Butler

Project Alpha Prototype Report

Schweitzer Engineering Laboratories

Horizon

Brendan Crebs

11/13/23

TABLE OF CONTENTS

I. Introduction

1.1. Butler Introduction

In software engineering the practice of continuous integration(CI) has become an increasingly important part of the software development process. Continuous integration is the idea that code from multiple contributors should be frequently built and tested as it is being pushed to a main branch. A tool that automates this process solves a number of common development problems which if solved, will lead to significantly greater efficiency in the long run. The aim of this project is to develop a generalized CI tool called BuTLer(build, test, lint) which we seek to use in multiple repositories at SEL.

One of the primary motivations of this tool is to have it generalized so that it will work in a number of drastically different code bases. At SEL, there are already existing hardcoded CI tools that some teams have developed for themselves. Our team seeks to provide a standardized CI tool that can parse through any repository and construct a series of build, test, lint and publish tasks if the tool is being run on a publish branch. For SEL this could greatly increase the productivity of a number of teams who do not use CI or use inferior CI tools/processes.

1.2. Background and related work

More broadly the software domain of this project would be automation, more specifically the domain would be “continuous integration” which was defined in the introduction. One of the biggest problems a CI tool like Butler seeks to solve is “integration hell” where it becomes incredibly difficult to integrate new code due to integration errors which arise when a child branch greatly deviates from the master branch. Having constant feedback on how your code may break things upon integration prevents a multitude of issues from arising and saves the time and resources that would be needed to fix them. In the domain of continuous integration one of the most well-known and state of the art CI tools would be Jenkins. Jenkins is a highly flexible tool that primarily provides a syntax for a “pipeline” feature which can be setup to run a number of CI steps on a dedicated build server[1]. The flexibility of Jenkins allows for more in depth and specific CI tools like Butler to be easily integrated as a step in the pipeline. Jenkins itself is not sufficient for the task that Butler tries to solve. Without Butler we would have to define an enormous number of hardcoded tasks for Jenkins to perform. Instead, Butler will automatically determine project packages, construct build, test and lint tasks for each package and execute said tasks. The automated nature that Butler will have of parsing through the entire repository looking for application packages is a functionality that other CI tools do not supply. To accomplish this without leaving Butler error prone we will supply a syntax for special Butler configuration files that can be thought of as similar to makefiles where a team could specify Butler’s behavior in their repository.

1.3. Project Overview

Butler will be implemented as a CLI tool written in Golang. Butler could be run locally, but most importantly it would be run on a build server as a step in a Jenkins pipeline. Each time code

would be pushed from a local branch to the remote origin Butler would be run. When it is run it will construct a series of tasks that can be separated into build, test, lint, and in certain situations publish tasks.

Butler will start by reading configurations from a Butler config file that will be defined by the team. Since Butler could be operating in a variety of different repositories that use different languages, it is important that the users supply some information such as where to find dependencies for a language and what the relevant test and lint commands would be for that language. A .butlerignore file will also be supported which will allow a user to input paths for Butler to not check. After this, it will determine the git branch and the git diff of the current branch compared to the main branch. Checking the git diff will allow us to dynamically determine what application packages to create tasks for. We can then avoid always making tasks for all packages in the repo. This will greatly improve performance by excluding irrelevant packages. Determining the git branch is also very important. If a branch is master or a publish branch, then we will indeed create tasks for every application package along with the creation of special publish tasks. More on that later.

After the setup for Butler has been completed it will go on to satisfy the first objective which is linting. Linting is checking the code for stylistic issues, redundancies, or code that could potentially break. Butler will walk through the repo it is running in and execute a language specific linting command for each package. Anything flagged by the linter will cause the Butler to fail. It is our intention to make it so it is required that the most recent run of Butler on a non-publish branch is successful before that branch can be merged with the master branch. This is so we can avoid merging incorrect code.

The next objective we want Butler to achieve would be test tasks. In a similar fashion to linting Butler will execute a language specific test task for every relevant package. The test command will execute unit tests and also gather coverage. Failure of any test or failure to achieve 100% test coverage will result in Butler failing the build. In addition to running unit tests, we also want Butler to run special fuzz tests. These fuzz tests will generate millions of random inputs for each package in an attempt to break them or find vulnerabilities. Discovery that an input breaks a package will result in Butler failing.

The next objective would be build tasks. Build tasks will be generated based on docker files that accompany an application package. In the various repositories at SEL there exist many applications within individual repos which Butler will have to parse through looking for relevant dockerfiles. To perform build tasks Butler will simply execute the instructions contained within the dockerfile adjacent to an application package.

Project Description and Clarification 4 On child branches from the main branch Butler will finish with build tasks and print a large collection of data collected from the build to a “butler results” json file. The user will also be told whether the build succeeded or not in the terminal. If it fails, we want the user to know exactly why it failed by printing the failing package and the specific task that failed such as a particular unit test.

