ECSE 321 - Intro to Software Engineering
Release Pipeline Plan - Deliverable 4

Harley Wiltzer
Camilo Garcia La Rotta
Jake Shnaidman
Robert Attard
Matthew Lesko

April 2, 2017

Contents

Part I

Integration

1 Description

The integration pipeline is responsible for maintaining the integrity of the source code throughout the development phase. Without proper integration, mistakes from one software developer can propagate into other people's work. Like an infectious disease, bugs can propagate. Infected code (Code that has bugs) that is shared can easily go unnoticed and affect larger systems where it can become even harder to notice. The goal of the integration pipeline is to reduce the amount of mistakes that will be integrated into the project and to keep track of all changes. The tools used for this purpose are discussed in the Tools section.

2 Tools Used During Integration

This section describes the tools that were used in the integration pipeline. The tools used for the integration pipeline include but are not limited to: Git, Github, TravisCI, JUnit, PHPUnit, and Slack.

2.1 Git/Github

Git is the name of the version control system that is used to track the changes of our source code. Github is the website that is used to host the source code.

The Git tool allows us to see how changes are made to the source code by our team members. Very importantly, it keeps our software developers accountable for the code that they write by keeping history of their contributions. Indeed, you can pinpoint the exact moment where software developers have made terrible mistakes. In these cases where the code has been rendered properly damaged, git allows you to revert to earlier stages in development before the mistake was made. Git also allows our members to work on code simultaneously which improves our productivity!

Github allows our team to not only seamlessly share our code, but provides shelter for our code. Github, as a reliable and very commonly used website, serves to host our repositories and keep it safe from accidental deletion.

Git was chosen because of its almost universal use for project source-code version control, no real alternative was considered because of how ubiquitous and industry-standard Git is.

While there are other websites used for hosting repositories, GitHub is well known and respected as a good platform. Added to this, the repository is provided / is monitored by those in charge of ECSE 321 and we were not given a choice. Had this not been the case, we still would have used GitHub because of it's reliability and many integrations with software like Slack and TravisCI.

2.2 TravisCI

TravisCI is a continuous integration service that reacts to each new version of source code pushed onto Github. When the software detects that a new version of the source code has been added to github, it automatically tries to build the project and run tests. This is useful as testing automation ensures that no code that is pushed onto the repository goes untested, and can immediately

bring problems to light if implemented correctly.

TravisCI was chosen because of the utility it provides in automated testing. Even if one of the team members forgets to test code before committing and pushing it, this software ensures that it is tested automatically. For the security it provides us in terms of not having to worry about whether the code was tested or not, it is worth implementing and was thus chosen.

2.3 JUnit/PHPUnit

JUnit and PHPUnit testing frameworks were used throughout development to throughly test our code. By ensuring that code is tested properly while it is being written and before being shared, it serves to minimize the amount of mistakes that are committed to our group's source code. These adheres to the Test-Driven Development paradigm that was chosen to be followed for the purpose of this application.

As one does in test-driven development, tests were written before the code was implemented. This provided the team with clear, good insight concerning which functionalities are still required to be implemented or fixed.

JUnit and PHPUnit were chosen as the desired testing methods because we were shown in class how useful they were for full-scale testing and flexibility compared to just running tests within a main method.

2.4 EclEmma

EclEmma is an eclipse plug-in and was used in conjunction with JUnit to determine code coverage for the tests that were written so that the team could implement more tests so the maximum amount of coverage could be achieved.

EclEmma was chosen as our designated test coverage analysis tool because it was simple enough to implement quickly and because the results can be easily read. It was also introduced to us in class and so as we were already familiar with it, so it made sense that EclEmma was the coverage tool that we would implement.

2.5 Slack

Slack is a powerful communication tool that allows the team to be able to keep the communication seperate for each different task, and boasts many 3rd party integrations. Slack was not only used to aid communication across our team, but was integrated with Github to notify all of our team members of commits made. This made it easy to keep track of the progress made, which was important when certain elements within the pipelines had dependencies.

