# Literatura

**[1] Shahin, Mojtaba & Zahedi, Mansooreh & Ali Babar, Muhammad & Zhu, Liming. (2018). An empirical study of architecting for continuous delivery and deployment. Empirical Software Engineering. 10.1007/s10664-018-9651-4**

**Introduction:**

Development and Operations (DevOps) is a relatively new software development paradigm that promises many benefits such as solving the disconnect between development and operations teams, faster development and deployment of new changes, and faster failures detection [1-3]. Continuous integration, delivery, and deployment practices are key DevOps practices to fully realize the promises of DevOps without compromising quality [1]. Continuous Integration (CI) is the practice of integrating code changes constantly (i.e., at least daily) into the main branch and entails automated build and testing [4, 5]. As shown in Figure 11, CI is the first and core enabler for adopting continuous delivery and deployment practices [2, 6]. Whilst continuous delivery is focused on keeping software releasable all the time, continuous deployment extends it in order to continuously and automatically deploy new changes into production [7].

Precise definitions of continuous delivery and deployment are often missing in both the literature and industrial circles [5, 7, 8]. Continuous deployment is a push-based approach, by which code changes are automatically deployed to a production environment through a pipeline as soon as they are ready, without human intervention [9]. Continuous delivery is a pull-based approach, in which a person (e.g., a manager) is required to decide which and when production-ready code changes should be released to production [9]. Figure 1 shows the relationship between these practices and how an application is deployed to different environments [10]. Whilst a production environment is where the applications or services are available for end users, a staging environment aims at simulating the production environment as closely as possible. Continuous delivery and deployment share common characteristics and are highly correlated and intertwined [11, 12]. Hence, we refer to Continuous Delivery and Deployment (CD) as CD practices in this paper.

To support CD, organizational processes and practices may need to change and/or be supplemented by innovative tools and approaches. To this end, several efforts have been allocated to study and understand how organizations effectively adopt and implement CD practices. A number of studies emphasize the importance of choosing appropriate tools and technologies to design and augment modern deployment pipelines to improve the software delivery cycle [13-15]. Other research argues that the highly complex and challenging nature of CD practices requires changes in a team’s structures, mindset, and skill set to gain the maximum benefits from these practices [16, 17]. The other areas of interest in CD research are focused on improving automation in testing and deployment, and integrating performance and security into the deployment process [18].

**Architecture:**

The software architecting process aims at designing, documenting, evaluating, and evolving software architecture [43]. Recently, Software Architecture (SA) research has been experiencing a paradigm shift from describing SA with quality attributes and the constraints of context (i.e., the context and requirement aspect) and structuring it as components, connectors and views (i.e., the structure aspect) to focusing on how stakeholders (e.g., the architect) make architectural decisions and reason about the chosen structures and decisions (i.e., the decision aspect) [28, 44]. Whilst current research and practice mostly consider architecting as a decision-making process, it has recently been argued that SA in the new digitalization movements (e.g., DevOps) should cover the realization aspect as well [28]. The realization aspect of software architecture design is expected to

deal with operational considerations such as automated deployment, monitoring, and operational concerns. However, the SA research community has provided few guidelines and systematic solutions for this aspect of software architecture [29]. Accordingly, our findings in this study are placed in all four aspects of software architecture in the CD context, as Sections 5.1 and 5.2 mostly focus on the structure aspect, Section 5.3 is about the context and requirement aspect and Section 5.4 concerns the realization aspect. Finally, the decisions and rationales (the decision aspect) that are made at the level of architecture for CD, are embedded in these aspects of software architecture.

**[2] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. (2017). The impact of continuous integration on other software development practices: a large-scale empirical study. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017). IEEE Press, Piscataway, NJ, USA, 60-71.**

TRAVIS being adopted as part of a larger restructuring effort, which lasted several days. Examples include changing the build system from Ant to Gradle and, consequently, reorganizing project folders; updated library dependencies; and restructured tests. Based on this anecdotal evidence, we concluded that the activity immediately prior to and immediately following the introduction of TRAVIS CI might not be representative of the project's overall trends. Therefore, to limit the risk of bias due to extraordinary project activity in the transition period, we excluded one month of data centered around the adoption event in our quantitative analyses below. Measures: We collected global and time-window measures:

