Automated characterization of build and test failures on a continuous integration system

Memoria de Proyecto Final de Máster

Máster Universitario en Desarrollo de sitios y aplicaciones web

Realizado en colaboración con:

Autor: Gerson Esquembri Moreno

Consultor: Carlos Caballero González

Profesor: César Pablo Córcoles Briongos / Julià Minguillón Alfonso

HP Consultant: Francisco Javier Viella Fernández

8 de junio de 2020

License:

Attribution-NonCommercial-NoDerivs 3.0 Unported (CC BY-NC-ND 3.0)

Licencia:

Reconocimiento-NoComercial-SinObraDerivada 3.0 España (CC BY-NC-ND 3.0 ES)

Abstract

Continuous integration systems allow for easy identification of build and test failures in software projects. However, in projects with many developers and deliverables, where the same code is used with different configurations for many different products, it is not easy to identify what changes in the code caused the failures. This usually results on involving a team to make sure that the failures are identified as soon as possible so the broken code can get reverted, and it does not affect future rounds. Early failure detection prevents the products which are going to be released to be faulty, and helps developers work with a stable codebase that will help them identifying the result of their modifications without the noise generated by other developers' issues.

A way to solve this problem is developing a strategy that allows the build and test failures, and their root causes, to be identified automatically. The strategy must be based on the identification of possible candidates for the broken builds/tests, evaluating the changes between the last successful integration and the failing one. Furthermore, a strategy to evaluate if the offending commit (the one which caused the failure) should be automatically reverted or not could be developed, as well.

The state of this project after the development phase demonstrates the possibility of developing a system capable of automating failure detection on a complex integration system, where manually finding the responsible changes of the failures can take from a few minutes from a few hours, meaning that the figure of a full time system integrator is needed. With the *automated failure detection tool,* finding the offending commits takes just a few seconds, making it much easier for the system integrator to revert the changes that caused the failure.

Keywords: integration system, commit, build failures, test failures, automated integration, CI/CD.

Index

1. Introduction 5
   1.1. Context 6
   1.2. Objectives 7
   1.3. Approach and methodology 7
   1.4. Planning 9
2. System architecture 11
3. Development platform 14
4. Usability/UX 16
   4.1. Usability 16
   4.2. UX 24
5. UML Diagrams 28
   5.1. Use cases 28
   5.2. Data flows 30
6. User profiles 32
7. Viability 32
8. Failure detection algorithms 33
9. Future projection 35
10. Installation requirements 36
11. Installation instructions 37
    11.1. Preparing the environment 37

1. Introduction

Continuous integration is a coding manner that enables development teams to introduce small changes and commit them to the team's version control repository frequently. Modern application development is done through different platforms and tools, so the developers need a method to integrate and validate the changes. Its goal is to establish an automated and logical way to build and test the applications.

Continuous delivery provides a way to automate the applications deployment to specific environments.
Usually there are other environments than the production one, like the testing environment. Continuous delivery provides a way to push the changes to them automatically.

Continuous testing is required by continuous integration and delivery, to ensure that the applications deployed to the end users meets the quality criteria defined by the quality team. Continuous testing usually involves regression and performance tools, among other tests, that are automatically executed on the CI/CD pipeline.

However, the CI/CD pipeline has some drawbacks. The cost of the automation for the delivery can be high, both in involved people and hardware cost for the integration tests. As we will see, in our case of study the hardware cost can be high, since the tested devices are printers which cost thousands of dollars.

Also, when within this CI/CD system there are lots of different products and platforms involved, making sure that the committed code is stable, has a high cost. Since actual CI/CD systems do not have an automated way to detect the responsible commit for a failed build or broken integration test, there must be a team taking care of the pipeline.

Nowadays, in environments that require to enable developers to commit code for different products, usually there is a system (Jenkins, Atlassian Bamboo, Travis CI) that provides that functionality. But often, for very specific environments that require a non-standard functionality, a proprietary system must be developed.

1.1. Context

In our case of study, there is an integration system already implemented that allows many developers (around 200) to commit easily into the same repository, with different products being released from that code. Those products are different firmware versions of the same code, for different large format printers (Technical Graphics, 3D, etc.). That is why it's important to enable developers to commit their code into the same repository. That way the whole set of products can be easily controlled, and keeping track of their integration status, issues can be detected early, before they get into a product release or product branch.

The system automatically executes build rounds. In each of those rounds, which last around 1 hour, every product is built, and then the integration tests are run for a total of around 20 products, and 100.000 tests on each round. The rounds are executed if a certain number of commits is reached, or if having at least one commit that has not been included within a round, a certain amount of time has passed.

In those build rounds, build failures can happen even if the developer has compiled the code in his sandbox before the commit. That happens because the same code goes into different product platforms, and the build settings are different for each product because of their nature, and there may be conflicts with commits from other developers that enter the same build round.

Also, intermittent build or test failures can happen. Reasons will be addressed during the development phase of the project. Most of them happen without an easily identifiable root cause, but sometimes the failures appear due to the hardware on which the tests are run.

All these issues involve an extra cost when trying to address them, most of the time because in an integration round there are usually several commits, and the developer, or the team taking care of the integration system, have to check if the issue was originated by a single commit, if it's an intermittent issue, if it's a hardware failure (i.e., there is a paper jam on the tested printer, there's no ink, etc.).

1.2. Objectives

To develop strategies to identify automatically detect problems in continuous integration, which are:

- Identification of possible candidates evaluating the changes between the last successful integration and the failing one.
- Matching of modified files and files that caused the failure.
- Assignment of certainty degrees on the identification for the offending commits.
- Communicate the developer that his commit may be the offending one.
- Develop a strategy to identify if a commit that caused a build failure should be automatically reverted or not.
- Dangerous commit identification: number of files, number of changes, author.

In conclusion, at the end of this project is expected to reduce the workload of the integration team, since a lot of the processes involved in the identification of the integration failures will be automated.

1.3. Approach and methodology

In this project, the approach for scheduling the tasks is the well-known Agile methodology, which is a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

The agile tasks are divided in sprints, which are a period allocated for a particular phase of a project.

Dividing the tasks this way allows us to plan the development of a large project, without the constraints of a Waterfall approach, that wouldn't allow us to have flexibility when deciding if a functionality is going to be added to the project at a certain point or if it will be added later.

Other of the advantages of the Agile methodology is that task names are auto explicative. The features are the epics, and within the epics, there are user stories, which are smaller parts of the functionality.

1.4. Planning

1. **Indication of candidate commits for broken compilations**
   1.1. Investigation of UI alternatives for showing candidate commits on the Continuous integration system
   1.2. Investigation of parsing algorithms for compilation logs and commits
   1.3. Algorithm implementation?
   1.4. Indication of temporal history (first broken round, log intermittent failures, etc.)
   1.5. Detection on single commit build rounds
   1.6. Detection on multi-commit build rounds
      1.6.1. Missing files.
      1.6.2. Syntax errors.
      1.6.3. Build configuration.
   1.7. Detection of intermittent build failures (EXTRA)
      1.7.1. Identification of first occurrence.
      1.7.2. Comparison with previous build success.
      1.7.3. Identification and report of differences between builds.
2. **Indication of candidate commits for broken Google tests**
   2.1. Investigation of UI alternatives for showing candidate commits on the continuous integration system
   2.2. Investigation of parsing algorithms for Google tests results
   2.3. Algorithm implementation?
   2.4. Indication of temporal history (first broken round, log intermittent failures, etc.)
   2.5. Detection on single commit build rounds
   2.6. Detection on multi-commit build rounds
   2.7. Detection of intermittent build failures (EXTRA)
3. **Indication of candidate commits for broken integration tests**
   3.1. Investigation of UI alternatives for showing candidate commits on the continuous integration system
   3.2. Investigation of parsing algorithms for printer logs, integration tests results and commits
   3.3. Indication of temporal history (first broken round, log intermittent failures, etc.) (EXTRA)
   3.4. Detection on single commit build rounds (EXTRA)
   3.5. Detection on multi-commit build rounds (EXTRA)
   3.6. Detection of intermittent build failures (EXTRA)

The tasks **1.1, 1.2, 2.1, 2.2, 3.1** and **3.2** are spikes, which are time investments on investigation, to figure out what needs to be built and how the developers are going to build it. The tasks with the (EXTRA) tag are going to be developed after the ones without it, since those are the main features. Therefore, if the project's timeframe allows it, the (EXTRA) tasks will be developed before the due date.