Slack was chosen over something like Facebook Messenger because of the utility and focus that it provides to group conversations. Being able to seperate each kind of task to a seperate conversation definitely made organisation and teamwork simpler and more reliable. The added funcitonality of plug-ins for slack also make a variety of tasks simpler and less time-consuming.

3 Tools Considered But Not Used

3.1 Trello

Team management software like Trello (and it's competitors) allow users to create lists and have cards to show a variety of different things relating to group workflow. Such software sees a lot of use in industry for large-scale projects requiring multiple teams. However because of the scale and the way the project was structured, it was determined that such a software was overkill and not entirely necessary. Though the learning-curve to be able use the software for those unfamiliar with it is relatively small,it was not worth the risk introducing a new variable into the management of the project and potentially causing extra confusion and taking time off other tasks.

3.2 Subversion

Subversion (SVN) is another commonly-used version control system and a rival to Git. Although it is far less complex than Git and much easier to learn, SVN faces some serious problems that would inhibit the productivity of the team, which lead to its dismissal from this project. Most relevantly, SVN only allows modification of code when connected to the repository, and local repositories do not exist in SVN. This means that if a team member has no internet connection, it would be impossible for him to work on the code. This is a severe inconvenience, as there are many situations in which code development has been done without internet connection, which has sped-up the development process considerably. On the other hand, Git features local repositories, so the user can edit code and commit it to his local repository and finally push the code to the remote when internet connection has been established.

4 Design

This section specifies our team's design choices.

The following sections contain lists that are enumerated to link to the respective subsections to the Rationale section of the document

4.1 Naming Conventions

Our team came to decisions on certain naming conventions to keep our code consistent across platforms. This choice also makes it easier to read code written by other team members since the style remained consistent. This way the code was easier to interpret because it formalized the expectations of methods and variables.

1. By convention, all variables are listed at the top of a class if it is used throughout the class
2. It was decided to keep method names as descriptive as possible
3. It was decided to keep the code split into three modules: Model, View and Controller in all platforms
4. It was decided to have method signatures followed by an open bracket on the same line

5. Manager and Controller variables were named with two letters for brevity

6. Although somewhat standard, we emphasized that all variables and methods that do not need to be accessed outside of the class it's declared in must be declared private

4.2 The Integration Process

It is important in the integration phase to ensure that all team members have a consistent understanding of the state of the code in the repository. Tools like Ant and TravisCI were used extensively to achieve this. It was decided as a team that code should only be pushed to the repository once it has been compiled, run, and has passed its corresponding tests. An exception to this rule is that if a developer requires assistance on a piece of code, he may push it (assuming it compiles) so long as he leaves a detailed report of its failures in the Git repository and in the commit message. It is also expected that the developer gives a summary of what needs to be done in Slack. Also, of important note, it is expected that the developer *must* push his code after it has been successfully compiled and it has passed all tests.
Furthermore, adhering to the rules of Test-Driven Development, the team members are expected to always write unit tests for new features before implementing the features themselves in the code.

4.3 Differences Between Platforms

There aren't any differences between platforms generally. They use the same naming conventions. The only difference is that the mobile application uses PHPUnit whereas the desktop/mobile uses JUnit. They're designed to be similar, so there isn't much of a difference there.

5 Rationale

This section specifies the rationale for our team's design choices. The enumerated lists correspond to the Design section

5.1 Naming Conventions

1. We decided to list all the variables at the top of classes and not create a separate class for variables because we don't have many variables that are used across classes.

