LUT University

School of Engineering Science

Bachelor's Degree in Information Technology

Bachelor's Thesis

Automation tools in software development and production

Sami Kohvakka, **2020**

Supervisor: Ari Happonen D.Sc. (Tech.)

ABSTRACT

Author:	Sami Kohvakka
Title:	Automation tools in software development and production
Year:	2020
Faculty:	School of Engineering Science
Major:	Information Technology
Bachelors's Thesis:	Lappeenranta-Lahti University of Technology LUT 39 pages, 4 figures and 1 appendix.
Examiners:	Associate Professor Ari Happonen
Keywords:	Continous Integration, Continous Delivery, DevOps, developer tools, CI/CD

This thesis surveys automation tools available to aid in web application development. It presents common process phases in software development in the most popular software development process models. Emphasis is given to improving software quality and process productivity through continuous integration and *DevOps* practices.

Automation tools and their future development directions are examined based on the common process phases identified from literature. Such tools help developers write better code faster and provide insight to quality and boost to productivity through automated testing and delivering of the code.

TIIVISTELMÄ

Tekijä:	Sami Kohvakka
Otsikko:	Automaatiotyökalut ohjelmistokehityksessä
Vuosi:	2020
Tiedekunta:	School of Engineering Science
Pääaine:	Tietotekniikka
Kandityö:	Lappeenrannan-Lahden teknillinen yliopisto LUT 39 sivua, 4 kuvaa, ja 1 liite.
Tarkastajat:	Associate Professor Ari Happonen
Hakusanat:	jatkuva integraatio, jatkuva toimitus, DevOps, kehitystyökalut, CI/CD

Tässä tutkielmasssa selvitetään ohjelmistotuotannossa käytettäviä web-sovellusten kehitysprosessiin liittyviä automaatiotyökaluja. Tutkielmassa käydään läpi ohjelmistokehitysprosessin perusvaiheet eri prosessimalleissa, sivutaan ohjelmiston laatua ja prosessin tuottavuutta sekä paneudutaan jatkuvaan integraatioon ja *DevOps*-käytänteisiin.

Kirjallisuudesta tunnistettujen prosessivaiheiden perusteella työssä avataan nykyisin tarjolla olevia koodin kirjoittamista, testaamista ja toimittamista tukevia työkaluja sekä niiden tulevia kehityssuuntia. Nämä työkalut auttavat kehittäjiä kirjoittamaan parempaa koodia nopeammin ja parantavat ohjelmiston laatua sekä kehitysprosessin tuottavuutta poistamalla manuaalisen työn ohjelmiston testaamisesta ja toimittamisesta.

CONTENTS

List of Figures

1 Introduction

With the rise of open source community, tools that are used to develop and deliver software have taken a major leap forward. Many of these are backed by some of the largest companies in the world, such as Jest (Facebook) and Bootstrap (Twitter), others are created by non-profit organizations that deliver free products for a greater good, such as Apache Software Foundation. As the number of technologies increases, synergies can be achieved though co-operation.

In wider software tools development perspective, software development tools can enjoy from the general industry 4.0 effect boosting better development tools generation in almost all industry sectors. For example, from mass products by large companies to digital mass customization (Piili et al. 2013) and implementing AI software tools support for better and more sustainable product designs between industry specific solutions and cross discipline context. (Happonen, Ghoreishi, and Pynnönen 2020; Ghoreishi and Happonen 2020).

In software tooling context, it does not make much sense to write many decent editors for all languages instead of one actually good editor which can serve multiple languages. Similarly, synergies can be found in common business processes between industries. The fourth industrial revolution is driven by information systems enabling efficient automation. Industrial demand for IT-solutions propels demand for more efficient and flexible ways to make software and improve productivity in software production.

1.1 Objectives and Restrictions

The goal of this thesis is to provide insight to current situation and trends on software development processes and tooling in Web development, considering both the frontend technologies and backend web services. The main focus area is in improving the productivity in software development through process automation.

The main research question is

What kind of tools exist to help automate software development processes in web development and related backend services?

This thesis does not cover change management processes or tooling, and neither it covers service desk operations or service desk related tooling. User interface generators are left outside the scope of this thesis. Reasoning for excluding those topics is that both of them have scope so large an entire thesis could be written about either of those.

Emphasis is given to JavaScript and related technologies, as it is the native scripting language which runs in browsers, and can be used to create the required backend services. Java will be used as an example of strongly typed, non-scripting language for backend services.

1.2 Research methodology

This is a qualitative thesis summarizing findings from several case studies and interviews enriched with data scraped from popular software development forums in internet. It builds on opinions from many developers summarizing the common factors and tries to extract knowledge about future directions in software development.

1.3 Structure of the Thesis

This thesis begins with an introduction to the topic, is followed by literature review on build systems, software development, DevOps, agile method and version control systems, continues through the empirical part where more recent trends are discussed and finishes with a brief summary of automated development to deployment pipelines.

Literature review forms a base of understanding in changes of software development process which have lead to the need of process automation in several phases. It identifies the common process phases for which supporting automation tools are presented in the empirical section.

The empirical section begins with tooling for writing code, continues by examining testing tools and delivering tools. It ends with an overview of a software production pipeline, binding together the tools that support each process phase.

2 Literature review

This section covers a summary of research conducted within previous 20 years on search terms *build systems*, *software development DevOps*, *agile software development*, *test driven development*, *version control systems* and *software quality management*.

2.1 Software development process

Software development is characterized by projects that are too large for individual developers, and thus require co-operation between project members. Another striking characteristic is uncertainty, particularly regarding the functional specification of the software. Developing software is unpredictable and often done through trial and error due to lack of existing blueprints. This results in budget overruns, project cancellations, poor satisfaction and/or user acceptance and late finish. (Kraut and Streeter 1995; Buhrer 2003).

2.2 Conventional model

Traditional software development process can be thought of as a V-shaped model, where specifications are on the left-hand side, coding happens at the bottom and testing is performed on the right-hand side reflecting to the corresponding specification on the left-hand side. (Berger 2003; Regulwar et al. 2010) The key takeaway here is that extensive plans are made at the very beginning of the process, coding happens in the middle of the process and is followed by extensive testing phase, where pieces of software are tested first as individual units, then integrated to be a part of a more comprehensive system and tested as a part of the complete system. Moving to the next phase happens after previous phase is completed.

The main drawback of this model is that perfect plans are really hard to make. The model works, and it does produce high quality software, but the initial specification is often flawed and therefore the product is not what customer wanted. Compared to the agile model, changes to the specification are really expensive to make later in the process. (Stober and Hansmann 2010b, pp. 4–5).

Early hints about agile practices were identified in Sawyer (2004), who noted that open source software development is distributed to an interactive network of independent, product-oriented developers. Frequent, iterative small changes became the new normal.

