ISSN 2347 - 3983
Volume 9. No. 7, July 2021
International Journal of Emerging Trends in Engineering Research
Available Online at http://www.warse.org/IJETER/static/pdf/file/ijeter28972021.pdf
https://doi.org/10.30534/ijeter/2021/28972021

Towards continuous deployment of a multilingual mobile app

Christian Schindler [1], Kirshan Kumar Luhana [2], Wolfgang Slany[3]
[1,2,3]Graz University of Technology Graz, Austria
Corresponding authors E-mail:cschindler@ist.tugraz.at
kirshan.luhana@student.tugraz.at

ABSTRACT

In this paper, speed and reliability improvements of the deployment of Catrobat's Pocket Code are described. Pocket Code is open source and has over 500 contributors and about 28,000 active installs. It is a multilingual application, also supporting right-to-left languages such as Arabic, Farsi, and Sindhi. A major challenge to continuous deployment is the mandatory manual acceptance testing done by product owners. A second major challenge is the maintenance of an up-to-date app description in multiple languages: For Google Play, the app-description including screenshots must be translated to all supported languages. This leads to a huge number of repetitive tasks. These tasks, when carried out by humans are not only prone to errors but also the time needed, and the quality of the outcome differs between their executions. For instance, if screenshots for the descriptions are created manually, deployment is further deferred. Therefore, automatic screenshot creation for all languages is highly desirable. This paper describes our solution for continuous deployment facing these challenges using Fastlane (app-release tool), and Jenkins (continuous integration server), and the staged deployment approach of Google. The latter supports postponing deployment steps that are not easily automatable to later phases. Overall, the approach was successfully implemented for Pocket Code. It was also tested with another Catrobat app, Pocket Paint. This shows, the approach can be transferred to fit the deployment process of other multilingual apps.

Key words: Continuous deployment; Internationalization (I18n); Mobile application; Pocket Code

1. INTRODUCTION & BACKGROUND

1.1. Catrobat Project

The Catrobat[1]project was initiated in 2010 by Prof. Slany at the Institute of Software Technology at Graz University of Technology. The name Catrobat is used for a set of creativity tools for various platforms and mobile devices as well as a visual programming language inspired by the well-known Scratch[2]framework which was developed at the MIT Media Lab. It is hosted on GitHub; it is free and independent and uses an open source license. More than 500 volunteers from over 20 countries contribute to this project. Agile methods such as Extreme Programming and their underlying principles are used for development and management of the project [1]. The focus of the Catrobat project is the development and improvement of Pocket Code for Android [2] and iOS.

1.2. Pocket Code

Pocket Code is the integrated development environment (IDE) for the brick based visual programming language Catrobat. Within Pocket Code, programs can be created, executed, uploaded to, and downloaded from its web sharing platform[3]. It is publicly available for the Android platform and in beta testing phase for the iOS platform. Pocket Code is thoroughly internationalized (I18n) and localized (L10n) and can also be deployed in various flavors with custom features, e.g., Phiro[4] or Create@School[5]. At the time of writing, the Google developer console statistics show that Pocket Code (for Android) has been downloaded over 500k+ times in more than 140 countries.

1.3. Continuous practices

In the software industry, continuous practices based on agile methods are emerging to mitigate the gap between development and deployment [3, 4]. Through the implementation of continuous practices feedback loops between developers and customers can be shortened which improves the quality of the delivered product. Frequent releases lead to a perceived increase in developer confidence, improved customer satisfaction and bonding as well as an increase of productivity [3,5].

1.4. Continuous integration

The term *continuous integration* (CI) was introduced by Martin Fowler along with the twelve practices of the development process called *Extreme Programming* (XP). CI advocates frequent integration of new code to a centralized repository [6–8]. One goal of continuous integration is to create and maintain a codebase which, potentially, could be rolled out at any given moment. Core capabilities of the integration environment that enable such continuous

[1] https://catrobat.org
[2] https://scratch.mit.edu/
[3] https://share.catrob.at/pocketcode/
[4] http://www.robotixedu.com
[5] https://edu.catrob.at/no1leftbehind-for-teachers

integration are automatic building and testing with different granularity levels [3]. This supports teams to improve integration and quickly find bugs while reducing the risk of a last minute release cancellation [6]. Therefore, every new code which is to be integrated into the code base is issued via a pull request (PR) and must be automatically built and tested to ensure system stability. Many tools are available, e.g., Bamboo CI[6], Codeship[7], TeamCity[8], and Travis CI[9], to name a few, which all support automatic building, testing, and packaging of new versions of the software with or without human intervention [9].

1.5. Continuous delivery

Continuous delivery (CDe) is an important practice for agile software development. In [10] the author found out that from an organization's point of view continuous delivery is rated the second most important agile software development practice. The combined practices of continuous integration, automated testing and quality checking which enable teams to continuously keep their software 'ready to release' defines CDe [3, 11,12,13].

1.6. Continuous deployment

Continuous deployment means to install every change in the software directly, so it is used in production. Similar to CDe, where the final deployment step is manual, in CD, this step as well is fully automatic [3]. The main advantage of CD is shortening the time to market for any changes which leads to improved productivity [12, 14]. There are several application domains where CD is challenging [5]. For example, in the telecommunication domain deploying applications to the network requires configuration for their different clients. In the medical sector standard checks and certifications are required before a new system can be implemented. At factory sites automation control systems must be shut down for deployment which cannot be done frequently due to financial loss. Also, certain app stores, such as Apple's App Store, have quality assurance measures which prevent CD.

1.7. Deployment pipeline

A staged release process (the way to progress through the release process in stages), is called a deployment pipeline [12, 15]. Downloading the latest code from the repository and building the binaries for further use is usually the first stage of a deployment pipeline [15]. The term *pipeline* does not necessarily imply *automatic*. Depending on the business needs, the used tools or any other technical limitation of each stage of the pipeline can be either manual or automatic. Sometimes human intervention or authorization is required, which poses a delay in the pipeline. Deploying the software to the production environment is usually the last stage of the pipeline. A manual deployment pipeline poses the following challenges:

- People with tacit deployment knowledge. Consider the possibility that the person who usually deploys is on vacation or gets ill - who will take over?
- Deployment documentation of a manual deployment process is in most cases outdated. Chances of errors during deployment escalate, especially when the responsible person changes.
- Usually manual deployment needs a long time; therefore, minor bug fixes do not trigger a new deployment. A severe bug, which is discovered late during the deployment process, renders the current deployment obsolete, i.e., a lot of expensive (manual) work is wasted.