2. Keeping method names as descriptive as possible allowed for code to be easily understood, making it far more productive for other team members to use and integrate into the system

3. Splitting code into three modules made it easier to know where to find classes within the projects

4. Having method signatures followed by a bracket rather than have the opening bracket on the following line kept the code consistent which made it easier to track whether or not code has syntax errors

5. Since Managers and Controllers were a large part of how our application works and everyone knew what they were, it was convenient to keep the names short. eg: pm - profile manager, ac - application controller

6. We decided to enforce the idea that methods and variables should default to private to keep people from using code that they didn't write from another class where it might break something. There are people in our group who have only taken comp 202, so we wanted to be explicit here.

5.2 The Integration Process

The integration process was designed this way by the team to reduce confusion between team members. This process of integration guarantees a certain quality of code to the developers, so that when the pull from the Git repository they understand which code can be used reliably, and which code still needs work. The use of TravisCI will help to enforce the rule that only code that has passed its corresponding tests should be pushed, because TravisCI will automatically run the tests. Furthermore, with a convenient build system described in the next part of this report, testing code throughout the development phase will not be a hassle. As described later, this build system requires code to be tested before any executable is created as well, which will further convince the developers to test the code.

Finally, it is also important for the developers to be constantly pushing their working and tested code. Not only does pushing the code to the Git repository create a reliable backup, but it also keeps the team up to date with the progress that has been made. If certain code is a dependency of another feature, it is obviously important that this code is submitted as soon as possible, so another developer make work on another feature promptly.

Part II

Build

6 Description

The build pipeline is responsible for the successful and efficient building of the source code and automated testing. As the system evolves, the processes required to carry out certain tasks like compiling the code, creating executables, and running tests will evolve as well, and will ultimately become more complex as time fades away. For this reason, it is important to design a robust automated build system that provides the developers with the power to do any of the aforementioned tasks by running simple build commands.

Of course, such power does not come from nothing; in fact, the developers must actively maintain this build system such that it is always kept up to date with the rest of the system. Fortunately, certain tools exist for creating such build systems that will significantly aid in the process. These tools will be discussed in the Tools section.

7 Tools

Although designing a Monad in Haskell for automating build procedures sounds very interesting, such a task would be a rather horrendous decision. Given the plethora of tools available that can already be used to aid in the development of build systems, it is much preferred to use tools that have already been tested and have already been shown to be efficient rather than developing one anew.

Firstly, designing and implementing build scripts, the Apache Ant tool will be used. A major convenience of this tool is that it has support for both Java and PHP languages, which cover 100% of the languages used to implement the TAMAS system. Ant provides the ability to design multiple *build targets*, which are basically sets of building instructions that can be chosen and sequenced when the build is invoked. If implemented properly, this would allow the code from multiple platforms to be compiled individually or all at once, it would allow tests to be run automatically (and perhaps different sets of tests depending on the user's choice), and would allow executable files to be generated at will. Ultimately, Ant would provide the ability to create a unified build suite for compilation, testing, packaging and friends, maximizing the convenience and effectiveness of the developers in the build phase.

Furthermore, the Travis CI tool will be used for automated testing with Git integration. When synced with Github, Travis CI can automatically run tests and communicate test results automatically whenever a new commit is pushed. The Travis functionality is itself governed by a build file, which will take advantage of the Ant build file that will be in the Git repository. Travis CI will be incredible for ensuring that the entire development team is always up to date and informed concerning the latest effectiveness of the code.

8 Design

This section will discuss the various design decisions made concerning the design of the build system, and will provide a high level view of the build system implementation.

8.1 Concerning the unification of the application platforms

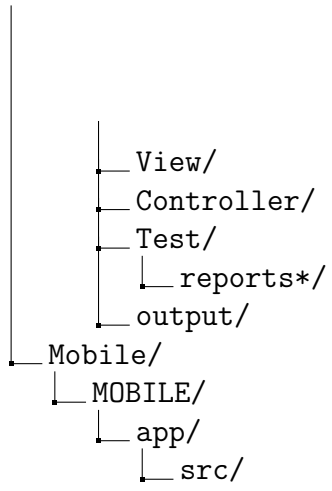
The first design decision made after deciding to use Apache Ant was to make one master build system for all platforms of the application. Although all three application platforms serve different purposes, and to the end user all platforms are literally separate, it would behoove the development team to be able to manage the build procedures for all three with the same build script. This way, should they find it appropriate, the developers may test code for more than one platform at once, for example. It is projected that this will make the build process much more efficient. Of course, the build system will be robust enough to give the user control over which platforms to target when the build is invoked. Of course, this design choice was made possible thanks to only the magic offered by Apache's Ant tool, which supports the testing protocols and compilers used for all platforms in this system.

