CARDIFF UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND INFORMATICS

MSC ADVANCED COMPUTER SCIENCE

60 CREDITS

Command Line Interface to Generate CI/CD Configurations for Projects

Author
Miles BUDDEN

Supervisor
Carl JONES

November 3, 2021

Abstract

CI/CD (Continuous Integration/Continuous Delivery) is an important aspect of modern software development. A limiting factor when it comes to implementation of CI/CD workflows is the configuration overhead. Often, very similar configurations are used for multiple projects, thus duplicating work. Notwithstanding the need for the base knowledge required for CI/CD deployments, this project proposes a solution to this problem. This project provides tooling to automate the integration of source control, build systems, storage systems, as well as deployment targets and databases.

Contents

List of Figures

1 Introduction

1.1 Aims

The main aim of this project is to produce tooling that will to some degree, reduce the amount of work required to set up CI/CD workflows such as those that can be run by GitLab pipelines. In doing this, the benefits will be two-fold: Those with experience of CI/CD will be able to deploy a CI/CD pipeline faster and with more ease; Those with less DevOps (software development and IT operations) experience will be able to use CI/CD tools where they might have otherwise not been able to or severely struggled to.

In order to realise this overarching aim, I will need to aim towards several other lesser goals. An important aspect of this project is to integrate with several platforms in order to provide a choice for the developer. This would help differentiate this tool from other tools as for most platforms, there are user-friendly scripts and tools to help with that platform in particular. We will look further into this in Section 3.2.

As well as a choice of platform, it is important that the tool should abstract away from the process as much as possible. This is to help with the initial aim to reduce work needed by the developer. Although it can be detrimental to user configuration and choice at times to abstract to a too greater level, at that point the tool proposed by this project would not be the correct tool to use.

1.2 Audience

The audience for this tool is a broad one. Depending on past experience with CI/CD tools, this tool will provide different helpful factors. Should the user be someone with extensive CI/CD experience, then the tool will assist them in boilerplating their CI/CD pipeline. Although their knowledge could be extensive, the use of the tool will cut down on the amount of time required for them to configure the CI/CD pipeline.

On the opposite end of the spectrum, you have the user who has little to no experience with CI/CD. This tool consequently allows them to implement a basic CI/CD pipeline with sane defaults which will allow them, in turn, to harness the power of CI/CD without the need to learn the configuration steps that are required.

1.3 Scope

The scope of this project must be clear to avoid infringing on the functionality of existing tools (See section 3.2). There are tools that exist to configure configuration files, as well as tools to interact directly with the APIs provided by the target platforms (see Section 5.1). In between these tools, however, exists an area with little existing tooling. This area is the automatic configuration and deployment of CI/CD pipelines for many combinations of different platforms.

This begins with the generation of the Dockerfile. Although not a simple process, a template Dockerfile for the project can be generated which can then be updated by the user in order to accurately match the specific architecture of the project. Once this Dockerfile is generated, the code will be uploaded to a code store. Once uploaded, a build process can begin which will generate a Docker image from the Dockerfile. This image can then be stored in a registry. Once accessible from a registry, the deployment platform of choice can then pull the image and deploy it. As well as an image, the deployment platform should also be able to deploy a database and return the credentials of the said database so that it can be connected to the project.

A clear end needs to be defined at this point as any more configuration beyond this point is infringing on the functionality of the platform-specific tools. Although there is no legal or functional harm in implementing functionality found in these tools, there is the issue of both time and user experience. As the project has a final deadline, time needs to be spent on providing new functionality to the end-user instead of re-implementing existing tools. As well as the time aspect, more functionality adds more complexity to the tool which could reduce accessibility and usability for less experienced users. This is an issue as one of the audiences specified in Section 1.2 is the inexperienced group, whom this additional functionality would impact.

1.4 Assumptions

The breadth of the audience of this project means that there are few assumptions that need to be made. The largest assumption is that the end-user wishes to use one of the platforms developed for during this project. We can overcome this assumption slightly by implementing a modular approach to the project such that in a future use case where the platform of choice is not supported, the platform can be added to the project with relative ease.

To help with the packaging and distribution of the project, we must make the assumption that the user has, or has access to, the dependencies required by the project. Although it could be possible in future for the project to be packaged along with its dependencies, due to the relatively short time-frame for the project, combined with the high rate of change of the project, it is infeasible to repackage the project after each build without a relatively large overhead that could be spent delivering greater functionality to the tool.

In order for the tool to successfully store, build, distribute, and deploy a project, I assume that the user has access to the platforms that support those aforementioned functionalities. Without these platforms, the project will not function as it is considerably beyond the scope to provide those platforms to support those actions.

This list of assumptions is not an exhaustive list as many of the assumptions made in this project are such common assumptions that they are not worth examining, for example, does the user have access to a computer.

1.5 Summary of Important Outcomes

Due to the exploratory nature of this project, defining important outcomes is difficult beyond a point. A minimum viable product (MVP), in the case of this project, would take the form of a tool that allows the user to automatically configure deployment of their code to an endpoint. Within this outcome, there are many avenues to explore, and from a planning point of view, it is difficult to determine which of these are important. The overarching idea will be a system that takes a set of inputs and transforms them into a more useful output. The input, in this case, is the code, and the output is a deployment of the code.

Some examples of these avenues are automatic deployment triggers, intermediary storage of built applications, as well as the automatic configuration of databases to accompany the application. Although potentially vague, I would determine an important outcome to be some combination of these features to produce a tool that provides real functionality and improvement of the development process to all of its target audience. This improvement would mostly come from a change of the application just running on the developer's computer to an application that can be run on the service that the developer chooses. Although not a precise and measurable outcome, it will hopefully become obvious during and after the development process, whether this tool is of any significant value to an end-user.

2 Planning

2.1 Issues Surrounding the Nature of the Project

Although appearing as a simple, linear project at first glance, the nature of the project is more complex than that. Much of this project consists of research sections that were difficult to predict the length of. This is due to the development methods used in this project. In order to reduce work in progress (WIP), I opted for short phases each of a similar format. This format consisted of a research phase, followed by an implementation phase. Should the project have consisted entirely of development phases, based on prior knowledge and research, then the time required would have been easier to calculate.

This approach, however, helped to reduce WIP, which in turn, reduced the risk of wasted effort should the deadline arrive and I have much invested in an unfinished WIP.

2.2 Methodology

My methodology for this project was developed in an organic way that loosely followed agile principles. I initially planned for three phases, following three different themes. The first sprint focused on research and the generation of ideas. This research included the platforms that I wished to target as well as ways to structure the project. The second sprint was the main development sprint which I aimed to have built the MVP by. The final sprint was the additional feature sprint. This was very open-ended in what I would be doing for its duration as it depended entirely on how much I had achieved in sprint two, as well as what I had discovered during sprint one.

Although I roughly stuck to the three sprint format for the duration of the project, it proved flawed for several reasons. The main issue that I encountered was information retention. Much of the research that I carried out was about the underlying architecture of each of the target platforms. This included the data flow as well as the specific quirks of each platform. Expressing this information in a way that I could learn from in a more concise manner than from the platforms documentation was challenging and was adding significant overhead. In order to overcome this issue, I adapted to the aforementioned short, multi-discipline sprint method. This meant that when I had finished researching a new platform, I could implement it straight away without having to make extensive notes. This allowed me to implement a new platform most weeks as well as minimising WIP by researching topics I would not have time to implement later on.

The second major advantage to this method was that I lined up well with my weekly meetings. This meant that after each of these minor sprints, I could look at the progress I had made in order to assess whether any further work was needed as well as to determine the direction the project would take for the next sprint. This allowed for more rapid changes in direction as opposed to three, much longer sprints.

2.2.1 Lean

Although the methodology developed organically, both from prior knowledge and also from direct experiences from the project, the methodology closely represented the lean methodology [1]. This methodology centres around the idea of minimisation of WIP and hypothesis-driven development. Although it is hard to apply this methodology at a macro scale to this project (due to the nature of a fixed dissertation title), it is very applicable at the micro-scale. The short sprints each tested a hypothesis surrounding a good new feature to implement. For the majority of these sprints, the hypothesis was sound. This meant that by the end of the sprint, no work was lost as there was deliberately no WIP by the end of the sprint post the MVP implementation phase. In the event that I still had WIP at the end of the sprint, I could evaluate whether I had just misjudged the scope of the feature I was implementing or whether the hypothesis that it was a good feature to implement was bad. This is where my final methodology finds its similarities to lean. With the lean methodology, one of these failed sprints is equivalent to testing a startup idea that was unsuccessful, as can be seen in Figure 1. Although there is some lost work, the lost work is minimised by the minimisation of WIP and the short, *lean*, evaluation period.

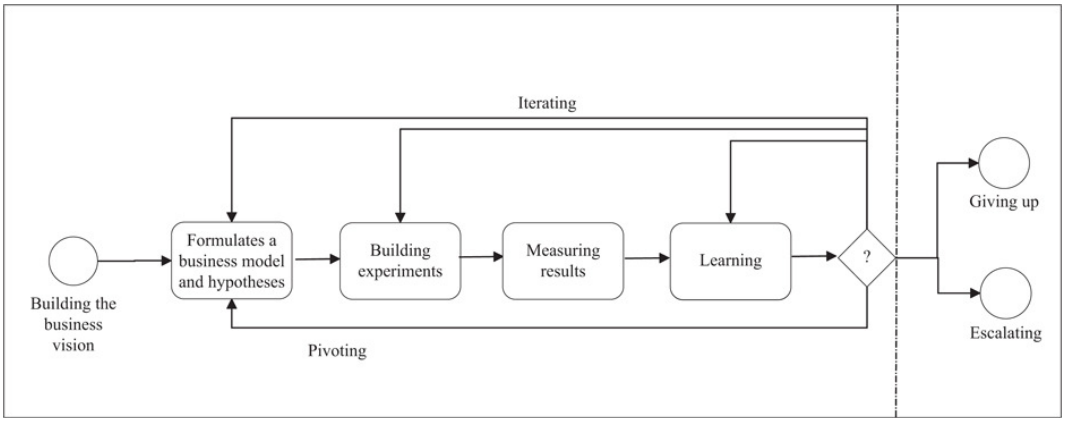

Figure 1: The lean startup cycle [2].

2.2.2 Kanban Board

In order to visualise and implement the *minimisation of WIP*, I decided to use a Kanban-style board to manage the smaller aspects of the project. By using three columns (to-do, doing, and done), I was able to clearly manage the number of cards in the "doing" column. This allowed me to gain efficiencies both in the lean sense by minimising WIP, but also by achieving this minimisation, reduced overhead associated with context switching. This project differed from Kanban as there was no formal backlog to pick from. Instead, tasks were added directly to the *to-do* column. Due to the short discovery-implementation cycle, there were few tasks to pick form at any one time and therefore the board functioned more as a rich to-do list as opposed to a Kanban board.