Repetitive and boring tasks increase the chance of errors during manual deployment. In automatic deployment, this is not an issue, and therefore, automation of deployment reduces the risks of such errors. Secondly, release time is significantly reduced and the reliability (stay timeboxed) is increased with automation. Having an automated deployment pipeline any authorized person can trigger the deployment without detailed or technical knowledge of the deployment steps. The automated deployment approach does not need extra documentation since all steps are implicitly documented by the executable deployment script.

1.8. Release strategy

Any new version of an app due to a bug fix, a new feature, a feature change or a feature upgrade is potentially a new release. Nayebi et al. [16] state that the release strategy of an app has a direct impact on the success of a mobile app. The release strategies of free open source software (FOSS) projects usually can be categorized either as time or feature based [17]. No matter which strategy is chosen (time or feature based), in FOSS projects a tendency can be observed that frequent releases are increasingly favored [18]. Frequent releases mean that the scope of new features and hence the amount of new code is limited which in turn reduces the risk of errors for the release [19]. Three main motivations for a frequent release approach adoption were identified by [18], a.) the increase of project attractiveness, b.) improvement of maintenance, and c.) the increase of market share. Project attractiveness and market share are also apparently influenced by the level of internationalization and proper localization of the app. According to a study by Google and AdMob, the number of users who have stopped using an app, because it was not localized properly, varies between 34% and 48% [20]. The difference in percentage depends on the origin of the data(USA; China; Japan; UK and South Korea). Both, a frequent release approach, and the amount of work to release a multilingual app properly, imply the necessity of an automated deployment pipeline. Due to the frequent developments in hardware, services and platforms the consumer IT market rapidly grows [21]. Consumers can

[6] https://www.atlassian.com/software/bamboo
[7] https://codeship.com/
[8] http://www.jetbrains.com/teamcity
[9] https://travis-ci.com

pick an app from a vast library of apps with similar functionality according their business needs and personal preferences. Therefore, software vending companies must pay attention to what customers want and react and adapt properly to match the changes in the IT market [21]. Being faster than competitors in terms of feature development according to customers' wishes and faster quality releases of the software is an important success factor [22].

1.9. Localization and internationalization

Localization (L10n) means to adapt a product or its content to meet language and cultural requirements of a specific market [23]. Whereas internationalization (I18n) is the creation of the prerequisites for that a product can be easily localized [23]. Both are considered as important factors to attract users and increase market share globally [24]. When apps are properly translated and users' cultural values are reflected they feel more comfortable and productive. Pocket Code follows design principles for I18n and L10n [25]. Support for different languages including right to left languages (Arabic, Urdu, or Farsi etc.) is implemented on application level which means switching between languages can be done in the app hence there is no need to change system settings. Furthermore, with this approach it is possible to support even languages which are not supported by the system. Localizing an app needs translation by professionals or volunteer native speakers. Any user can contribute to the translation of a project by using one of the many user friendly desktop and online applications for managing multilingual projects. With such tools users are enabled to edit translations online or to export the project's translated string resources in various formats. Since Pocket Code is a free and open source project it can make use of the Crowdin [10] localization management platform free of charge [11]. It facilitates all stakeholders to translate an application's string resources easily and in a reasonable amount of time. Crowdin maintains a RESTful API over HTTP. With GET and POST commands files can be up- or downloaded. Also, web-hooks are supported for integration with source code management platforms such as GitHub[12]. For the Catrobat project in Crowdin more than 500 volunteers are registered and help to translate Pocket Code into various languages. More than 57 languages have been translated (some only partially due to the voluntariness of the project) which can be selected and used in Pocket Code. On Google Play [13] in contrast the applications details (description including screenshots) are available only for 26 languages.

1.10. Integration & deployment tools in Catrobat

The Catrobat project uses the Jenkins-CI server to continuously integrate the code for the Android application. For continuous deployment the tool Fastlane is used.

1.10.1. Jenkins

Jenkins [14] is a tool for build automation which facilitates building and testing. Using Jenkins, integration can be triggered manually, or by external events, e.g., GitHub pull requests, or can be configured to run the build on a regular basis. Jenkins is free and open source and is designed around a simple extension mechanism, so anyone can write plugins to customize its behavior. By using different plugins the functionality of Jenkins can be changed and adapted to the project's needs, e.g., from being a mere continuous integration, a continuous delivery, or a full blown continuous deployment tool. Furthermore, there are many convenience plugins to facilitate reporting of build and test-results.

1.10.2. Fastlane

Fastlane[15]is a tool to automate a deployment pipeline[16]. It is free and open source and can be used for Android and iOS apps. When releasing a multilingual app to an app store, such as Google Play, the description including screenshots must be translated to properly reflect the app's multilingualism. When screenshots are created manually, usually different people are required who are capable of understanding and operating the device in the various supported languages. This is monotonous and error prone. Fastlane helps to automate these tasks. Also signing the app and uploading it to an app store can be done automatically with Fastlane. Fastlane was chosen since the provided functionality fit the project's needs and there were no proper and free alternatives found. Another reason Fastlane was chosen is the large and active community which drives the development of the Fastlane feature set, provides excellent documentation and support in differentcommunity forums such as Stackoverflow[17]and on their GitHub project page[18].

1.11. General Pocket Code deployment challenges

Developing tools to support critical tasks and interactions are increasingly being focused by human factor researchers [26]. To increase reliability, availability and performance, these tasks must be automated as far as possible which in turn also improves performance, and productivity, hence automation saves time and money. In Catrobat's Pocket Code deployment process the main challenges are:

[10] https://crowdin.com
[11] https://crowdin.com/pricing
[12] https://support.crowdin.com/github-integration
[13] https://tinyurl.com/gplay-langsupport
[14] https://jenkins.io
[15] https://fastlane.tools
[16] https://docs.fastlane.tools
[17] https://stackoverflow.com/questions/tagged/fastlane
[18] https://github.com/fastlane/

- Pocket Code's many supported languages must be reflected by the description including screenshots which is shown in the app store.
- The product owners have to accept the app's changes before release and therefore, the acceptance tests are currently done during the deployment preparation which delays the actual deployment.
- Currently, the release responsible person has to fulfill many manual steps to set up the environment to build, sign, and align the Android application package (APK) as well as manually test the release candidate for the actual app store upload.