2. System architecture

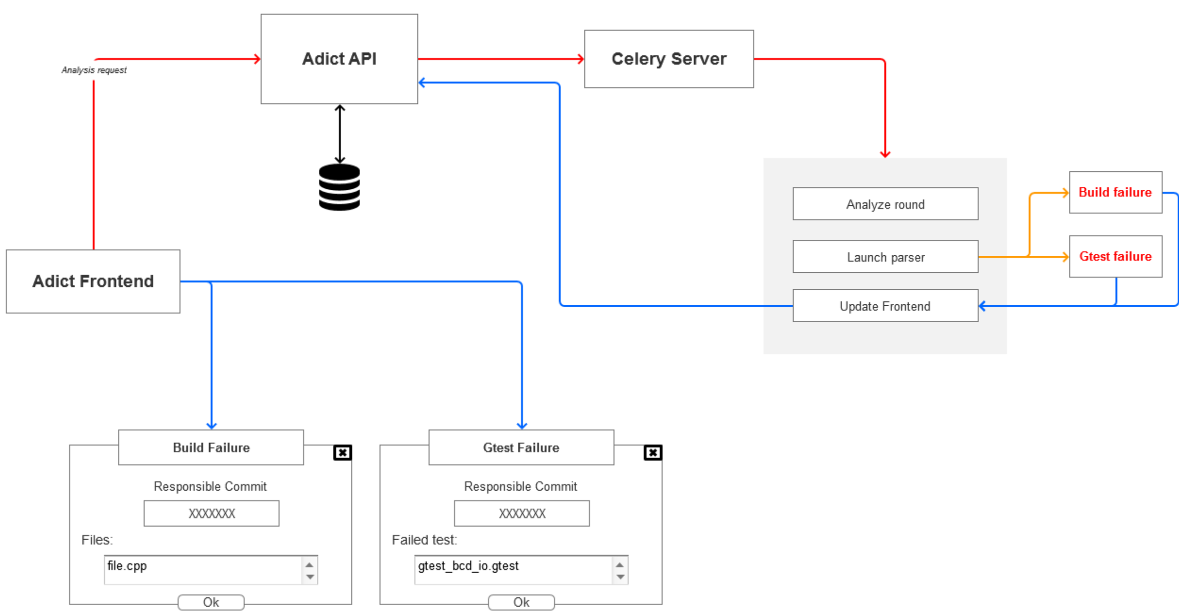
When developing a system that will automatize the analysis of build and test failures, there are several architectures that can be implemented. It depends on different factors, but mainly on the way the system is expected to work.

If the system analyzes the failures automatically without any request from the system integrator, the root causes of the failure will be analyzed *on-the-go*. This means that whenever there is any failure, the integration system API will notify the automated failure detection system, and this system will analyze it.

In order for this architecture to work, a task managing module would be needed. There is an interesting option, called Celery Project, which is an asynchronous task queue based on distributed message passing. It receives all the notifications from the integration system API, and interacts with the failure detection system, executing its tasks, depending on the failure that must be analyzed.

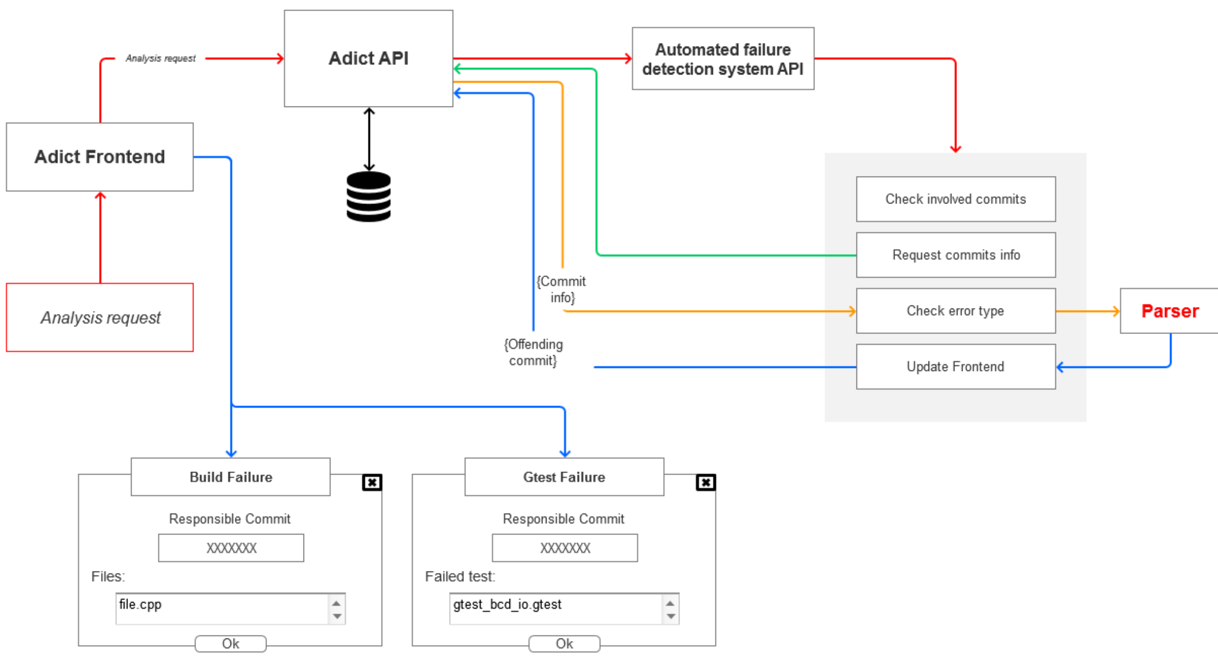
Each of those tasks is a Python script, that will distinguish, depending on the request data, between the different kinds of failures. After it, a parsing script will be launched, determining the reason of the failure, and sending it back to the integration system API.

The following diagram describes the architecture:


Analysis request
Adict API
Celery Server
Analyze round
Launch parser
Update Frontend
Build failure
Gtest failure
Adict Frontend
Build Failure
Responsible Commit
XXXXXXX
Files:
file.cpp
Ok
Gtest Failure
Responsible Commit
XXXXXXX
Failed test:
gtest_bcd_io.gtest
Ok

However, this architecture requires two factors. First, a lot of resources will be used when parsing continuously the error logs, which requires us to ponder the system overload the failure analyzer will add. This problem might be mitigated since the build failure analyzer system can be loaded into a different server than the integration system's server.

The second problem is that for the system to be able to analyze every kind of failure, a lot of research is required, to find adequate patterns able to discover the errors. Since the *Agile* methodology is being applied on this project, it makes much more sense to have an architecture that allows us to develop functionalities incrementally, such as different types of errors. Thus, the architecture shown on the following diagram has been designed, so the continuous system integrator requests the analysis for a concrete kind of failure:



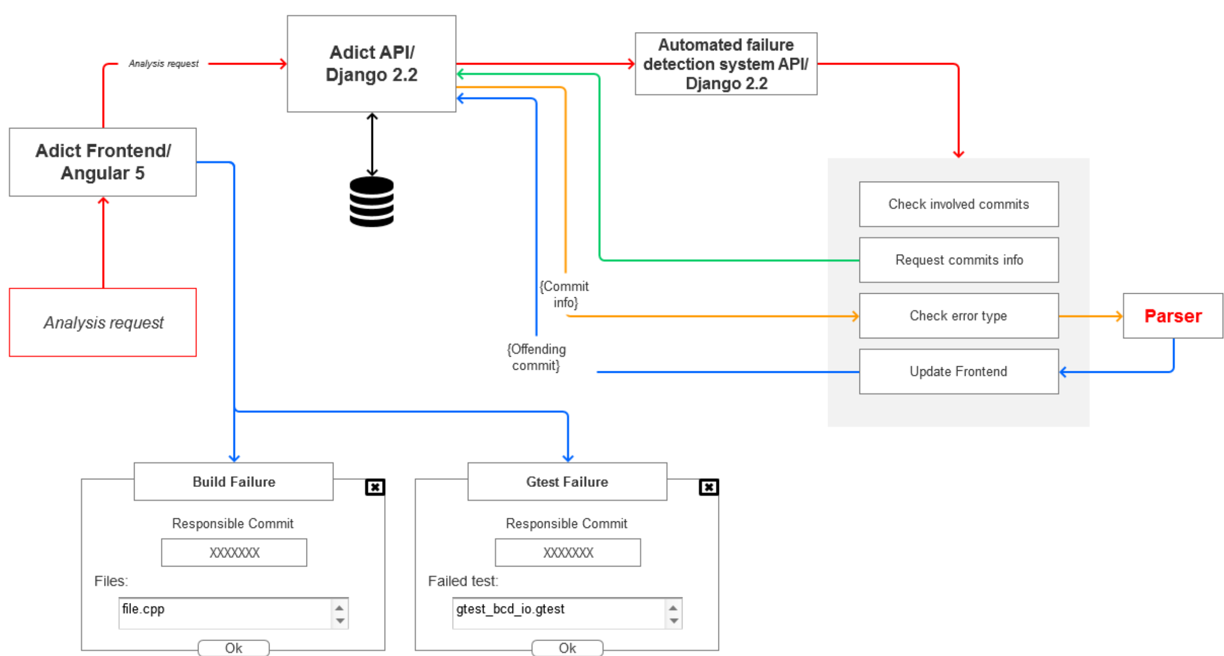
When the integrator requests an analysis, the *Adict API* (continuous integration system API), notifies the automated failure detection system's API, which will treat the failure appropriately.