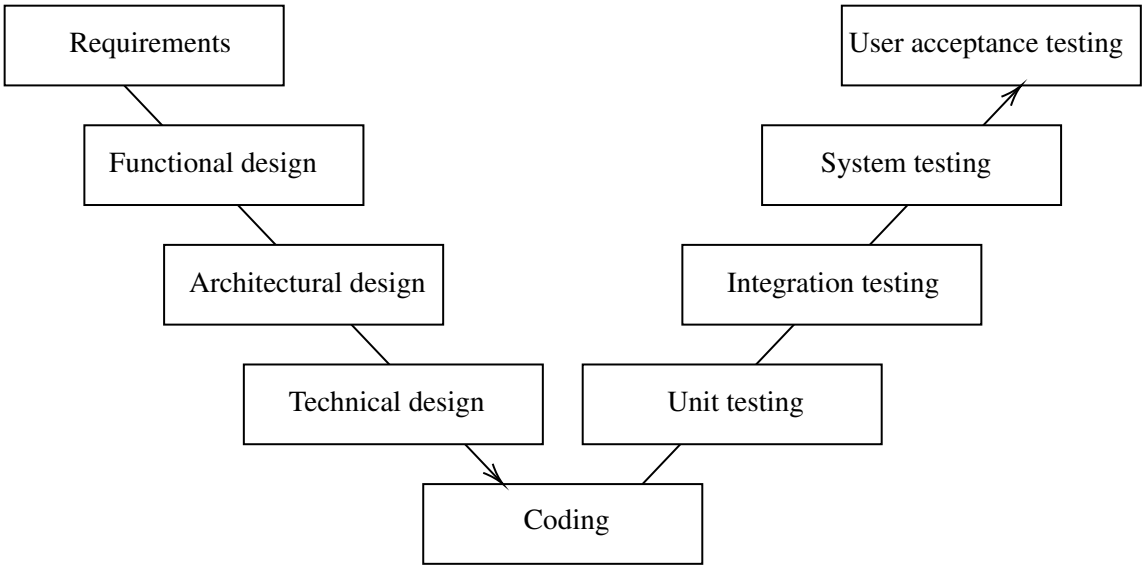


Figure 1. V-model of software development

2.3 Agile model

Instead of the conventional big bang model, agile model aims to enter the markets with minimum viable product to which incremental improvements are delivered often. It enables adaptive software development where focus areas can be switched often based on the most pressing demands. (Abrahamsson, Oza, and Siponen 2010). In agile model, specifications are locked only to a short period of time - often only a week or two - during which all new development happens. After this period requirements can be adjusted, changed or added, and the new development period begins.

Stober and Hansmann (2010a) define agile model as a continuous cycle of iteration. (Figure 2). It builds on agile manifesto, summarizing the twelve principles behind agile software development. The highest of which is to satisfy customer through early and continuous delivery of valuable software. (Kent et al. 2001).

In Agile setting, Scrum is the default project management model. It replaced eXtreme Programming, Kent Beck, Alistair Cockburn, Feature Driven Development, Crystal and other early agile project management models. (Gloger 2010). The base block in scrum model is sprint, during which a predefined amount of new features ought to be implemented by developers. A sprint can be thought of as a one full cycle in Figure 2. Due to frequent delivery of small changes, automating all test cases is essential to achieve good productivity in agile model. In practice, agile model may be too intensive and overwhelming for some projects. Daily scrum meetings are organized every other day or so. (Stober and Hansmann 2010a).

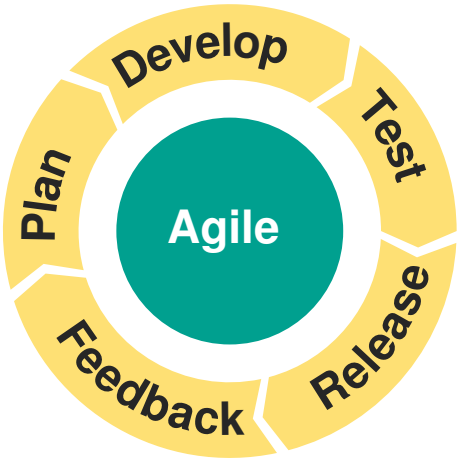


Figure 2. Agile model

2.4 Test-driven development

In test-driven development, programmers write tests against predefined interfaces before they start writing the actual code. It is a time-consuming activity and increases time to release, but ensures that all parts of the software are tested against specifications before the product is released. (Amrit and Meijberg 2018; Stober and Hansmann 2010a, pp. 47–48). However, programmers often skip some phases of test-driven development process because they feel some phases are less relevant than others. As a result, unit tests are rarely up-to-date and programmers ship quick-and-dirty code that barely passes the minimal test cases they were told to write. (Romano et al. 2017).

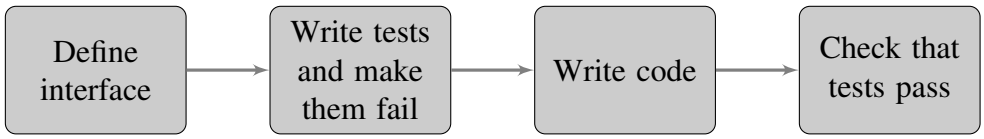


Figure 3. Test driven development

It is also possible to incorporate agile practices, such as test driven development, use cases, user stories, continuous integration etc. to the V-model to create some kind of a hybrid model. Often times this is motivated by reluctance of abandoning already approved, inefficient but working practices in industries, such as medical device manufacturing, where regulatory approval is required for working practices. (McHugh et al. 2013).

2.5 Web development

Web development refers to activities used to provide a user interface through web browsers. It is characterized by the use of frontend markup languages like HTML and CSS and emphasizes visual presentation, usability and aesthetics. (Miller and Connolly 2015). Newman, Agioutantis, and Schaefer (2017) refer to this as *presentation tier*, which is separate from *logic tier* and *data tier*. Presentation tier consists of a HTML template declaring the view, stylesheet defining the visual presentation of the components within the view and JavaScript sections defining the actions of buttons etc.

Schewe and Thalheim (2019, pp. 10–18) divide web information systems to six main categories based on their intended usage. The main categories are *Electronic Business and Electronic Commerce*, which includes B2B web services and online stores, *Communities and Groups* like Facebook, which are devoted to social interaction between users, *Entertainment* as in Netflix and other streaming services, *Identity and Personal Presentation*, for building your personal brand or authenticating your identity, *Learning and Edutainment*, to educate people and *Information Services* like search engines and encyclopedias.

In the early 2000s web frameworks emerged to simplify web development and reduce errors, which were frequent because all applications were being coded manually from scratch. These frameworks can be grouped to client-side and server-side frameworks based on where the code is being executed. Client-side frameworks such as Angular, Vue and React are executed in web browser and focus on presentation. Server-side frameworks such as Django, Zend and Ruby on Rails interlink with the web server process, but suffer from scalability issues as all pages for each client are rendered on the server instead of the client's browser. (Curie et al. 2019).

2.6 Backend services

Backend services refer to domain logic accessible to the frontend via standard interfaces and communication protocols such as HTTP. (Song, Hu, and Yu 2018). Those services can be consumed by desktop and web users, as well as mobile users. Most of those are provided as RESTful web services. (G. Liang et al. 2015). Provided backend services usually manage database access and may be linked to mail providers or similar integrated services. Dynamic web pages used scripting languages like PHP, JSP or ASP in Common Gateway Interface architecture to run in web server process and access backend services through socket, ODBC or COM modules as described in Chen and Mohapatra (2005), but have since been replaced by

separate client-side script communicating with server-side (micro-)services using JavaScript Object Notation (JSON). Through the rise of Node.js, it is now possible to construct backend services with the same scripting languages used in modern browsers. (Barna et al. 2017)

2.7 Process automation

Prior to 2010, IT departments typically consisted of separate development teams and operations teams. Development team was responsible for developing the software, and the operations team for running it. To enable shorter release cycle times, large companies such as Google, Facebook. Mozilla and Netflix invested heavily in modern release engineering techniques and technology. The roles of build engineer, sysadmin and tool developer were merged to create a new position of *release engineer* or *DevOps engineer*. (Kerzazi and Adams 2016).

Development and operations, or *DevOps* practices first emerged in 2009 to narrow the gap between development and operations teams. It tackles the inefficiencies in software development, release and operations processes by providing agile methods to deliver resilient systems. (Lwakatare et al. 2019).

In 2000, Nelson et al. wrote that up to 90% of the software life-cycle costs are spent on operations support phase after the application has been developed and installed. He divides software operations to mature and evolving context, and identifies some key differences between the groups. In mature group, focus is on maintenance, user assistance and system upgrades. Very little emphasis is put on change management and new development. Expertise in this group is application-specific and not easily portable. Experience is valued over formal education. In the evolving context, new development plays more important role and formal education in computer science or business with emphasis on computer science is preferred over application-specific experience. In this group, expertise is seen more portable and less dependent on environment.