3 Background

3.1 Why Use CI/CD

The advantages to using CI/CD for a project are multifaceted. CI/CD automates several aspects of the development that would otherwise be slow, error-prone tasks. This automation process, once implemented can improve the build/release time of a project by 330× – 1110× [3].

As well as the build/release cycle, CI/CD can be applied to ticketing and issue systems such as that implemented by GitHub. CI/CD implementations to automatically test pull requests attached to issues have been shown to significantly increase productivity without a decrease in quality [4]. This is significant as it allows core developers to spend less time testing and checking pull requests and more time writing code, thus improving project output.

Even teams that already have good metrics benefit greatly from CI/CD, especially when it comes to incident recovery. Well performing teams still recovered 2.604× faster from incidents as well as having a 7× lower failure rate to begin with [5].

Within an enterprise context, the main barriers to CI/CD implementation are training costs, technology costs, downtime when upgrading, consulting services, as well as other expenses [6]. This tool helps reduce some of these costs by lowering the barrier of entry, thus reducing possible costs. The addition of DevOps to an enterprise context results in some significant benefits which include but are not limited to reduction in duplicated work, reduction in staff needed, added value from reinvestment in new features, time recovery, and frequency of experiments.

As well as the benefits of CI/CD in industry, there are benefits to introducing students to this important concept in education. This tool aims to lighten the learning curve of CI/CD and therefore provide a gentle start with little prior knowledge required. This is a good example of spiral learning [7]. If students are exposed to the concept of CI/CD early on without requiring much background information, it makes fully learning the concept and filling in any gaps in their understanding further on much easier and with a greater success rate.

3.2 Existing Solutions

In this section, I shall highlight some of the existing solutions to areas that this tool covers. In order to count as existing solutions, I am only looking at tools that are predominantly CLI, as well as simplifying or wrapping, to some extent, the underlying API. This look into existing tools is also non-exhaustive as there are so many tools that encompass some of the functionality of this project that it would be infeasible to cover them all.

3.2.1 GitLab

Arguably, git itself could be included as a CLI tool to access GitLab. However, the tool this project seeks to implement interacts with the CI/CD aspects of GitLab as well as the git aspects of it. Therefore, git would be unable to fulfil all of the required functionality of the tool. There does exist a

CLI wrapper around the GitLab API [8] that provides complete access to all of the API endpoints that GitLab exposes. Although powerful, this means that the user is faced with many more features than they are ever likely to use, thus making the tool harder to use.

3.2.2 OpenShift

There is an in-house CLI for OpenShift that exposes all of its functionality [9]. Similarly to the GitLab CLI, this tool exposes more functionality and configuration to the use than that which is needed. Therefore, additional functionality complicates its usage and can therefore be improved by this project. An example of this additional complexity is the deployment of a Docker image. For example, in order to deploy an image from GitLab, the OpenShift CLI requires 14 commands [10].

For more basic control over OpsnShift's functionality, there is the Kubernetes (k8s) CLI: `kubectl` [11]. The OpenShift CLI is built on top of this, extending its functionality to include the additional features that OpenShift provides. Therefore, the same issues prevail with this tool.

3.2.3 Azure

Similarly to OpenShift, Azure provides an in-house CLI solution: `az` [12]. This tool provides rich functionality with all of Azure's products. Although useful for an expert, this functionality comes at a cost, both in learning difficulty, as well as file size[1]. This CLI is built using the Python bindings for the API which provide an alternate interface into Azure.

3.2.4 Project Type Detection

Currently, there are few projects that a built with the sole purpose of detecting project type. The best, currently maintained project is Netlify's framework-info [13]. This detects which JavaScript framework is being used in a project. The two issues with using this project are, firstly, that it is written in JavaScript. This would make it difficult to integrate into the current code-base. Secondly, it is only able to detect JavaScript libraries. This means that in order for it to be useful, it would need to be modified to also detect frameworks in other languages. These two issues are major enough for the lowest friction path to be reimplementation. This is not a huge task, however, as the core architecture of framework-detector can be used which will remove the need for detailed planning and trial and error.

---

[1] The Azure CLI's total installed size is over 1GB.

3.3 Problem Areas

The project can be broken down into a series of problems. Although these problems may not line up directly with each phase, they can be used as a measure of progress. Due to the nature of the project, more problems arose during the development phase of the project.

The first problem area to address was the architecture of the project. This included ways to make the project modular, as well as specifying some kind of consistent design between said modules. This had to be the first problem area addressed as it blocked the development of the other sections until completed.

Once the core architecture was addressed, there was potential for all of the other problem areas to be addressed with relative asynchronicity. As mentioned in Section 2.2, a truly asynchronous approach to the following problems was unfavourable. Each problem area from this point onward revolved around a provider from the desired CI/CD tool-chain. For each provider, a problem area arose around each of the services that I wished to wrap. Each of these problem areas lasted roughly one of the one-week phases ultimately decided upon. It was important that some aspects of the project were implemented in parallel, however. For example, improvements to the core architecture, as well as some documentation, were made over the development process. It was important to balance which tasks were done in parallel due to the aforementioned faults with parallel tasks. Building each section in isolation would have made an implementation of a full end-to-end solution difficult.

4 Specification and Design

4.1 Requirements

In order to formally specify the requirements for this project, I shall break them down into functional and non-functional requirements. Although there was an approximate idea of requirements at the start of the project, I shall also be listing new requirements decided upon during the development of the project. Although perhaps an unorthodox approach, this approach meant that I was able to adapt the direction of the project during its development in order to react to new discoveries made along the development path.

Due to the heavy use of time-boxing during this project, as well as the short duration of the time-boxes, I chose to use an approximation of the MoSCoW method [14] for defining my requirements. Due to the short duration, it was imperative that I prioritised certain requirements over others, lest I waste my time on unimportant features.

There is scope here to define the requirements as user stories and write acceptance criteria for them. I have decided to not use this approach as these user stories will either be too specific to allow for discovery and addition of new features upon said discovery, or they will have to be written so vaguely that they will be difficult to definitively test and be of little use to the project.

The requirements assume the pre-conditions laid out in Section 1.4.

4.1.1 Functional

- The user must be able to upload their code to a code store.
- The user must be able to trigger the CI/CD pipeline.
- The CI/CD build stage must result in a valid artefact.
- The artefact must be stored.
- The user must be able to specify how the artefact is deployed.
- The tool must be usable through a CLI.

- The user should be able to deploy a database.
- The user should have a choice over which services they use.
- The tool should guide the user through each stage of deployment.
- The tool should recommend reasonable defaults.
- The tool should expose an interface to allow the development of new plugins.

- The tool could automatically configure the project with the database's credentials.
- The tool could expose an interface to allow ease of script development involving the tool.

- The tool won't implement the full APIs exposed by the chosen service integrations.
- The tool won't offer configuration options post-deployment.

4.1.2 Non-Functional

- The tool must be usable by someone with only minor experience with DevOps.
- The tool must fail cleanly when an error arises.
- The future operation of the tool should not be affected by any previous errors that have occurred.

- The tool should be usable with only minimal interaction and without the need for extensive menus.
- The user should be able to use the tool on their platform of choice.
- The tool should notify the user as to the cause of any error which may occur.
- The user should be able to add support for more services.

- The tool could suggest fixes for any error that occurs.
- The tool could work on future versions of the services that it targets.

- The tool won't affect the targeted service should an error arise.

4.2 Architecture Overview

In order to support the addition of new services, I decided to architect the project in such as way as to accept the implementation of new services post-development. I chose to implement this through a plugin system. A key aspect of this plugin system was loose coupling such that there was little dependence of one plugin on another. This architecture bore strong similarities to Hexagonal Architecture [15]. The final architecture resulted in moderately loose coupling. This was because although each module was technically independent, there was still a core application that derived its functionality from the plugins. This architecture is also beneficial to testing as it clearly separates the functionality of the final tool into separate spaces.

The plugin system allowed for the separation of service providers, each into their own plugin. This approach proved useful from an implementation aspect as it allowed each provider to be implemented separately from each other in an isolated manner. Depending on the user, however, this approach could be detrimental to the user experience as it may have made more sense from an inexperienced user's perspective to divide up the functionalities into plugins, and not providers.

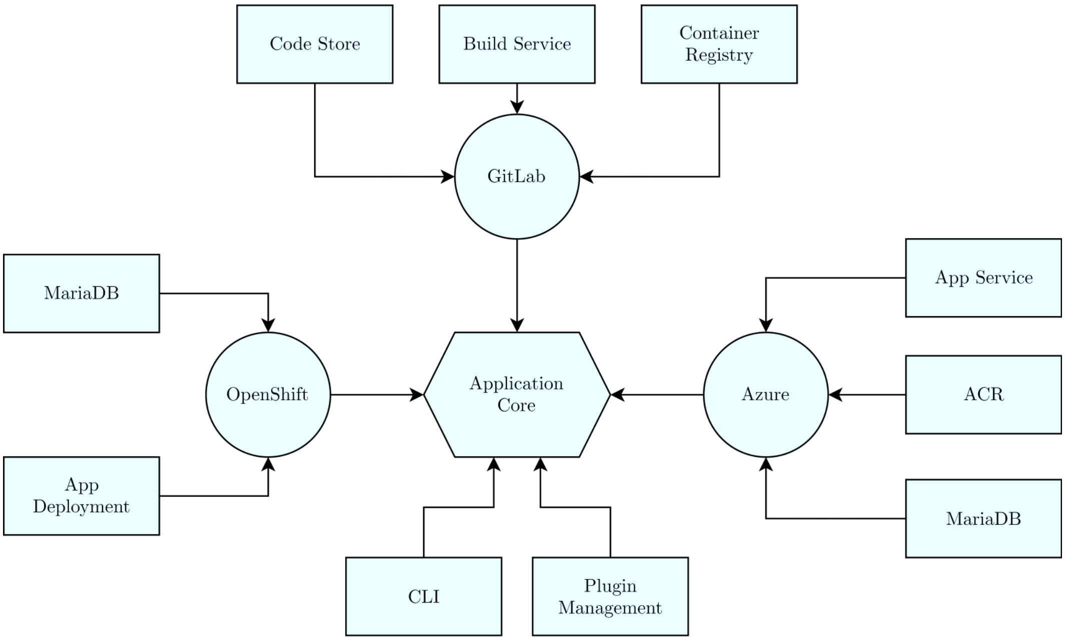

Figure 2: The plugin architecture.

4.3 Architecture Development Process

The use of plugins developed in relative isolation from each other meant that they could be implemented in parallel with little interference between each other. The first step, however, is to implement the core project. This was comprised of the overarching CLI, the plugin system itself, as well as the CI/CD pipelines for the project. During this phase, I implemented a basic interface with one of the providers such that the core project could be tested.