With this architecture, the integration system will not experiment a noticeable overload, since not every failure will get analyzed, just those requested by the system integrator. However, the failure detection system will still be implemented on a separated server.

Like in the previously analyzed architecture, every analysis request will be run on a Python script, launched from the failure detection system, which also acts as entry point for the requests. This makes the system very easily expandable. Also, achieving a higher efficiency is feasible, since each of those scripts can be run on a separated thread, so it is possible to take advantage of the processor cores.

3. Development platform

As we can see on the following diagram, the system has been developed using several platforms.



The part of the system named *Adict API* is a prior service, existing before the implementation of the failure detection system started. Its platform is *Django 2.2*, and runs continuously on a Linux server, retrieving the integration's system data, and providing *Adict Frontend* the data it needs to show the developers the integration's status: compilation status, integration tests failures and successes, etc.

Adict Frontend is the existing user interface in the enterprise's system. It retrieves the integration data from *Adict API* and shows the developers the failures/successes on the integration rounds, which include product platforms compilations and integration tests results. It has been developed using *Angular 5*, which is a Typescript based framework for frontend development.

The automated failure detection system has been developed using the same platforms used in the existing system, in order to make its integration feasible on the future.
The system's backend is based on Django 2.2, retrieving the data from *Adict API,* and applying several algorithms on that data to find candidate commits for an integration failure. It runs as a service on a Linux server, listening for analysis requests sent from the user interface by the integrator.
Django provides a lot of flexibility to develop search algorithms, because it is based on Python, a programming language that offers lots of analysis capabilities at different levels, from analyzing JSON data to parsing failure logs, which have thousands of different patterns. Also, the readability of an algorithm developed in Python is quite easy, so maintaining and adding functionalities to the code does not require a lot of effort.

The automated failure detection system's frontend is based on Angular 5. The upgrade of the existing system to a newer Angular version was taken in account, but the effort and time that would have to been invested are too high compared to developing a new version of the system.

It receives the requests from the integrator, and sends them to the backend, which will, as seen on the previous section, calculate which are the candidate commits for the failure. The frontend will later receive the results of the calculations and show them to the integrator.

The frontend's architecture follows the Redux pattern, with the help of NgRx's library. The data is kept on the state, the only source of truth. The state oversees retrieving the data from the system's API and passes it to the different components.

4. Usability/UX

4.1. Usability

We can think of usability as the system features that enable the user to achieve the results the system is expected to produce, such as the UI and the actions needed by the user to achieve those results.

Given that the development methodology applied within this project is the well-known *Agile,* the usability features will be implemented in an incremental process, adding functionalities on each step, that are testable and doable by the user.

Following that philosophy, the development of the system has been divided into tasks, each one adding functionalities that bring closer the main objective of the project, the automatization of failure research. Those tasks are described as *User Stories,* which are simple descriptions of the system features from the point of view of the person who will make use of the new capability, usually the user of the system, in this case, the system integrator.

Also, in those tasks, there are acceptance tests, usually describing the functionality the user story is supposed to add to the system, that must pass in order to the task to be approved by the quality team.

For the Google test failures, an *Epic* has been defined. An epic is a functionality that is too large to be developed within a *sprint* (usually periods of 2-3 weeks), so it has to be split into smaller pieces of work, the mentioned user stories. The epic's description is:

As a system integrator I want to get the candidate commits for broken Google tests so that I can quickly decide what commit to revert.

As we can see, the functionality described by this epic is too large to be developed within a sprint. The user, the *system integrator,* wants to know which commits are the ones that might be the cause of the Google tests failures, without having to investigate the integrated code, and wants the system to show clearly what commits are the candidates.
But for the system, achieving this objective involves a few functionalities, that will not need user's interaction, but are needed in order to discover the candidate commits. Those functionalities are the pieces of work that the epic has been divided into, the user stories:

As a system integrator I want the system to show me the list of the last build rounds and show me the Google test failures in the one I select.

This user story describes the part of the system that will get the build rounds data, that will be used later for checking the build rounds where the responsible commit of the failure most surely has been integrated, and when the failure started happening. Also, the system integrator requires the system to show the failed tests for a given test suite.
As previously mentioned, for each user story there are acceptance tests, called *acceptance criteria* as well. In this case, those tests are:

Test	Status
The last build rounds IDs are shown	Not passed
The Gtest failures for the selected round are shown	Not passed

The tests describe the functionality expected to be delivered when the user story has finished. The user story cannot be closed until the test have passed. Usually, this is checked by the quality

team, but sometimes the same developer can run those tests, although it is not the most recommended manner to do it.

For enabling the system integrator to make a request, a UI mock-up has been defined:

The system integrator can make a request just by clicking on the *Request failure analysis* button. The next user story describes the part of the workflow where the system receives the request, which involves developing an entry point to the failure detection system API:

As a system integrator I want the system to acknowledge that I have requested the investigation of a given Google test failure.

In this case, the acceptance criteria test is:

Test	Status
The integrator gets notified that the system received his request	Not passed

Now, the system has to search for those commits that have been integrated between the last test execution success, and the first failure, in order to analyze the committed files, and correlate the failures found in the logs, and the changes done to the code:

As a system integrator I want the system to show me the list of commits involved in the failure (from the last success to the first failure).

From the system's user point of view, it might not be important if the system shows the commits where the responsible code of the failure could be, since the integrator just wants to know which commit is the candidate of the failure. But for the system to achieve its goal of finding the responsible commit of the failure, it is needed to know the set of commits where the candidate could be. The acceptance criteria test is described as follows:

Test	Status
The system shows the possible candidate commits involved in the failure	Not passed

When the system has the set of possible candidates for the test failure, the next step is knowing the test suite that the test belongs to:

As a system integrator I want the system to analyze the Google test failure log and show me the name of the suite that test belongs to.

With this information, the system will later be able of matching the suite with a commit, getting closer to the output the user is expecting.

Finally, with all the pieces of the system together, it can deliver its main feature: showing the integrator the candidate commits for the broken tests, so they can be reverted, without requiring any time spent on further investigation:

As a system integrator I want to get the candidate commits for broken Google tests so that I can quickly decide what commit to revert.

It may look like the system's architecture has not been described directly on this chapter, but in fact, putting all the user stories together, the system gets developed. From the first task, when only the list of last build rounds was being requested to the integration system's API, to the last story, the whole system has been developed, since for delivering all the user stories, the system's architecture described on the previous chapter, had to be implemented. The results of the analysis will be exposed to the system integrator on a dialog that looks like the following mock-up:

Gtest Failure

Responsible Commit

XXXXXXX

Failed test:

gtest_bcd_io.gtest

Ok

Another epic has been defined for the build failures analysis:

As a system integrator I want to get the candidate commits for broken compilations so that I can quickly decide what commit to revert.

It describes the expected functionality from the failure analysis system related to the build failures. Once again, the task must be split into smaller slices, since it is too large to be developed in a 2-3-week time period. The first part of the build failure analyzer system to be developed will just show the list of the last build rounds, as described by the epic's first user story:

As a system integrator I want the system to show me the list of the last build rounds.