2.8 Build systems

A build system is a set of predefined rules and software tools that convert source code into deliverable targets ready for deployment or testing. A classical example of this is *Make*, which builds executable programs and libraries from C source code by reading *Makefiles*.

(Feldman 1979). For Java, ANT and Maven are two popular build languages. (McIntosh, Adams, and Hassan 2012).

A complete build should take less than 15 minutes in order to provide quick feedback to programmers. For larger projects, this is not always possible. (Mårtensson, Hammarström, and Bosch 2017; Stober and Hansmann 2010c). Therefore, incremental builds are the cornerstone of modern build systems. (Bezemer et al. 2017)

2.9 Virtualization

Virtualization is a key technology to enable clouds. It allows running multiple configurations or even different operating systems simultaneously on the same physical machine, meaning that the available resources can be provisioned to multiple separate clients based on their distinct needs. The most prominent virtualization services include Amazon Elastic Compute Cloud (EC2), Google App Engine, Hadoop and Microsoft Azure. (Cafaro and Aloisio 2011, pp. 10–18)

Containers are created from a blueprint, referred to as *an image*. The image contains all of the resources and configurations required to run an app. Multiple containers can be created from the same image. (Zammetti 2020, pp. 346–350) Because containers can be versioned and started ad-hoc, container-based virtualization improves fault-tolerance and recovery in mission-critical systems and enables automatic scaling based on the current load. (Lee et al. 2018).

Because git was written as a POSIX-compliant software to be run on a POSIX-complaint operating system, emulation is currently required to run git on Windows. This imposes a significant performance penalty on Windows platform, making git-dependent developers favor macOS and GNU\Linux over Windows. To overcome this, Microsoft has begun work on virtualization systems to enable running a true Linux kernel and native POSIX-software within Microsoft Windows (Lewis et al. 2018).

2.10 Productivity

In software development, productivity is a broad concept which cannot be simplified to a single metric like time taken to crate a working solution. Although achieving the same goal, some solutions consume more processing power or memory, meaning that those are *ceteris*

paribus less efficient solutions. (Sadowski and Zimmermann 2019, pp. 3–18). Wagner and Deissenboeck (2019) define productivity in knowledge work through *effectiveness* and *efficiency*. Effectiveness refers to the relation between the produced functionality and quality and the ideal functionality and quality. Efficiency refers to the ratio between used effort and ideal effort. Efficiency determines the productivity of the input and effectiveness is the quality of the output.

2.11 Quality control

Based on user surveys by Pinto et al. (2018), CI systems increase the confidence that code is in a known state - either it compiles or it does not. If it does not, it is most likely caused by insufficient pre-commit tests, mostly caused by the lack of time. Often times, it would be faster to run the tests locally before submitting the commit to the CI pipeline than to commit first, then get a broken build, fix issues and commit again.

Another common reason for breaking the build is poor dependency management on local environments. Users may have globally installed packages on their devices, and thus everything will run smooth until they submit just to see that they have forgotten to add the package to project-specific list of dependencies. (Pinto et al. 2018).

2.12 General tools

Software is most commonly developed by teams consisting of many developers (Dooley 2011, p. 3). Teams need tools to collaborate. Version control system is perhaps the most important collaboration tool, which coordinates file sharing amongst multiple developers and can be used as a base for automated builds and tests. In systems that allow multiple developers to work on same file simultaneously, branching is often used to separate lines of progress. Branching creates an independent copy of the original version, and changes made in that branch can later be merged back to the original version. (Crookshanks 2014b)

In larger projects, builds are rarely performed directly from the integrated development environment. Instead, a build system like *Ant*, *make* or *Maven* is often used. Such systems consist of rules that define build targets and their dependencies. Builds may be initiated by each change or scheduled to occur e.g. hourly, nightly or weekly. The shorter the cycle, the easier it is to find the change which caused a build to fail. (Crookshanks 2014a)

2.13 Main challenges

According to Brindescu et al. (2020), code associated with a merge conflict is twice as likely to contain a bug than code that does not create a merge conflict. If the merge conflict requires manual intervention to resolve, code is 26 times more likely to create a bug. Avoiding merge conflicts is hard because there are no well-known predictors of merge conflicts (Leßenich et al. 2018). In popular repositories merge conflicts happen no matter how actively developers engage in conversations. (Vale et al. 2020).

Incremental builds are the core of modern build systems. This creates a challenge of ensuring the correctness of each build, when parts of the previous builds are preserved. Because builds often take some time, developers may have incentives to try a small modification by altering a single piece of the built artifact. A good build system should be able to tell that the artifact no longer corresponds to the source files and initiate rebuild of that altered piece during the next build sequence. (Maudoux and Mens 2018).

Complicated delivery process and long build times are the main reasons why developers do not deliver often to the mainline. This leads to self-feeding loops of trouble where few deliveries causes the build system to build larger parts of the system at once, further increasing the build times. Delivering rarely keeps the developers from learning the tools and processes used for delivery, which promotes the feeling of not knowing how to deliver. It also leads to complicated merge conflicts that developers do not want to resolve, preventing the developer to commit often because of fear of those merge conflicts. (Mårtensson et al. 2017).

At Rare (A British console gaming company currently owned by Microsoft), games were previously developed in two stages - first each team built their features independent of one another, and then all features were merged to one codebase to one codebase and developers started to integrate changes and fix bugs. After switching to continuous delivery model, integrating changes became much easier and the number of bugs reduced significantly, but this created a number of new issues. Compiling a large C++ codebase takes a lot of time, and running all test suites takes even more time. In game industry, artifacts consist mostly of artwork, which takes a lot of storage especially in HD or 4K. Transferring multi gigabytes of packages between agents serving the same pipeline is an enormous stress to networking. (Soltani 2016).

3 Current situation

In a case study Clarke, Elger, and O'Connor (2016) highlight four technological enablers of efficient software development. First, multipurpose languages like JavaScript allow developers to create frontend and backend stuff using the same language, boosting their productivity through better knowledge of the programming language compared to a situation where a full-stack developer must use multiple programming languages. Secondly, decoupled (microservice) architecture keeps individual components small and relatively easy to understand. Thirdly, virtualization services like Docker enable platform-independent rapid deployments and recovery in case of system breaking changes get accidentally deployed. Finally, quality can be controlled through git hooks and code quality review steps coupled to the continuous integration and delivery platform.

Code can be written using general purpose text editors, such as Notepad++, gedit, Vim etc. or purpose-built integrated development environments (IDEs), such as Android Studio, Apple's XCode, IntelliJ IDEA by JetBrains or Microsoft's Visual Studio. Some general purpose editors are feature-rich and customizable, meaning that they can be extended through plugins to turn them into full-featured IDEs. These include Emacs and Visual Studio Code.

IDEs support compiling and debugging the code within the editor context. More advanced editors provide syntax highlight, code completion by analyzing the context, peaking and jumping to variable declarations or function definitions, code refactoring, linting and options to automatically fix some semantic or programming style related errors.

3.1 Libraries and Frameworks

Recent advances in web frameworks have focused on cross-platform code reuse. Globally, more than 50% of site visits are made from mobile devices. 74% of requests made from mobile devices are made from Android devices and 25% from iOS devices. In the United States of America however, iOS is more popular with 58% market share followed by Android at 41%. (StatCounter 2020a; 2020b; 2020c). To provide a native experience on all commonly used platforms, application should be developed for four operating systems - Windows, Android, iOS and macOS, and compiled binaries ought to be provided for each main version of those operating systems. A non-native experience can be achieved through web apps, as all of those platforms include a web browser. (Shahzad 2017).