The Build Targets section will be split up by platform and will describe the build targets for each. Note that although platform discrimination is implicit in this structure, all targets will be included in the same build file.

8.2 The calm before the storm: an overview of the intended file structure

This section will give an overview of the file structure that is inherent to the build system's design. Note that the directories and files suffixed with an asterisk denote directories and files that are generated by the build system.

```
APP/
├── build*/
│   ├── jar*/
│   │   └── <Executable Desktop JAR archive>*
│   ├── apk*/
│   │   └── <Android APK>*
│   ├── web*/
│   │   └── <Web Application zip archive>*
│   ├── desktop-classes*/
│   │   └── <Desktop Class Files>*
│   └── mobile-classes*/
│       └── <Mobile Class Files>*
├── Desktop/
│   └── TAMAS/
│       ├── src/
│       ├── test/
│       │   └── reports*/
│       └── output/
└── Web/
    └── ca.mcgill.ecse321.group10.TAMAS/
        ├── index.php
        └── Model/
```



8.3 Build targets by platform

The following section will provide a catalog of build targets structured by the target platform. Feel free to refer to directory map above when reading the paths mentioned below.

Desktop

The build targets intended for the desktop system will be enumerated below.

1. `init-desktop`
   Creates the requisite directories that will be needed for the remainder of the Desktop build targets

2. `build-desktop`
   Sources the source files from the `APP/Desktop/TAMAS/src` directory and compiles them. Puts the compiled class files in `APP/build/desktop-classes`.
   Depends on `init-desktop`.

3. `test-desktop`
   Runs all JUnit tests from the `APP/Desktop/TAMAS/test` directory and generates test reports to go to `APP/Desktop/TAMAS/test/reports`.
   Depends on `init-desktop`.

4. `export-desktop`
   Makes executable JAR archive with the class files in `APP/build/desktop-classes` directory, and places JAR in `APP/build/jar`.
   Depends on `test-desktop`.

5. `clean-desktop`
   Recursively removes all directories and files suffixed with an asterisk under the `APP/DESKTOP` directory and `build/desktop-classes` directory.

6. `run-desktop`
   Launches the desktop application, executing the executable JAR file in `APP/build/jar`.
   Depends on `export-desktop`.

Web

The following section will outline the build targets for the web application. Note that as PHP is an *interpreted* language, the code is not compiled, thus no compilation steps are included.

1. `init-web`
   Creates the `APP/build/web` and `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test/reports` directories.

2. `test-web`
   Runs the PHPUnit tests found in the `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test` directory, sends test reports to the `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test/reports` directory.
   Depends on `init-web`.

3. `export-web`
   Makes a zip archive of the PHP files and stores it in `APP/build/web`.
   Depends on `test-web`

4. `run-web`
   This target will be included if it is found to be convenient. It will call a script that starts the XAMPP server and opens the `index.php` file in a browser.

Mobile

Finally, the build system design targets related to the mobile platform will be shown. Note that some of these targets depend on Desktop targets: this is because the Mobile application is heavily dependent on the logic and model of the Desktop application.

1. `init-mobile`
   Creates the directories requisite for the remainder of the mobile build targets.

2. `build-mobile`
   Compiles the source code from the `APP/Mobile/MOBILE/app/src` directory, places compiled class files in `APP/build/mobile-classes`.
   Depends on `init-mobile,export-desktop`.