As mentioned earlier we want functionality for a “publish task” which bundles up application packages into their distributable form then pushes them to antifactory. These tasks will only occur on a publish branch or a merge into the master branch.

It should be noted that due to the enormous number of expected tasks, Butler will take a long time to run. So, performance considerations are of high importance here. My planned solution is to make Butler multithreaded. Butler will determine how many available threads there are and will then spin off a child process on each thread. These child processes will be to execute the tasks that we constructed.

It is our intention for Butler to be used in conjunction with Jenkins pipeline. We plan to instruct users to set up a pipeline and a build server that will be connected to their Bitbucket repo. This build server will run on a virtual machine however, the instance of Butler will run on a container on that virtual machine. The reason for separating them is because we also plan to distribute a specific container image for Butler to be run on. This image will be set up so that Butler will have access to any tools it needs on other team’s build servers.

1.4. Client and Stakeholder Identification and Preferences

My client is Schweitzer Engineering Laboratories where my boss Matt Halladay will serve as my mentor and contact liaison. Stakeholders in this project will first be myself and my boss Matt. Other stakeholders will include my team who will be my first client when it is finished. Other future stakeholders would include any team that decided to adopt Butler as a CI tool in their development process. Other stakeholders that I will not be directly interacting with would be companies that supply dependencies such as Jenkins.

The clients for this project would need clean, working code that is easy to use. This will primarily be accomplished by providing extensive and easy to read documentation. This documentation would also include guides for usage such as editing the Butler config and setting up a build server. The needs of my boss and team who are expecting this application will be prioritized above all others.

II. Team Members - Bios and Project Roles

Brendan Crebs is a student at WSU pursuing a BS in computer science and is the sole worker on this project. Some of Brendan’s skills include Golang, Node, C/C++, Python, general algorithms, CI, and Docker. Brendan is an intern at SEL where he has had the opportunity to enhance these skills and work on this project. In general, Brendan is interested in backend application development. However, he has a special personal interest for certain niche areas such as chess engines and fractal generation software which have been the subject of previous projects. On this project, his responsibilities have been comprehensive, and he has made all repo commits as the sole contributor. His design, however, has been assisted with mentorship from the customer Matthew Halladay who is also his boss.

Project Requirements

III. Project Requirements introduction

This section is an overview and analysis of Butlers requirements. This will start with the functional requirements, which is a breakdown of what specific features must be implemented with Butler. Next will be the non-functional requirements which will be a set of descriptions on how Butler should function. Lastly, this section will include how Butler will be affected by system evolution. In other words, as technology and user needs change over time, how might Butler be affected? This is a major concern considering the nature of Butler as a supporting tool for other technologies.

System Requirements Specification

IV. Use Cases

V. 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 like 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 need 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[8] 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.

VI. 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.

VII. 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 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.

Solution Approach

VIII. Solution Approach introduction

The intention behind this section is to be an in-depth overview of the Butler's design. This will include a high-level discussion on Butler's architecture, a lower-level discussion on Butler's subsystems, and specification regarding Butler's data structures and UI. An additional purpose for this section is to provide a well thought out design to reference during development. This doubles as an appropriate section to reference to customers so they can better understand the product they are receiving.

Architecture Design III.1. Overview The intended design of Butler is for it to be the main step in a repos CI process. Butler itself will essentially be a series of steps that should execute independently from each other. For this reason, we will be using the Microservice Architecture Model [2]. For Butler to be a highly customizable application, the different parts must not be too interdependent on each other. For this we need an architecture model that supports the loosely coupled nature of Butler's various modules. For example, the creation of build, test, lint, fuzz, and publish tasks will all be separate microservices that can operate completely independent of each other. This will accommodate our design of wanting to allow the user to shut any of these task steps on or off depending on their needs. Another benefit of the modularity of the microservice model is that it will be easier and cleaner to test when modules are mostly independent of each other. Below is the subsystem decomposition that represents the overarching architecture of Butler.

IX. System Overview

The main functionality of Butler is to provide the various benefits of continuous integration to our customers. From a design standpoint we intend on having Butler function as a CLI application. Our design calls for a user to not have to interact with Butler during runtime. Rather, the users focus will be towards setting up configurations so that Butler will run the way they want to. Butler is designed this way because it is a form of automation. We want the user to be able to setup Butler, attach it to their build server, and not need to regularly interact with it other than to view the result of a build. We will have instructions so that a user can setup their remote repository with web hooks so that code being pushed automatically triggers a Butler build. When Butler is run, it will walk through the repository constructing a series of build, test, lint and publish tasks. Test tasks for example will contain a test command that Butler will execute adjacent to a specific workspace. To be more efficient Butler will determine how many threads are available on the host machine and spin a task child process off onto each one. This is due to performance concerns with the potential for hundreds of tasks in larger repositories.