• total number of commits in a project's history, as a proxy for project size / activity.
• total number of commit authors, as a proxy for the size of a project's community; commit authors include both core developers, who are also committers, and external contributors, without "write" access, whose commits are merged in by the core developers. Since it is common for open-source developers to contribute under different aliases (name-email tuples), e.g., as a result of using different machines and changes in git settings over time, we first performed identity merging. We used heuristics that match first and last names, email prefixes, and email domains, cf. prior work [27], [28], and found an average of 1.15 (max 7) aliases per person in our dataset.
• project age at the time of adopting TRAVIS CI, in months, computed since the earliest recorded commit.
• main programming language, automatically computed by GITHUB based on the contents of each repository and extensions of file names therein.
• number of non-merge commits, number of merge commits per time window. Since git is a distributed version control system, developers can work locally, in increments, before pushing their changes to GITHUB or opening a pull request. E.g., a developer can choose to partition a large change into many smaller ones, and make multiple local commits to better manage the work. This would have no effect on CI runs, since CI is only triggered by GITHUB push events, and events happening after (and including when) a pull request is opened. Consequently, to study Commits To the Mainline Every Day (Fowler's best practice), as opposed to potentially local git commits, we distinguish between non-merge commits and merge commits as a proxy. We recognize non-merge commits as those having at most one parent, and merge commits otherwise.
• mean commit churn per time window. Churn is the total number of lines added plus the total number of lines removed per commit (in git modified lines appear as first removed, then added), extracted from git logs. The mean is computed over all commits in that time window.
• number of issues opened / closed, number of pull requests opened / closed per time window, extracted using the GITHUB API.
• mean pull request latency per time window, in hours, computed as the difference between the timestamp when the PR was closed and that when it was opened. The mean is computed over all PRs in that time window.

• number of tests executed per build. Each TRAVIS build runs at least one job, corresponding to a particular build/test environment (e.g., jdk version). Once the job starts, a log is generated, recording the detailed information of the build lifecycle, including installation steps and output produced by the build managers. On a sample of Java projects that used Maven, Ant, or Gradle as their build systems, for which we could make reasonable assumptions about the structure of their build log files, we parsed the TRAVIS build logs and extracted information about the number of tests executed (we take the maximum number of tests across jobs as the test count for a build), and the reasons causing builds to break (similar to [29]).

**[3] Ståhl, Daniel & Bosch, Jan. (2014). Modeling continuous integration practice differences in industry software development. Journal of Systems and Software. 87. 48–59. 10.1016/j.jss.2013.08.032.**

**[4] Barrie Sosinsky. Cloud Computing Bible. (2011). Willey Publishing, Inc. ISBN: 978-0470903568**

Książka opisuje najważniejsze elementy potrzebne do pracy z chmurami obliczeniowymi. Pozycja ta zawiera takie informacje jak:

- definicja chmury
- architektura towarzysząca chmurom
- informacje na temat serwisów oraz aplikacji działających w chmurze
- opis podstawowych chmur obecnie wystepujących na rynku
- zarządzanie chmurami
- komunikacja z chmurami
- bezpieczeństwo w chmurach
- praca z dostępnymi zasobami dyskowymi
- Amazon web services
- Microsoft Cloud Service
- Google web services

Cloud computing takes the technology, services, and applications that are similar to those on the Internet and turns them into a self-service utility. The use of the word “cloud” makes reference to the two essential concepts:

- Abstraction: Cloud computing abstracts the details of system implementation from users and developers. Applications run on physical systems that aren’t specified, data is stored in locations that are unknown, administration of systems is outsourced to others, and access by users is ubiquitous.
- Virtualization: Cloud computing virtualizes systems by pooling and sharing resources. Systems and storage can be provisioned as needed from a centralized infrastructure, costs are assessed on a metered basis, multi-tenancy is enabled, and resources are scalable with agility.

Computing as a utility is a dream that dates from the beginning of the computing industry itself. A set of new technologies has come along that, along with the need for more efficient and affordable computing, has enabled an on-demand system to develop. It is these enabling technologies that are the focal point of this book.

Cloud computing is a natural extension of many of the design principles, protocols, plumbing, and systems that have been developed over the past 20 years. However, cloud computing describes some new capabilities that are architected into an application stack and are responsible for the programmability, scalability, and virtualization of resources. One property that differentiates cloud computing is referred to as composability, which is the ability to build applications from component

parts. A platform is a cloud computing service that is both hardware and software. Platforms are used to create more complex software. Virtual appliances are an important example of a platform, and they are becoming a very important standard cloud computing deployment object. Cloud computing requires some standard protocols with which different layers of hardware, software, and clients can communicate with one another. Many of these protocols are standard Internet protocols. Cloud computing relies on a set of protocols needed to manage interprocess communications that have been developed over the years. The most commonly used set of protocols uses XML as the messaging format, the Simple Object Access Protocol (SOAP) protocol as the object model, and a set of discovery and description protocols based on the Web Services Description Language (WSDL) to manage transactions.

