# Butler

*Project Solution Approach*

Schweitzer Engineering Laboratories


**Horizon Group**

Brendan Crebs

10/2/23

# TABLE OF CONTENTS

# I. Introduction

The intention behind this document 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 document is to provide a well though out design to reference back to during development. This doubles as an appropriate document to provide to customers so they can better understand the product they are receiving.

To give a brief description of the project, Butler is a multithreaded, Continuous Integration (CI) tool that constructs and executes build, test, lint and publish tasks. Butler is intended to be a CLI tool that is intended to be automatically run on build servers when code is pushed, or a branch is merged. Butler is a generalized tool that allows for a high degree of configuration to fit the needs of the user's repository.

# II. 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 they 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 can not 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 similar to that of a makefile with dynamic syntax. These considerations will be discussed in detail in the following sections.

# III. 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[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.

## III.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 of 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 environment handler will take the information obtained from the config handler to prepare Butler for the environment it is working in. Using this information, it will walk through the repository to obtain an array of file paths along with some other relevant information.

#### 2.2.2 Concepts and Algorithms Generated:

First it will run several functions to gather more information it doesn't have such as the current branch and if the Butler config has been modified since last commit. Using this information, it will determine if a full build is required. If so, it will override what the user has supplied in the config as a safety measure. Then it will start walking though the repository to gather file paths while not touching any path or pattern explicitly blocked by the user in the config. It will also not go down a path until at least the root of that path has been accepted by the user in the config. After the file paths have been determined, we will have to gather a list of all of the dependencies for each language in the repository. There is no general way to do this due to the variety in dependency information storing methods language to language. Therefor we will need to have hard coded methods for every major programming language. We gather these dependencies to check if any have changed in version. If so, we will require a build for any code that uses that dependency.

#### 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. |
| Build 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 in order 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 fuzz tasks in particular an interesting problem arises. Each language does not have a clearcut generalized way of performing fuzzing. Therefore, this may be another section where we have to provide a large number of language specific functions for every major programming language. This is my current solution to this issue.

### 2.4.3 Interface Description:

Services Provided:

| Service Name | Services Provided To | Description |
|---|---|---|
| Build Fuzz tasks | Task Executer | Builds fuzz task object for each task and pushes it to the task queue. |
| Build 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 a number of 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 in order 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 |
| Parse Config | Config 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 |
| Parse Config | Config Handler |

### 2.6. [Task Executer]

#### 2.6.1 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.

#### 2.6.2 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.

#### 2.6.3 Interface Description:

Services Provided: N/A

Services Required:

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

## IV. 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.

## V. 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 similar to 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

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.

## VI. 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.

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

Dockerfile: Docker script containing commands that construct a Docker image.

## VII. References

[1]Martinekuan. "Microservice Architecture Style - Azure Architecture Center." *Azure Architecture Center | Microsoft Learn*, Azure, 2023, learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.