The nature of tasks will vary heavily depending on how the user configured Butler. We intend for Butler to be a very generalized tool, meaning that it can fulfill its purpose in any repository if the user supplies the necessary information. An interesting problem with this approach is that we cannot assume what technologies the user will have in their repository, nor can we assume the way their repository is structured. Therefore, our methods must account for a wide range of possibilities and edge cases. A tool like this is difficult to make without foresight into the nature of the repository it will live in. To mitigate this concern our plan is to write well thought out methods and have the user supply information to fill any gaps. To do best establish the latter, our configuration structure will operate like that of a makefile with dynamic syntax. These considerations will be discussed in detail in the following sections.

X. Architecture Design

1. Overview

The intended design of Butler is for it to be the main step in a repos CI process. Butler itself will essentially be a series of steps that should execute independently from each other. For this reason, we will be using the Microservice Architecture Model[1]. For Butler to be a highly customizable application, the different parts must not be too interdependent on each other. For this we need an architecture model that supports the loosely coupled nature of Butler's various modules. For example, the creation of build, test, lint, fuzz, and publish tasks will all be separate microservices that can operate completely independent of each other. This will accommodate our design of wanting to allow the user to shut any of these task steps on or off depending on their needs. Another benefit of the modularity of the microservice model is that it will be easier and cleaner to test when modules are mostly independent of each other. Below is the subsystem decomposition that represents the overarching architecture of Butler.

2. Subsystem Decomposition

2.1. [Config handler]

2.1.1 Description:

The config handler subsection deals with the deeply important Butler configuration options. It parses config information and determines both what algorithms need to be used and what tasks need to be created in future steps.

2.1.2 Concepts and Algorithms Generated:

The config Handler defines 3 interfaces. One will be for a base Butler config object. This interface will be used only once, and it will contain all the relevant settings defined in the base butler config file. The next interface will be the Language interface. On object will be created using this for every language supplied in the Butler config. Based on the information provided in each language object, a language specific set of algorithms will be selected. Then the cli arguments passed with the Butler call will be applied.

2.1.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Parse config	Environment Handler, build tasks, publish tasks	The parse config service will parse the config and returns objects to represent the receiving information.
Parse cli	Environment Handler	Parse Cli parses the command line arguments and applies them to the base config object.

Services Required: N/A

2.2. [Environment handler]

2.2.1 Description:

The config handler subsection deals with the deeply important Butler configuration options. It parses config information and determines both what algorithms need to be used and what tasks need to be created in future steps.

2.2.2 Concepts and Algorithms Generated:

The config Handler defines 3 interfaces. One will be for a base Butler config object. This interface will be used only once, and it will contain all the relevant settings defined in the base butler config file. The next interface will be the Language interface. On object will be created using this for every language supplied in the Butler config. Based on the information provided in each language object, a language specific set of algorithms will be selected. Then the cli arguments passed with the Butler call will be applied.

2.2.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Walk the repository	Lint task, test task, build task, publish task	The gather file information service will give all task services the info necessary for executing task commands.
Determine dependencies	Lint task, test task, build task, publish task	Gathers language dependencies and version to determine if tasks should be created for a code package.

Services Required:

Service Name	Service Provided From
Parse config	Config Handler
Parse cli	Config Handler

2.3. [Lint tasks]

2.3.1 Description:

Lint tasks will be the first type of task created and also the easiest. The point of these tasks will be to execute a linting command for every code package for a given language. All lint tasks will be created here and then pushed into the task queue.

2.3.2 Concepts and Algorithms Generated:

For lint tasks we will take the file information obtained in the environment handler and separate the files into groups based on language. Irrelevant files such as txt files will be discarded at this step. Then we determine code packages that linting tasks must be created for. Lint tasks will then be created using the task interface. After this, the set of code packages will be passed to the next available task step if there are any.

2.3.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Clean files	test task, build task, publish task	Removes any irrelevant files from the file path array.
Create lint tasks	Task Executer	Builds lint task object for each task and pushes it to the task queue.

Services Required:

Service Name	Service Provided From
Walk the repository	Environment Handler
Determine dependencies	Environment Handler

2.4. [Test tasks]

2.4.1 Description:

Test tasks will be created in a similar fashion to lint tasks. There are a few differences though, the main being the addition of fuzz tasks which send thousands to millions of random inputs to code packages to probe for failures and flaws. This is useful for finding edge cases in code so we would provide it as an optional functionality to the test task service. Beyond that test tasks are determined in an identical fashion to lint tasks but are kept separated so the user can easily shut one or the other off without disrupting the remaining services.