4.4 Modules

In order to implement a plugin system, I chose to implement the functionality for each of the providers in separate modules. These modules were entirely separate from the core project. Although ultimately, all of the code was stored in a single mono-repo, the plugins were functionally separated from the core project from Python's perspective.

As can be seen in Figure 2, the tool revolves around a central application core. This core then gains functionality from each of the plugins. In the diagram, the plugins are represented by the circles, the core by the hexagon, and features by rectangles. Although the core does implement some functionality, the majority of the functionality of the tool is derived from the plugins. Another important aspect of the architecture to note is that there are no direct connections between each of the plugins. They each only interface with the core.

I chose to architect automatic plugin detection into the project. This further decoupled the modules from the core project by removing any explicit mention of them. The architecture I chose in order to achieve this was to use a specific naming convention for the plugins, install them as Python modules, and then scan the Python namespace for matching modules.

There are two alternative architectures for implementing this method of discovery. The first architecture is the similarly named namespace packages [16]. This is where each plugin implements features in the same namespace. This has the advantage that discovery does not depend on the package name but on the contents of the package. I chose not to implement this architecture as it proved simpler to scan the global namespace instead of implicitly importing everything in a namespace. The second alternative architecture is to harness the metadata that can be included when packaging with `setuptools` [17]. `setuptools` allows for the specification of entry-points. These can then be scanned for by the parent package using `importtools` in order to find all of the installed plugins. I chose not to use this architecture as firstly, I was not explicitly using `setuptools`, but secondly, this seemed functionally similar to scanning the global namespace for packages, except with the extra steps of having to specify entry-points for each of the plugins.

4.5 Code Stores, Builders, Image Stores, and Consumers

In order to allow interoperability between plugins, the project requires an intermediary format to use between providers. One method could be the source code of the project. This has the advantage that the size of transfer between providers will always be small. The disadvantage to this approach is that it introduces many more variables to the mix. These variables are to do with the environment that the code shall be run in. With this approach, the dependencies are determined by the provider and not by the author. As well as this, as this tool is language agnostic, languages that need compilation must also need to be supported. This means that a compilation step is needed and the artefact from the compilation process is required instead of the source code.

An alternative to the transmission of the source code is to use containers. These are snapshots of an operating system that can be deployed in a reproducible way. These can be deployed to a compatible run-time in which the author can specify all of the dependencies in a reproducible manner. This run-time differs from a virtual machine in that a virtual machine virtualises an entire machine from the ground up, whereas a container run-time utilises features in the Linux kernel to isolate the container from the host without the need for virtualisation. The implementation of this format has been standardised across providers [18]. This standardisation specifies both the image format [19], as well as how the image should be run [20].

The use of containers, therefore, works as a good intermediary format to use between providers as the dependencies can be guaranteed, as well as the format being universal. This means that the chance of a provider being able to accept the container with few extra steps to deploy it is higher.

In order to use the containers, a flow of data through the proposed system is needed. As can be seen in Figure 3, this flow can be broken down into four distinct phases: code storage, building, storing the building artefact, and deployment. In between these phases are flows of data.

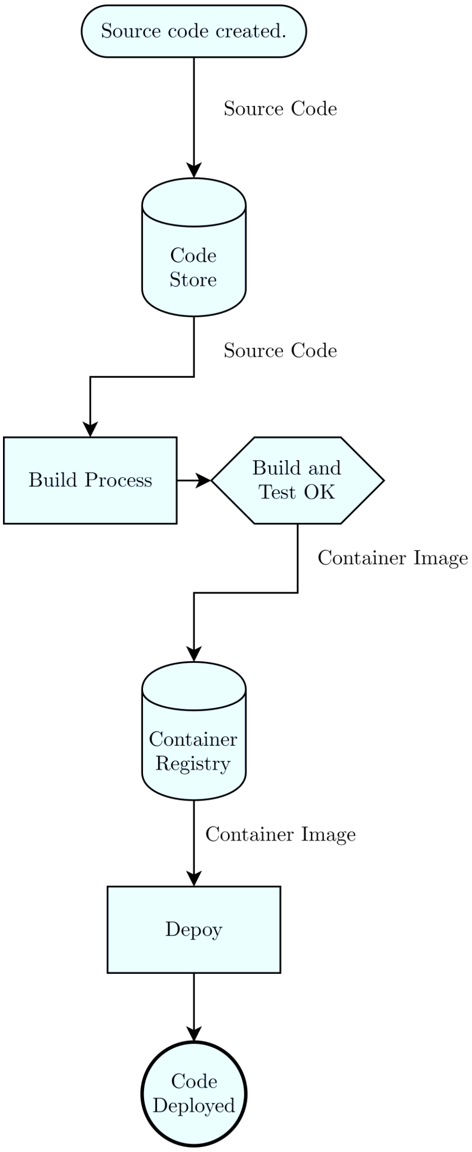

Figure 3: Data flow through the providers.

The first phase occurs when the user submits their code. This uploads the source code for the project to the code store. From the code store, the source code flows to the build process. This process takes the source code, builds it and runs the specified tests. The output from this phase is then checked. If the project built successfully and the tests passed, then the container image produced by the build service is sent to the container registry. The registry is responsible for storing the images produced by the build process and providing them to the deployment service when required. The deploy service pulls the container image from the container registry and deploys it using the aforementioned standardised run-time. This deploy service is also required to set up the routes, DNS, and storage to allow the application to function and be accessed.

Through the combination of all of these phases the tool, and consequently the user, is able to deploy the source code through a CI/CD pipeline consisting of multiple providers. Each of these providers uses an interoperable format to facilitate modularity of service providers, and thus an overall modular system.

4.6 Choice of Tools

An important aspect when starting this project was the choice of tools that I would use to develop it. These tools include but are not limited to: which langauge/s, framework/s, and how the tool shall be deployed/distributed.

4.6.1 Language

An important aspect of the project, which was specified from an early stage was that it should be multi-platform. Although most modern languages support this to some extent, higher-level languages such as Python and Java support this to a greater extent as they run on their own VM, thus separating them from the base operating system. Due to the tight time constraints, requirements, and scope of the project, it was important that I picked a language that I could develop in with good productivity. I have extensive experience writing Python in many contexts, such as web development, data science, as well as CLI applications. This made Python a perfectly suited language for this project. Although there are many other languages that could have worked well for this project, my prior experience with Python was such an overwhelming advantage versus any other language that it was the obvious choice.

Once I had chosen Python, I needed to choose which version to use. This choice was a trade-off between compatibility and feature set. The older the Python version I chose to use, the greater compatibility I could achieve as each Python 3 version is backwards compatible with all previous versions. On the other hand, the newer the Python version I picked to target, the more features I had as well as the longer security patch support. After weighing up the feature/compatibility trade-offs, I chose to target Python 3.9. At the time of development, this was the latest full release of Python which offered typing built directly into the language [21]. This meant that, due to the time pressures of the project, although I would have relatively little time to unit test each element of the project, I could avoid the most common type of programming error: “a certain kind of value was used where a different kind of value was expected” [22]. Similarly to the advantages TypeScript bring to JavaScript, Python 3.9 types combined with appropriate error checking tooling negate this kind of error.

4.6.2 Libraries

The main area of this project that could be accelerated by the use of a library was the CLI interface. There are many ways to implement CLIs in Python although there are a handful of popular libraries to do so. One of the most popular is click [23]. This provides tooling to create a comprehensive CLI including auto-generated docs, auto-completion, and argument parsing. In order to make the most of the typing advantage brought about by Python 3.9, I chose to use a library built on top of click called typer [24]. This library wraps click but utilises the type hinting introduced by Python 3.9, thus helping to eliminate bugs introduced through incorrect typing.

In order to interact with GitLab in Python, the most popular library is `python-gitlab` [25]. This wraps all of the endpoints of the GitLab API, allowing for easier use than manually using the REST API. As well as the GitLab API, this project needs to be able to interact with the target project's git data. For this, GitPython [26] is a popular library. This library is able to emulate most of git's features through its Python API.

Azure has multiple options when it comes to libraries. As its main CLI is written in Python [27], it doubles as the Python library. This resulted in an interesting development cycle. The initial implementation was done by calling the Azure CLI. Although this has several issues including potential security ones, it meant that the implementation phase could closely follow what had been documented during the discovery phase. This use of the raw CLI could then, in future, be upgraded to the use of the Azure CLI's internal CLI parser, and ultimately, using the actual Python API.

OpenShift exposed several methods of interaction. The first was the oc CLI. This, similarly to the Azure CLI, could be called in Python and could therefore minimise the jump between discovery and implementation. As OpenShift is built on top of k8s, the Python k8s library could be used for basic interaction with the service.

A final library that I chose to use was rich [28]. This provides styling of output into a terminal. Although this is not strictly required, it results in a more user friendly, and specifically, beginner friendly output.

Although this list of libraries is non-exhaustive, it covers the main libraries used to structure the project. Where other libraries are used, they play a minimal role in the project.

4.6.3 Deployment

I chose to deploy this project as a Python package, installable with pip. I chose this method as firstly, as Python was already a dependency, the user was almost guaranteed to have pip installed as well. This means that the project can be easily installed with one command. This command installs all dependencies as well as the project its self. The alternatives to this were to either distribute the source code or to compile the application. The first method would have been more difficult than distributing using pip because even when the user downloads the source, they will still have to use pip to install the dependencies, thus negating any benefits found by directly downloading the project. The second alternative would have been to compile the project. Although potentially easier for the end-user to run, it produced several cons. These being firstly the development time required to get the project to compile. Python is an interpreted language first, with compilation being retro-fitted by third-party libraries. This introduces scope for new errors and issues surrounding the unofficial method of running the code. The second issue is that of distribution. Several different builds would have to be maintained in order to target all of the possible platforms and architectures that the end-user could be using.

5 Implementation

In this section, I shall outline how I went about implementing different sections of the project. Although technical aspects shall be addressed, this section is not a substitute for the full code listings. Details of these listings can be found in the Section 10.1.

5.1 Choice of Integrations

Although the ideal outcome of this project is a tool that has modular support for all common services associated with a CI/CD pipeline, due to the time constraints I targeted a subsection of the possible services. This allowed me to focus on a full implementation of each service.

5.1.1 Code Stores

The first store of the data flow in this project was the code store. This is where the user submitted their code to, in order to trigger the pipeline. For this service, I chose to implement tooling for GitLab. This was because it is Cardiff University's primary service for source code storage. The alternative to this would have been GitHub which offers similar services. The modular nature of this project, however, means that in future, GitHub support can be easily implemented.