Mobile app architecture can be divided to four distinct groups each with some advantages and drawbacks. Native apps offer the best performance but introduce the most overhead as the code cannot be shared between Android and iOS devices. Hybrid apps are essentially native skeletons with an embedded webview in which the application is executed using the same HTML/CSS/JS codebase for both Android and iOS. Mobile web apps are just regular web apps that scale properly to small display sizes. Progressive Web Apps (PWAs) are web apps that can be installed (cached) on the device for native-like features such as push notifications and offline usage. (Khan, Al-Badi, and Al-Kindi 2019).

PWAs eliminate most of the programming overhead introduced by multiple target The cost and time savings of having one codebase instead of multiple codebases often outweighs the drawbacks of the PWA approach. platforms. (Khan et al. 2019).

To bridge the gap, *web-to-mobile* frameworks have emerged. These are hybrid frameworks which usually contain a collection of native widgets operate by transpiling the code from one language to platform-specific language to enable native builds.

Curated, proprietary or open source user interface libraries such as Bootstrap by Twitter enable quick prototyping and rapid development of web interfaces. Those libraries define the default style of each general widget, such as input box, table, button and toggle.

3.2 Emmet

Emmet toolkit is an editor plugin providing CSS-like syntactic sugar for rapid HTML development. It is available for Eclipse, Sublime Text 2, Visual Studio Code and many other common text editors. (Emmet 2020).

For example,

```
#page>div.logo>img#mylogo^ul#itemlist>li*5>a{Item $} \
    [href="/items/$"]^button.btn[@click="myFunction"]
```

expands to

```
<div id="page">
    <div class="logo"><img src="" alt="" id="mylogo"></div>
    <ul id="itemlist">
        <li><a href="/items/1">Item 1</a></li>
        <li><a href="/items/2">Item 2</a></li>
        <li><a href="/items/3">Item 3</a></li>
        <li><a href="/items/4">Item 4</a></li>
        <li><a href="/items/5">Item 5</a></li>
        <button class="btn" @click="myFunction"></button>
    </ul>
</div>
```

The power of Emmet is not limited to abbreviations, it can be used to select and update matching tags, toggle comments, evaluate math expressions in CSS and encode or decode image to data:URL. (Emmet 2020).

3.3 Language Server Protocol

To overcome the issue of writing a separate extension for each editor someone prefers to use, Microsoft and RedHat initiated an open source project to separate language feature provision and its visual representation from each other. Language Server Protocol (LSP) defines an interface against which contributors can create LSP clients for each editor and language servers for each programming language.

In theory, one language server can serve multiple editors, each having their own LSP client. In practice, language server process is most commonly spawned by the extension which provides the LSP client, thus reserving the LSP server for the editor instance.

3.4 Snippets

Code snippets are pieces of code that can be inserted with a shortcut or via an abbreviation. Some editors support placeholders to which developer can jump by pressing a tab button. In Visual Studio Code, snippets can contain a selection list or predefined variables like current date, content on the clipboard or file path to be expanded and inserted automatically. (Microsoft 2020)

For example, TextMate snippet defined as

```
{
    "Print to console": {
        "prefix": "log",
        "body": [
            "console.log('${1:text}')",
            "$2"
        ],
        "description": "Log output to console"
    }
}
```

in Visual Studio Code expands to

```
console.log('text')
```

when user begins to type `log`. The focus is on *text*, any inserted text replaces the content between apostrophes and pressing tab jumps to the next line.

3.5 Building the project

Because building large projects can take a lot of time, some build tools support incremental builds. For example, the Linux kernel is made of multiple smaller projects organized as a monotree. It does not make sense to compile all projects at once when changes are committed to one of the subprojects, such as a device driver. A robust and efficient build system is essential, as some studies have showed that on average 12% of development effort is spent on maintaining build scripts. (Erdweg, Lichter, and Weiel 2015).

Incremental builds in Ant and GNU Make are based on recording which files are used to make an output file, and trigger rebuild whenever one of those files is changed. For GNU Make, rules are defined in Makefiles. When target T is requested, Make looks for the proper rule which makes T from statically defined dependencies. (Erdweg et al. 2015).

3.6 Code review

O'Regan (2019, pp. 79–96) defines static testing as systematic examination of the source code without execution of the code. It is a well-established practice spawning from the late 1970s and aiming to provide built-in quality through elimination of unreachable code blocks, violations of coding standards and variables that are never used. This process can be automated by running a source code analysis system, such as ESLint or SonarQube.

Some version control server software can scan source code for known security issues. Dependency scanning is a feature where dependencies defined in `package.json` are compared against a database of known issues in *npm packages*. When a new vulnerability is discovered, services like GitLab Ultimate and GitHub will notify developers about the vulnerability if it is affects repositories stored on the version control management service. (GitLab 2020a; GitHub 2020a).

Another form of static code analysis is code scanning, where semantic code analysis is performed to find problems in code. Those problems can range from notifying about useless expressions to detecting cross-site scripting related attack vectors and insecure parameter calls. (GitHub 2020b).

3.7 Unit testing

Unit testing is the lowest level of software testing, focusing on individual functions, methods, procedures or similar low-level units. Common tools for unit testing include Jest, Jasmine, JUnit, Mocha with Karma etc. On JS/TS side, most tools use describe-it-expect syntax borrowed from Ruby testing framework called RSpec.

A sample test is written in JavaScript in RSpec-like test frameworks as a combination of describe-block which groups tests together, and it-blocks which create test cases. Tests can be run automatically before or after the build phase and the results of such tests are given as number of successful test cases out of all test cases. Modern unit testing tools output code coverage metrics to indicate how comprehensively the source code was tested. (O'Regan 2019, pp. 49, 75–76)

```
describe("cat". () => {
    it("has four legs", () => {
        expect(cat.numberOfLegs).toBe(4)
    })
})
```

3.8 Integration testing

Integration testing ensures that defects in interactions between separate interacting components are detected. Often times those defects emerge when either of the components one-sidedly breaks the previously negotiated contract on interface through which the components interact. (Xu et al. 2016).

In web applications, integration testing can be automated by emulating user actions through predefined scripts running in a browser-like environment such as Cypress. It uses similar syntax to define test cases than Jest and other RSpec-like test frameworks.

```
describe("My Page", () => {
    it("Gets, types and asserts", () => {
        cy.visit("https://mypage.mydomain.io")
        // Find link and click it
        cy.contains("link to page 2").click()
        // Should navigate to a new URL which includes "/pages/page2"
        cy.url().should("include", "/pages/page2")

        // Focus on element by css class,
        // type into it and assert the new value
        cy.get(".input-field")
            .type("sometext")
            .should("have.value", "sometext")
    })
})
```

Another option for integration testing is Robot Framework, which is a generic process automation framework intended for robotic process automation. Both of these are open-source and free to use. Syntax for Robot Framework is presented in Appendix x.

3.9 Version control

Version control systems are the foundation of source code management. They enable tracking changes over time and describe how the software came to be. Dooley (2011) separates version control strategies to two main groups. *Lock-modify-unlock* strategy locks each source file for edit, enabling only one developer to work on that file until the developer unlocks the file. *Copy-modify-merge* strategy enables simultaneous editing of separate copies of a source file, after which the copies are merged to form a new main version of that source file.

Git is a highly scalable version control system developed for the Linux kernel development. For each release, between 50k and 70k commits are added to the kernel. (Wilde and German 2018). As of April 2020, there are 3.9 Gigabytes of version history in torvalds/linux kernel source tree repository dating back to 1990s. With 31 000 forks and over 400 weekly authors, contributor network is too difficult for GitHub to parse. (Torvalds 2020).