2.4.2 Concepts and Algorithms Generated:

For lint tasks we will take the file information obtained in the environment handler and separate the files into groups based on language. Irrelevant files such as txt files will be discarded at this step. Then we determine code packages that linting tasks must be created for. Lint tasks will then be created using the task interface. After this, the set of code packages will be passed to the next available task step if there are any.

2.4.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Create Fuzz tasks	Task Executer	Builds fuzz task object for each task and pushes it to the task queue.
Create Test tasks	Task Executer	Builds test task object for each task and pushes it to the task queue.

Services Required:

Service Name	Service Provided From
Walk the repository	Environment Handler
Determine dependencies	Environment Handler

2.5. [Build tasks]

2.5.1 Description:

Build tasks are for executing commands to build code packages. The methods of doing this can vary greatly repo to repo so we plan on having much of the build task sections behavior controlled by the user via the config file.

2.5.2 Concepts and Algorithms Generated:

For build tasks, we will supply several methods that may or may not be used depending on how the user configures the config file. It will be encouraged though, that the build task method will be contained in a user defined script such as a bash script or dockerfile. Then the user will supply that information in the config and Butler will know how to simply execute the file to complete the build task.

2.5.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Create Build tasks	Task Executer	Creates build task object for each task and pushes it to the task queue.

Services Required:

Service Name	Service Provided From
Walk the repository	Environment Handler
Determine dependencies	Environment Handler

2.6. [Publish tasks]

2.6.1 Description:

Publish tasks will be for packaging then publishing application units of code in a repo. For this we would have the user point Butler to a registry such as one on Artifactory to push the package to.

2.6.2 Concepts and Algorithms Generated:

This is another instance of a problem that can only reasonably be solved by implementing a multitude of methods for each major language so that we can know how we are to package an application unit of code. We will also need the user to supply packaging configurations in the butler.yml.

2.6.3 Interface Description

Services Provided:

Service Name	Services Provided To	Description
Create Publish tasks	Task Executer	Creates publish task object for each task and pushes it to the task queue.

Services Required:

Service Name	Service Provided From
Walk the repository	Environment Handler
Determine dependencies	Environment Handler

2.7. [Task Executer]

2.7.1 Description:

The task executer will execute the entire queue of tasks. It will cease and exit with an error if any of these tasks fail.

2.7.2 Concepts and Algorithms Generated:

Since many of the tasks take long periods of time to execute, it is essential that the performance of this section is considered carefully. If we run each task sequentially, builds could potentially take hours in large repos. To solve that, this section will be multithreaded. First, we determine all available threads on a machine. Then, we assign a child process for each open thread that will each run a task from the task queue. When a task successfully exits, we assign the next task in queue to the newly opened thread. All the while the task results are being printed to the command line for the users viewing.

2.7.3 Interface Description

Services Provided: N/A
Services Required:

Service Name	Service Provided From
Create Lint tasks	Lint tasks
Create Test tasks	Test tasks
Create Fuzz tasks	Test tasks
Create Build tasks	Build tasks
Create Publish tasks	Publish tasks

XI. Data design

In terms of data structures, we will use numerous objects to represent configuration data and tasks. These objects are described in the system architecture above. Beyond that a queue data structure will also be used to hold the task objects. This will be a proper data structure to represent the problem of executing every task in order. Each time a task is completed and the thread it was using released, the queue will release the next task to occupy that thread.

XII. User Interface Design

We do not have a traditional interactive UI since it would be pointless for Butler purpose. However, the configuration file and Butler results could serve as the visual user interface. For the config, we want it to function like that of a makefile. We don't necessarily want every option to just be a key value pair like in other configs. This change would be appropriate due to the large number of options that we will be supplying to the user. For the Butler results, we want to print out the status of the task execution in real time as they are being executed. This gives Solution Approach 9 the user an indication of how long the build is taking, what tasks are taking the longest, and at what stage in the build they are in. If the build fails, we want to print out the exact reasons for the failure so that the user knows where to look to fix the problem. This portion alone could greatly improve code quality by exposing issues with the user team's code.

XIII. Test Plan Introduction

The main objectives in testing Butler would include assuring high code quality and assuring that Butler is flexible enough to work in any environment a customer may attempt to use it in. Besides code reviews, assurance of code quality will be accomplished through our unit testing and CI process. The approach to unit tests will be a requirement of 100% test coverage and all tests passing before Butler can be considered for release. This will help assure that every part of Butler is ran and correctly exits with expected outputs. After a working prototype of Butler has been produced, it will be important for a CI process to be set up at this point. This will allow the continuous testing of the code as more features are added to Butler, hopefully catching any bugs when they are introduced. Assuring that Butler is flexible will be partially accomplished with unit tests but will mostly be achieved through functional testing. Functional testing will see us using Butler in a multitude of real and mock repositories to see how well it responds. This will be a highly important step before release.