Another advantage to choosing GitLab is that it has a mature service for running CI/CD pipelines integrated into the GitLab instance. This reduces the amount of research required for building these two parts of the pipeline as they share a common API. This did, however, introduce the challenge of maintaining the hexagonal architecture. As both stages of the pipeline use the same service ultimately, I shared some code between the two stages. This resulted in coupling between the two services. This choice resulted in the plugin architecture that modularised service providers instead of services.

5.1.2 Builders

In order to take advantage of the services provided by GitLab, I chose to implement an integration with GitLab Pipelines: GitLab's CI/CD runner service. The GitLab pipelines service provided all of the features required to build Docker images from source code with only minimal configuration.

5.1.3 Image Stores

Due to the service-module architecture that I chose to implement, as well as the fact that several of the services I chose to implement contained container stores, I implemented two different container stores, each with their own advantages and disadvantages. The first store that I implemented was the GitLab container registry. As with GitLab Pipelines, implementation of a further GitLab service was made easier by the use of existing procedures for authentication with, and access to, GitLab. As well as this, GitLab Pipelines requires very little configuration to push images to the GitLab container registry as they both reside within the same service. A major disadvantage I encountered later on with this container registry was that for the university's deployment of GitLab, the container registry was only accessible from within the university's network. This meant that only deployment targets within the network would be able to pull images from this registry.

The other container registry I chose to implement was the Azure Container Registry (ACR). This had the advantage that it could be configured to be accessible from anywhere. There was, however, significantly more configuration required, both from the pipeline's perspective as well as from ACR's perspective. Creating an ACR instance required the setup of authentication, which then had to be passed to the build stage so that it could upload built images. This then created more complexity for the CI/CD configuration as it had to accept these dynamic values from ACR.

5.1.4 Deploy Targets

I chose to implement two targets that the user could deploy against: OpenShift, and Azure Web App Service (AWAS). OpenShift was in important target if the GitLab container registry was going to be used as there was an OpenShift instance hosted on the same network. As OpenShift is a wrapper for k8s, the process to deploying from a container registry to it was well documented as k8s is a very common service. OpenShift also manages routing, which once configured, allowed the container to be accessible from the outside world.

The second target I chose was AWAS. This is a managed container service provided by Azure. This, given the requirements of the project, was the best offering provided by Azure. It takes a Docker image and deploys it with routing, DNS, SSL, and monitoring with little configuration. This made it ideal for this project as the small amount of configuration made it more approachable for beginners.

5.1.5 Databases

An important aspect of containerised applications is persistence. This is usually achieved by storing the deployment's state in a database, instead of the container it is housed in. This reliance on the database means that speed and latency are important factors. These considerations mean that the database is usually housed, topologically speaking, close to the container. Therefore, I chose to target the same services as the deploy targets for the database integrations.

OpenShift has a concept of templates which made configuration of a database on the platform easier than building a database configuration from scratch. These templates take a set of variables and deploy a service on the platform in accordance with the configuration those variables described. OpenShift, by default, has a template to deploy a MariaDB database with persistence. Therefore, the user can be prompted for the important variables and the tool can use reasonable defaults for the less important variables.

The official guidance from Azure when it comes to deploying databases is not to use containers to do so, contrary to OpenShift's documentation. The recommendation is to use a managed database service run by Azure. One of the database services that they provide is a managed MariaDB service which takes similar configuration variables to OpenShift's MariaDB template. This sentiment is common amongst the DevOps community [29]. The reasoning for this is that the most common methods to deploy containerised applications such as k8s assume that the containers that they are deploying are stateless. Therefore, using this assumption, most management services will spin up and tear down containers with no regards to persistence. In a production setting, hosting databases in containers adds unnecessary complexity for little added value.

5.2 Project Configuration and Storage

Different operating systems use different methods for storing configuration data for their applications. For example, commonly on UNIX systems, configuration data is stored in the `□/.config/` folder. On Windows a common place is in `%APPDATA%`. In order to get the path to the OS's usual configuration directory, I used the `typer.get_app_dir("ci-plumber")` method, provided by typer. This returns a Python Path object which points to the correct configuration path for the OS.

At this location, I created a JSON configuration file which contained a dictionary. Each key in the dictionary was a different git remote, and each value was the configuration information for that remote. I chose to identify projects by their remote as this should be unique between projects. A positive side effect of this was that should the user clone a repo twice, then the configuration of that project would persist between local versions of the project.

In order to interact with this configuration system easily, I wrote a wrapper on top of this typer method to reduce complexity. This wrapper consisted of two helper functions: `get_config` and `set_config`. These two functions had two arguments: the key/value to get/set, as well as the remote address of the current repo in order up update the correct repo in the configuration file.

5.3 Core Application and Module System

The structure of the project closely followed the structure of the end CLI. The core of the project, and as per hexagonal architecture the only dependency, was a typer project. This project acted as the CLI entry-point for the whole project, as well as the coordinator in order to locate plugins. The actual functionality for this part of the project was very little as far as the end user experience went. As well as discovering plugins and providing an entry-point, this core component housed code that was used by multiple plugins. I named these helpers, and although not directly used by the core, this approach helped reduce inter-plugin reliance. These helpers consisted on mostly plugin-agnostic methods such as managing project configuration, generating project configuration files, interacting with git, and running commands.

After researching various methods of plugin detecting in Python, I decided upon the namespace search method. This is performed in the initialisation step of the CLI. Once a matching module is found, it is checked to see whether it has matching top-level variables which contain the metadata about the plugin, such as its name, as well as what stages in the CI/CD pipeline it can provide. Once the plugin has been verified, is is loaded into the main module. Each of these plugins is a separate Python module that exposes a typer CLI. The main module utilises the sub-command feature of typer which allows a main typer instance to contain other sub-instances which are made accessible through sub-commands. Each of these sub-commands represents one plugin and therefore one service which the project integrates with.

In order for this tool to bridge the gap between the different stages of the CI/CD pipeline, there were several approaches that could have been taken. The main approach I implemented was manipulating of the default values for the CLI parameters based off previous commands. For example, if the user created a Docker registry using ACR, then when they ran the deploy command, the default value for the container registry address would be the address returned in the previous step. This approach meant that the tool could maximise its usefulness for both experienced and inexperienced users. For the experienced users, there was still an option to overwrite the defaults with their own custom values. For the inexperienced users, it meant that they did not need to worry about understanding, recording, and reusing the results from the previous steps, the tool could do that automatically for them.

One potential issue when a user installs the module is that their pip bin folder is not on their system path. For an inexperienced user, this could be difficult to fix. In order to reduce dependency on the system path, I made the tool accessible from the Python executable. By including a `__main__.py` file in the root of the module, it allows the tool to be called from the command-line using `python -m ci_plumber` instead of directly. Another advantage of this approach is if the user accidentally installed two versions of the tool in two different versions of Python. This feature means that they can call the tool for a specific version of Python, for example: `python39 -m ci_plumber`.

5.4 GitHub Actions

During the development of this project, there were several tasks that had to be carried out each time I committed code. I automated these using GitHub Actions.

5.4.1 CodeQL

CodeQL is a tool recently acquired by GitHub that allows for automatic scanning of code. It scans for bugs, errors, but most importantly security vulnerabilities. Running this after each commit meant that I was always alerted if I had written any erroneous or potentially vulnerable code.

5.4.2 Docs

An important aspect of this project was the documentation available to the end user. This is because even though the project provided documentation on the command-line, less experienced users may require the help of tutorials or more visually appealing documentation. I chose to write the documentation for this project using MKDocs. This is a tool that allows you to write markdown documentation which can then be compiled into a static website. This compilation step needed to be run every time I edited the documentation. Therefore, I used GitHub actions to automatically compile the documentation and then deploy it to GitHub pages in order to be viewed by the users.

5.4.3 Testing

Due to all of the external calls made by the tool, unit testing was of very little use for most of the functionality of the project. Integration was useful but could be largely carried out by hand during the development process. Therefore, the main type of testing that I could automate was checking that the code-base contained no Python specific errors. I automated this by creating a GitHub action that could take the project and attempt to run commands on each commit.

5.5 Git Hooks

Although it is easier to automate many long running actions on the remote side using GitHub actions, many tasks can be automated locally using git hooks. These are commands that are automatically run when a commit is made. If there are any errors, then the commit is not made.

5.5.1 Pre-commit-hooks

In order to automate the installation, updating, and integration of these hooks, I used a Python package called pre-commit-hooks. This package takes a configuration file of all of the hooks you wish to add to the project and registers them with git. As the hooks are specified by pointing pre-commit-hooks at a remote repository containing the hook, it is also able to update the hooks to newer versions.

5.5.2 Flake8

Flake8 is a Python linter. Its purpose is to flag any style errors in a Python code base. Python has a style guide called PEP8 which provides details on how you should format your Python. For example, how many spaces you should indent by and what maximum line length should be. If the code base is not flake8 compliant, then flake8 will return an error and the commit will not go through.

5.5.3 MyPy

One of the main reasons I chose to target Python 3.9 was for its new type syntax. Although types are now fully supported in Python 3.9, you don't *have* to use them. Mypy enforces typing on all variables used throughout the project, thus forcing the developer to make use of the new typing features present in Python 3.9. If any variables are used without explicitly declaring their types in the project, mypy will return an error and the commit will fail.

5.5.4 Black

Black is a formatter that works in tandem with flake8. Black works through the whole project and formats the project in accordance with flake8. Although it changes the code, it never changes the logical structure of the code. Although black does not typically fail and does not typically block committing, it helps reduce the number of times flake8 prevents committing by formatting the code in compliance with flake8.

5.5.5 Isort

Much like black, isort does not typically block committing. Its purpose is to sort imports in a Python program into a logical order. By default, it splits imports into three sections: standard library, third party, and absolute imports. In each of these sections, the imports are ordered alphabetically. Although PEP8 does not specify an order to import, and there is little performance difference, it helps to make the code base more readable and logical.

5.6 GitLab

I chose to implement most of the GitLab functionality into one initialisation function. The reason I chose to do this was that the code store functionality that GitLab offers would usually already have been configured by the user. This is an assumption that the project makes throughout the development process. For example, the configuration mechanism assumes that the user's project resides in a folder that is a git repository and has a remote set. Therefore, due to this assumption, the initialisation step of the GitLab plugin largely revolves around setting up the CI/CD configuration as opposed to the git configuration.

5.6.1 Initialisation

The initialisation starts by determining which folder the tool is being run from, and then determines whether that folder is a get repository. Finally it checks to see whether there is a remote for the repository. Should any of these stages fail, then the tool will report the error.

Once the tool has determined the remote address, it authenticates the user with GitLab. This is done either by an access token and URL passed as arguments, or prompted for interactively. Once the user has been authenticated against GitLab, a list of the repositories that they have access to is retrieved. This list of repositories is checked against the remote for the local repository. If a match is found, the ID of the remote repository is stored. If no match is found, an error is raised, informing the user that the remote for their local repository does not match any of the repositories on the GitLab instance that they have authenticated with.