Most large Infrastructure as a Service (IaaS) providers rely on virtual machine technology to deliver servers that can run applications. Virtual servers described in terms of a machine image or instance have characteristics that often can be described in terms of real servers delivering a certain number of microprocessor (CPU) cycles, memory access, and network bandwidth to customers. Virtual machines are containers that are assigned specific resources. The software that runs in the virtual machines is what defines the utility of the cloud computing system.

Cloud computing has lots of unique properties that make it very valuable. Unfortunately, many of those properties make security a singular concern. Many of the tools and techniques that you would use to protect your data, comply with regulations, and maintain the integrity of your systems are complicated by the fact that you are sharing your systems with others and many times outsourcing their operations as well. Cloud computing service providers are well aware of these concerns and have developed new technologies to address them. Different types of cloud computing service models provide different levels of security services. You get the least amount of built in security with an Infrastructure as a Service provider, and the most with a Software as a Service provider. This chapter presents the concept of a security boundary separating the client’s and vendor’s responsibilities. Adapting your on-premises systems to a cloud model requires that you determine what security mechanisms are required and mapping those to controls that exist in your chosen cloud service provider. When you identify missing security elements in the cloud, you can use that mapping to work to close the gap. Storing data in the cloud is of particular concern. Data should be transferred and stored in an encrypted format. You can use proxy and brokerage services to separate clients from direct access to shared cloud storage. Logging, auditing, and regulatory compliance are all features that require planning in cloud computing systems. They are among the services that need to be negotiated in Service Level Agreements.

**[5] Martin Fowler. Continous Integration. (01 May 2006).**

Artykuł osoby aktywnie rozwijającej oraz pracującej przy metodykach programowania zwinnego oraz ekstremalnego. Martin Fowler opisuje dosyć dokładnie jak działają serwery ciągłej integracji, skąd się wzięła potrzeba wykorzystania ciągłej integracji oraz problemy jakie występowały podczas programowania kilka lat temu. Artykuł ten opisuje również dobre praktyki towarzyszące implementacji takiej infrastruktury, optymalizacje budowania aplikacji, zalety jakie niesie ze sobą ciągła integracja oraz sposoby na automatyzacje procesów przy realizacji ciagłej integracji.

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

**[6] Lorin Hochstein, Rene Moser. Ansible: Up and Running. (20 Jul 2017). O'Reilly Media, Inc. ISBN: 978-1491979754**

Książka opisująca krok po kroku jak zacząć prace z Ansiblem. Zaczynając od podstawowych elementów tego narzędzia jakim jest wstępna konfiguracja, poprzez podstawowe playbooki aż do bardziej zaawansowanych zagadnień jak na przykład tworzenie własnych pluginów lub modułów. Książka opisuje między innymi takie zagadnienia jak:

- wprowadzenie do ansibla i konfiguracja środowiska
- podstawowe playbooki
- praca z plikiem inventory
- zmienne i fakty
- role i podstawowe moduły
- pluginy
- optymalizacja pracy
- własne moduły
- współpraca ansibla z dockerem
- zagadnienia związane z kluczami ssh
- zagadnienia sieciowe z wykorzystaniem ansibla

**[7] William E. Shotts Jr. The Linux Command Line. (2015). Helion. ISBN: 978-1593273897**

Podstawowe komendy pozwalajace na swobodne poruszanie się po systemie linux. Książka ukazuje sposoby na prace z systemami typu linux z poziomu linii komend oraz przy pomocy skryptów bashowych. Książka zawiera podstawowe informacje na takie tematy jak:

- wprowadzenie do systemu Linux
- korzystanie z komputera pracującego pod kontrolą systemu Linux
- zarządzanie zasobami komputera
- administrowanie systemem
- tworzenie skryptów powłoki
- prawa dostępu, zmiana hasła i zmiana powłoki
- demony usług
- użytkownicy i grupy

**[8]** https://jenkins.io/doc/

Dokumentacja Jenkinsa.

**[9]** https://gerrit-review.googlesource.com/Documentation/

Dokumentacja Gerrita.

**[10]** https://about.gitlab.com/install/?version=ce

Dokumentacja Gitlaba.

**[11]** https://zuul-ci.org/docs/zuul/

Dokumentacja Zuula.

**[12]** https://docs.ansible.com/

Dokumentacja Ansibla.

**[13]** https://docs.openstack.org/install-guide/overview.html

Dokumentacja OpenStacka.

**[14]** http://docs.grafana.org/

Dokumentacja Grafany.

**[15]** https://prometheus.io/docs/

Dokumentacja Prometheusa.