Many people of the Catrobat team work together to achieve the goal of a releasable deployable artifact and all involved individuals depend on each other, if one works slow the whole deployment process is deferred. One of the responsible development team members creates the release branch, after that a senior member with access to the credentials for the keys signs, aligns and uploads the APK to our internal cloud for acceptance testing by the product owners who test for the final go/no-go decision. On final approval, an authorized member uploads the APK to Google Play. Usually the app's metadata (i.e., description and screenshots) for all languages is not updated frequently since this is, when done manually, a tedious and monotonous process. Therefore, the application and its description including the screenshots diverge with the time, which potentially irritate customers when they don't get what they see. In the following sections, we describe how the current deployment is transformed to a continuous process with almost no human interaction.

1.12. Fastlane and Crowdin challenges

To combine the Fastlane and Crowdin tools the file-structure and directory naming pattern has to be adapted. Fastlane's Screengrab tool is capable of capturing screenshots by changing the system locals settings. It then retrieves the screenshots from the emulator or the device for further use in the metadata compilation process. Screengrab names its folders with the following naming convention *languageCode-CountryCode*, e.g., *en-US*, *en-UK*, *de-DE*, *ur-PK* (see the source code of Screengrab[19]). Crowdin on the other hand uses a different naming convention for its export feature. It exports the project's languages compressed as a zip file having a separate folder for every language. All folders contain an XML file named *google play.xml*. This XML-file has four properties the *title*, the *description*, the *promotion text* (i.e., the short description), and the *app updates* (*what's new* section which is not yet maintained with Pocket Code). At the time of writing, Pocket Code offers app localization (L10n) and internationalization (I18n) for 57 languages. This includes languages which are not supported by the Android system, for example Sindhi, and Pashto. Pocket Code, furthermore, supports different dialects such as *French-African* and *French-French*. These dialects are also not supported by Google Play. Crowdin cannot know and is not aware of which languages in general and which dialects in particular are supported by Google Play. In its language export all available translations are included even those which are not used and therefore cannot be uploaded to Google Play but which are used in the Pocket Code app. At the time of writing, Google Console supports app listings in 78 languages[20]. Unfortunately, this is not done in a consistent and uniform way. Some languages are named using only the *languagecode* and others with *languagecode-CountryCode*, e.g., *ar* for all Arabic languages, *hr* for Croatian, *ca* for Catalan but *cs-CZ* for Czech, *en-US* for English United States, *en-UK* for English United Kingdom. This behavior seems to be unspecified or random. In any case it must be taken care of, which means that the directories created by Fastlane must be made compatible with Google Play for that the app descriptions and screenshots are accepted for each language.

2. CATROBAT STATUS PRIOR TO AUTOMATION

2.1. Code Quality and Releases

In the Catrobat project about a third of the volunteers are students who participate less than 6 months. Student volunteers enter the project with heterogeneous knowledge skills related to object oriented programming (OOP), agile development methods and XP-practices. From our experience, development of any new feature needs approximately six months due to getting acquainted with the project and the development. In average students need about three months to become productive. This frequently leads to features which are finished, if at all, at the end of the volunteers' participation. These features are either rushed to release, handed over to succeeding student volunteers for further development or finalization, or in the worst case are abandoned. All three options are less than perfect, however features which are released too early show the lack of clearly defined acceptance criteria this must be addressed by the product owners. An established frequent release plan distributed over the year can help in this regard. In Figure 1 the commits per week over the last two years are depicted whereas the releases are marked with diamonds. Catrobat project's release strategy is arbitrary. When having a look at the releases documented on GitHub[21]one can see that there are features, e.g., 'NFC', 'Backpack', '2D physics' or 'landscape mode' which appear in more than one release. In case of 'NFC' it was introduced in version 0.9.22 in June 2016 and last time improved with version 0.9.29 in September 2017. The 'Backpack' feature was introduced in version 0.9.21 in February 2016 and usability improved with version 0.9.28 in March 2017. Both examples appear in four releases over a span of a year, which indicate quality issues.

[19] https://tinyurl.com/fastlane-screengrab-src

[20] https://support.google.com/googleplay/androiddeveloper/answer/113469

[21] https://github.com/Catrobat/Catroid/releases

Evidence for quality issues are pull requests (PR) like #2689[22]or #2682[23]where with the latter over 44k lines were removed. This shows that there is a lot of useless code in the code base. Such quality issues in the Catrobat project can be approached from various sides, e.g., decreasing time to production by improving the deployment process. The Catrobat project relies on participation of its volunteers and frequentreleases raise motivation and status of its volunteers. Working on a project with an arbitrary release strategy potentially discourages volunteers when their contributions are not released until they leave the project. Knowing that through development work market share is gained also increases motivation. Furthermore, [18] states that the main advantages of frequent releases are along quick feedback and bugfixes, increased efficiency, new collaborators and an increased quality focus on behalf of development.It is worthwhile to investigate the effect of frequent, regular, releases on the Catrobat project, the motivation of the volunteers as well as the code quality. In [27] we lay the base for a frequent release strategy in the Catrobat project and further elaborate the shift from arbitrary towards continuous deployment in this paper.

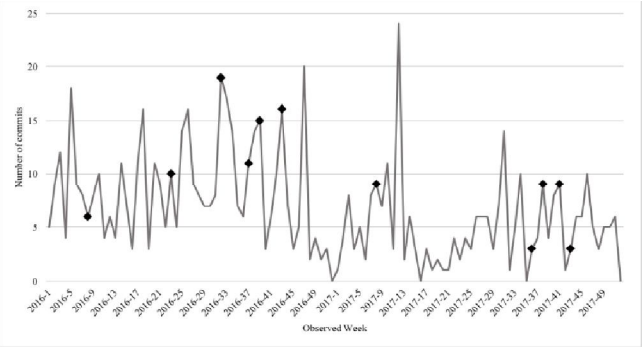


Figure 1: Commits per week to the main Catrobat repository (Catroid) in 2016 and 2017. Public full-stage releases are marked with diamonds

