is dedicated full time to software development, and particularly that people travel a lot in order to promote the actual research work they do.

A possible way to work around the lack of a central pillar of the overall approach is to isolate responsibilities as much as possible and to only assign people from individual organisations to particular software components and have them organise the development process internally. This is the task of the project management.

A further core task of the project management besides organising the necessary infrastructure is to make sure that the integration strategy is rigorously followed from the beginning. This comprises the absence of shadow code, a common, shared understanding of how to use version control systems and when to apply changes to the master branch and other branches.

### 7.2. Software development culture

From all methodologies discussed, Continuous Integration can bring the most benefit. It is important to note, though, that everyone in the team has to agree on these practices being used. This might create some new pain points within the development team, but it is crucial that everyone understand the principles and share a common goal. This explicitly means that a non-trivial effort should be spent on test coverage. It is worth mentioning that for research-oriented environments daily commits of code are not necessary, but rather weekly or half-weekly commits are sufficient.

Continuous Delivery is especially helpful for research projects, since the described Big Bang Integrations usually happen multiple times during a project lifecycle; each time with a high risk of failing. The risk to failure puts a lot of stress on the whole consortium. In contrast, realising Continuous Delivery does not introduce new challenges except agreeing on a common packaging format.

Once Continuous Delivery has been realised, implementing Continuous Deployment is low effort. It is a crucial step to lower the effort for all consortium members to get access to a running instance of the project outcome.

## 8. Conclusions

In this chapter, we have presented our approach of Continuous Anything, a combination of Continuous Integration, Continuous Delivery, and Continuous Deployment in order to support the prototype integration to distributed research projects.

Our approach makes prototype integration a core element of the project plan and puts it on the same level as project management and financial administration. It does so by defining a framework that distributes and shares responsibility of integration work while at the same time clearly holding individuals responsible for dedicated software components. Through a high degree of automation, it keeps the overall integration work low, but still provides immediate feedback on the quality of the integration status. It is important to note that the quality of individual software components remains in the hands of their developers. It is them who decide which and if unit tests are necessary. In contrast, our framework requires that integration tests be available that ensure that interfaces between components work as intended. This approach allows an easy isolation of errors and the identification of responsible programmers in case of failures or problems.

## Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 732667 (RECAP), 732258 (CloudPerfect), and 644690 (CloudSocket).

## Author details

Simon Volpert, Frank Griesinger and Jörg Domaschka*

*Address all correspondence to: joerg.domaschka@uni-ulm.de

Institute of Information Resource Management, Ulm University, Ulm, Germany

## References

[1] Rother M. Toyota Kata. United States: McGraw-Hill Professional Publishing; 2009

[2] Duvall PM, Matyas S, Glover A. Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education; 2007

[3] Marick B. How to misuse code coverage. In: Proceedings of the 16th International Conference on Testing Computer Software; 1999. pp. 16-18. http://www.exampler.com/testing-com/writings/coverage.pdf

[4] Fowler M. Testing Strategies in a Microservice-Architecture [Internet]. Nov 18, 2014. Available from: https://martinfowler.com/articles/microservice-testing/ [Accessed: July 15, 2017]

[5] Fowler M. Continuous Integration [Internet]. May 1, 2006. Available from: https://www.martinfowler.com/articles/continuousIntegration.html [Accessed: July 15, 2017]

[6] Sandberg AB, Crnkovic I. Meeting industry: Academia research collaboration challenges with agile methodologies. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. Piscataway, NJ, USA: IEEE Press; 2017

[7] Eddy BP et al. CDEP: Continuous delivery educational pipeline. In: Proceedings of the SouthEast Conference. New York, NY, USA: ACM; 2017

[8] Guillot I et al. Case studies of industry-academia research collaborations for software development with agile. In: CYTED-RITOS International Workshop on Groupware. Springer; 2017

[9] Fagerholm F et al. The RIGHT model for continuous experimentation. Journal of Systems and Software. 2017;**123**:292-305

[10] Koetter F, Kochanowski M, Maier F, Renner T. Together, yet apart – The research prototype architecture dilemma. CLOSER 2017 Proceedings of the 7th International Conference on Cloud Computing and Services Science. Porto, Portugal: SciTePress; April 24-26, 2017. pp. 646-653

[11] Eckstein J. Agile Software Development with Distributed Teams: Staying Agile in a Global World. United States: Addison-Wesley; 2013

[12] Sharma S, editor. The DevOps Adoption Playbook: A Guide to Adopting DevOps in a Multi-Speed IT Enterprise. United States: Wiley; 2017

[13] Humble J, Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. UK: Pearson Education; 2010

[14] Davis J, Daniels K. Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale. US: O'Reilly Media, Inc.; 2016

[15] Kim G et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations: IT Revolution Press; 2016

[16] Boehm BW. A spiral model of software development and enhancement. Computer. 1988; **21**(5):61-72

[17] Beck K et al. Manifesto for Agile Software Development [Internet]. 2001. Available from: http://agilemanifesto.org [Accessed: July 15, 2017]

[18] Booch G. Object oriented design with applications. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.; 1991. ISBN: 0-8053-0091-0

[19] Baur D, Seybold D, Griesinger F, Tsitsipas A, Hauser CB, Domaschka J. Cloud orchestration features: Are tools fit for purpose? In: 8th International Conference on Utility and Cloud Computing. Piscataway, NJ, USA: IEEE Computer Society; 2015

IntechOpen

IntechOpen