To accomplish the testing goals, we will utilize several testing technologies. For unit testing we will use multiple third-party testing dependencies for Golang to assist testing. These will be "delve"[3] for debugging, "golangci-lint"[4] for linting, "gotests"[5] for assisting with table driven tests, and "gomock"[6] for mocking. In addition to this, the CI tool Jenkins [1] will be used for our CI process and will be run on a dedicated build server specifically for Butler. At first, we will manually include the testing locations and commands into a Jenkins pipeline. When a working version of Butler has been produced, these manual steps will be removed and Butler itself will be used as our CI tool and included as a step in the Jenkins pipeline. Major testing deliverables for Butler will first include unit and integrations tests that achieve 100% test coverage over Butler. These tests will be included for the first time with the initial Butler prototype. From there on out, each version of Butler will require that all added features be tested with 100% test coverage before merges to the main branch can occur and a new version released. The next major deliverable will be a markdown document containing the results of functional testing. This document will be pre-prepared with a table of functional tests to run and what the results were. Much like the unit and integration tests, a 100% pass rate will be required with the functional tests for release to be an option. The next deliverable will be a build server and CI pipeline. For

the pipeline, Jenkins will be used as discussed in the previous paragraph. The build server will consist of a container run on a virtual machine. The container on the VM will be the same container image we are using for development. The reason for this is to assure consistency between the two environments.

The intention behind this section is to outline the plans and processes that will be used in the testing of Butler. This section does not intent to document the results of the tests, nor will it go into exhaustive detail on specific tests. Rather, this section is meant to provide a reasonable strategy for testing Butler properly.

XIV. Testing Strategy

The approach to testing will have CI and functional testing as the main focus. The latest stable version of Butler will be used in the CI pipeline to test itself. For a branch to be merged to main, and for a new version to be released, Butler must pass all the conditions for a successful build. These conditions will include, 100% off unit/integration tests passing, 100% code coverage, no linting errors, and a successful build task for Butler before publishing. If any of these conditions fail, then the entire build will fail, and we will be prohibited from merging into main or publishing a new version. For functional testing, all tests will be predetermined, and all tests must pass before Butler can be considered ready for release. These two processes will remain separate and are listed below:

CI Testing Process:

1. **New code is written:** The first step in the process will be writing code instead of writing tests.
2. **Unit tests written for new code:** After new code is produced there should also be new unit tests or updated unit tests to achieve test coverage over the new features. It will be good practice for these tests to include a large set of inputs and edge cases to cover a wide variety of possibilities. We want to avoid merely establishing coverage will bad tests.
3. **Check test results:** The unit tests will be run locally where they should be fixed if they fail or do not produce 100% coverage.
4. **Push to remote and trigger a build:** After a developer pushes code a CI build will be automatically triggered which will lint the code, run all tests including tests for code separate from the new feature, and perform a build task by running a dockerfile which we will supply. If anything fails on the build server then the developer is to return to step 3 to investigate the issue. Here they will likely need to use the debugger to establish root cause.
5. **Perform Functional testing:** Perform functional tests and document the results. See “Functional Testing Process” for a detailed description of this step.
6. **Merge branch:** After a successful build, functional tests and approval from the reviewers, the branch can be merged into main. When a merge to main happens, this will trigger a special Butler build which will test everything in the repository, lint

everything, build everything, and then create a special publish task to publish a new version of Butler. The publish task will be optional and will need to be manually enabled from the Butler config. Success of the merge will hinge on the success of this Butler build. If the build fails, then the branch will not be merged, and the developer will need to look to the Butler results to determine what needs to be fixed before another merge attempt can be made. Success of this step will satisfy the CI testing requirements.

Functional Testing Process:

1. **Get instance of working Butler version:** For functional testing, there first must have been a successful butler build. Functional testing will occur after step 4 in the prior process and before a merge attempt to main is made.
2. **Create functional testing document:** For functional testing, the developer must first create a markdown document that contains a table populated with functional tests in the first column. In the other columns there will be space for a description of the test, observations during the test, any inputs used, and outputs generated, and whether the functional test passed or failed.
3. **Perform functional tests:** The developer must run the functional tests to the specification of the test description. The user should attempt to also do abnormal actions to break Butler so obscure issues can be revealed.
4. **Document results:** The developer should document the results of the functional tests in the empty fields of the table. If any tests fail, then the new version cannot be released and must undergo revision. After cause is determined and a fix is produced, start back at step 1 and reperform functional tests for the new instance of Butler.