2.2. Deployment process status quo

An app release must be planned carefully since failures in the app or errors during the release process potentially have a negative impact on the organization's reputation, the budget, and the users [28]. The Catrobat project's deployment process is easily automatable in large parts but this was not considered until now and therefore, this venture was always postponed. The project is hosted on GitHub and therefore, uses git as its versioning system. A branching model is used where a *master* branch reflects the status of the currently released project and a default *development* branch reflects the current state of development which nevertheless is all time potentially releasable. Code which would break the integrity of the development branch will not be integrated and cannot be merged. Severe and urgent bugs in the released app are handled in a way that a hotfix branch is created from the master branch. The bug is fixed there and the changes are merged back into the master branch and the development branch and the APK of the app is updated in the app store. This method is under constant evaluation and discussion whether this is the optimal solution but it is established within the project and accepted by the development team. The actual deployment phase of the Catrobat project starts after feature development has been finished. The code must have been reviewed by peer developers, integrated as agreed upon with the development team's coordinator and flagged as potentially shippable by the product owners after they successfully went through their feature acceptance tests. The single steps of the deployment are very similar. Yet they have not been automated since there are still many manual interactions in the workflow (see Figure 2). When the feature development phase including continuous integration has been finished the release responsible person will create the release branch by branching off the development branch. This branching-off can only be done when the development branch is potentially shippable which means that it contains only code which has been positively tested and considered as not breaking the application. This freshly created release branch is once again thoroughly tested on Jenkins by executing all automated test (lint, PMD, unit, integration, and acceptance) to ensure integrity of the release branch. When during execution no errors have occurred, the outcoming APK is considered a potentially shippable artifact. This APK along with the release notes is published in our internal information system. As soon as it is accessible the product owner acceptance testing starts which means the app is thoroughly manually tested according to predefined scenarios but also in an exploratory manner. The product owners either accept or reject this release candidate as a whole. A rejection by the product owners means that they found issues with the release candidate which are rated as "showstoppers" and which need to be fixed immediately. The product owners create a list of prioritized issues in the issue tracking system, the release is cancelled and the issues are handed over to the development to fix these issues and enhance the testbase ensure that a reintroduction of these issues is found during future testruns. When the issues are fixed the deployment chain is entered again from the beginning. After acceptance through the product owners, the release candidate is eligible for deployment. When there have been features which changed the UI (user interface) the screenshots for the app-store have to be captured. Up until now this means that the screenshots have to be taken manually for all languages to update the metadata for the app store. For being able to upload to Google Play, the APK must be signed and aligned. After this step the upload of the APK to the app store, along with the description and screenshots in all supported languages, can be done. The final step is the announcement of the release via Catrobat project's communication channels. There exists a detailed

[22] https://github.com/Catrobat/Catroid/pull/2689
[23] https://github.com/Catrobat/Catroid/pull/2682

step by step documentation in the internal information system but it is subject to continuous adjustments and optimizations. The team constantly works on improvements to streamline the workflow with the goal to reduce errors and alleviate long-winded deployment steps. Human beings have problems to keep their concentration on boring tasks such as creating screenshots of the same app configuration in different languages and sorting them into appropriate folders. Put simply, they are bad at repetitive tasks [29]. A deployment process, when it contains manual tasks, is error-prone. This is especially true if the number of repetitive manual tasks scale up due to, e.g., an increased number of supported languages, or different flavors. Both examples significantly increase the number of manual tasks in Catrobat's deployment process. Senior team members despite their skills become even faster bored than rookies and feel unchallenged which quicker leads to an increase of introduced errors because of repetitiveness of the manual tasks [11, 30].

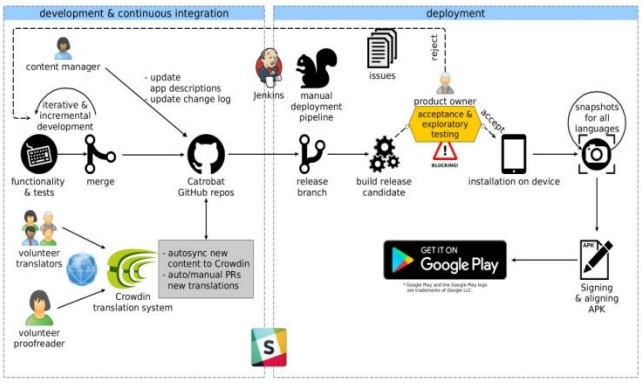


Figure 2: Catrobat manual deployment workflow

2.3. Rapid increase of manual steps

The Pocket Code app exists in different flavors, which means that there are custom build versions for partners available having a different feature set and a different outlook. Such flavors must be separately released in the same way as the original app, with all metadata for the different languages. A future goal is to support different app stores and increase the efforts to further extend the number of supported languages. The downside of this idea is an explosion of repetitive manual steps. This poses a real problem which can only be tackled by automation of manual deployment tasks. If automation is not possible, such manual interventions must be removed or shifted from the deployment phase into a pre- or postdeployment phase.

3. STEPS TOWARDS CONTINUOUS DEPLOYMENT

Continuous software engineering (including continuous deployment and release) heavily depends on applying automation to the overall software development process [31]. Therefore, the goal is to eliminate all repetitive manual steps. The deployment phase must be automated as much as possible to eradicate errors introduced by manual (i.e., human) intervention. The meanwhile *old* Catrobat deployment workflow is depicted in Figure 2. The single blocking activity in the deployment phase is the final product owner approval of the release candidate. This step must be automated, omitted or moved either before or after the actual deployment phase. It is clear that this step is mandatory but neither can be omitted nor fully automated with reasonable effort and hence the only option is to move it out of this phase. With an automatic app deployment (see Figure 3) it becomes possible to move the final app approval tests after the deployment to Google Play has happened. The only premise is that this APK must not reach the public without the final product owner approval. Fortunately, the Google Play Developer API supports this plan by allowing one to deploy new APKs of an app to one of the following different default release tracks [24] :

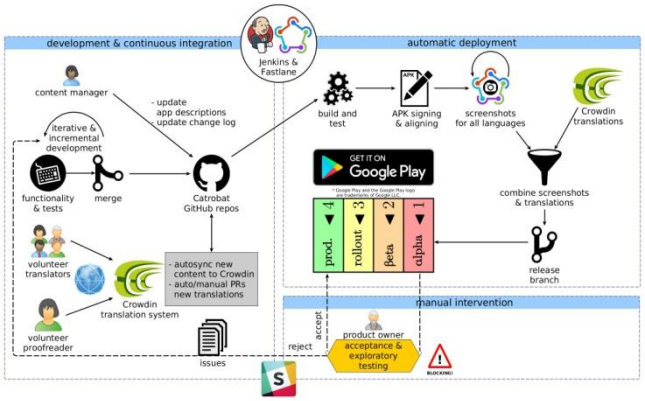


Figure 3: Catrobat automatic deployment workflow

- The **Alpha** track: For this track only alpha testers are subscribed (e.g., product owners) and are able to install the APK.
- The **Beta** track: This track is only for a limited number of invited beta testers.
- The **Rollout** track: Via this track a defined percentage (ranges from 5% to 50%) of the app's users (randomly selected) can be reached and are informed about the possible update.
- The **Production** track: This track is the public track and is used to publish the app for all users.