3. `export-mobile`
   Builds the APK file and places it in `APP/build/apk`.
   Depends on `build-mobile`.

4. `clean-mobile`
   Removes all created directories and files (directories and files suffixed by an asterisk in the directory map) from within the `APP/mobile-classes` directory.

5. `install-mobile`
   Calls a script to install the apk file onto the connected mobile device, or to run the apk on a virtual device if no physical device is connected.
   Depends on `export-mobile`.

Convenience Targets

The following targets were designed merely for the convenience of the developers.

1. `clean-all`
   Calls `clean-desktop,clean-mobile`.

2. `build-all`
   Calls `build-desktop,build-mobile`.

3. `export-all`
   Calls `export-desktop, export-web, export-mobile`.

4. `test-all`
   Calls `build-all, test-desktop, test-web`.

5. `start-fresh`
   Removes all XML persistence files and calls `clean-all, build-all, test-all, export-all`.

6. `pandemonium`
   Calls `clean-all, run-desktop, run-web, install-mobile`.

9 Rationale

This section will attempt to explain the design choices that have been made. Though this be madness, yet there is method in't.

The build system was designed to encapsulate each of the repetitive processes that may be automated to save the developers trouble and keep the build process organized. As described above, build targets for all three platforms are included in the same build script, so as to simplify the automated build process for the developer.

The most intense platform to build is Desktop, and that comes as no surprise - the Desktop application is used as a library for the Mobile application, and it also must be compiled unlike the PHP application. Included in the Desktop build suite are an initialization routine, a compilation routine, a testing routine, an exporting routine, a clean-up routine, and of course an execution routine. This dismembers the build process into many parts, making the exact build procedure very flexible. For example, the developer may compile code with `ant build-desktop`, or make a *clean build* by running `ant clean-desktop build-desktop`. A nice build process with automated testing may be achieved with `ant build-desktop test-desktop`.

The build systems behave very similarly for the other platforms. As described previously, the Web build system does not require a "build" target because the code does not get compiled. Furthermore, the Web build system does not create an executable file in its "export" target, but rather a zip file for packaging purposes (also, a website is not executed like an application).
On the other hand, the Mobile build system does not include any testing targets, because its logic will all be tested within the Desktop system. Furthermore, instead of *running* the application, like in the Desktop's and Web's `run` targets, the Mobile system simply installs the application on a device.

Finally, it is likely that the developers may need to build all systems at once. For this reason, the Convenience Targets were conceived. Should the developer wish to do a clean build and test everything, for example, this can be done with `ant start-fresh`. If the developer wishes to compile everything, that may be done with `ant build-all`. All targets listed in the Convenience Targets section were designed *only* for convenience and are not crucial to the build system.

Finally, it is important to note how the developers should make use of this build system. As discussed in the integration process, developers are expected to frequently run tests on their code to ensure that the code that is being pushed to the repository functions according to the unit tests. Therefore, it is imperative for the developers to run the source code using this build system with `ant run-`, for example, which will only deploy code that has passed its tests. This build system provides a simple, yet robust way for the developers to keep themselves updated on the validity and reliability of their code, which will considerably enhance both the integration and deployment processes.

Part III

Deployment

10 Description

Software deployment concerns all the tasks that cover the process of making a system available for public use. This phase will be be activated once the integrated code has tested and approved by the preceding phase. At this moment the code is ready to be released to the public, In the case of the Desktop application through a web market or physical mean,for the Mobile application through the Android marketplace or an APK sharing forum and for the web application through a a production server, such as a hosting web server to ensure its availability on the web. In this section we will go through the tools, design choices and specifics of each platform's deployment phase.

11 Tools

Within the paradigm of automatizing and avoiding manual repetitive tasks that is applied throughout the release pipeline, we look now at tools which can help with the process of deployment. is saying "These tools must be able to transfer the binaries, libraries and all other required files for the proper functioning of the product within each platform in a seamless, error-less, automated way. All while monitoring the process, gathering metrics which can help troubleshoot any errors or improve future deployments.