The last build rounds will be needed by the analysis system to obtain the set of possible candidate commits, as explained in the Google tests case. The request can be done by the system integrator on a dialog that looks like the following mock-up:

SUCCESS zynq-threadx-ass Build log

FAIL wrl17-intel_x86_64-dbg Build log Request failure analysis

Backtrace

```
build_log: /tmp/cibprod5256:18000/bcd/295211/2-autoL2/polestar/wrl17/wrl17-intel_x86_64-dbg/wrl17-intel_x86_64-dbg.txt
build_type: atlas
make[1]: *** [all] Error 1
```

Secondly, as in the previous epic, the user will be grateful if the system displays a message when the analysis request is received. The next user story describes that case of use:

As a system integrator I want the system to acknowledge that I have requested the investigation of a given build failure.

Now, when the system has received the request and acknowledged the user, the next step in order to add functionality that brings the system closer to its main task, is to select from the last build rounds set only those that might be involved on the build failure, that is to say, the commits done between the last success until the first failure happened:

As a system integrator I want the system to show me the list of commits involved in the failure when error type x happens (from the last success to the first failure).

The system now has bounded the original set of commits to a few, the ones within most surely will be the responsible commit of the build failure. Thus, there are only a few build logs where the failure could be. The system must be able to analyze those logs, for a determined type of error, and check with product does the compilation error belong to. This information will help discarding false candidates. To do so, the next user story has been defined:

As a system integrator I want the system to analyze the build failure log when error type x happens and show me the name of the product that the compilation belongs to.

Finally, with all this information gathered, the system will be able to correlate which commits, from the previous candidate sets, may be the commit that caused the build failure. The last step is to notify the user that a candidate or candidates have been found (since on each build round

more than one commit could be done, there may be several candidates, that break different products, or maybe the same product):

As a system integrator I want the system to notify me of commit candidates when error type "x" happens.

Developing these small tasks, the failure detection system is now able to identify a build failure, and correlate it with the commit that may be breaking the compilation, so the system integrator now doesn't have to invest a lot of time looking for a candidate, and can spend that time on a more important task for the company.
The candidate commits will be shown to the system integrator in a dialog that looks like the following mock-up:

Build Failure

Responsible Commit

XXXXXXX

Files:

file.cpp

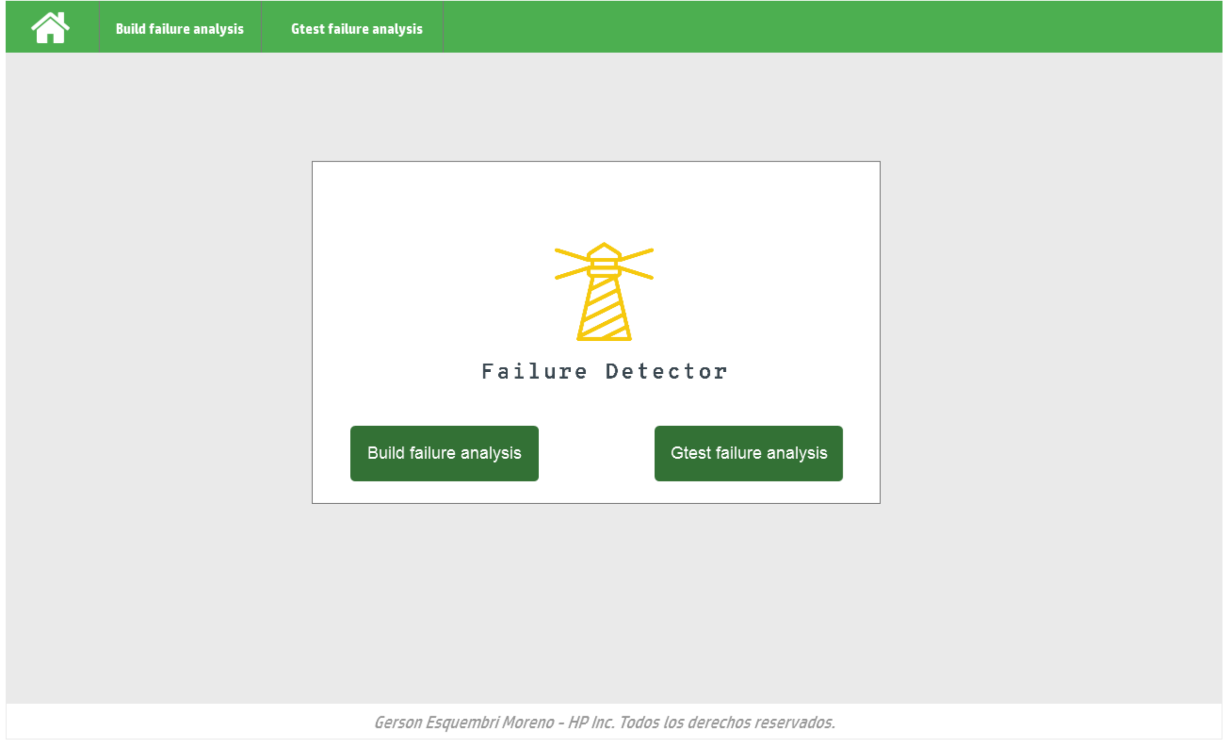
Ok

The investment needed to develop a system with the mentioned features is high, as seen in this chapter. But when the system starts being functional, the developers will spend less time looking for the build and test failures.

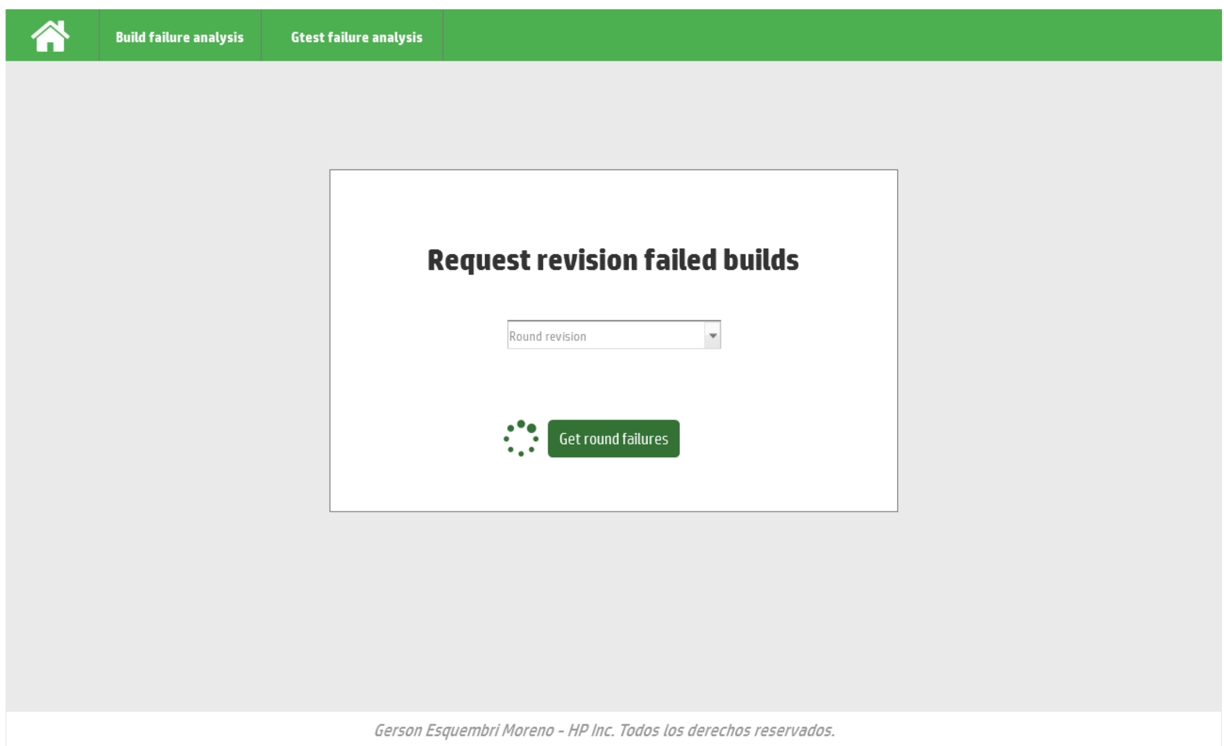
4.2. UX

The design of the user interface follows the styles of the already existing system, since the only parts that will remain after this thesis are the dialogs where the results of the failure analysis are shown. However, to summarize the real workflow the system integrator will do when using the failure detection tool, a complete user interface has been designed.