At the time of writing, the Catrobat project uses only the alpha and production tracks. The only subscribers of the alpha track are the product owners who are informed about the availability of the new APK in their track. They can now install the app via Google Play on their devices. This is an advantage compared to the manual installation in the previous deployment approach, where they had to copy the APK to their devices, allow "manual installation" by activating the "Unknown sources" option in the Android security settings, and then manually install the APK on the device. On product owner approval of the alpha track

[24] https://developers.google.com/android-publisher/tracks

version a Jenkins job is triggered to finalize the deployment. Finalization means that the APK is promoted to the production track and the metadata, i.e., the app description and screenshots in all languages are uploaded. If there are problems, either during the product owner approval tests or during automatic deployment, the communication happens via Catrobat's Slack [25] infrastructure in the appropriate channel. The best case is, that the deployed APK is moved from the alpha to the production track and its metadata is uploaded and displayed for the different languages. In case of errors, reports are communicated to the developers, translators or designers who have to fix them. With Jenkins and Fastlane the following steps have been automated. The deployment in the best case boils down to triggering two events, a.) the automatic deployment to the alpha track, and b.) after the product owner approval the promotion from the alpha to the production track and the upload of the updated metadata (including screenshots for all languages). The whole deployment pipeline job with Jenkins/Fastlane, is a simple sequence of steps (see Figure 4), which only progresses if the previous step succeeded.

3.1. Preparations for alpha track deployment

The first part of the Jenkins job with the name *deploy to alpha track* sets up the environment by cloning the developmentbranch with the latest releasable code. Then it builds the release candidate, i.e., the release-APK and the debug-APK, and runs all necessary tests (LINT, PMD, UI) using the Android emulator.

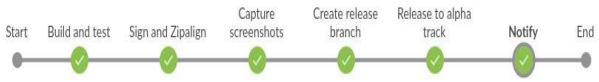


Figure 4: Catrobat's Pocket Code deployment pipeline

3.2. Signing and aligning the APK

Once the release-APK was built, the next step of the Jenkinsjob is to sign and align the release-APK. For the signing process a certificate is needed which is managed and stored by the credential plugin of the Catrobat Jenkins server. The next step is to align the APK. Aligning ensures that all uncompressed data such as images, are aligned on a 4-Byte boundary which allows direct access to all parts and reduces RAM consumption even if a part contains binary data with alignment restrictions [32]. It is recommended to always use zipalign before distributing APKs to end-users [32]. All android apps must be digitally signed with a certificate prior to be able to distribute them via Google Play [33]. To be able to publish updates for an app it is important that all versions are always signed with the same certificate.

3.3. Screenshots for all languages

Within this part of the deployment the most interesting and fruitful improvement possibilities in terms of speed and reliability are contained. To get a consistent multilingual app description on Google Play not only the description text has to be translated but also the screenshots have to be taken in the different languages. Therefore, the language settings have to be changed and the app has to be put into the same configuration for all languages to take the screenshots. The person who is responsible for this work most likely does not understand all the different languages which makes it hard to operate the app and navigate to the desired configuration. When this must be done manually not only is this a time consuming task but it is a very error-prone one too. Furthermore, the screenshots are stored on the device and must be downloaded and then combined with the various translations of the app's description to form the metadata used on Google Play. These tasks can fortunately be automated with Jenkins and Fastlane. When the previous pipeline steps were successfully executed Jenkins runs the Fastlane screengrab tool to capture screenshots for the same app configuration in all supported languages. These screenshots are to be used for the app description on Google Play. The screengrab tool automatically changes the system language and then captures the screenshots with the tests in the Espresso[26]test package. In Figure 5 automatically created screenshots for six different languages of the same simple app configuration can be seen. Some of those screenshots are only partially translated which is due to the voluntariness of the project. The app is put automatically into the same simple configuration for all languages by the screeengrab-Espresso tests. The only purpose of these tests is to navigate the app to the desired configuration and taking the screenshot. Therefore, these tests are kept plain and simple since no functionality must be tested. They are created during feature development and due to their simplicity they are cheap. Using the package structure, the tests can be combined to certain test collections. The following code snippet shows a typical Espresso test to create a screenshot of the script-area in Pocket Code. In line 5 and 7 bricks are added to the script-area and in line 14 the screenshot is taken:

```
1        package
org.catrobat.catroid.uiespresso.fastlaneScreenShots;
2        ...
3        @Test
4        public void scriptsScreens() {
5         addBrick(WaitBrick.class,
6       R.string.category_control,false);
7        addBrick(VibrationBrick.class,
8        R.string.category_motion, true);
9        try {
```

[25] https://catrobat.slack.com

[26] https://developer.android.com/training/testing/espresso/index.html

```
10                  Thread.sleep(100);
11          } catch (InterruptedException e) {
12          e.printStackTrace();
13          }
14           Screengrab.screenshot("scriptScreen");
15          }
```

The following script is executed by Jenkins which triggers the execution of above described espresso tests. The tests are executed on an emulator, therefore, the screenshots are to be downloaded from the instance and stored to the defined locale folders (lines 7-11) to be further used as app store metadata.

```
1           fastlane screengrab
2           --app_package_name 'org.catrobat.catroid'
3           --use_tests_in_packages
4           'org.catrobat.catroid.uiespresso.fastlaneScreenShots'
5           --app_apk_path  'debugapk'
6           --tests_apk_path 'debug-androidTestapk'
7           --locales '"en-US' ,'ar','zh-TW','da-DK','nl-NL',
8           'en-GB','fr-FR','fr-CA','de-DE','hi-IN','hu-HU',
9           'id','it-IT','ja-JP','ko-KR','fa','pl-PL','pt-BR'
10                  ,'pt-PT','ru- RU','es-419','es-ES','es-US','sv-SE'
11                  ,'th','tr-TR'"
12                  --clear_previous_screenshots true
```