The project will use Travis CI as the deployment tool for all platforms. The project's hosting provider shall be GitHub Releases. Travis CI can automatically upload assets from the Travis build directory to the team's git tags on the team's GitHub repository.
The main repository will contain a .travis.yml build file which will be used by Travis CI in order to build, automatically test and deploy the software system. The configuration for GitHub releases will be directly written in the .travis.yml file:

```
deploy:
provider: releases
api_key: "GITHUB OAUTH TOKEN"
file: "FILE TO UPLOAD"
skip_cleanup: true
on:
tags: true
```

It is highly recommended to use "travis setup releases" which will automatically create a GitHub oauth token with the correct scopes and encrypts it for deployment.

12 Design

The Deployment phase can be split into a set of well delimited activities which cover the entirety of the process. In this section we will describe them with relation to each available platform of the product.

- **Release:** Deploying GitHub Releases works only for tags, not for branches; hence the team will tag the current version of the software system if it is meant for deployment. The team will name tags that fit with semantic versioning: MAJOR.MINOR.PATCH. Every release

will have a title and description. If the release is unstable, the team shall select GitHub's "This is a pre-release" option to notify users that it's not ready for production.

Once the approved code is received from the Build Phase, the global repository containing the three platform specific applications will be tagged with an incremental numerical ID which fits with semantic versioning; this helps mark the end of the addition of a concrete sequence of fixes and upgrades to the existent applications.

The sub-repositories containing only each target platform application will then be delegated to the specific tool (addressed in the **Tools** section) for it to be deployed to the client. We will discuss this process for each platform:

- **Desktop:** The desktop application and its repository shall be released on GitHub Releases as an Open Source Software project; allowing users to fork the repository and make their own changes. The desktop program shall be released as a .jar file with the repository; so that users may access and open the program directly through their explorer. The release process will release a certain branch that contains the project the team wishes to release.
- **Mobile:** The mobile application and its repository shall be released on GitHub Releases as an Open Source Software project. The mobile application shall be packaged as an .apk (Android Application Package) file. Users will be able to fork the repository and make their own changes.
- **Web:** Because of the small size of our product, all the required libraries and executable can be found under the sub-repository containing the web application. It suffices to transfer this folder to the web server's repository. It can be achieved through the web deployment tools explained in the previous section. The repository shall also be released on GitHub Releases as an OSS project and may be forked by other users.

Any and all applications may be downloaded and packaged in a zip folder from the GitHub repository.

- **Install/Activate:** This activity is related only to **Desktop** and **Mobile** targets, as the **Web** application is accessed through the web and doesn't host any permanent or temporary files client-side.

  - **Desktop:** The `.jar` file can be placed anywhere the client wants. To avoid the need to handle OS specific file system architectures and scripting languages, once the executable is downloaded by the client the .jar will reside in the default download folder. From this point, it suffices for the client to run the executable through the method of his choice (Command Line or GUI) to access the application.
  - **Mobile:** Once the client downloads the APK the Android application handler will automatically decompress it and install it. The client requires no further action in order to access the now installed application.

- **Uninstall/Deactivate:** In the given case that one of our platforms stops being supported by the development team, a deactivation process starts. This process involves notifying the client of the end of support followed by the deactivation/uninstall of the application module.

  - **Desktop:** Uninstalling the desktop application involves deleting the .jar file of the main program and any output data that is stored in xml. If the user has cloned the repository on a local machine, then a full deactivation would require the deletion of the local repository and the user's forked repository on GitHub.
  - **Mobile:** Uninstalling the mobile application invloves uninstalling the application from the mobile phone and deletion of the .apk file by the use of a file manager for Android OS. If the user has cloned the repository on a local machine, then a full deactivation would require the deletion of the local repository and the user's forked repository on GitHub.
  - **Web:** The simplest target to deactivate, we only require to stop the web server daemon or the physical server itself.