XV. Test Plans

1. Unit Testing

For unit testing we will establish that individual functions will our unit to test. A unit test will be generated for every function, including helper functions. For each file, there will be an adjacent test file which will contain the unit tests for functions in that file. Our goal with unit tests is to provide the functions will a large and diverse set of inputs to detect any edge cases. The unit tests are where 100% test coverage will mostly be established. Here is also where we will check to make sure that Butler errors and returns when it is supposed to.

2. Integration Testing

The integration tests will work well with our microservice model approach. With these tests we will check to see how one component flows into the other. For example, we will test the lint component and build component in the same test to make sure nothing failed when they were included together. Since naturally, Butler flows from one component to the next, integration tests will be very helpful and not difficult to implement.

3. System Testing

3.1. Functional testing:

The functional tests will be performed extensively in a multitude of different repositories with diverse environments to assess how Butler performs. This process I believe will help make Butler much stronger and more flexible. This effort is to expand the usefulness of Butler and reduce the chances that it will fail in an environment which we didn't expect. The steps that will be taken during function testing are listed under the "Functional Testing Process" in Testing Strategy above.

3.2. Performance testing

One of the performance testing practices we will adopt will be fuzzing. Fuzz testing is the practice of feeding an application enormous amount of randomly generated inputs to break the application. Other measures will be generation unreasonably large repositories with and unreasonable number of tasks which we will continue to increase until Butler breaks. During this time, we will look for how Butler's performance suffers and how much memory it is using under extreme circumstances.

3.3. User Acceptance testing

For user acceptance testing, prototype versions of Butler will be distributed to the primary customer which is my team and my manager so they can assess if Butler is adequately meeting their requirements. This practice will occur starting at the first prototype release all the way until the project has entered a maintenance phase.

XVI. Environmental requirements

The testing environment will require installation of Golang, all third-party testing tools, and a go.mod file containing the third-party dependencies used in the code. The testing environment will also require a build server and a computer to host said build server. On the build server and locally, a docker container will be used. To use the containers docker and the devcontainer cli will be required. For the CI pipeline Jenkins will also be used. It is expected all tests are run in a Linux environment. This can be WSL, a virtual machine, a container, or on a machine booted with any major distribution of Linux.

XVII. Alpha Prototype Demonstration

In terms of implementation, there is now a functioning prototype to work off going forwards. In this section I will discuss the status of the subsystems defined in section X, Architecture overview.

For the Butler configuration requirements set, only the config processing requirement has been fully implemented. This requirement is the most significant part of the subsystem and the only one that is needed to develop the remainder of the requirements in this project. The other two Butler configuration requirements are the dynamic config structure and nested config files requirements. These are not needed to finish the remainder of development and are mostly for

preventing the bloating of internal interfaces and easing the user experience. However, there are plans to implement both requirements by the sprint 3 report.

1. [Config Handler]

a. Functions and Interfaces Implemented

For the config handler subsystem, a first working implementation was created by the first sprint to accommodate the remainder of development. This system works by determining the Butlers configuration based on the Butler config and any command line arguments. This is currently working but there are several goals that remain unimplemented for this subsystem. One such goal is a nested structure to have Butler's behavior differ at different levels of the repo. However, the most important piece left to implement for this subsystem will be a special syntax to enhance the user experience and reduce the massive bloating of internal interfaces due to the large amount of option fields which have been added to the config. As of now the Butler config is a series of key value pairs and the interfaces to accept the config information reflects this. Below is a picture of a populated ButlerConfig object being printed to the command line to show this subsystem is functioning.

[INCLUDE PICTURE]

b. Preliminary Tests

Up to this point, most testing on the Butler Configuration subsystem has been informal functional testing to validate outputs so that the other subsystems can function with expected values. This is also where the first unit tests have been written. These unit tests, however, do not establish 100% coverage over the subsystem and will not be finished until all requirements for this subsystem have been implemented.

2. [Environment Handler]

a. Functions and Interfaces Implemented

The Environment Handler subsystem immediately follows the work of the Config Handler. This section walks through the repository and determines several important details about the environment Butler is being ran in. Ultimately this section will determine what sections of the repository need to be built, if a full build is needed, and if Butler is being ran on a publish branch. As of now the Environment Handers successfully walks the repo and gathers files and directories down the paths allows by the user and does not walk down the paths/patterns blocked by the user. It then successfully conducts an environmental analysis including diffing the current repo to the remote main branch to determine what needs to be built. After this the Environment Handler successfully constructs workspaces for units of code. Later on, tasks will be constructed for each of these workspace objects. As of now the Environment Handler is functioning properly and is one of the most complete subsystems in Butler. Below is a picture of all paths and workspaces gathered printed to the terminal to show that the Environment Handler is functioning in this regard.

