Butler

Project Requirements and Specifications

Schweitzer Engineering Laboratories

Horizon Group

Brendan Crebs

9/28/23

TABLE OF CONTENTS

I. Introduction 3

II. System Requirements Specification 3

II.1. Use Cases 3

II.2. Functional Requirements 3

II.3. Non-Functional Requirements 6

III. System Evolution 7

IV. Glossary 8

V. References 8

I. Introduction

Butler is a multithreaded build, lint, test and package application that is intended for use as a continuous integration (CI) tool. Continuous integration is the practice of frequently building and testing code as it is being developed as a method of upholding code quality and finding issues early[1]. The use of CI is becoming standard practice for many companies[2] due to its numerous benefits which is what prompted the need for a tool like Butler at SEL. Butler builds a series of “tasks” for build, test, lint and package processes and then spins off child processes to fulfill these tasks. An example of a test task would be a set of unit tests for an application in a user’s repository. If these unit tests fail, then Butler would fail, and you would be prevented from merging your code onto the main branch. This is how Butler would uphold code standards and improve code quality and team productivity.

The intention of this project is to have Butler be a generalized tool that can be used in any repository regardless of what languages and technologies are used. This could replace many of the CI tools that teams have hardcoded for their repositories, thus allowing them to not have to maintain yet another application. It could also be distributed to teams that lack CI processes, allowing them to benefit from CI without having to take the time and effort to make a tool on their own. The generalized nature of Butler is also helpful for a team changing what technologies and languages are used in their repository. A hardcoded CI tool would need to be partially rewritten in such cases. With Butler only small configuration file changes would be needed to accommodate the technology change.

II. System Requirements Specification

II.1. Use Cases

For Butler, a use case diagram is somewhat unnecessary since there is really only one use case which is to run Butler. After Butler is run, it will produce a result that is binary. Either the build succeeded or failed. While Butler is running there is nothing the user can do to interact with how Butler functions. The configuration setup that occurs before butler is ran, I would argue is a prerequisite to using Butler rather than a use case. So, I would argue that Butler only has one high level use case between itself and the end user.

II.2. Functional Requirements

1. Butler Configuration

Dynamic config structure:

Description	If Butler is going to be a generalized tool it will need a dynamic config file structure that handles for the vastly different development environments it could be placed into. This leaved a more static approach of having a series of key value pair options inadequate for this task. We will need to supply a syntax similar to that of a makefile to allow for a wide range of options pertaining to different possible languages and repo setups.
Source	Requirement for a generalized tool elicited from the client.
Priority	Priority Level 0: Essential and required functionality.

Nested config files:

Description	Butler cannot assume that a team is going to have consistent practices throughout all their application packages. Some applications of the same language may require different commands to correctly execute tasks. To handle this, we could provide the ability to create nested Butler config files that would pertain only to a particular nested workspace in the repo. Whatever is defined in this file would override options defined in the global Butler config for that workspace.
Source	Internal team requirement elicitation.
Priority	Priority Level 1: Desired functionality.

Config processing:

Description	Before Butler creates and executes any tasks it needs to determine several essential repository attributes based off the information it parsed out of the config files. Most importantly, it needs to correctly determine what languages it will have to deal with and how the constraints supplied in the config file will affect what tasks it creates, how it creates them, and how it will execute them.
Source	Requirement for a generalized tool elicited from the client.
Priority	Priority Level 0: Essential and required functionality.

.butlerignore file:

Description	There are several situations in which we would not want Butler to walk though parts of a repository. Examples of this would be special test directories or massive auto generated folders such as node_modules directories which are created by Nodejs to house dependency code. We could solve this issue by providing a .butlerignore config where specific paths and path patterns could be supplied so that butler would ignore those paths while walking the repo. We would certainly add this functionality to the regular butler config but a special ignore file would be useful in cases where there are many paths to ignore and putting them in the base config would make it too bloated.
Source	Internal team requirement elicitation.
Priority	Priority Level 2: Extra features or stretch goals

2. Task Construction

Walking the repository:

Description	After Butler has processed all the information from the config it will then need to walk through the repository looking for workspaces to create tasks for. Butler needs a method for determining the workspace such as finding a package.json. An assortment of language specific methods will be created to satisfy this task.
Source	Requirement for a tool that dynamically determines workspaces elicited from the client.
Priority	Priority Level 0: Essential and required functionality.

Creation of build, lint and test tasks:

Description	On the condition that Butler has the information it needs, and that build/lint/test tasks are enabled, Butler now must construct build, lint and test tasks for each workspace. A special task object will be created for each task based on information obtained from the config and the walk through the repository.
Source	Requirement for a tool that runs lint and test tasks elicited from the client.
Priority	Priority Level 0: Essential and required functionality.