- **Update:** Following the same logic of a full release, a subtag (i.e. x.1) would be given to the global repository containing the updates applications.

  - **Desktop:** The old `.jar` must be overwritten with the new compiled executable. If the user desires to keep an older version of the application, he simply needs to store the file under different name. Multiple versions can work concurrently.
  - **Mobile:** The android APK handler will receive the new APK and release a notification to all users who have the older version installed.
  - **Web:** Through a blue-green deployment procedure, one of the servers web daemon will he halted, the necessary configuration or script files updated and the service restarted. Once the application is up online again without problem the green server will be taken down to pass through the same procedure. Hence no downtime will be perceivable client-side

- **Version Tracking:** Versions of the release will be tracked by tags on GitHub. Since GitHub Releases requires the release to be tagged, each tag will contain all Patches, Minor, and/or Major updates specific to that software version. Each version and its changes can be viewed within the GitHub repository. Every release will be accompanied by release notes that specify the rationale and description of the release version. Along with the release notes are the commit number identifier, the time since the update has been released, the software version number, and download options (zip, and tar). In order to provide a broader coverage of our application for the customers, an archive website will be kept from where all past versions of the application can be stored and retrieved from. This log of past versions will be linked to the GitHub repository containing the open-source code of the application. In addition to this link, GitHub provides the functionality of publishing a compressed archive version of the current code. This useful feature will be used to maintain an accessible, visible release of the latest and past versions of the application in all its target states.

- **Monitoring:** Based on the aforementioned monitoring tools, we would keep an extensive log of the current amount of installed applications on all platforms, including their up-time, their error-logs and their amount of transactions. This would help identify and prioritize the requirements for the next iteration of development. In this section of the deployment phase we enter a more continuous analysis of the production application's state. Log analysis tools such as loggly, Splunk, logstash among others would help retrieve useful, tangible metrics on the deployed applications.

13 Rationale

Why all this process: As with any phase of the Release pipeline, the design choices for the Deployment are based on the goal of achieving an automated release where the versioning, archiving and delivery of the application produces minimal errors, if not none.

Desktop Deployment: Using Travis CI as our deployment tool and GitHub Releases as our release host simplifies our deployment process. Travis CI has GitHub integration, it is quick to learn and easy to use; making it an optimal choice for the deployment tool. It has the capability of automatically releasing versions of our software tagged for release; which simplifies the release process. Using GitHub Releases as the release hosting solution helps reduce the time it takes to provide release notes (can be done as we tag the software version). Anyone who has a GitHub account can easily fork and download the release from GitHub releases; hence, our potential user base is over 20 million (20 million GitHub users).
Mobile Deployment: Similarly to the desktop application, deployment using Travis CI as the tool and GitHub Releases as the release hosting solution helps simplify the process to deploy the mobile application. The reason why we include the .apk file is because then anyone with an Android phone or emulator may install and use the mobile application quickly. The Android OS dominates roughly fifty percent of the smartphone OS market in the USA and Canada; hence, making our user base large.
Web Deployment: Choosing approaches such as the blue-green technique for the web application can be backed by security reasons:

- Disaster Recovery in case a major bug went undetected through the Integration and Build Phases.
- Reducing, or even nullifying the downtime of a web service to the clients
- Being able to maintain multiple versions of an application on separate web-servers to ensure a broader support of client-side configurations

Monitoring: The main difference with the other two phases comes from the support tasks related with the monitoring of the delivered applications. It is a continuous process that's doesn't stop since the first public release of the application is made. This part of the Deployment is intrinsic to discover faults or new requirements that appears with the continuous use of the application. As such, the usage analysis tools for big log files comes as a natural automatisation step. The metrics gathered by these applications, as well as by the Android marketplace and GitHub itself ensure a better understanding of the usage, improvements and fall-pits of the product as sa whole.