5.6.2 CI Pipelines

The penultimate step of the GitLab initialisation is to generate the requisite files for the CI/CD pipeline. The first step of this is to generate the `.gitlab-ci.yml` file. This contains the steps required to build the project. In the case of this project, to build a Docker image of the project and push it to the correct registry. The default `.gitlab-ci.yml` file uses a tool called Kaniko [30], which builds a Docker image using a Dockerfile. As the GitLab CI runners use Docker themselves, it is considered bad practice to use Docker again in a build file. This results in a scenario called Docker-in-Docker (DinD) [31]. Although this does work, it requires the runner to be running in a privileged mode, which can cause security issues. Kaniko behaves in a similar manner to the Docker build system but runs entirely in user space without the privileges needed by Docker. Once Kaniko has built the image, it uploads it the the GitLab container registry. It can retrieve the details of this without further configuration by the user as these are accessible as environment variables as standard in all CI runners.

5.6.3 Dockerfiles

The final stage is to generate a Dockerfile that can build the project. This is a difficult stage to complete automatically as each project can be architected very differently and therefore require a very different Dockerfile. In order to solve this problem, I created a Python package called framework-detector which, given a folder, attempts to work out what the project type is. I detail how this sub-project works in Section 5.9.

Once the Dockerfile is generated, the user can push all of their changes the the remote server. To prevent the build and deploy pipeline running for every commit, by default, the CI runner is only triggered when a new tag is pushed to the remote. This allows the user to control exactly when their project is deployed. The benefit to using a tag as opposed to a manually triggered runner is that a tag can be created and pushed from the local command line using vanilla git commands whereas a manually triggered event must either be done through the GitLab web interface or through the API.

5.6.4 API

I interacted with GitLab through channels: git, and the API. In order to fulfil the code store aspect of GitLab, all of the functionality is available through the vanilla git interface. Through git, the user is able to upload their code, push tags, and trigger builds. The only aspect as far as storing code is concerned that cannot be done using the git binary is the creation of a repository on the GitLab instance.

All other actions to do with GitLab are done through the API using python-gitlab. This includes determining which repos the user has access to, as well as editing CI variables when pushing the image to an external repository. The main drawback to using the API is authentication. Although an OAuth application can be created and the user authenticated through that [32], this feature was disabled on the university's GitLab instance. Therefore the only way to authenticate the user was to get them to generate a token using the GitLab web UI. This added unneeded complexity to the authentication process.

5.7 OpenShift

I chose to divide the OpenShift plugin up into several commands. These were deploy, list projects, create a database, and create a database configuration. These commands provided all of the most basic functionality needed to deploy a project to OpenShift.

5.7.1 Interfacing

I used two approaches for interfacing with OpenShift: the CLI, and the Python k8s module. The CLI was useful for issuing configuration commands as the documentation largely used these commands for examples. The k8s API was useful for retrieving complex data as this data was returned in a native Python format and therefore did not require parsing. Overall though, the CLI proved more useful as this closely followed the documentation. The k8s API also did not allow for interaction with the more advanced features of OpenShift which are not available in vanilla k8s. In order to interface with the CLI, I used a method previously mentioned in Section 5.3 which took a command and ran it in a relatively secure manner. Any errors were handled and the result of the command returned.

5.7.2 Application Deployment

The deployment process in OpenShift is at first glance needlessly long. Upon further inspection, however, this is due to the level of customisation and diversity of authentication provided. Although this project is in a working state, the OpenShift implementation is not at its ideal state of completion due to the coupling still implemented in this module to the GitLab module. This is because at the time of implementation, GitLab was the only source of Docker images as well as implementing a non-standard method of authentication for getting those images. Therefore, the OpenShift implementation implements these non-standard methods, thus creating a dependency on the image store being one provided by a GitLab instance.

The first step to my method of deployment was to first authenticate with GitLab and OpenShift. Once authenticated with GitLab, the project ID could be found. This was important as it was needed to determine the path to the image on the GitLab image store. In hindsight, it would have been better for the user to specify the path, but this method required less configuration on the part of the user and therefore made sense at the time of implementation. I improved upon my method of specifying image paths when I implemented the ACR integration.

After authentication with GitLab and OpenShift, the GitLab authentication secrets needed to be inputted into OpenShift. The first step of this was to create a new OpenShift project. Once created, two sets of secrets needed to be created. Usually only one needs to be created but due to the quirks of GitLab, the authentication URL and the image store URL were different and therefore needed to be authenticated against independently. Although OpenShift supports this method of delegated authentication [33], it is non-standard. For each endpoint, the username, password, and email needed to be specified. Once these were inputted, they needed to be linked to actions within OpenShift in order to let OpenShift know which secrets it needed to use for which actions. These action being build, deploy, and default. Although it is likely that the secrets only needed to be tied to either default or deploy, I chose to bind them to all three should the user choose to continue using the GitLab project in future for more advanced deployments but wished to continue using GitLab as the image store.

Once the secrets have been created, a new image stream can be created. It is important that an image stream is used instead of directly importing an image. This is because an image stream can point towards an external image and when that external image updates, the image stream pulls the most recent version of the image and can trigger a redeployment of an application with the most up to date version of the image. This is an important aspect of the CI/CD pipeline this tools aims to create as without this automatic step, it becomes less of a pipeline and more of a series of steps that the developer must step through each time they wish to deploy. This image steam can be used by different apps within the project, although without additional configuration, cannot be used outside of the project.

An app is then created using this image stream as the base image. The app is then exposed externally. This process of exposing the app only works if the Dockerfile and consequently the Docker image specify the default ports using the `EXPOSE` directive. Once the application is exposed, the routes generated can then be displayed to the user. The project should be ready or close to ready at this point.

5.7.3 Database Deployment

I chose to divide the deployment of a database on OpenShift into two steps: creating the configuration file, and deploying the database using the configuration file. I chose this method as the MariaDB template required a lot of variables. Therefore, in order to make the tool accessible to less experienced users, I had to choose a lot of the defaults myself. By splitting the process into two steps, it allows for more experienced developers to look at the configuration file and make any changes they with to make before deploying the database.

The command to create the database configuration takes lots of arguments, but most of them have defaults and are therefore not required. If the user does not specify any of the required variables then they will be prompted for them when they attempt to run the command. The command then checks to see if there is already a file called `maria.env`. If there isn't, a configuration file of that name is created with the configuration data inputted into the command. An entry into the project's `.gitignore` is also created so that the configuration file containing the password is not committed into source control. The second command creates the database using this file. The file is already in the correct format of OpenShift and can therefore be directly uploaded along with the name of the template it is the variable set for; In this case it is `openshift/mariadb-persistent`.

5.8 Azure

Azure was the final integration I developed for this project and therefore most resemblant of its full implementation. I applied what I had learned developing the other plugins to the development of this one. Therefore, I was able to avoid some of my previous pitfalls such as dependencies between plugins, structure, and targeting the plugin architecture from the start of development as opposed to refactoring code into the plugin system as with the GitLab and OpenShift implementations.

5.8.1 Interfacing

In order to interface with Azure, I created a three step implementation plan. Unfortunately, due to time constrains, I was only able to implement the first step of this plan. The three steps I identified were:

1. Wrap the CLI.

2. Use the command parsing part of the Python library to parse the commands without using the commandline.

3. Directly use the APIs provided by the Python library.

The only advantage to each of these steps were and increase in security, speed, and package size. As none of these steps impacted the actual functionality, they were of less importance that implementing further features. Given a greater amount of time, I would have implemented this plugin using the Python API. The main drawback to using the CLI, besides security, was the number of dependencies that the Python Azure CLI library required. As it is able to interact will all of the services that Azure provides, it requires the Python library for every service, therefore severely bloating the install size of this project. The ideal final state would be to only require the Python libraries for the services used by this tool and to interact with them directly instead of with the CLI.

5.8.2 Registry Creation

Due to the limitation that the university's GitLab instance's container registry is only accessible from the university's network, it was vital that the Azure plugin should contain functionality to create a container registry so that Azure could be used in the deploy stage of the pipeline.

The ACR functionality was implemented as a single command that took configuration information for the registry and created it. As with most other commands in the project, reasonable defaults were chosen for each of the configuration values and the user prompted should they not specify any of the required variables.

I broke the task of creating an ACR down into a series of steps. The first of these was to manage the current resource group. Resource groups are how Azure groups a set of resources together so that they can be managed as a group [34]. When the request is made to create a new resource group, a success will be reported whether it already existed or not. Therefore, I always try to create the resource group whenever one is specified in the command.

Once the resource group is created, the registry can be created within it. At this point the SKU for the resource group also needs to be specified. For most of the projects using this tool, the most basic SKU is good enough and is therefore the default. Once the ACR is created, an admin needs to be created on the registry for authentication. Once the admin has been created, their credentials can be returned.

At this point, the tool has the URL of the ACR as well as the credentials. Therefore, these need to be sent to the build process in order for it to be able to upload the artefact it creates during the build. In this case, the credentials are set as environment variables. This is another case of a dependency where there shouldn't be one. The tool, at this point, communicates directly with GitLab. In future, a generic interface should be exposed by the core application for communicating the each stage in the pipeline. The core would then be able to direct the message to the correct plugin as opposed to hard coding one of the plugins in.

As well as the environment variables, the `.gitlab-ci.yml` needs to be updated as well to reflect the names of these new environment variables. This new CI file then needs to be committed to GitLab before a new build can take place so that the new build sends the images to ACR instead of the GitLab package registry.

5.8.3 Application Deployment

Once an image has been deployed to ACR, an application can be created from it. Similarly to the "creating the ACR command", this command has several good defaults as well as required arguments that will get prompted for should they not be specified. The first step to the creation of the web app is to create a service plan. This serves as a wrapper around the app and defines how it should be run. For example, which OS should be used. Using this service plan, a web app can be created. This app uses the service plan as well as the path to the image uploaded to the previously created ACR. In order to use this image, authentication needs to be set up. The first step is to create a managed identity for the web app. This serves as a user account that can access ACR on behalf of the web app. This user can then be granted pull access to the ACR so that they have permission to pull images from it. Finally, the managed identity is tied to the web app so that the web app can communicate with ACR through the identity.

Finally, the application can be deployed. The web app now has permission to pull images from ACR and can therefore pull the image of the user's project. This deploy process automatically manages deploy stages such as routing and management of SSL certificates. Finally, the URL the project has been deployed to can be displayed to the user.

5.8.4 Database Deployment

I implemented the database creation functionality for Azure into a single command. Due to the huge number of variables associated with the database, I had to choose a lot of the defaults, however, they are all accessible as arguments for the command. The process of creating the database is relatively simple. A command is issues with all of the variables specified in the command. This then returns a set of credentials for the database.

In order to maintain similarity between the plugins, I chose to also generate a `maria.env` file with the database details in as well as add it to the `.gitignore`.