3.4. Metadata creation using Screengrab and Crowdin

Google Play expects a certain naming format of the language folders for the app metadata. Unfortunately, this naming format is not consistent. Furthermore, unsupported languages such as Urdu and Sindhi must be removed in a later step since they must not be included in the app's metadata. Fastlane is used to deploy an app and its metadata to Google Play. Therefore, the Fastlane deployment pipeline needs in each language folder the app's title, a full description, a short description, and an app update information as separate text files (*title.txt*, *short description.txt*, *full description.txt*, and *whatsnew.txt*). Pocket Code's deployment pipeline, which in a later step uses Fastlane to publish the app to Google Play, first downloads all metadata translations as a zip file from Crowdin. This is done using the Crowdin console client[27]. Within that archive all language folders with the following (consistent) naming scheme are contained: *languagecode-CountryCode* (e.g., *en-US*, *ar-SA*, *sd-PK*). Unfortunately, Crowdin's naming scheme is incompatible with the Google Play naming scheme for languages. To make the Crowdin output compatible with Fastlane the *google play.xml* file's content is split up into three individual text files which are mandatory for Fastlane, i.e., *title.txt*, *short description.txt*, and *full description.txt*. As a next step the previously created three app description files for every language are merged with the screenshot directory structure created by Fastlane's Screengrab. The third step makes the directory structure naming scheme Google Play compatible by renaming the folders according to the Google Play naming convention, e.g., *ar-SA* to *ar*, *bg-BG* to *bg*. The directories of the translations for languages unsupported by Google Play, such as *sd-PK* and *ur-PK* are deleted in this step. At the time of writing, Pocket Code has translations of the app description for 57 languages whereas Google Play does not support ten languages of those Pocket Code does. Therefore, ten translations are removed prior to the upload and Google Play displays the app descriptions and screenshots for only 47 languages. This is subject to change since Google frequently increases the number of supported languages.

3.5. Release branch creation

When the release-APK has been signed and aligned, the metadata (app description) has been combined with the screenshots for all supported languages. Everything is in place for the actual deployment, therefore, a release branch is created (line 1) and all artifacts, i.e., code, APK, and the app's metadata in different languages are committed and pushed to this branch (line 3).

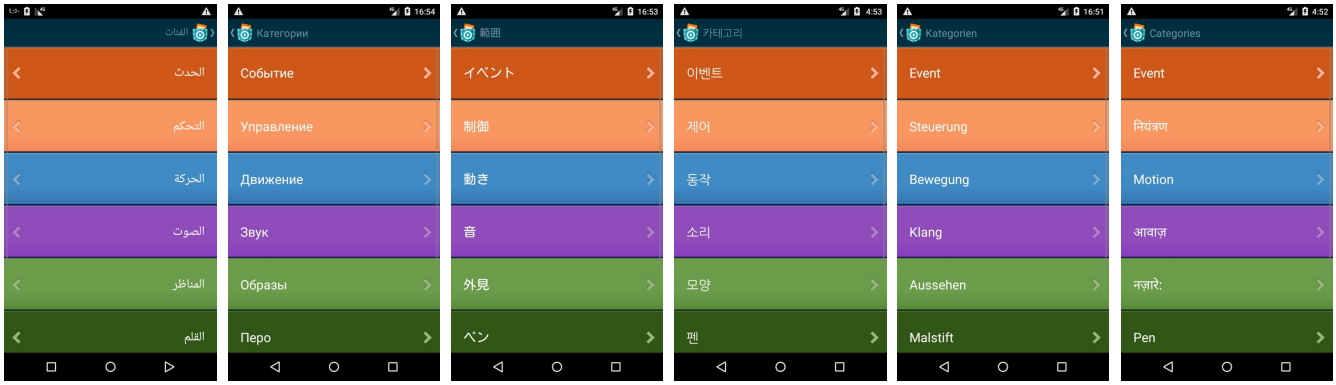
```
1           git checkout -b $branch_name
2           ...
3           git push origin $branch_name
4           ...
```

3.6. Deployment to the alpha track

The release branch now contains the release candidate and the app's metadata. The APK can now be uploaded to the project's alpha track on Google Play. Unfortunately, the alpha track has no separate metadata. That means uploaded metadata would be public which must not happen at this point. Therefore, at this stage of the Jenkins job only the APK is uploaded to the alpha track without any metadata and images (line 5, 6 of the following Fastlane supply tool script).

[27] https://support.crowdin.com/cli-tool/

Metadata will be uploaded at a later stage when the APK is



a.) Arabic (ar) b.) Russian (ru-RU) c.) Japanese (ja-JP) d.) Korean (ko-KR) German (de) e.) Hindi (hi-IN)

Figure 5: Automatically created screenshots for different languages (some are only partially translated).

promoted from the alpha track to the production track by the product owners.

```
1       fastlane supply
2       --apk $release-signed-alignedapk
3       --track alpha
4       --json_key GOOGLEPLAYSTOREKEY.json
5       --skip_upload_metadata true
6       --skip_upload_images true
```

3.7. Post alpha track deployment actions

The Catrobat project has various Slack channels for maintaining timely communication. This is especially true for everything related to a release. There exists a dedicated Slackchannel *ci-status* where the Jenkins server notifies about success or failure of a job. Every deployment build job status is also posted to this channel. If every stage of the pipeline including the deployment to the alpha track was successful there are only two more stages until public accessibility of the new app version.

3.7.1.Final app approval

The final app approval by the product owners (POs) is completely unpredictable in terms of time and therefore, was moved after the alpha track deployment. Although possible and surly beneficial striving towards automatic acceptance tests with Cucumber [34, 35], it is not possible with reasonable effort to fully automate app approval in our case. Since the Google Developer API supports a staged deployment approach the product owner app approval tests (i.e., acceptance and exploratory tests) can be relocated after the alpha track deployment. The alpha track in the Catrobat project is only subscribed by the product owners and so at this stage of the deployment the APK is only available for the POs for testing. 3.8. Promote to production trackPromotion from the alpha to the production track only happens when the product owner approval was finished positively, i.e., the acceptance and exploratory tests were successful. Once this is the case there is a Jenkins job triggered which is following two main goals 1.) to upload the metadata (the app description and screenshots for all languages) to Google Play, and 2.) to promote the alpha track APK to the production track since only then the new version of the app is publicly available for download.

4. TIME CONSIDERATIONS

4.1. General reflection

The manual deployment process as it is depicted in Figure 2 is afflicted with four main disadvantages: First, due to the fact that there is human interaction needed during the whole workflow additional operating delays are generated. This task switching, adds up and leads to an overall process delay. Second, the manual creation of screenshots not only is slow but very prone to errors due to its repetitive and tedious nature. Third, manual setting of the language (either through changing the system language or the language of the app itself) is another disadvantage, especially if the language is not understood by the person who operates the device. Fourth, the final approval of the app through the product owners is the most unpredictable manual element in the deployment process and must be postponed after deployment to the alpha track and after the creation of the metadata. This is perfectly possible through the staged release approach supported by Google Play. Disadvantage two and four have the most impact on the delay of the deployment whereas two (along one and three) can be mitigated through automation and four rather through stage relocation within the pipeline. As a benchmark, a simple manual deployment to Google Play without running the full test-suite (lint, PMD, unit and UI) and without the update of the metadata, i.e., the app description with the screenshots for all languages, takes about 25-35 minutes according to the release responsible person. The test-suite in its size at the time of writing adds about 15-20 minutes to the deployment time. In reality when