The next mock-up shows the design of the main page, where the user can choose between analyzing build or integration test failures.

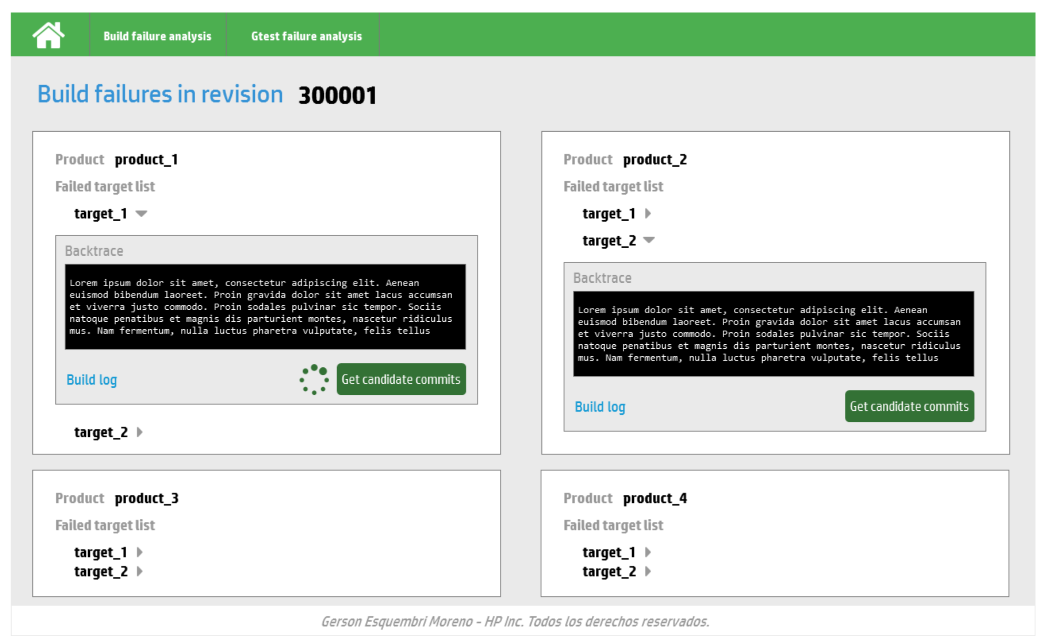


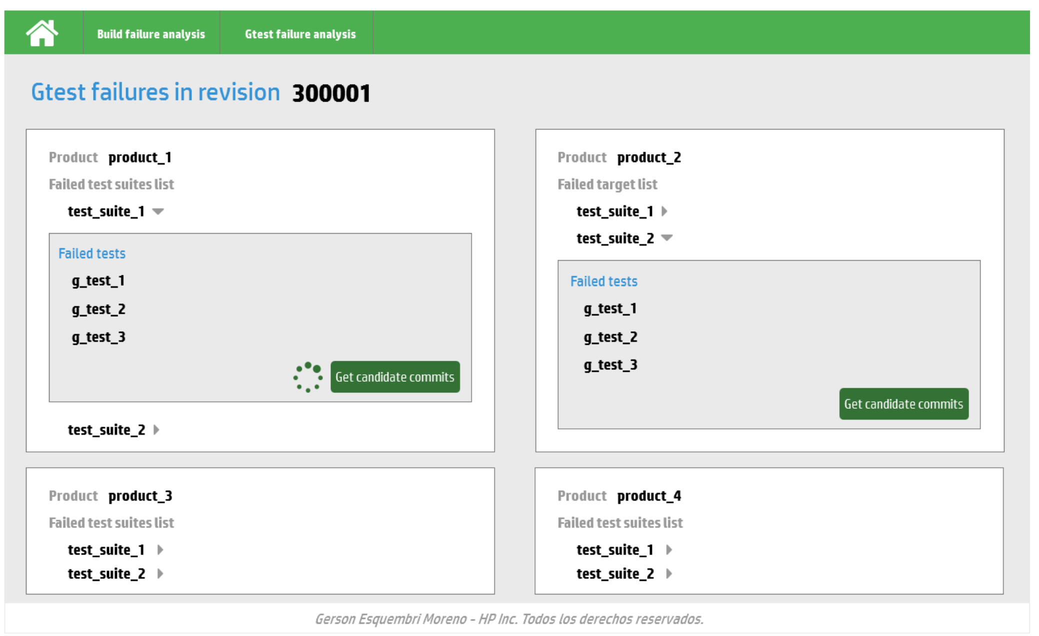
The next part of the workflow represents the existing integration's system interface, where the integration rounds are exposed. Here the system integrator can choose between the last integration rounds and request the failure detection system the failures that happened on a round. For both the build and integration test failures, this step is similar.



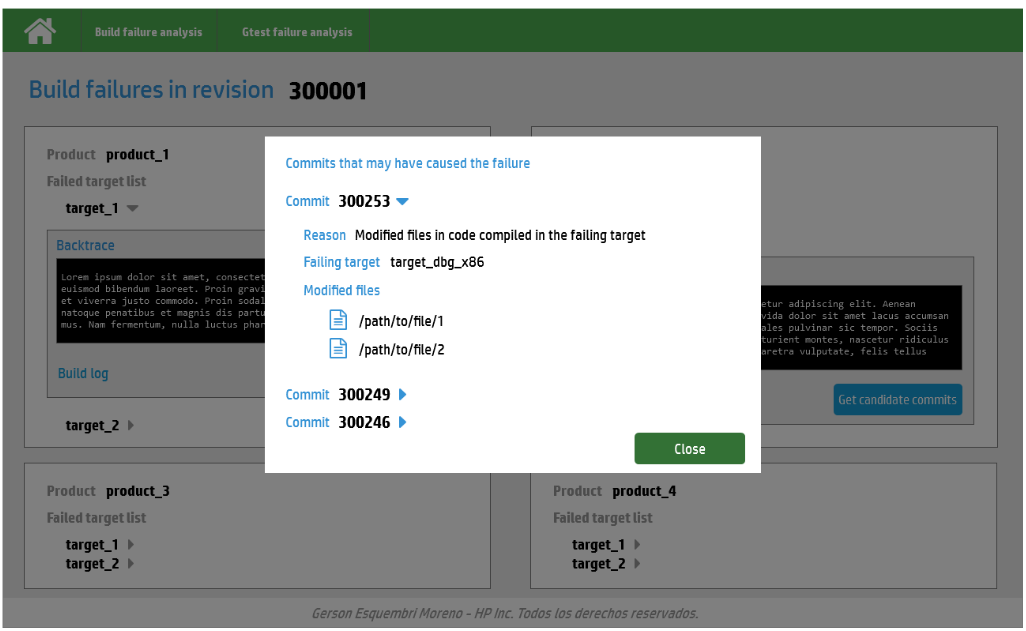
After choosing a round there are two possible outcomes:

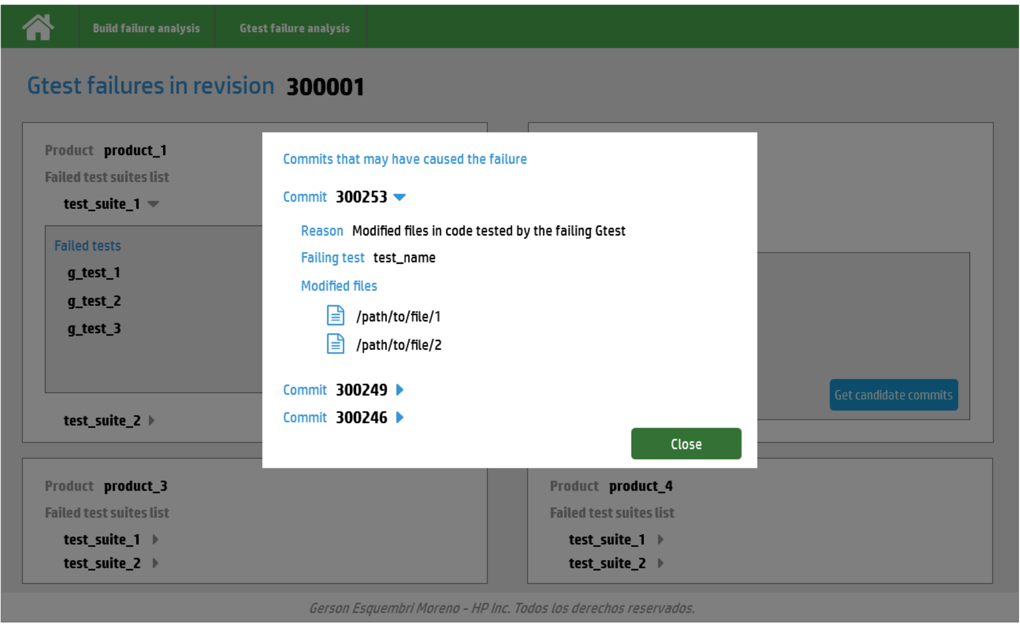
- There are no failed builds nor failed integration tests on that revision, so a message will appear indicating that there are no failures in that round.
- There are failures, from one to many. If that is the case, the next step's interface will show those failures, as it can be seen on the next images:


Build failure analysis
Gtest failure analysis
Build failures in revision 300001
Product product_1
Failed target list
target_1
Backtrace
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean euismod bibendum laoreet. Proin gravida dolor sit amet lacus accumsan et viverra justo commodo. Proin sodales pulvinar sic tempor. Sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nam fermentum, nulla luctus pharetra vulputate, felis tellus
Build log
Get candidate commits
target_2
Product product_2
Failed target list
target_1
target_2
Backtrace
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean euismod bibendum laoreet. Proin gravida dolor sit amet lacus accumsan et viverra justo commodo. Proin sodales pulvinar sic tempor. Sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nam fermentum, nulla luctus pharetra vulputate, felis tellus
Build log
Get candidate commits
Product product_3
Failed target list
target_1
target_2
Product product_4
Failed target list
target_1
target_2
Gerson Esquembri Moreno - HP Inc. Todos los derechos reservados.


Build failure analysis
Gtest failure analysis
Gtest failures in revision 300001
Product product_1
Failed test suites list
test_suite_1
Failed tests
g_test_1
g_test_2
g_test_3
Get candidate commits
test_suite_2
Product product_2
Failed target list
test_suite_1
test_suite_2
Failed tests
g_test_1
g_test_2
g_test_3
Get candidate commits
Product product_3
Failed test suites list
test_suite_1
test_suite_2
Product product_4
Failed test suites list
test_suite_1
test_suite_2
Gerson Esquembri Moreno - HP Inc. Todos los derechos reservados.

Finally, when candidates to be the responsible commits of the failures are found, a dialog with the related information will appear on the screen.





5. UML Diagrams

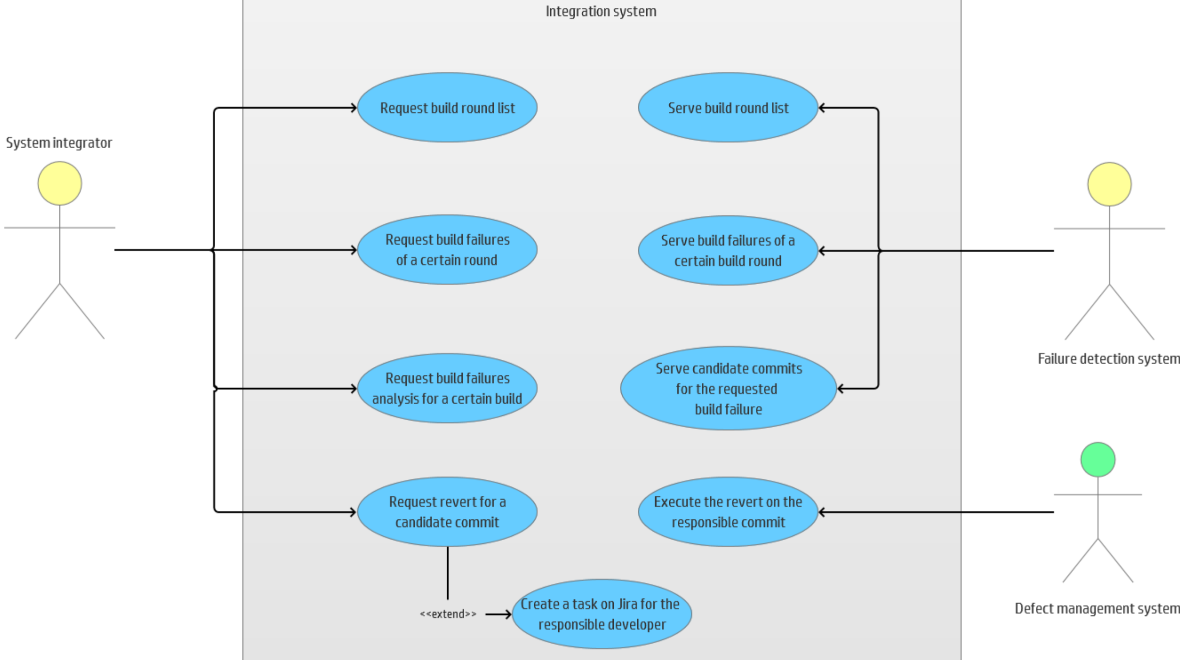
On this section, the system will be described by its functionalities and data flows.

5.1. Use cases

The system could be described as a workflow conformed by three main parts, also called *actors*:

- The **system integrator,** who is the responsible of the stability of the integration system, keeping track of the introduced changes, and how do they affect the system.
- The **failure detection system,** a *non-human* actor, software based, which analyzes the build/test failures requested by the system integrator and provides the candidates to be the responsible commit of the integration failure.
- The **defect management system,** another *non-human* actor, software based, which includes the revert monitor (reverts a commit whenever the integrator requests it) or Jira, the task management system, where the integrator communicates the developers that their changes need to be reviewed and fixed.

In the case of the build failure analysis, the use cases are summarized on the following diagram:

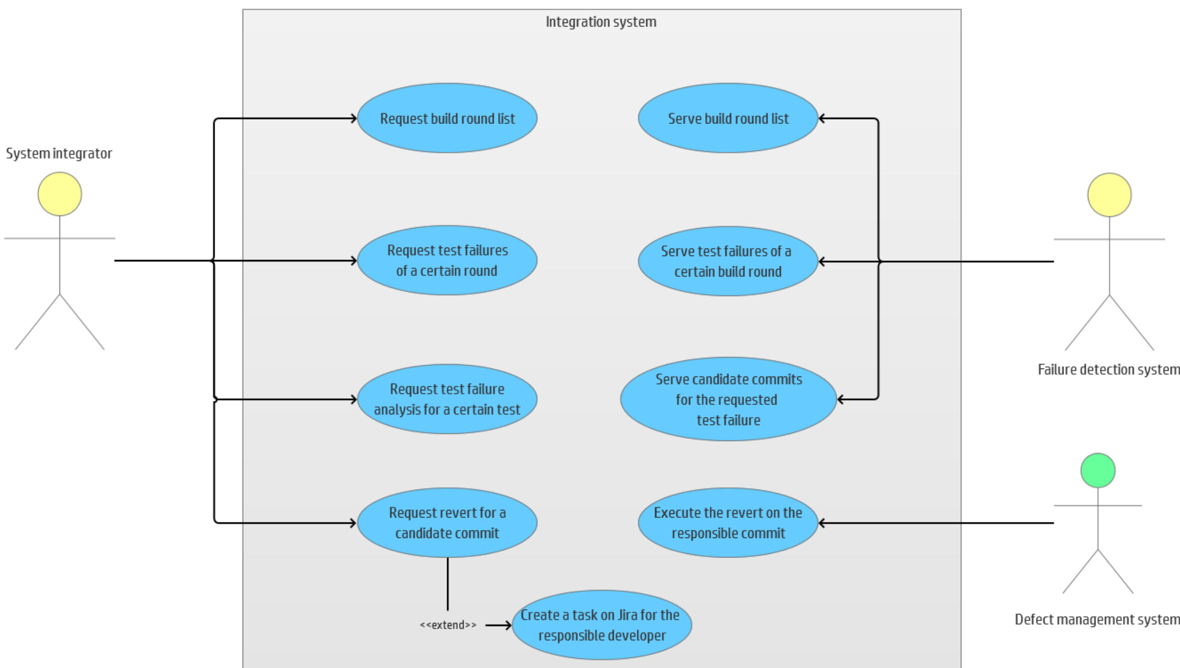


As it can be observed, the *system integrator* can request the last build rounds list, the build failures of a certain round from that list, the analysis of a certain failure, and the revert for the responsible commit that the failure detection system.