5.9 Detection of Project Type

An important aspect of the project was creating an appropriate Dockerfile for the project. Although it might not be fully set up for the project, it can be useful to have a good tarting point when writing a Dockerfile. The first stage of providing an appropriate Dockerfile is to determine what kind of project the Dockerfile is being created for. Once the project type has been determined, the corresponding Dockerfile can be provided.

In order to detect the project type, I created a JSON file for each type of project that I wished to detect. Each file specified metadata to do with that language or framework as well as information on how to identify a project of that type. This information consisted of files to look for in the project as well as optionally the contents of those files. For example, if the project uses Spring Boot and Gradle, there will be a `build.gradle` file in the root of the project containing the string `org.springframework.boot`. Should this criteria match with the specified directory, then the default Spring Boot Dockerfile can be provided.

This method of determining what framework or language a project is using allows for a modular and therefore easily extensible tool. Should a new framework wished to be added, it would be as simple as adding the requisite JSON file as well as a corresponding Dockerfile.

5.10 Wizard

Unfortunately, due to the time constraints of the project as well as the architecture I chose, I was unable to complete the wizard feature. The intended purpose of this feature was to guide the user through the whole process of creating the pipeline. The choice of dividing the targeted platforms into plugins hidden behind typer meant that it was difficult to interface with them beyond typer's functionality from the core project.

In order to implement this feature in the time left, I created a one week time box and a branch in which to attempt to create this feature. Therefore, if I wasn't successful, I could just delete the branch without leaving half finished code in the project. Despite not finishing this feature, I chose to keep some of the functionality I had built as it was not detrimental to the project and would be a starting point if I wished to complete the wizard functionality in future. The functionality created was each plugin exposing to the core project what stages in the pipeline they provided. For example, Azure exposed the image store, deployment, and database stages.

5.11 Documentation

Due to the requirement of this project being beginner friendly, I chose to create online documentation [10]. This contains the usage documentation for the tool as well as tutorials for the most common pipeline configurations. The documentation was written in markdown and then compiled using MKDocs. As was mentioned in Section 5.4.2, this was done automatically using GitHub actions. This action then published the docs to GitHub pages where is is available to be viewed by the end user.

6 Results and Evaluation

Due to the open-ended nature of this project, it is difficult to determine a finished state for it. Although each requirement can be checked to see whether it has been met, significant scope for further development and scope still exists. Therefore, I think it is important to reflect and evaluate the success of this project on not just the measurable, quantitative outcomes such as requirements met, but also on qualitative outcomes such as my opinion on how the project went. The requirements only measure features whereas a truely positive outcome for this project should be measured in capability, where capability reflects on the whole process that the tool implements. Success should therefore be ideally measured, for example, on whether a novice developer can implement a pipeline as opposed to a specific feature within that process.

6.1 Requirements

6.1.1 Functional

- *The user must be able to upload their code to a code store.*
  This requirement was met as a requirement in the first place to use the tool is to have a code repository that code can be uploaded to. In the case of this project, it was GitLab although there is scope to include other stores such as GitHub.

- *The user must be able to trigger the CI/CD pipeline.*
  This requirement was made by creating the `gitlab-ci.yml` file in the GitLab plugin. This file created a trigger to start the CI pipeline by creating and pushing a new tag to the source code repository.

- *The CI/CD build stage must result in a valid artefact.*
  Although the definition of a valid artefact is undefined for this requirement, it could be interpreted to mean an artefact that is compatible with the other stages of the pipeline. In this case this artefact is an OCI compatible container image. The final result of the GitLab build process is indeed such an artefact. Therefore this requirement can be considered complete.

- *The artefact must be stored.*
  This requirement is met by two of the plugins. Firstly the GitLab plugin fulfils this requirement by providing an integration with the GitLab container registry. This is configured automatically by the tool as the upload destination of the build artefact. The second plugin that fulfils this requirement is the Azure plugin. The tooling for ACR means that ACR can be configured to be the upload destination for the image outputted by the build stage.

- *The user must be able to specify how the artefact is deployed.*
  The user is provided with choice at two levels when it comes to deploying the application. The first level of choice is the platform that the artefact is deployed on. Both the OpenShift and Azure plugins support deployment. The second level at which the user has choice is with the configuration of their chosen service. Both OpenShift and Azure allow for customisation to some extent the way that deployment in configured.

- *The tool must be usable through a CLI.*
  The tool is installable and usable through a CLI. The tool can be installed using `pip` entirely through the command line. It can then be called using either `ci-plumber` or `python -m ci_plumber`.

- *The user should be able to deploy a database.*
  This requirement is met in two plugins. In both the OpenShift and Azure plugins, the user is able to configure and deploy a MariaDB database that supports persistence and external connections.

- *The user should have a choice over which services they use.*
  This requirement is partially met in the current stage of implementation although can be fully met once more plugins are creating using the existing plugin interface. There is choice of service for each stage of the CI pipeline except for code storage and the build stage. In future, services such as GitHub and Jenkins could be added so that this requirement can be fully met.

- *The tool should guide the user through each stage of deployment.*
  This requirement was not met. The aim of the wizard implementation was meant to fulfil this requirement but due to issues presented in Section 5.10, this section of the project was left uncompleted.

- *The tool should recommend reasonable defaults.*
  This requirement was met firstly by the defaults included in the tool. These provided reasonable defaults for all of the services the tool provided. Secondly, the defaults were automatically updated to reflect the

outputs of previous commands. This meant that the defaults were not only reasonable, they were also relevant.

- *The tool should expose an interface to allow the development of new plugins.*
  The plugin system as discussed in Section 5.3 allows for new plugins to be developed, installed, and then automatically discovered by the tool with no change to the tool's central code-base.

- *The tool could automatically configure the project with the database's credentials.*
  This requirement was partially met. Providing that the project used environment variables to store the database credentials and that these environment variables matched the naming standard used by the `maria.env` file, then the database could be automatically used by the user with little to no extra configuration.

- *The tool could expose an interface to allow ease of script development involving the tool.*
  The tool allows for easy scripting by exposing every configuration value used as arguments for each of its commands. This means that the full functionality of the tool can be extracted through the use of scripting with no user interaction.

- *The tool won't implement the full APIs exposed by the chosen service integrations.*
  Due to the complex nature of the services interfaced with for the project, the full APIs of each of them were not implemented. This allowed for a much more streamlined interface for the user and only provided the tools that they absolutely needed with little complication.

- *The tool won't offer configuration options post-deployment.*
  Configuration of services post deployment was beyond the scope of this project and therefore not implemented, thus fulfilling this requirement.

6.1.2 Non-Functional

- *The tool must be usable by someone with only minor experience with DevOps.*
  This requirement cannot be tested as no user testing was conducted during this project.

- *The tool must fail cleanly when error arises.*
  When error occurs, it is caught and displayed in a readable manner to the user thus fulfilling this requirement.

- *The future operation of the tool should not be affected by any previous errors that have occurred.*
  Although this requirement is not explicitly handled, the main risk for errors occurs is when calls are made to external services. These calls can either succeed and therefore change the state of the service and therefore future running of the tool, or they can error. In the event of an error, the state of the external service will not change and therefore the future running of the tool will not be affected.

- *The tool should be usable with only minimal interaction and without the need for extensive menus.*
  The user can either enter all of their inputs in one go as command line arguments or they can be prompted for them. Either way, the minimum amount of interaction possible is made, thus fulfilling this requirement.

- *The user should be able to use the tool on their platform of choice.*
  As the tool is written in Python, distributed in an uncompiled format, and uses platform agnostic methods when using the file system, it can be used on most platforms.

- *The tool should notify the user as to the cause of any error which may occur.*
  Should any error occur, the error and stack trace are presented to the user to analyse and act upon.

- *The user should be able to add support for more services.*
  The plugin system allows for the user to implement extra services as they see fit.

- *The tool could suggest fixes for any error that occurs.*
  Although the errors are descriptive and could potentially suggest a fix by their verbose nature, there is no explicit functionality to suggest fixes to errors.

- *The tool could work on future versions of the services that it targets.*
  The backwards compatibility of the APIs of the services that the tool target is beyond the control of the project. Should the future versions of the services maintain backwards compatibility of their APIs, then the project will remain compatible.

- *The tool won't affect the targeted service should an error arise.*
  This requirement is fulfilled for the same reason the requirement "*The future operation of the tool should not be affected by any previous errors that have occurred*" was fulfilled as the future operation of the tool is dependant on the state of the target service.

6.2 Development Retrospective

Overall, I think the development process was a positive experience, both for the project and for my personal development. The tool met most of its requirements and those that it did not meet were met to some extent. There are many positive and negatives to do with the actual development process of this project so I will only address the major ones in this section.

6.2.1 Positives

One of the my proudest achievements and one of the most significant features of this project is the plugin system. This was not a feature that I had previously implemented in a Python project and was therefore apprehensive about. Due to the time constraints and the cost of implementing the plugin system meant that I only had one attempt at implementation. If this had of failed I would have to have reset the project back to a previous state as there wouldn't have been time to implement another module system as well as finish the integrations I wished to complete. Therefore, I think it is a big positive that the plugin system worked and I could therefore include it in the end-tool.

A key aspect of this project was that is was friendly towards less experienced developers. Therefore, the simplified interface that was implemented for the complex services that the tool wrapped was a positive. This aspect of the project required a significant amount of research and was therefore another higher risk area of the project as the research required a large investment of time without any return in terms of requirements met. Although this outcome should be tested user research and testing, I am treating any simplification of the standard interfaces of the target services as a success due to the complex and opinionated nature of them.

The most obvious positive outcome of this project to the end user is the CLI experience when using the tool. I am pleased with my choice of libraries used to display the user interface as well as my implementation of them. Typer allowed me to rapidly build a fully featured CLI including docs, autocomplete, and command arguments. The use of rich to display the output allowed me to display informative, live outputs to the greatest extent that can be displayed in a standard terminal and to do that in a cross platform way.

6.2.2 Negatives

Despite the overarching success of this project, there are a significant amount of negatives associated with the development process of this project.

The lack of testing, specifically user, unit, and integration tests resulted in a tool that is not as reliable as it could be. Although the majority of bugs are associated with type issues and were therefore addressed using strict type checking for the whole project, all other types of bugs such as those caused by logic errors or faulty interfaces with other services. These issues could have been solved using the aforementioned methods of testing. Although lack of testing was not a conscious decision during the development process, I chose to implement further features and integrations instead of reducing the technical dept that had been accrued. This did, however, have the benefit of more of the requirements being met. Perhaps a better way to address this problem in future would be to make testing a requirement and therefore it could be implemented during the development process without impacting velocity. An alternative perspective for this negative is to in fact question the efficacy and purpose of testing. Although testing is important for increasing confidence in stakeholders, it was a greater priority to deliver functionality. [35] talks about "*shifting testing left*". This does not mean testing earlier, but instead improving on all of the dimensions of quality that are not reflected in testing, such as compliance and accessibility from early on on the life-cycle of the project. This stage of software development can be called the discovery phase and does not warrant thorough testing as many aspects of the project are still likely to change [36].