3.10 Integration platforms

Continuous integration (CI) platforms, such as CircleCI, Travis CI, GitLab CI/CD provide a way of binding together source control management, test automation, build automation and delivery automation. Whenever a change made by an individual developer is integrated to the mainline codebase, it triggers a series of commands that are executed to build a new version of the application and check that it did not break anything. The main motivation for using a CI platform is saving time and improving quality. This includes fast feedback on breakage and ensuring quality of releases by asserting that new changes do not break existing features. (Pinto et al. 2018).

3.11 Hooks

Hooks are actions tied to commits in version control system like git. These hooks can be either server-side or client-side. In git, client-side hooks are defined in *hooks* directory within hidden *.git* directory, which defines the local repository. Client-side hooks interact with lifecycle events on user's machine and do not affect other developers. Server-side hooks are similar, but shared between users and executed on the common remote server. For example, Google runs initial tests in pre-commit phase on the client-side, after which results are communicated to developers. If pre-commit tests pass, programmer is entitled to submit the

commit to common repository. This initiates post-commit hooks on the server side to trigger the build process. (J. Liang, Elbaum, and Rothermel 2018).

3.12 Crowdsourcing

Croudsourcing refers to the use of collective knowledge to resolve an issue or implement a common feature. StackOverflow is a popular Q&A platform used by software developers, and it includes subpages for different programming languages. Each month about 50 million developers look for answers in StackOverflow, and per StackOverflow Developer Survey 2020, almost 91% of users visit StackOverflow when they get stuck with a problem. This is significantly higher than the amount of developers who switch to other work to come back later (54%), watch help (53%), call a coworker or a friend (50%) and go for a walk or do some other physical activity (43%). (StackOverflow 2020).

Abdalkareem, Shihab, and Rilling (2017) studied the use of StackOverflow in popular GitHub projects. They noted that the most popular use cases are learning about programming language features and best practices, e.g. *what is the most efficient way of casting string or integer to Boolean in JavaScript*, or *how the format function can be used to achieve a specific format in Python*. Another very common use case is learning about how to use a library and gain knowledge about the undocumented features of some common API. This includes deprecation, possible arguments and identifying the best method to perform a task.

3.13 Build optimization

A typical flow when user accesses a web page is that the browser fetches *index.html* file, which references some stylesheets and script files. These are all text files. For developers, those are easier to understand and faster to debug when variables and functions are named using real words describing their purpose. However, browser does not care whether the variable is called `UserName` and function is referenced as `fetchUserInfo(UserName)` or `a(b)`. The latter is more concise, requires less bandwidth and downloads faster. The process of converting natural language names to single letters is referred to as code compression, or minification and is handled by a bundler, such as Webpack, and takes place during the build phase. (Zammetti 2020, pp. 141–150)

Similar steps are performed on compiled languages, where unreachable code may be re-

moved, loops optimized by replacing memory references with register references, array boundary checks eliminated where possible and so on. (Suganuma et al. 2005). This generally improves program performance and eliminates some overhead caused by poor code. (Pan and Eigenmann 2008).

3.14 Release engineering

Kerzazi and Adams (2016) analyzed job descriptions of *release engineers* and *DevOps engineers* to determine the skill set required to oversee and develop modern delivery pipelines. Among 211 job postings, scripting was by far the most wanted skill. Most commonly this involves automation of manual release engineering tasks using scripting languages like Bash, Python or PowerShell. Other commonly required tasks are managing automated build and quality assurance tools, defining and managing the infrastructure for development, testing and production and setting up branching and merging strategies that support parallel development.

Releases are often delivered with delivery agents, daemons, that monitor integration platform for new deployments. Such agents support multiple modes, e.g. Docker, Kubernetes, shell, ssh and VirtualBox and can be run either on a dedicated server or in the cloud, where they can scale automatically and utilize idle capacity with cost-effective spot instances. (GitLab 2020b).

3.15 Deployment monitoring

Automated deployment health monitoring provides insights to application performance, stability and load. Event logging services like Sentry capture application errors and send detailed info about client's operating system and browser as well as a description of what the user did before the error occurred to developers, giving them a rich set of data to boost debugging efforts. (Sentry 2020)

System health monitoring services like Prometheus collect time-series of response times, resource usage etc. They can be used to detect bottlenecks in applications, optimize the number of resources available based on observed load at different time of day and gain insight to performance problems that occur only on production environment where the amount of data is often significantly higher than in development or test environment. (Prometheus

2020)

3.16 AI

Software development environments can be enriched with Artificial Intelligence. Running source code through machine learning algorithms help provide better, context-aware suggestions for code completion. A more advanced form of assisted development can look for solutions to presented questions and answer them based on existing software resources. These resources can include source code, it's comments, version control logs, emails, issue reports, official web pages, blocks, Stack Overflow Q&A etc. (Lin et al. 2017).

AI can also be implement to improve builds and optimize test strategies. Richter et al. (2020) present a learning ranking algorithm to identify which test suites would give best compromise between runtime and test coverage. This is important, given that tests suites often run more than 10 minutes, and developers feel like they do not have time to run them locally. (Pinto et al. 2018).

3.17 Overview

Figure 4 walks through the automation opportunities in software development process. Modern, extensible and purpose-built general text editors provide a rich set of features that speed up development. Intelligent code completion consists of features like *jump to definition* and *show available member functions*. Linters can be used to enforce common coding conventions and hint for potential pitfalls. Formatters can be combined with linters to provide automatic correction for some violations of defined conventions. Smart snippets contain tabstops with placeholder hints. Crowdsourcing through StackOverflow and similar sites is a common practice used to implement solutions to common problems.

A source control management system and an continuous integration platform are the enabler of a continuous delivery pipeline. Remote, shared git repository can be configured to run a series of commands, called hooks, for all incoming commits. Those hooks can for example be used to reject a commit if it fails to satisfy requirements set for coding style or commit message. Some providers can run dependency scanning on npm packages notifying about known security vulnerabilities. Static application security testing is based on evaluating the source code within the repository against a list of known patterns that can create security

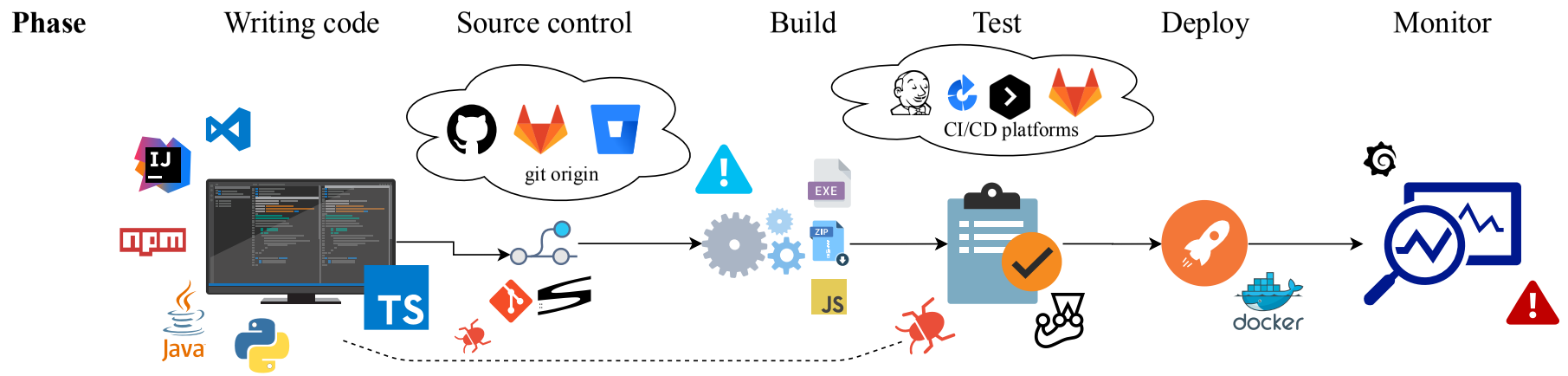
vulnerabilities.