The *failure detection system* serves the requested data by the system integrator, analyzing which could be the candidate commits.

Finally, the *defect management system* executes the revert requested by the system integrator and supports the creation of tasks for the responsible developers of the failures to fix them.

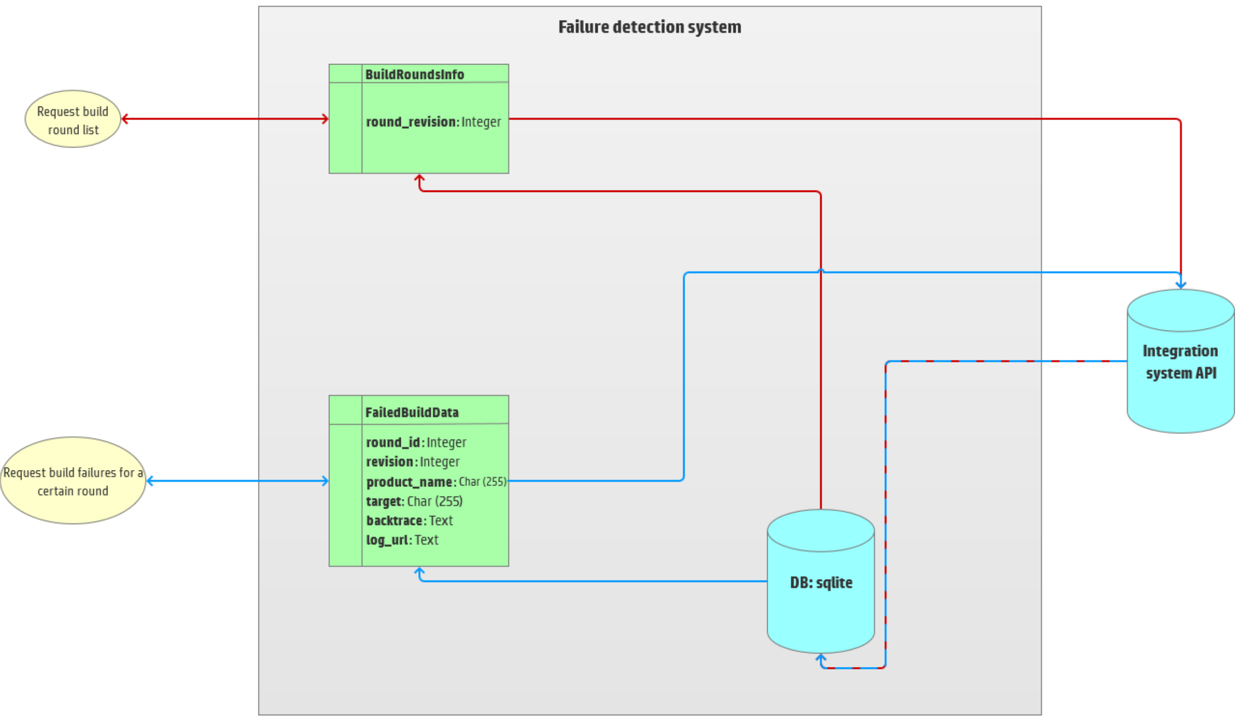
The case of the Google test failures is similar, as it can be observed on the following diagram:



5.2. Data flows

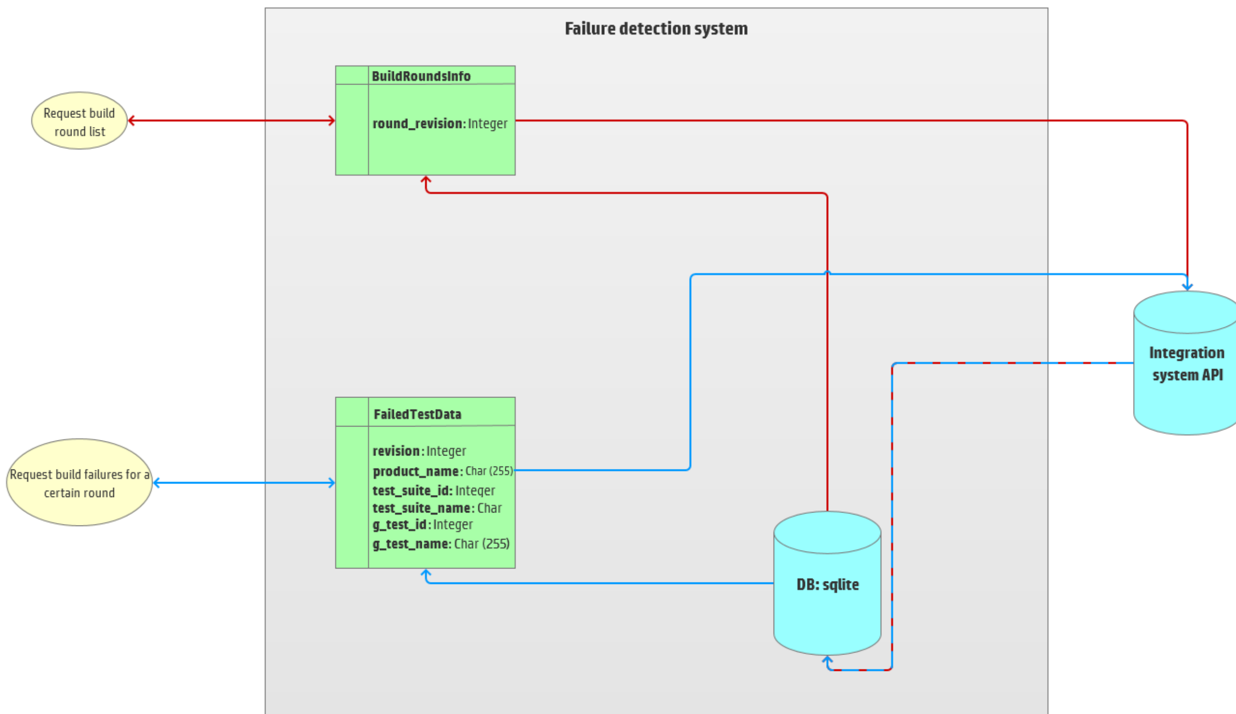
Within the different kinds of requests that the system integrator can make to the failure detection system, there are several data models involved.

For the *build failure analysis* requests, the data flow is described in the following diagram:



As it can be observed, first, the system integrator requests the last build rounds revision numbers from the system API. Then, for each of those rounds, the related information can be requested. The data obtained in both of those requests gets stored on the failure detection system's database, which uses *mysql*.

The data flow for an integration test request is similar, as it can be observed on the next diagram:



6. User profiles

As we have seen in the previous chapters, this system is oriented towards development environments with many developers that commit code changes into the same development branch. Usually, in those environments a lot of products are developed, involving thousands of integration tests that are run at the same time, resulting in a very complex system, that has somehow to be taken care of. The people in charge of the system must be engineers with a strong technical background, that have knowledge about the products that are being developed, and how they are being developed.

These engineers are the users of this system, so it must present the results clearly, and as fast as possible.

Probably, the engineers in charge will be more interested on not having to invest much time on finding the candidate commit(s) and on an intuitive UI, making the tool's usage easy to learn.

For the scope of this project that is not hard to achieve, since the time needed for the system to correlate the failures with the committed changes will be short, due to a small set of different kinds of failures taken in account. But for the future, it may be interesting to explore different possibilities to show the candidate commits to the system integrator, such as receiving the analysis request, and notifying the integrator later through an e-mail, desktop notification, etc.

7. Viability

Setting up an automated failure detection system can be quite expensive, not in hardware, but in years of investigation, researching how to automatically find the reason of the failures, unless the set of failures that may happen is clearly identified.

On the other hand, for the company it might be an interesting option, since nowadays a lot of resources have to be spent in order to keep their integration systems stable, and those

resources could be used to develop better products.

Usually, environments with a large number of developers are only found on large companies, that have a wide product portfolio. For those companies it is always important to automatize repetitive tasks as much as possible, such as controlling their integration system.

An automated failure detection system like the one described on this project just needs a server, or even could be run on a server that also runs other processes, depending on the failure analysis workload. It could be even developed by a third party, and adapted to different environments, even allowing to personalize the failures detected.