Due to the process of developing my technical skills as well as the architecture in parallel with the development of the tool, small mistakes were made in the implementation of features of the integrations that became vital components. The most common and potentially most serious mistake that I made in the start of the project was reliance of one plugin on another. This resulted on deviation away from the ideal hexagonal architecture and to something needlessly more complex.

Similarly to the last point, due to the less formal nature of the planning implemented in this project, the early project lacked structure. This was due to my first implementation being little more than scratchpad code. Although not a bad thing in its own right, the final implementation should have been built from the ground up as opposed to modifying existing code. The scratchpad implementation was an important stage in learning the services involved in this project and how they behaved. This code was refactored to an extent but the core structure remained the same. Therefore, in its current state, the code base has a lot of large files that contain code that performs a variety of functions instead of more segmented, organised code. Although this can be fixed with relative ease, the time constraints and few drawbacks to this issue mean that it has yet to be fixed and could lead to maintainability issues in future.

When I was coming to the end of the project, I encountered difficulties to do with managing the documentation. As I have previously mentioned, I spent the majority of time developing new features instead of solidifying what I had already built. Although this had the benefit of increasing the number of requirements I fulfilled, it meant that I had disregarded some of the other important aspects of software development with documentation being one of them. Although I had left enough time to write the documentation at the end of the project, the process was more difficult that if I had been writing it as I had been developing the tool.

7 Future Work

The most important aspect that needs changing in future is to add an appropriate open source license to the project so that it can be used by a wider audience that just those in the faculty. Wider use would be good reason and motivation to improve upon this project further. The use of a permissive licence would allow other developers to use the tool without fear of infringing on the license but would allow correct attribution to still be made for the tool.

In order to continue development on the project in a sustainable manner, significant refactoring will need to be done on the existing code base to address issues previously mentioned. This is an important first step as the sooner the refactoring is done, the sooner that the refactoring is done, the less code will need to be refactored. This is because all new code added will build upon the existing problematic code. The refactoring will also help speed up future development as it will make the code base more readable and logical and will therefore be able to be built upon more easily.

A shortcut that I used throughout the project to reduce the differences between my implementation steps and discovery steps was to wrap the CLI instead of using the API provided by the service. This had the advantage that I could implement features faster but at the cost of speed, security, and reliability. Therefore, once the refactoring has been completed, it would be a good idea to change the CLI based implementations for API based implementations.

As well as solidifying existing integrations, the addition of more integrations in the form of plugins would benefit the project. Due to the limited choice of existing integrations, the tool could not provide the functionality in terms of features or platforms that a user could require. This addition of more integrations would increase the potential audience of the tool as well as improve the feature set for existing users.

The wizard was the largest planned feature that was not implemented in the final submission. Therefore, it would be a good future task to complete the implementation. The groundwork in the plugins has already been made and therefore only major changes to the core would have to be made and not to the existing plugins.

A potentially confusing aspect of the tool to a new user could be that depending on the image store, the CI file changes. This stage could be simplified by using a standard set of CI variables amongst providers of image stores, thus negating the need for different CI files for each provider.

8 Conclusion

In conclusion, I am pleased with the overall outcome of the project. The product of this project was a tool that provides benefit to both experienced and inexperienced developers. It has taught me much about planning, project management, as well as software development and architecture. The tool fills a gap that there are few tools for and therefore delivers value, despite the improvements that can be made.

9 Reflection

The greatest positive that I can take away is the personal development gained from completing this project. This is the largest solo project I have built and therefore I encountered and had to overcome issues that I had not in previous project. The main challenge that I encountered and had to improve upon was time management with regards to task estimation. I consistently marginally underestimated every task that I undertook which resulted in a lack of time to work on non-core features such as tests and documentation. Were I to do the project again, I would explicitly factor in time to write tests and documentation.

This time spent on implementation instead of testing was not wholly bad, however. It allowed me to ensure that the architecture was correct and implemented properly opposed to rushed. It also allowed me to develop my own skills further instead of using the limited time I had to write tests and polish what I had already built. Furthermore, it could be argued that through the time spent of making sure that the architecture was correct, I built the project in a more testable, modular way that may not have been possible should I have spent the development time elsewhere.

I am pleased with the approach that I took with the project with regards to methodology. Although it took some time to develop a structure, my approach of one week time-boxed periods in which I attempted to produce a pre-decided set of features worked well. The weekly evaluations allowed me to work in a flexible way and to react quickly to any hindrances which I encountered during my research.

In hindsight, my approach was potentially too ambitious for the timeframe. Although there are few tools that meet the criteria that this tool meets, there are tools that bridge the gap between services in a highly configurable way. An example of this is Terraform. Although the initial learning curve is steep, Terraform is an extremely powerful and robust tool for defining infrastructure as code. Everything that this project achieves is possible in Terraform. Therefore, it might have been a better decision to take the time to research tools like Terraform more deeply, both to understand why they work in the way that they do and whether there is anything to be learned from that, as well as to look into possibilities of integrating those projects into my own. Although an avenue unexplored to its full potential, I think that a better approach to this project would have been to wrap Terraform instead of the individual platforms. This could have potentially lowered the barrier of entry to use Terraform, allowing less experienced developers to use it. The main advantage to this is breadth of platform support. With my chosen approach, I have to manually integrate each new platform into the tool. With the Terraform wrapping approach, I could have utilised the existing the existing integrations it provides and written a generic wrapper around it.

10 Appendix

10.1 Source Code

The source code for the project is available in the following locations:

- https://github.com/pbexe/ci-plumber
  The main project repository.

- https://github.com/pbexe/framework-detector
  The framework-detector source code.

- https://git.cardiff.ac.uk/c1769331/ci-plumber
  A mirror of the main repository on the university's GitLab.

- https://doi.org/10.5281/zenodo.5545987
  A snapshot of the project in a citable format.

10.2 Package Manager

The project is published on PyPI and is available at the following links:

- https://pypi.org/project/ci-plumber/

- https://pypi.org/project/ci-plumber-openshift/

- https://pypi.org/project/ci-plumber-gitlab/
- https://pypi.org/project/ci-plumber-azure/
- https://pypi.org/project/framework-detector/

10.3 Installation

10.3.1 Requirements

- Python 3.9+
- Azure CLI
- Openshift CLI
- Windows/Mac/Linux. Others may work but are untested.
- A supported project type. Currently supported:
  - Spring Boot
  - Flask
  - Or just a Dockerfile

10.3.2 Installation Steps

```
// Install CI Plumber as well as all of the modules:
$ pip install ci-plumber[all]

// You can also install individual modules instead of the entire package:
$ pip install ci-plumber
$ pip install ci-plumber-azure

// Once installed, you can add tab completion:
$ ci-plumber --install-completion
```

10.4 Usage

10.4.1 GitLab and Openshift Tutorial

First we need to initialise the project. All of the commands can either be run interactively or using the CLI options. For this tutorial we shall be using the interactive mode.

```
$ ci-plumber gitlab init
Gitlab url [git.cardiff.ac.uk]: <The URL to your gitlab instance>
```

```
Username: <Your username>
Email: <Your email>
Access token: <Your access token>
Docker registry url [registry.git.cf.ac.uk]: <The URL to your Docker registry>
Getting remote
[12:41:23] Logging in to Gitlab
           Getting projects
[12:41:24] Matching remote with Gitlab projects
           Found project: Flask Demo
           Generating .gitlab-ci.yml
           Generating Dockerfile
           Gitlab configured!
```

This command will do several things:

1. It will ask you for the gitlab url, username, email and access token. These will be stored in order to authenticate against GitLab.
2. It will also ask you for the Docker registry url. This is the url that other plugins such as Openshift will be able to pull images from.
3. It will then try to find the project that you are working on on GitLab.
4. It will then Genrate the .gitlab-ci.yml file and the Dockerfile if approprate for the project.

We next need to push our changes to GitLab so that GitLab CI will run the new configuration:

```
// Stage the changes
$ git add .

// Commit the changes
$ git commit -m "Add .gitlab-ci.yml and Dockerfile"

// Create a new tag to trigger the pipeline
$ git tag -a v0.0.1 -m "Release v0.0.1"

// Push the changes to GitLab
$ git push --follow-tags
```

Once GitLab is set up, we can set up the Openshift project. Openshift should pick up on the credentials left by `ci-plumber gitlab init`. We can deploy the app to Openshift using the following command:

```
$ ci-plumber openshift deploy
Project: <A name unique to your project>
Username [c1769331]: <Your username. The default should be yours>
Password: <You won't be able to see what you're typing here. It's not broken.>
Repeat for confirmation:
[13:08:46] Logginginto GitLab
           Getting the Gitlab project
[13:08:47] Loggin in to Openshift
[13:08:49] Creating a new project
[13:08:52] Creating secrets
[13:09:11] Importing image-stream
[13:09:13] Creating a new app
[13:09:16] Exposing the service
[13:09:18] Here are the details
[13:09:20] <The details as well as the URL will be written here>
```

To deploy a database as well, you can use the following command:

```
$ ci-plumber openshift create-db
Mysql password:
Mysql root password:
[13:19:28] Creating database config
           Creating MariaDB pod from openshift/mariadb-persistent template
[13:19:31] Exposing DB
[13:19:33] Getting DNS
[13:19:36] Writing config to maria.env

// You can now find the credentials in maria.env
$ cat maria.env
ADMIN_PASSWORD=<Your password>
USER=maria_user
PASSWORD=<Your password>
NAME=mariadb
HOST=<The database DNS>
```

10.4.2 GitLab and Azure Tutorial

To deploy to Azure, we shall use a different architecture for the project. We will begin in a similar manner to the GitLab + Openshift section:

```
$ ci-plumber gitlab init
Gitlab url [git.cardiff.ac.uk]: <The URL to your gitlab instance>
Username: <Your username>
Email: <Your email>
```

```
Access token: <Your access token>
Docker registry url [registry.git.cf.ac.uk]: <The URL to your Docker registry>
Getting remote
[12:41:23] Logging in to Gitlab
           Getting projects
[12:41:24] Matching remote with Gitlab projects
           Found project: Flask Demo
           Generating .gitlab-ci.yml
           Generating Dockerfile
           Gitlab configured!
```

This will create GitLab credentials similarly to before. However, we will now be using Azure instead of GitLab to store the images. We must begin by creating a new Azure container registry:

```
$ ci-plumber azure create-registry
Registry name [registry887130626]:
Resource group name [myResourceGroup]: sub1
[16:00:16] Creating resource group sub1
[16:00:24] Creating registry registry887130626
[16:00:40] Enabling admin user
[16:00:43] Getting admin credentials
[16:00:46] Logging in to Gitlab
           Gettingthe Gitlab project
           Creating Azure access keys in CI
           Azure access keys already exist in Gitlab CI for c1769331/flask-demo
[16:00:47] Creating .gitlab-ci.yml

// Stage the changes
$ git add .

// Commit the changes
$ git commit -m ”Add .gitlab-ci.yml and Dockerfile”

// Create a new tag to trigger the pipeline
$ git tag -a v0.0.1 -m ”Release v0.0.1”

// Push the changes to Gitlab
$ git push
```

We have now instantiated a new Azure container registry, pointed GitLab CI to push new images to the registry, and triggered a build which should push the new image to the registry.