the manual workflow is followed the app's metadata is updated only when the gap between the published app description including the screenshots and the UI of the app becomes apparent. This is the case only because the process to update the metadata involves taking the screenshots for all languages manually which is tedious and error-prone and hence is avoided whenever possible. This can lead to a confusing situation for the users since the app and its description on Google Play deviate too much when the metadata update is skipped too often. With the manual workflow the overall time needed for taking screenshots is determined by, i.) the number of languages the screenshots have to be captured, and ii.) the complexity of the navigation to reach the app's desired configuration. As an example for the complexity of the matter, where not necessarily navigation but consistency is the challenge, consider the case, somebody wants to create screenshots of the Pocket Code IDE to document programming. Using Pocket Code scripts for these screenshots which make use of variables, for consistency's sake the variables used in these scripts should also be translated, since a Thai locale using German variable names does not look appealing, although it is of course technically valid. So usually it is the easiest way to be aware about this challenge and avoid it by not using variables in Pocket Code scripts for screenshots. In any case manual navigation tofeatures which are hidden in the depths of the Pocket Code GUI defer the screenshot creation significantly and should be avoided whereas this does not affect the process negatively when screenshots are created via automated tests.

4.2. Time measuring experiment

The magnitude of time needed for taking screenshots was determined empirically using the following simple setting: eight different languages including Japanese, Chinese, and Arabic; six screenshots, i.e., six simple app configuration which could be navigated to within a second. It must be mentioned that meanwhile Pocket Code supports language changing on application level which is helpful for the manual scenario since the person taking screenshots can stay within the app and does not need to switch to the Android system settings to change the language. In this setting it was possible to take six screenshots per language within three minutes in average, independent of the language. That shows that the configuration was simple enough since during the experiment (48 screenshots in total) only one error was introduced. The effect of this single error for that language was, that due to the activities to fix the app's configuration and retake the screen shot, tripled the amount of time needed for taking the screenshots from an average of three minutes to a little more than nine minutes. We think it is safe to assume that the number of errors positively correlates with the configuration complexity. Furthermore, the language itself also comes into account but depends on the cultural background of the person conducting the experiment, since a typical educated western person does not have language skills for, e.g., Chinese, Japanese, Thai or Arabic. Having to take screenshots in such a language poses the problem that the person who operates the device must be able to navigate the app in this language to the desired configuration. If that person is not aware of the language, e.g., Arabic, only very simple configurations can be reached with a reasonable amount of time, otherwise it is likely that the operator gets completely lost in the UI and has to switch again the language to fix the navigation, switch back again to Arabic to finish the screenshot.

4.3. Time extrapolation

At the time of writing, Pocket Code's app description on Google Play is available in 26 different languages. When we assume, that no errors are made during screen capturing, and the setup is simple enough to have the same speed changing the app's configuration for all languages, the minimum possible time to take all screenshots is about 78 minutes (three min in average by 26 languages). On filesystem level screenshots have to be put into the corresponding folders together with the translations of the app description to form the app's metadata in a compatible format for Google Play. This means as an example, Arabic screenshots have to be copied to the Arabic language folder, Chinese into the Chinese language folder, etc. This is not as straightforward as it might seem when done manually since it is often not obvious for a person who is not capable to understand the 26 languages and the differences between them, such as, Arabic and Farsi, or Russian and Serbian using the Cyrillic script. In our experiment, we didn't explicitly target this challenge and chose languages that can be easily differentiated by looking at a screenshot. We can therefore only assume that this will be very likely a reason of significant time loss when having to deal with similar languages using the same characters (glyphs/symbols) which are hard to distinguish for a layperson.

4.4. Comparison manual vs. automated

As described before, the manual approach takes at least 78 minutes. The automated approach, for all 26 languages with six screenshots each, including the navigation of the app to the desired configuration, downloading, and sorting the screenshots to the correct folders, in contrast takes less than 10 minutes (545 seconds). For this comparison the product owner approval of the app can be neglected since there are no automatizations of these tests done yet, but, of course, this will be a topic for the future. The overall app approval by the product owners, in the case of the Catrobat project, takes up to two days. This means when the manual workflow is executed this leads to a delay of up to two days until the app reaches the app store. Since this approval step happens before the metadata is compiled the deployment is delayed even more. In the staged approach, PO-approval is done after deployment to the alpha track. If the approval was

successful the app can be promoted within seconds to the production track and the metadata deployed within minutes. For the duration comparison between automated and manual workflow the following activities were considered: the mandatory automatic test runs, the screenshot creation, the metadata compilation, and the upload to Google Play. The manual deployment adds up to 2hr 8 min with 15 min for the automated test runs, 35 min for manual intervention, and 78 min for the screenshot creation (where the sorting was neglected). The automated deployment adds up to less than 25 min with 14 min for the automated test runs, 10 min for the screenshot creation (including the sorting to the language folders), and less than 1 min for uploading to Google Play. This means a total saving of at least 1 hour and 43 minutes. Pocket Code was manually released about 42 times to Google Play from the first release in 2013 up until 2018. The accumulated delta between manual vs. automatic deployment is at least 72 hours. At the first glance that is not the most impressive figure but the true benefits of this approach are the gained flexibility and ability to deploy without too much of a lead time, the improved accuracy and the saved human interaction. Furthermore, this approach enables the project to implement a frequent release policy with automatic metadata creation without stressing out people with boring repetitive tasks.

5. CONCLUSION

In this paper, we presented the deployment pipeline of Pocket
Code. Using Fastlane, Crowdin and Jenkins build server within a staged deployment approach significantly reduces deployment time and enables automatic creation of the app's metadata for Google Play. The deployment of a new feature or a bug fix can be triggered as soon as it is ready for release. Automation in deployment decreases release time, increases accuracy of, and confidence in the process. Furthermore, by limiting repetitive and error prone tasks the deployment activity becomes more engaging and hence raises the satisfaction and self-esteem of the release responsible persons. In the best case of the Catrobat project's deployment, i.e., the product owners' acceptance tests were positive, only two mouse clicks are required for a full deployment and update of the app's metadata in all available languages on Google Play. The first click creates all metadata, i.e., the screenshots and app description for all languages, commits to the release branch and uploads the APK to the Google Play alpha track. Only the users who are registered for the alpha track are informed about the update and can download the app. In our case only the product owners are registered who will then start with acceptance testing. The second click uploads the latest app's metadata to Google Play and promotes the app to production for that it becomes publicly available. An interesting part of our deployment pipeline, and where most of the time is saved is the automatic capturing of app screenshots and the handling of compatibility issues which usually arise when different tools are used in conjunction. The origin of the time gain is in our case the automation of the metadata creation process, i.e., the compilation of the app description translations and the app's screenshots for all languages, for that the app can be fully presented in all supported languages on Google Play. The Pocket Code deployment pipeline approach can therefore, be of interest for all app providers who publish multi-language apps and strive to maintain an internationalized and localized Google Play presence. This approach supports a software product team to keep its deployment efforts low and focus on development. By automating repetitive deployment tasks, this approach helps to properly present multi-language apps, reach a broader audience, and potentially increase market share.