Build automation systems are a set of programs that execute compiler commands based on identified dependencies of the changed source code. Incremental builds are a must for keeping the build times under 10 minutes when commits are made frequently and builds are initiated after each commit. Module bundlers have changed the way web apps are made. They create a dependency graph based on import statements and can perform source-to-source translation from TypeScript, CoffeeScript, Dart etc. to ECMAScript and extended CSS languages like SCSS to CSS. Resulting assets are optimized for lazy loading.

Test automation can be based on dedicated test frameworks or robotic process automation software. In both cases, test suites are a series of scripts which define the part of application to be tested, the inputs to be used and the outcome to be expected. When commits are made frequently, it is important to pinpoint directly which commit broke the application and prevent breaking commits getting deployed to production. For web applications, production environment is the internet and often includes a cloud provider. Containerization is a great way of publishing a highly scalable web application, as updated containers can be spawned on the fly through orchestration software such as Kubernetes. Because container images are versioned and tagged, containerization provides a quick way of recovery in case of failure. If previous images were conserved on the machine, the broken service can be reverted back to previous, functioning version within seconds.

Monitoring provides insight to server load, response times and error sources. Event logging can be initiated to report about errors automatically, and in scripting languages that are used to build web apps it can point directly to the line of code which triggered the error in browser, giving the developers a head start to debugging before user feedback is received through the conventional channels.

Figure 4. Automation opportunities



4 Discussion

This thesis began by describing the typical workflows in software development process and actions in web development. Those actions are process phases that can be summarized under three categories – *writing code*, *testing code* and *deploying code*. Process phases are shared between common software development process models – *the V-model*, *the agile model* and *Test-Driven Development*. In V-model, phases are performed in an ordered sequence, in agile model phases overlap and in TDD the order between writing tests and writing code is reversed.

The goal of this thesis was to identify automation practices and tools in those three process phases. Practical examples were given with ECMAScript, more commonly known as JavaScript. The need for those practices stems from popular agile software development process model, which puts more emphasis on multiple small but frequent deliveries over extensive one-time testing before a single major delivery. Because changes can be delivered to codebase multiple times a day, manual testing is not feasible.

The main enabler of process automation is an integration platform integrated to a version control system. Generally this is git, a version control system popularized by the rise of open source community, but other options are also available. At simplest, server-side git hooks combined with some shell scripting can be used to serve as an integration platform. However, visually more attractive, pre-made integration service with a graphical user interface is often preferred over self-hosted, self-managed git instance combined with self-made shell scripting. Such integration services are available from many vendors, predominantly GitLab, Microsoft (Azure, GitHub), Atlassian, JetBrains and CircleCI.

A rising trend in process automation is artificial intelligence. Via semantic context mapping, it already provides developers with better, more accurate suggestions in code completion. Some authors have demonstrated the use of AI in answering common questions made by developers. Current models are based on searching old documentation, version history, current source code and other supporting material for answers on how to implement a generic feature, or how to use some external API. Other authors have studied the use of artificial intelligence to create better dependency mapping to prioritize tests and improve incremental build quality and speed.

5 Summary

To support agile software development process model, automation tools are required in software development. Tools have evolved rapidly, but their general structure is quite simple and relies heavily on shell scripting and daemons. The main issues are related to keeping the build times and test times reasonable yet reliable. To be able to identify breaking code quicker, it is desirable to perform a build and run necessary tests after each change. In larger projects this is not always feasible due to resource constraints and developers must resort to hourly or nightly builds.

The build system should be modular yet well integrated where parts of it can be replaced when requirements change or a tool is no longer maintained and has been replaced by a better one in the markets. The pool of available tools is large, which creates some challenges in selecting the right tools that support the development of that particular software.

Academic research focuses on build prioritization and problem solving through artificial intelligence, but production-grade solutions utilizing such techniques are not yet available on the markets. Version control system vendors like GitLab and GitHub have complemented open source version control solutions with proprietary and premium supporting features like dependency scanning and static code analysis.

The rapid development of software production tools hinders the credibility of the results presented in this thesis. Some tools will become obsolete and new tools will emerge. As with most of the research on software development tools, it is likely that the results will not stand the test of time, and the reader is therefore advised to use caution and check more recent papers for an updated list of software development practices and automation tools. The general principles hold for the time being – source code goes into the system, some tricks are done and a product comes out from the other end. Revisions of the source code must be managed, and software should be tested before deploying to production.

The rapid development of software development and production tooling presents opportunities to revise papers like this and conduct more up-to-date research. Some of the great journals for up-to-date information on software engineering trends are *Information Systems Journal*, *Business & Information Systems Engineering*, *Empirical Software Engineering* and *IEEE Transactions on Software Engineering*.

REFERENCES

Abdalkareem, Rabe, Emad Shihab, and Juergen Rilling (2017). "What Do Developers Use the Crowd For? A Study Using Stack Overflow". In: *IEEE Software* vol. 34 (2), pp. 53–60 (cit. on p. 24).

Abrahamsson, Pekka, Nilay Oza, and Mikko T. Siponen (2010). "Agile Software Development Methods: A Comparative Review1". In: *Agile Software Development: Current Research and Future Directions*. Ed. by Torgeir Dingsøyr, Tore Dybå, and Nils Brede Moe. Berlin, Heidelberg: Springer, pp. 31–59. Link (visited on 04/05/2020) (cit. on p. 10).

Amrit, Chintan and Yoni Meijberg (2018). "Effectiveness of Test-Driven Development and Continuous Integration: A Case Study". In: *IT Professional* vol. 20 (1), pp. 27–35 (cit. on p. 11).

Barna, Cornel, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu (2017). "Delivering Elastic Containerized Cloud Applications to Enable DevOps". In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 65–75 (cit. on p. 13).

Berger, C. (2003). "Vorgehensmodell für die Entwicklung von Software und Hardware im Traktionsbereich". In: *e & i Elektrotechnik und Informationstechnik* vol. 120 (1), pp. 23–28. Link (visited on 03/22/2020) (cit. on p. 9).

Bezemer, Cor-Paul, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan (2017). "An Empirical Study of Unspecified Dependencies in Make-Based Build Systems". In: *Empirical Software Engineering* vol. 22 (6), pp. 3117–3148. Link (visited on 04/05/2020) (cit. on p. 14).

Brindescu, Caius, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma (2020). "An Empirical Investigation into Merge Conflicts and Their Effect on Software Quality". In: *Empirical Software Engineering* vol. 25 (1), pp. 562–590. Link (visited on 02/23/2020) (cit. on p. 16).

Buhrer, Hans Konrad (2003). "Software Development: What It Is, What It Should Be, and How to Get There". In: *ACM SIGSOFT Software Engineering Notes* vol. 28 (2), p. 5. Link (visited on 05/11/2020) (cit. on p. 9).

Cafaro, Massimo and Giovanni Aloisio, eds. (2011). *Grids, Clouds and Virtualization*. Computer Communications and Networks. London: Springer London. Link (visited on 07/27/2020) (cit. on p. 14).

Chen, Huamin and Prasant Mohapatra (2005). "Using Service Brokers for Accessing Backend Servers for Web Applications". In: *Journal of Network and Computer Applications* vol. 28 (1), pp. 57–74. Link (visited on 07/09/2020) (cit. on p. 12).

Clarke, Paul M., Peter Elger, and Rory V. O'Connor (2016). "Technology Enabled Continuous Software Development". In: *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*. 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED), pp. 48–48 (cit. on p. 17).

Crookshanks, Edward (2014a). "Build Tools and Continuous Integration". In: *Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software*. Ed. by Edward Crookshanks. Berkeley, CA: Apress, pp. 65–77. Link (visited on 07/28/2020) (cit. on p. 15).