Creation of publish tasks:

Description	On the condition that Butler has the information it needs and is being run on a publish branch, publish tasks must be created. Publish tasks must be created for applications that are marked for need of publishing. The publish task will bundle the application and then push said application to a preconfigured artifactory registry.
Source	Requirement for a tool that runs publish tasks elicited from the client.
Priority	Priority Level 1: Desired functionality.

Creation of fuzz tasks:

Description	Butler will include the option of running fuzz tasks as a subsection of test tasks. These tasks will run fuzz tests for specified workspaces and a for a specified amount of time or inputs determined by the user.
Source	Requirement for a tool that runs fuzz tasks elicited from the client.
Priority	Priority Level 2: Extra features or stretch goals.

3. Execute tasks

Multithreaded execution of tasks:

Description	After tasks have been created, Butler will need to execute said tasks and then display the result if the task passed or failed in real time to the user. If the task failed Butler must include why it failed. Due to the enormous number of tasks that may be created, this will be a multithreaded process where each task will be spun off on an available thread.
Source	Requirement for a tool that executes tasks elicited from the client.
Priority	Priority Level 0: Essential and required functionality.

Saving and displaying results:

Description	At the conclusion of a Butler build the results must be printed to a butler_results.json file for future reference. These results will be a detailed record of the results of every executed task. The file will be created and pushed to a database specified by the user. In addition, a small summary of butler results must be printed to the command line wherever Butler was run at its conclusion. This will occur every time regardless of if Butler was successful or not.
Source	Requirement for a tool that supplies information about the build elicited from the client.
Priority	Priority Level 1: Desired functionality.

II.3. Non-Functional Requirements

Documentation:

The system configuration should be as intuitive as possible. However much of the setup for a user will require knowledge of precisely what needs to be supplied to Butler. Because of this the system must include extensive documentation that must be detailed and clear so that even a non-experienced developer can properly configure Butler to be used in their repository.

Performant:

The system must be optimized to complete all tasks as fast as possible. As a CI tool, Butler will be run frequently throughout the day as developers push code to remote repositories. Often times developers will need to know if a build has succeeded or failed before they progress with what they are working on. Therefore, performance is a major consideration for Butler.

Visually helpful:

Butler should not hide most information from the user and instead should provide the user with helpful feedback. This feedback should be especially detailed if Butler fails so that a user will quickly know what to do to fix the issue.

Environmental constraint:

It is an expectation that Butler will be executed on a Linux machine or within a virtualized Linux subsystem. This is true regardless of whether Butler is run locally or on a build machine.

Machine access:

For Butler to be used to its full potential it's expected that Butler should have access to all available threads on the machine it's running on.

Adaptability:

The system must be highly adaptable so that it can handle any reasonable situation by the user supplying the relevant information.

III. System Evolution

For this project one of the most important system evolution factors to consider is the potential changing nature of programming languages. Butler is going to ultimately need several hardcoded methods for every commonly used programming language. As programming languages are updated, their nature could change in a way that would make our methods useless. This may require tedious maintenance in order to be constantly monitoring how languages are changing version to version. Another possibility would be the release of new languages that teams could use but remain unsupported by Butler. We would have to include a clear list of what languages are supported at which versions. We could also potentially consider including a feature that allows a user to supply their own method for Butler to use in specific tasks such as determining language dependencies or versions. This would give teams the ability to still use Butler in a case that Butler temporarily does not support the technologies being used in their repository. Other solutions would include highly adaptable and general methods that can be used as a fallback if no specific method can be used for a language. This would hopefully cover most edge cases and prevent the need for a user to supply their own code, a situation which we wish to avoid. Further mitigation of this issue would come in the form of heavy contact with our customers. Since this application is to be used internally within SEL, we can reasonably maintain individual contact with our customers and maintain fast turnaround times for updates when they encounter these issues.

IV. Glossary

Fuzzing: the testing practice of feeding code enormous amounts of random inputs in order to find bugs or vulnerabilities.

Golang: A widely used object oriented programing language.

Continuous Integration(CI): development practice of continuously building and testing code. This is further defined in the introduction.

V. References

[1]“Atlassian Continuous integration page, ”Atlassian, 2023. [Online]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration. [Accessed: 9/28/2023].

[2]P. Sane, "A Brief Survey of Current Software Engineering Practices in Continuous Integration and Automated Accessibility Testing," *2021 Sixth International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, Chennai, India, 2023.[Online] Available: https://ieeexplore.ieee.org/document/9419464, [Accessed: 9/28/2023]