8. Failure detection algorithms

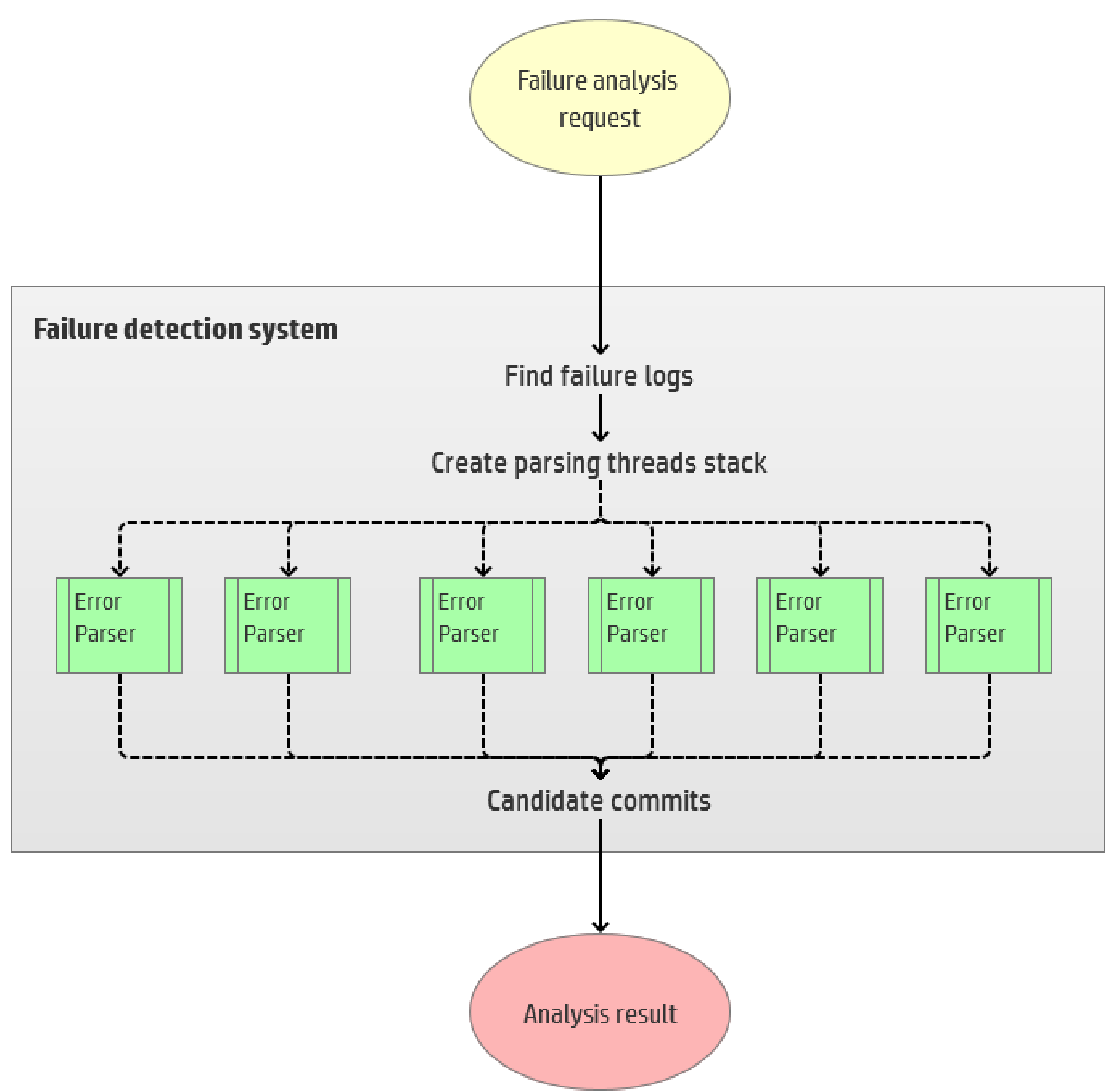
Given that the whole set of integration failures is almost infinite, for this project's time frame the implemented failure detection algorithms only detect a few kinds of failures. However, the analyzed cases cover around 80% percent of the integration failures that happen nowadays.

Failures that have never happened before are also a possibility, so the already mentioned set of integration failures keeps growing at every moment.

Other kinds of failures are infrastructure failures, that might happen when new technologies are included, or when part of the infrastructure is down. Some failures are intermittent, do not happen on every integration round, so finding a reason for them is not trivial.

The developed detection algorithms are based on error logs parsing, searching for keywords that might indicate where the reason of a failure is located. Also, the algorithms look for similarities between the component names which integration tests failed, and the names of the files modified on the previous commits.

In order to make the failure analysis as efficient as possible, a multithreading model has been designed, making use of Python's Threading libraries. A brief summary can be found on the following diagram.


Failure analysis request
Failure detection system
Find failure logs
Create parsing threads stack
Error Parser
Error Parser
Error Parser
Error Parser
Error Parser
Error Parser
Candidate commits
Analysis result

9. Future projection

During the timeframe for the development of this project, only the failure detection system's architecture and coverage for ~80% of the cases have been developed.
Integrating this tool on the already existing system is the main objective after the completion of the thesis. After its successful integration, since the architecture of the project's backend makes it easy to extend the failure cases coverage, more detection algorithms will be developed.

Beyond extending the coverage, more techniques could be applied, such as developing a neural network capable of learning how to detect the existing failures kinds, in order to detect unknown failure types that might appear on the future.

Given that the failure detection tool is highly coupled to the existing integration system, making it agnostic of the system's architecture and abstracting its components could help on its maintainability, and applying it to future versions of the integration system, without releasing major changes on its architecture.

10. Installation requirements

As explained on the previous chapters, the automated failure detection system requires three main components to work:

- A database with the integration failures logs. In our case of study, it is called *Adict API*, but some other options could be taken in account for the system to work, or at least to show its capabilities to the potential customers. For that purpose, the option selected is Mocklab, a web-based service that helps creating *stubs*, which are simulations of the integration system's different entry points, that reply with a predefined response to a predefined API call.

- A Django based service, containing all the functionalities related to the failures' analysis. The service can be run on several operating systems, such as Windows or any Linux distribution, though nowadays Linux is the most used system for server-side services, due to its stability. The actual's system implementation is run on Linux.

- The user interface, developed in Angular 5, does not need any special installation for its usage, since all the needed files will be bundled with the production version of the system.

11. Installation instructions

The installation of the *failure detection* system on a production server can be done following the next steps.

11.1. Preparing the environment

First of all, it is recommended to install Django on a *virtualenv.* In order to do that, the related packages have to be installed:

```
user@box ~ $ sudo apt install virtualenv virtualenvwrapper
```

After that, a *virtualenv* can be created. A *virtualenv* is a container that keeps all the packages needed by a Python project on a directory, isolating them from the system.

```
user@box ~ $ cd /var
user@box ~ $ mkdir virtualenvs
user@box ~ $ sudo chmod 777 /virtualenvs
user@box ~ $ virtualenv --python=/usr/bin/python3 /var/virtualenvs/failure-detector
```

After creating the *virtualenv,* it can be activated by issuing the following command:

```
user@box ~ $ source /var/virtualenvs/failure-detector /bin/activate
```

Finally, Django's dependencies can be installed. After downloading the project files and activating the *virtualenv*, from the project's directory, by executing the next command the required dependencies will be installed:

```
user@box ~/failure-detector-backend $ pip install -r requirements.txt
```

Now the routing stack has to be installed. The stack is composed by:

- **Nginx,** a reverse proxy that takes client requests, sends the request to the server, and delivers the server's response to the client.
- **uWSGI** *(Web Server Gateway Interface)*, that forwards clients requests from Nginx to Django.

Nginx can be installed executing the next order:

```
user@box ~/failure-detector-backend $ sudo apt-get install nginx
```

Prior to uWSGI's installation, a compiler of the C language must be installed, such as *gcc.* Also, Python 3 development packages have to be installed. Once these packages are set up, uWSGI can be installed by issuing the following command:

```
user@box ~/failure-detector-backend $ pip3 install uwsgi
```

Depending on the set up of the production server's URLs, different Nginx and uWSGI configurations may be applied. Examples can be found on the root of the project's files.

If uWSGI's instantiation script available on the project folder is applied, in order to initialize the project, the next commands must be executed:

```
user@box ~/failure-detector-backend $ sudo /etc/init.d/nginx start
user@box ~/failure-detector-backend $ sudo uwsgi –ini failure-detector/failure-detector-uwsgi.ini
```

After setting up the environment as described, the system can be accessed by introducing the configured URL on a browser.