— (2014b). "Version Control". In: *Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software*. Ed. by Edward Crookshanks. Berkeley, CA: Apress, pp. 1–30. Link (visited on 07/28/2020) (cit. on p. 15).

Curie, Dasari Hermitha, Joyce Jaison, Jyoti Yadav, and J Rex Fiona (2019). "Analysis on Web Frameworks". In: *Journal of Physics: Conference Series* vol. 1362, p. 012114. Link (visited on 07/09/2020) (cit. on p. 12).

Dooley, John (2011). *Software Development and Professional Practice*. [New York, N.Y.] : New York: Apress ; Distributed to the trade worldwide by Springer. 242 pp. (cit. on pp. 15, 23).

Emmet (2020). *Abbreviations*. Link (visited on 07/11/2020) (cit. on pp. 18, 19).

Erdweg, Sebastian, Moritz Lichter, and Manuel Weiel (2015). "A Sound and Optimal Incremental Build System with Dynamic Dependencies". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 89–106. Link (visited on 05/04/2020) (cit. on p. 20).

Feldman, Stuart I. (1979). "Make — a Program for Maintaining Computer Programs". In: *Software: Practice and Experience* vol. 9 (4), pp. 255–265. Link (visited on 04/05/2020) (cit. on p. 14).

Ghoreishi, Malahat and Ari Happonen (2020). "Key Enablers for Deploying Artificial Intelligence for Circular Economy Embracing Sustainable Product Design: Three Case Studies". In: *AIP Conference Proceedings* vol. 2233 (1), pp. 1–19. Link (visited on 09/11/2020) (cit. on p. 7).

GitHub (2020a). *About Alerts for Vulnerable Dependencies - GitHub Docs*. Link (visited on 08/20/2020) (cit. on p. 21).

— (2020b). *About Code Scanning - GitHub Docs*. Link (visited on 08/20/2020) (cit. on p. 21).

GitLab (2020a). *Dependency Scanning — GitLab*. Link (visited on 08/20/2020) (cit. on p. 21).

— (2020b). *GitLab Runner Docs — GitLab*. Link (visited on 08/25/2020) (cit. on p. 25).

Gloger, Boris (2010). ”Scrum”. In: *Informatik-Spektrum* vol. 33 (2), pp. 195–200. Link (visited on 04/05/2020) (cit. on p. 10).

Happonen, Ari, Malahat Ghoreishi, and Mikko Pynnönen (2020). ”Exploring Industry 4.0 Technologies to Enhance Circularity in Textile Industry: Role of Internet of Things”. In: Working Seminar on Production Economics. Innsbruck, Austria, p. 16 (cit. on p. 7).

Kent, Beck et al. (2001). *Principles behind the Agile Manifesto*. Link (visited on 08/20/2020) (cit. on p. 10).

Kerzazi, Noureddine and Bram Adams (2016). ”Who Needs Release and DevOps Engineers, and Why?” In: *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*. 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED), pp. 77–83 (cit. on pp. 13, 25).

Khan, Asharul Islam, Ali Al-Badi, and Mahmood Al-Kindi (2019). ”Progressive Web Application Assessment Using AHP”. In: *Procedia Computer Science*. The 16th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2019),The 14th International Conference on Future Networks and Communications (FNC-2019),The 9th International Conference on Sustainable Energy Information Technology vol. 155, pp. 289–294. Link (visited on 07/11/2020) (cit. on p. 18).

Kraut, Robert E. and Lynn A. Streeter (1995). ”Coordination in Software Development”. In: *Communications of the ACM* vol. 38 (3), pp. 69–81. Link (visited on 05/11/2020) (cit. on p. 9).

Lee, Jaemyoun, Haegeon Jeong, Won-Joo Lee, Hyo-Joong Suh, Dongeun Lee, and Kyungtae Kang (2018). ”Advanced Primary–Backup Platform with Container-Based Automatic Deployment for Fault-Tolerant Systems”. In: *Wireless Personal Communications* vol. 98 (4), pp. 3177–3194. Link (visited on 02/23/2020) (cit. on p. 14).

Leßenich, Olaf, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen (2018). ”Indicators for Merge Conflicts in the Wild: Survey and Empirical Study”. In: *Automated Software Engineering* vol. 25 (2), pp. 279–313. Link (visited on 02/23/2020) (cit. on p. 16).

Lewis, Nathan, Andrew Case, Aisha Ali-Gombe, and Golden G. Richard (2018). ”Memory Forensics and the Windows Subsystem for Linux”. In: *Digital Investigation* vol. 26, S3–S11. Link (visited on 03/28/2020) (cit. on p. 14).

Liang, Guangtai, Lijun Mei, Shaochun Li, and Liang Chen (2015). ”Exploiting Unique Characteristics of Mobile Backend Services to Recommend Right Ones”. In: *2015 IEEE International Conference on Mobile Services*. 2015 IEEE International Conference on Mobile Services, pp. 464–467 (cit. on p. 12).

Liang, Jingjing, Sebastian Elbaum, and Gregg Rothermel (2018). ”Redefining Prioritization: Continuous Prioritization for Continuous Integration”. In: *2018 IEEE/ACM 40th Interna-*

tional Conference on Software Engineering (ICSE). 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 688–698 (cit. on p. 24).

Lin, Ze-qi, Bing Xie, Yan-zhen Zou, Jun-feng Zhao, Xuan-dong Li, Jun Wei, Hai-long Sun, and Gang Yin (2017). "Intelligent Development Environment and Software Knowledge Graph". In: *Journal of Computer Science and Technology; Beijing* vol. 32 (2), pp. 242–249. Link (visited on 08/18/2020) (cit. on p. 26).

Lwakatare, Lucy Ellen et al. (2019). "DevOps in Practice: A Multiple Case Study of Five Companies". In: *Information and Software Technology* vol. 114, pp. 217–230. Link (visited on 02/23/2020) (cit. on p. 13).

Maudoux, Guillaume and Kim Mens (2018). "Correct, Efficient, and Tailored: The Future of Build Systems". In: *IEEE Software* vol. 35 (2), pp. 32–37 (cit. on p. 16).

McHugh, Martin, Oisín Cawley, Fergal McCaffery, Ita Richardson, and Xiaofeng Wang (2013). "An Agile V-Model for Medical Device Software Development to Overcome the Challenges with Plan-Driven Software Development Lifecycles". In: *Proceedings of the 5th International Workshop on Software Engineering in Health Care*. SEHC '13. San Francisco, California: IEEE Press, pp. 12–19 (cit. on p. 11).

McIntosh, Shane, Bram Adams, and Ahmed E. Hassan (2012). "The Evolution of Java Build Systems". In: *Empirical Software Engineering* vol. 17 (4), pp. 578–608. Link (visited on 04/05/2020) (cit. on p. 14).

Microsoft (2020). *Snippets in Visual Studio Code*. Link (visited on 08/20/2020) (cit. on p. 19).

Miller, Craig S. and Randy Connolly (2015). "Introduction to the Special Issue on Web Development". In: *ACM Transactions on Computing Education* vol. 15 (1), pp. 1–5. Link (visited on 07/07/2020) (cit. on p. 12).

Mårtensson, Torvald, Pär Hammarström, and Jan Bosch (2017). "Continuous Integration Is Not About Build Systems". In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 1–9 (cit. on pp. 14, 16).

Nelson, Kay M., Sucheta Nadkarni, V. K. Narayanan, and Mehdi Ghods (2000). "Understanding Software Operations Support Expertise: A Revealed Causal Mapping Approach". In: *MIS Quarterly; Minneapolis* vol. 24 (3), pp. 475–507. Link (visited on 05/05/2020) (cit. on p. 13).