ACKNOWLEDGEMENT

This is extended version of conference paper [27] presented in *IEEE International Conference*

REFERENCES

1. Slany, W. *Pocket code: a scratch-like integrated development environment for your phone* in *Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity* (2014), 35–36.
2. Girsang, A. S. Analyzing Android Users Based on Google Play Store Using K-Prototype Algorithm. *International Journal of Emerging Trends in Engineering Research* 8. ISSN: 23473983 (2020).
3. Shahin, M., Babar, M. A. & Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5, 3909–3943 (2017).
4. Fitzgerald, B. &Stol, K.-J. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software* 123, 176–189 (2017).
5. Leppanen, M.¨ *et al.* The highways and country roads to continuous deployment. *IEEE Software* 32, 64–72 (2015).
6. Geiss, M. Continuous Integration and Testing for Android. *Berlin Institute of Technology (TU-Berlin)* (2012).
7. Fowler, M. *Continuous Integration* accessed: 2017-12-24. https://martinfowler.com/articles/continuousIntegration.htm.
8. Claps, G. G., Svensson, R. B. & Aurum, A. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology* 57, 21–31 (2015).
9. Meyer, M. Continuous integration and its tools. *IEEE software* 31, 14–16 (2014).
10. De Cesare, S., Lycett, M., Macredie, R. D., Patel, C. &

Paul, R. Examining Perceptions of Agility in Software Development Practice. *Commun. ACM* 53, 126–130. ISSN: 0001-0782. http://doi.acm.org/10.1145/1743546.1743580(June 2010).

11. Davis, J. & Daniels, K. *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale* ("O'Reilly Media, Inc.", 2016).

12. Humble, J. & Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Pearson Education, 2010).

13. Chen, L. Continuous delivery: Huge benefits, but challenges too. *IEEE Software* 32, 50–54 (2015).

14. Savor, T. *et al. Continuous deployment at Facebook and OANDA* in *Proceedings of the 38th International Conference on Software Engineering Companion* (2016), 21–30.

15. Fowler, M. *Deployment Pipeline* accessed: 2017-12-15. https : / / martinfowler . com / bliki/DeploymentPipeline.html.

16. Nayebi, M., Adams, B. &Ruhe, G. *Release Practices for Mobile Apps–What do Users and Developers Think?* in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on* 1 (2016), 552–562.

17. Michlmayr, M., Fitzgerald, B. &Stol, K.-J. Why and how should open source projects adopt time-based releases? *IEEE Software* 32, 55–63 (2015).

18. Cesar Brandao Gomes da Silva, A., de Figueiredo˜ Carneiro, G., Brito e Abreu, F. & Pessoa Monteiro, M. Frequent Releases in Open Source Software: A Systematic Review. *Information.* ISSN: 2078-2489. http://www.mdpi.com/2078-2489/8/3/109(2017).

19. Feitelson, D. G., Frachtenberg, E. & Beck, K. L. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 8–17. ISSN: 1089-7801 (2013).

20. Google. *Share of App Users Who Have Stopped Using An App Because It Was Not Localized Properly as of March 2014.* accessed: 2017-12-05. https://www.statista.com/statistics/296304/mobileapp-abandoment- rate- due- to- lackinglocalization/.

21. Hamunen, J. *Challenges in adopting a Devops approach to software development and operations* en. G2 Pro gradu, diplomityo (2016), 69.¨ http://urn.fi/URN:NBN:fi:aalto-201609083476.

22. Dyck, A., Penners, R. &Lichter, H. *Towards definitions for release engineering and devops*in *Proceedings of the Third International Workshop on Release Engineering* (2015), 3–3.

23. Ishida, R. & Miller, S. K. *Localization vs. Internationalization* accessed: 2018-03-25. https://www.w3.org/ International/questions/qa-i18n.

24. Awwad, A. M. A., Schindler, C., Luhana, K. K., Ali, Z. &Spieler, B. *Improving Pocket Paint usability via Material Design compliance and internationalization & localization support on application level* in *Proceedings of the 19th International Conference on HumanComputer Interaction with Mobile Devices and Services* (2017), 99.

25. Usability Professionals Association. *Usability Body of Knowledge - Internationalization* accessed: 201803-21. https : / / www . usabilitybok . org/internationalization.

26. Reason, J. Human error: models and management. *BMJ: British Medical Journal* 320, 768 (2000).

27. Luhana, K. K., Schindler, C. & Slany, W. *Streamlining mobile app deployment with Jenkins and Fastlane in the case of Catrobat's pocket code* in *Innovative Research and Development (ICIRD), 2018 IEEE International Conference on* (2018), 1–6.

28. Erich, F., Amrit, C. &Daneva, M. Report: Devops literature review. *University of Twente, Tech. Rep* (2014).

29. Versluis, G. in *Xamarin Continuous Integration and Delivery* 1–5 (Springer, 2017).

30. Fallis, A. *Effective DevOps* 9, 1689–1699. ISBN: 9788578110796. arXiv: arXiv:1011.1669v3(2013).

31. Colomo-Palacios, R., Fernandes, E., Soto-Acosta, P. &Larrucea, X. A case analysis of enabling continuous software deployment through knowledge management. *International Journal of Information Management* 40, 186 –189. ISSN: 0268-4012. http://www.sciencedirect.com/science/article/pii/S0268401217308782(2018).

32. Developer, G. *zipalign*accessed: 2017-12-16. https://developer.android.com/studio/commandline/zipalign.html.

33. Developer, G. *Sign Your App* accessed: 2017-12-17. https://developer.android.com/studio/ publish/app-signing.html.

34. Kamalrudin, M., Sidek, S., Aiza, M. N. & Robinson, M. Automated acceptance testing tools evaluation in Agile software development. *Sci. Int,* 1053–1058 (2013).

35. Wynne, M., Hellesoy, A. & Tooke, S. *The cucumber book: behaviour-driven development for testers and developers* (Pragmatic Bookshelf, 2017).