[INCLUDE PICTURE]

b. Preliminary Tests

Most of the testing for the environment handler has been informal functional testing. During this testing, some issues were found that may indicate a design change. This was due to the burden of having the user pipe information to use from their own method file. We found that sometimes the data would simply be too large to pipe using more trivial methods. We did not want the user to been required to implement anything too sophisticated. The result of these finding is to remove the option for user defined workspace gathering and automate the process instead. We determined that automating this process would be reasonable even with the potential language variety in different repositories. Beyond the functional tests and its finding, some unit tests have been written for the Environment Handler. This subsystem will likely be the first to get 100%-unit test coverage.

3. [Lint Tasks]

a. Functions and Interfaces Implemented

A first version of the Lint Task subsystem has been successfully implemented. This subsystem initially functioned by piping the set of workspaces acquired by the Environment Handler to a user defined file and then expected the user to pipe back some information that would be used in the creation of lint tasks. After a customer demo, concerns were raised over the potential for pipes getting clogged due to too much data being passed at once. As a result, the new design is to ask the user for method file path and a command and then construct tasks based on that information. These task objects are pushed to a queue before progressing to test tasks. When the tasks are ready, the supplied command will be executed at the location of each workspace which was previously determined for the given language. We would then request that the user method return logs, and a status code. Test user methods have been created to accomplish this task. The same test user methods will also function as our built-in lint methods for when the user does not want to supply their own. The image below shows the constructed lint tasks printed from the task Queue to the terminal.

[INCLUDE PICTURE]

b. Preliminary Tests

All tests for the Lint Tasks subsystem were informal functional tests to validate functionality to continue development. During this testing the issues with using pipes for passing data between user methods was once again determined to be a poor solution. In response the design was changed as specified above in 3a. Piping is still being used but only for the logs and status code which are being written to stderr and stdout respectively.

4. [Test Tasks]

a. Functions and Interfaces Implemented

The Test Tasks subsystem functions in a similar fashion to the lint tasks subsystem and is currently just as functional. Test user methods have been defined for this subsystem as shown below. There is however a portion of this subsystem which has not been implemented which is the fuzz task creation. This portion of the Test Task subsystem is considered a future feature, so it is remaining a low priority for now. The test task objects which are produced by this subsystem are pushed to the same task queue containing the lint tasks and they will be executed after the lint tasks in order.

[INCLUDE PICTURE]

b. Preliminary Tests

The Test Task subsystem was testing in a similar fashion to the lint task subsystem. Unit tests have not been written but informal functional testing has been conducted to validate expected outputs. Testing the Lint Task subsystem has influenced this the testing of the Test Task subsystem. We have come to the same conclusion that this subsystem must change its design in the same way the Lint Task subsystem's design was altered due to pipes being clogged with too much data.

5. [Build Tasks]

a. Functions and Interfaces Implemented

For build tasks a system is in place to accept a user defined method for build tasks within there repo. A test file has been created to simply show that this implementation works. However, an actual build task method has yet to be supplied either in the built-in section of in the user defined method section. This will be a future goal. In the meantime, the system in place constructs the build tasks and then pushes them to the queue after lint and test tasks. Below is a picture showing a functional test result which shows that build tasks will theoretically work.

[INCLUDE PICTURE]

b. Preliminary Tests

For testing, only minor functional tests have been conducted to determine if this section works or not. In testing it has been determined that this section theoretically works. However, since actual build task methods have not been added, we will need to wait before this can be verified.

6. [Publish Tasks]

a. Functions and Interfaces Implemented

For Publish tasks the situation is like build tasks in terms of there only being a theoretically correct implementation. Going forward we will need to assure that there are sufficient test user methods as well as built-in methods for Publish Tasks. Below is a picture showing that publish tasks run when Butler detects it is on a publish branch.

[INCLUDE PICTURE]

b. Preliminary Tests

Tests for publish tasks have been minor functional tests which showed us that publish tasks will only be created when Butler is being run on a publish branch. Functional tests like that of the

Built Task functional tests have also been conducted in order to determine that publish tasks will theoretically work as expected when executed with a user method.

7. [Task Executer]

a. Functions and Interfaces Implemented

The task executer is the multithreaded task execution subsystem which accepts the queue we created in previous steps and executes them in order. This algorithm starts by checking for available threads and then spins off a child process to execute each task. This subsystem now has a working prototype and produces real time logs to the terminal about the status of the task execution. With the completion of this subsystem Butler can now be considered a working prototype. Below is a terminal output showing the intended usage of Butler. The picture shows a task failing which causes logs to be printed and Butler to exit.

[INCLUDE PICTURE]

b. Preliminary Tests

Tests for this section, like the task creation sections, is entirely functional at this current time. These functional tasks have been to verify that Butler is functioning properly and that tasks are passing when they are expected to pass, and failing when they are expected to fail. These functional tests have also validated that the actual terminal out puts are the expected value.