Newman, C., Z. Agioutantis, and N. Schaefer (2017). "Development of a New Web-Based Platform for Ground Control Applications". In: *Mining Engineering; Littleton* vol. 69 (10), pp. 38, 40, 42–44. Link (visited on 07/09/2020) (cit. on p. 12).

O'Regan, Gerard (2019). *Concise Guide to Software Testing*. Undergraduate Topics in Computer Science. Cham: Springer International Publishing. Link (visited on 07/28/2020) (cit. on p. 21).

Pan, Zhelong and Rudolf Eigenmann (2008). ”PEAK—a Fast and Effective Performance Tuning System via Compiler Optimization Orchestration”. In: *ACM Transactions on Programming Languages and Systems* vol. 30 (3), pp. 1–43. Link (visited on 08/18/2020) (cit. on p. 25).

Piili, Heidi, Thomas Widmaier, Ari Happonen, Jari Juhanko, Antti Salminen, P. Kuosmanen, and Olli Nyrhilä (2013). ”Digital Design Process and Additive Manufacturing of a Configurable Product”. In: *Journal Advanced Science Letters* vol. 19 (3), pp. 926–931 (cit. on p. 7).

Pinto, Gustavo, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças (2018). ”Work Practices and Challenges in Continuous Integration: A Survey with Travis CI Users”. In: *Software: Practice and Experience* vol. 48 (12), pp. 2223–2236. Link (visited on 02/05/2020) (cit. on pp. 15, 23, 26).

Prometheus (2020). *Overview — Prometheus*. Link (visited on 08/20/2020) (cit. on p. 25).

Regulwar, Ganesh B., P. R. Deshmukh, R. M. Tugnayat, P. M. Jawandhiya, and V. S. Gulhane (2010). ”Variations in V Model for Software Development”. In: *International Journal of Advanced Research in Computer Science; Udaipur* vol. 1 (2). Link (visited on 03/22/2020) (cit. on p. 9).

Richter, Cedric, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim (2020). ”Algorithm Selection for Software Validation Based on Graph Kernels”. In: *Automated Software Engineering* vol. Link (visited on 05/04/2020) (cit. on p. 26).

Romano, Simone, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo (2017). ”Findings from a Multi-Method Study on Test-Driven Development”. In: *Information and Software Technology* vol. 89, pp. 64–77. Link (visited on 03/22/2020) (cit. on p. 11).

Sadowski, Caitlin and Thomas Zimmermann, eds. (2019). *Rethinking Productivity in Software Engineering*. Berkeley, CA: Apress. Link (visited on 07/28/2020) (cit. on p. 15).

Sawyer, Steve (2004). ”Software Development Teams”. In: *Communications of the ACM* vol. 47 (12), pp. 95–99. Link (visited on 05/14/2020) (cit. on p. 9).

Schewe, Klaus-Dieter and Bernhard Thalheim (2019). *Design and Development of Web Information Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. Link (visited on 07/28/2020) (cit. on p. 12).

Sentry (2020). *Dashboards*. Link (visited on 08/20/2020) (cit. on p. 25).

Shahzad, Farrukh (2017). ”Modern and Responsive Mobile-Enabled Web Applications”. In: *Procedia Computer Science*. 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops vol. 110, pp. 410–415. Link (visited on 07/09/2020) (cit. on p. 17).

Soltani, Jafar (2016). ”Adopting Continuous Delivery in AAA Console Games”. In: *Proceedings of the 4th International Workshop on Release Engineering*. RELENG 2016. Seattle, WA, USA: Association for Computing Machinery, pp. 5–6. Link (visited on 04/07/2020) (cit. on p. 16).

Song, Yiguang, Li Hu, and Ming Yu (2018). ”A Novel QoS-Aware Prediction Approach for Dynamic Web Services”. In: *PLoS ONE* vol. 13 (8), pp. 1–21. Link (visited on 07/09/2020) (cit. on p. 12).

StackOverflow (2020). *Stack Overflow Developer Survey 2020*. Link (visited on 08/20/2020) (cit. on p. 24).

StatCounter (2020a). *Desktop vs Mobile vs Tablet Market Share Worldwide*. Link (visited on 07/11/2020) (cit. on p. 17).

— (2020b). *Mobile Operating System Market Share United States Of America*. Link (visited on 07/11/2020) (cit. on p. 17).

— (2020c). *Mobile Operating System Market Share Worldwide*. Link (visited on 07/11/2020) (cit. on p. 17).

Stober, Thomas and Uwe Hansmann (2010a). ”Overview of Agile Software Development”. In: *Agile Software Development: Best Practices for Large Software Development Projects*. Ed. by Thomas Stober and Uwe Hansmann. Berlin, Heidelberg: Springer, pp. 35–59. Link (visited on 03/28/2020) (cit. on pp. 10, 11).

— (2010b). ”The Flaw in the Plan”. In: *Agile Software Development: Best Practices for Large Software Development Projects*. Ed. by Thomas Stober and Uwe Hansmann. Berlin, Heidelberg: Springer, pp. 1–14. Link (visited on 03/28/2020) (cit. on p. 9).

— (2010c). ”Tooling”. In: *Agile Software Development: Best Practices for Large Software Development Projects*. Ed. by Thomas Stober and Uwe Hansmann. Berlin, Heidelberg: Springer, pp. 61–73. Link (visited on 03/28/2020) (cit. on p. 14).

Suganuma, Toshio, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani (2005). ”Design and Evaluation of Dynamic Optimizations for a Java Just-in-Time Compiler”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 27 (4), pp. 732–785. Link (visited on 08/18/2020) (cit. on p. 25).

Torvalds, Linus (2020). *Linux Kernel Source Tree.* Link (visited on 06/30/2020) (cit. on p. 23).

Wagner, Stefan and Florian Deissenboeck (2019). ”Defining Productivity in Software Engineering”. In: *Rethinking Productivity in Software Engineering*. Ed. by Caitlin Sadowski and Thomas Zimmermann. Berkeley, CA: Apress, pp. 29–38. Link (visited on 07/29/2020) (cit. on p. 15).

Vale, Gustavo, Angelika Schmid, Alcemir Rodrigues Santos, Eduardo Santana de Almeida, and Sven Apel (2020). ”On the Relation between Github Communication Activity and

Merge Conflicts". In: *Empirical Software Engineering* vol. 25 (1), pp. 402–433. Link (visited on 02/23/2020) (cit. on p. 16).

Wilde, Evan and Daniel German (2018). "Merge-Tree: Visualizing the Integration of Commits into Linux". In: *Journal of Software: Evolution and Process* vol. 30 (2), e1936. Link (visited on 02/23/2020) (cit. on p. 23).

Xu, Dianxiang, Weifeng Xu, Manghui Tu, Ning Shen, William Chu, and Chih-Hung Chang (2016). "Automated Integration Testing Using Logical Contracts". In: *IEEE Transactions on Reliability* vol. 65 (3), pp. 1205–1222 (cit. on p. 22).

Zammetti, Frank (2020). *Modern Full-Stack Development: Using TypeScript, React, Node.Js, Webpack, and Docker*. Berkeley, CA: Apress. Link (visited on 08/04/2020) (cit. on pp. 14, 24).

Appendix 1. Robot Framework example

```
*** Settings ***
Library             SeleniumLibrary

*** Variables ***
${SERVER}           localhost:7272
${BROWSER}          Firefox
${VALID USER}       usr
${VALID PASSWORD}   pwd
${LOGIN URL}        http://${SERVER}/login.html
${MAIN URL}         http://${SERVER}/main.html
${ERROR URL}        http://${SERVER}/404.html

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button
```

Appendix 1. (continued)

```
Welcome Page Should Be Open
    Location Should Be    ${MAIN URL}
    Title Should Be       Main Page
```