Next, we need to deploy the app to Azure. We will use the following command:

```
$ ci-plumber azure deploy
Service plan [myServicePlan]:
App name [myApp-159731108]:
[16:08:33] Creating app service plan
[16:08:43] Creating web app. This may take a while...
[16:09:20] Assigning managed identity
[16:09:28] Retrieving subscription ID
[16:09:31] Granting permission to access container registry
[16:09:42] Configuring app to use managed identity
[16:09:47] Deploying
[16:09:56] Deployed to https://myapp-159731108.azurewebsites.net
           It may take a moment to come online
```

As can be seen, the app is now deployed to Azure. We might also want to deploy a database for the project. We can use the following command:

```
$ ci-plumber azure create-db
Name [my-database-779171168]:
Admin username [myadmin]:
Admin password:
Repeat for confirmation:
[16:12:32] Initialising Server. This may take a while...
[16:15:41] Created Database
           The credentials have been written to maria.env
```

Similarly to the Openshift example, the details of the database are written to `maria.env`. This file uses standard syntax for environment variables, so it can be easily loaded using whatever method you prefer. For example, dotenv in Python.

References

[1] D. L. Frederiksen and A. Brem, “How do entrepreneurs think they create value? a scientific reflection of eric ries’ lean startup approach,” *International Entrepreneurship and Management Journal*, vol. 13, no. 1, pp. 169–189, Mar. 1, 2017, ISSN: 1555-1938. DOI: `10.1007/s11365-016-0411-x`. [Online]. Available: `https://doi.org/10.1007/s11365-016-0411-x` (visited on 09/02/2021).

[2] R. F. Bortolini, M. Nogueira Cortimiglia, A. d. M. F. Danilevicz, and A. Ghezzi, “Lean startup: A comprehensive historical review,” *Management Decision*, vol. 59, no. 8, pp. 1765–1783, Jan. 1, 2018, Publisher: Emerald Publishing Limited, ISSN: 0025-1747. DOI: `10.1108/MD-07-2017-0663`. [Online]. Available: `https://doi.org/10.1108/MD-07-2017-0663` (visited on 09/19/2021).

[3] V. Debroy, S. Miller, and L. Brimble, “Building lean continuous integration and delivery pipelines by applying DevOps principles: A case study at varidesk,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 26, 2018, pp. 851–856, ISBN: 978-1-4503-5573-5. DOI: `10.1145/3236024.3275528`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3236024.3275528` (visited on 09/19/2021).

[4] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, “Quality and productivity outcomes relating to continuous integration in GitHub,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery, Aug. 30, 2015, pp. 805–816, ISBN: 978-1-4503-3675-8. DOI: `10.1145/2786805.2786850`. [Online]. Available: `https://doi.org/10.1145/2786805.2786850` (visited on 09/19/2021).

[5] (2021). “What is DevOps? research and solutions,” Google Cloud, [Online]. Available: `https://cloud.google.com/devops` (visited on 09/19/2021).

[6] N. Forsgren, J. Humble, G. Kim, B. Washington, N. Kaul, and D. Smith, “ROI of DevOps transformation: How to quantify the impact of your modernization initiatives,” Google, Whitepaper, 2020. [Online]. Available: `https://cloud.google.com/resources/roi-of-devops-transformation-whitepaper` (visited on 09/19/2021).

[7] R. M. Harden, “What is a spiral curriculum?” *Medical teacher*, vol. 21, no. 2, pp. 141–143, 1999, Publisher: Taylor & Francis, ISSN: 0142-159X. DOI: `10.1080/01421599979752`. [Online]. Available: `https://doi.org/10.1080/01421599979752`.

[8] G. Pocentek and M. Mäenpää. (2018). “GitLab CLI usage,” python-gitlab 2.10.1 documentation, [Online]. Available: https://python-gitlab.readthedocs.io/en/stable/cli-usage.html (visited on 10/03/2021).

[9] Red Hat. (2021). “Get started with the CLI,” CLI Reference | OpenShift Container Platform 3.9, [Online]. Available: https://docs.openshift.com/container-platform/3.9/cli_reference/get_started_cli.html (visited on 10/03/2021).

[10] M. Budden. (2021). “CI plumber,” [Online]. Available: https://milesbudden.com/ci-plumber/ (visited on 10/24/2021).

[11] (Aug. 9, 2021). “Overview of kubectl,” Kubernetes. Section: docs, [Online]. Available: https://kubernetes.io/docs/reference/kubectl/overview/ (visited on 10/03/2021).

[12] Microsoft Azure, *Microsoft azure CLI*, original-date: 2016-02-04T00:21:51Z, Oct. 16, 2021. [Online]. Available: https://github.com/Azure/azure-cli (visited on 10/16/2021).

[13] Netlify, *Framework-info*, original-date: 2020-07-29T12:13:10Z, Oct. 15, 2021. [Online]. Available: https://github.com/netlify/framework-info (visited on 10/16/2021).

[14] K. Waters, “Prioritization using MoSCoW,” *Agile Planning*, vol. 12, p. 31, 2009.

[15] A. Cockburn. (Sep. 11, 2013). “Hexagonal architecture,” Alistair Cockburn, [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/ (visited on 10/03/2021).

[16] Python Packaging Authority. (2020). “Packaging namespace packages,” Python Packaging User Guide, [Online]. Available: https://packaging.python.org/guides/packaging-namespace-packages/ (visited on 10/10/2021).

[17] Python Packaging Authority. (2021). “Entry points,” setuptools 58.2.0 documentation, [Online]. Available: https://setuptools.pypa.io/en/latest/userguide/entry_point.html (visited on 10/10/2021).

[18] The Linux Foundation. (2020). “Open container initiative,” [Online]. Available: https://opencontainers.org/ (visited on 10/10/2021).

[19] *Oci-image-tool*, original-date: 2016-09-08T20:19:15Z, Sep. 19, 2021. [Online]. Available: https://github.com/opencontainers/image-tools (visited on 10/10/2021).

[20] *Oci-runtime-tool*, original-date: 2016-01-13T20:10:21Z, Oct. 3, 2021. [Online]. Available: https://github.com/opencontainers/runtime-tools (visited on 10/10/2021).

[21] (Oct. 5, 2020). “Python release python 3.9.0,” Python.org, [Online]. Available: `https://www.python.org/downloads/release/python-390/` (visited on 10/10/2021).

[22] (Oct. 8, 2021). “Handbook,” The TypeScript Handbook. in collab. with O. Therox, S. Sarker, T. Jiuding, and A. Savath, [Online]. Available: `https://www.typescriptlang.org/docs/handbook/intro.html` (visited on 10/10/2021).

[23] *$ click_*, original-date: 2014-04-24T09:52:19Z, Oct. 10, 2021. [Online]. Available: `https://github.com/pallets/click` (visited on 10/10/2021).

[24] S. Ramírez, *Tiangolo/typer*, original-date: 2019-12-24T12:24:11Z, Oct. 10, 2021. [Online]. Available: `https://github.com/tiangolo/typer` (visited on 10/10/2021).

[25] *Python-gitlab/python-gitlab*, original-date: 2013-02-07T17:23:16Z, Oct. 15, 2021. [Online]. Available: `https://github.com/python-gitlab/python-gitlab` (visited on 10/16/2021).

[26] *Gitpython-developers/GitPython*, original-date: 2010-11-30T17:34:03Z, Oct. 16, 2021. [Online]. Available: `https://github.com/gitpython-developers/GitPython` (visited on 10/16/2021).

[27] D. Taylor. (2019). “Building an open-source and cross-platform azure CLI with python,” Python.org, [Online]. Available: `https://www.python.org/success-stories/building-an-open-source-and-cross-platform-azure-cli-with-python/` (visited on 10/16/2021).

[28] W. McGugan, *Rich library*, original-date: 2019-11-10T15:28:09Z, Oct. 16, 2021. [Online]. Available: `https://github.com/willmcgugan/rich` (visited on 10/16/2021).

[29] V. Supalov. (Apr. 26, 2018). “Should you run your database in docker?” vsupalov.com, [Online]. Available: `https://vsupalov.com/database-in-docker/` (visited on 11/02/2021).

[30] *Kaniko - build images in kubernetes*, original-date: 2018-01-29T17:53:54Z, Nov. 3, 2021. [Online]. Available: `https://github.com/GoogleContainerTools/kaniko` (visited on 11/03/2021).

[31] S. Selhorn and M. Amirault. (Oct. 20, 2021). “Use docker to build docker images,” GitLab Docs, [Online]. Available: `https://docs.gitlab.com/ee/ci/docker/using_docker_build.html` (visited on 11/03/2021).

[32] S. Selhorn. (Oct. 13, 2021). “Configure GitLab as an OAuth 2.0 authentication identity provider,” GitLab Docs, [Online]. Available: `https://docs.gitlab.com/ee/integration/oauth_provider.html` (visited on 11/03/2021).

[33] Red Hat. (Aug. 26, 2021). “Using image pull secrets - managing images,” OpenShift Container Platform 4.6, [Online]. Available: `https://docs.openshift.com/container-platform/4.6/openshift_images/managing_images/using-image-pull-secrets.html#images-pulling-from-private-registries_using-image-pull-secrets` (visited on 11/03/2021).

[34] (Jun. 10, 2021). “Manage resource groups - azure resource manager,” Azure portal. in collab. with J. Gao, K. Sharkey, R. Lyon, D. Coulter, M. Singletary, M. Sahbi, and T. FitzMacken, [Online]. Available: `https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal` (visited on 11/03/2021).

[35] D. North. (Jul. 26, 2021). “We need to talk about testing,” DAN NORTH & ASSOCIATES LTD. Section: blog, [Online]. Available: `https://dannorth.net/2021/07/26/we-need-to-talk-about-testing/` (visited on 11/02/2021).

[36] D. North, “MiXiT - talk the three ages of innovation,” May 15, 2015, [Online]. Available: `https://mixitconf.org/2015/dan-north-the-three-ages-of-innovation` (visited on 11/03/2021).