XVIII. Alpha Prototype Demonstration

The prototype demonstration to my mentor was a two-part presentation. The first part was a display of Butlers functionality to show that it does indeed work. To do this I pre-prepared print statements throughout Butler to shows that the inputs and outputs of different subsystems were expected. Finally, I showed the task execution portion was functioning properly and tasks collected from the repository were succeeding or failing as expected. Next, we looked together at the design and implementation of the prototype. During this part my mentor brought to my attention the parts of my design he thought could be improved. Specifically in relation to the support of user defined methods. He thought that piping the code workspaces to the user for them to use was unnecessary and was also error prone due to the potential that the data would be too large to pipe without making the implementation needlessly complex. He proposed that instead of the user returning data for use in Butler, they only return logs and a status code, reducing the burden on the pipe and not causing the output to scale with the size of the repository. After this was explained to me, I realized that piping all the repository data could not be used for the user methods except for gathering third party external dependencies. He agreed with my proposal that we keep piping in for that section. My plan now to find the most robust way of piping information between files so that the external dependencies will not exceed the character limit for piping via stdout which is the current implementation. My mentor did not have many questions for me other than clarification questions as I explained the implementation.

XIX. Future Work

1. **Fuzz tasks:** This future feature will add fuzzing to Butler as an optional type of task. This would potentially expose users to vulnerabilities found in their code and/or failing edge cases that their unit tests didn't to cover.
2. **Enhance build/publish task construction:** This future feature will be enhancing the current build and publish task construction so that butler can work out of the box for users without them needing to define methods for this on their own.
3. **Built-in external dependency analysis:** This feature would add a language specific external dependency parsing method for every major language. This is again so Butler will work right out of the box without the user needing to supply a parsing method in the vast majority of cases.
4. **.butlerignore file:** A special .butlerignore file feature could be added so that a user could supply a large set of paths to ignore/include so they wouldn't need to maintain it in their base Butler Config. This will prevent the user from needing to bloat that file which will make it difficult to work in.
5. **Nested Butler config files:** This feature would allow a user to change the nature of butler at different point in the repo. The idea is that they would supply a file within a workspace and a new ButlerConfig object will be created for that specific workspace. They workspace object will contain a field which will either point to the base config or a special unique one created for that workspace. This will make Butler more flexible with different repositories.
6. **Packaging Butler:** This task is for going through the process of packaging Butler as a distributable, executable cli application. Modifications will likely be necessary to get it into the state where this is possible.

XX. Glossary

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

Devcontainer: A reference to a development container created using the devcontainer cli. See reference [7] for more information.

Dockerfile: A file that defines configurations for a docker image.

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

Golang: A widely used object-oriented programing language.

Jenkins: A CI tool used for establishing an automated CI pipeline. See reference [1] for more information.

Linting: The automated checking of code for potential bugs, stylistic issues, and poor coding standards.

Makefile: configuration file for Make, a dynamic build automation tool for C.

WSL: This is an acronym for “Windows Subsystem for Linux” which is a Windows feature that allows a user to run a Linux distribution in Windows without needing a virtual machine or container.

XXI. References

[1] Jenkins, www.jenkins.io/. Available: https://www.jenkins.io/ Accessed 26 Oct. 2023.

[2] Martinekuan. “Microservice Architecture Style - Azure Architecture Center.” Azure Architecture Center | Microsoft Learn, Azure, 2023, learn.microsoft.com/enus/azure/architecture/guide/architecture-styles/microservices.

[3] Go-Delve. “Go-Delve/Delve: Delve Is a Debugger for the Go Programming Language.” GitHub, github.com/go-delve/delve. Available: https://github.com/go-delve/delve Accessed 26 Oct. 2023.

[4] Golangci. “Golangci/Golangci-Lint: Fast Linters Runner for Go.” GitHub, github.com/golangci/golangci-lint. Available: https://github.com/golangci/golangci-lint Accessed 26 Oct. 2023.

[5] “Gotests.” Gotests Package - Github.Com/Cweill/Gotests - Go Packages, pkg.go.dev/github.com/cweill/gotests. Available: https://pkg.go.dev/github.com/cweill/gotests Accessed 26 Oct. 2023.

[6] Golang. “Golang/Mock: Gomock Is a Mocking Framework for the Go Programming Language.” GitHub, github.com/golang/mock. Available: https://github.com/golang/mock Accessed 26 Oct. 2023.

[7] Devcontainers, “Introduction to Dev Containers.” *GitHub Docs*, docs.github.com/en/codespaces/setting-up-your-project-for-codespaces/adding-a-dev-container-configuration/introduction-to-dev-containers. Accessed 12 Nov. 2023.

[8] Artifactory, “Artifactory - Universal Artifact Management.” *JFrog*, 20 Oct. 2023, jfrog.com/artifactory/.