Imperial College London

Bloomberg L.P.

# Industrial Placement 2019 Placement Report

## MEng. Computing 4

*Student Name:*
Shashwat Dalal
*Tutor:*
Marc Deisenroth

*Placement Manager:*
Chandan Singh
csingh43@bloomberg.net

October 3, 2019

**Acknowledgements**

I would like to give my thanks to the London Build Solution team for welcoming me to Bloomberg, and for making my placement enjoyable. I would like to thank Chandan and Benjamin in particular for their patience and for their invaluable guidance throughout the course of the placement.

# Contents

# 1 Company Overview

## 1.1 Profile

Bloomberg was started by Michael Bloomberg in 1981 following his departure from Salomon Brothers, and set out to build a computerized system that would provide market data, and perform financial calculations for Wall-Street firms. Since, Bloomberg has expanded globally and has extended its services to include a news channel, a luxury magazine, a venture capitalism business, an electronic brokerage, and a communications medium.

Bloomberg describes itself as the central nervous system of the financial industry. Its main product, the Bloomberg Terminal, is a desktop application that provides access to Bloomberg's software, data and media. Today the terminal is ubiquitous amongst traders as many of the world's most important trades are conducted through the terminal; often requiring traders and financiers to be within the Bloomberg ecosystem.

## 1.2 Philanthropy

A major part of Bloomberg's culture is its commitment to philanthropy. In 2018, Bloomberg invested 767 million dollars across its philanthropy projects in 510 cities.[4] As a placement student, I was fortunate enough to be involved in Bloomberg's day of service. We spent the day annotating maps for the `openstreetmap` initiative. The areas we annotated were of interest to Médecins Sans Frontières, Doctors Without Borders, as they were planning on launching projects in these areas.[9][8]

## 1.3 DevX

I completed my placement under Bloomberg's DevX, developer experience, division. This is Bloomberg's engineering division that focuses on building tools and work-flows that help make developers happier and more productive. Over the last few years this division has consolidated engineers across application and infrastructure divisions to help present a unified approach to how Bloomberg does software engineering.

## 1.4 Build Solutions London

The team I was part of focused on build and integration tooling for software at Bloomberg; the team's goal can be summarized by "Make everyone's builds go faster". To achieve this vision the team was tailoring BuildStream, an open source integration tool used by the likes of GNOME, for Bloomberg.[7] The team's technology stack consisted of:

- `Python3` for development
- `Docker` for packaging
- `Jenkins` and `GitLab CI` for testing
- `Ansible` for deployments
- `Graphana` and `Kafka` for monitoring and alerts

The team frequently alluded to the *Unix Philosophy* and the *Zen of Python* when justifying design choices, and followed a Scrum methodology with variable-length sprints.[12][13]

# 2 Project

## 2.1 BuildStream Overview

"BuildStream is a Free Software tool for building/integrating software stacks. BuildStream supports multiple build-systems (e.g. autotools, cmake, cpan, distutils, make, meson, qmake), and can create outputs in a range of formats (e.g. debian packages, flatpak runtimes, sysroots, system images) for multiple platforms and chipsets."[11] BuildStream accomplishes this by scheduling a dependency graph of smaller build and integration tasks, referred to as Elements. Elements expose declarative configuration options in `yaml` for users to specify for a certain flavor of task.

To speed build and integration processes, BuildStream utilizes a content-addressable cache, that can be either local or distributed, which keeps track of the mapping between elements and their respective artifacts. This helps prevent duplicate rebuilds, and makes incremental builds possible. BuildStream additionally heavily emphasizes reproducibility. Sandboxed environments are used to ensure elements execute in isolated file-systems. As a consequence, for the most part, this helps ensure artifacts are byte-by-byte reproducible.

## 2.2 Docker Image Element

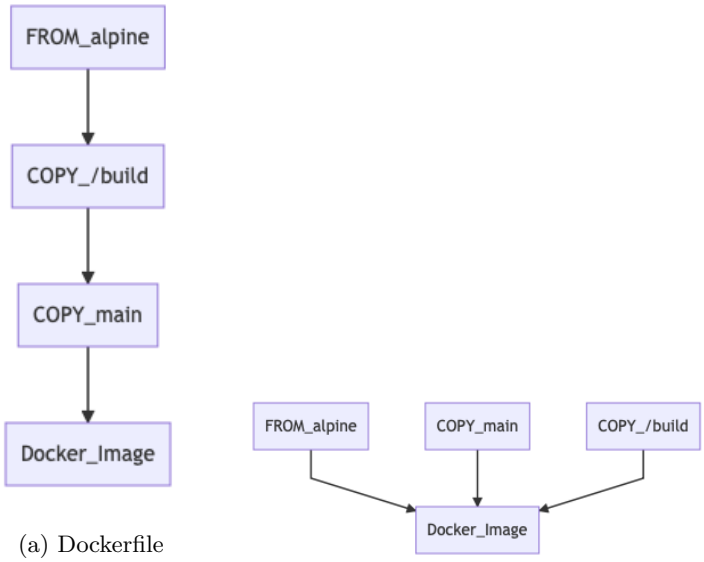
I was tasked with building an Element that would generate a Docker image artifact given dependent file-system(s). The following section summarizes the discussion, steps, and choices around the design process of this element.

### 2.2.1 Motivation

Dockerfiles internally represents dependencies as a linked-list. This is because commands are ordered line-by-line forcing each command to be dependent on the lines before it. BuildStream, on the other hand, internally represents dependencies as a directed acyclic graph; a more expressive model. The unavoidable dependencies that come with the linked-list model need not coincide with true software dependencies, and as such creating Docker images through BuildStream is often more precise and efficient. Take the following Dockerfile as an example.

```
FROM alpine
WORKDIR /main-app
COPY /build ./build
COPY main .
ENTRYPOINT /main-app/main
EXPOSE 8080
```

Figure 1 demonstrates the internal dependency representation of the above Dockerfile juxtaposed with the true dependencies of each of its commands. As $\rightarrow$, the relation for dependencies, is transitive, `FROM_alpine` $\rightarrow$ `Docker_Image`, `COPY_build` $\rightarrow$ `Docker_Image`, `COPY_main` $\rightarrow$ `Docker_Image` are valid relations in figure 1(a). Figure 1(a) in addition also has `FROM_alpine` $\rightarrow$ `COPY_build`, `COPY_build` $\rightarrow$ `COPY_main` which are the unnecessary and unavoidable dependencies. This has the consequence that if changes are made to the base `Alpine` image, `COPY_build` and `COPY_main` will have to be rebuilt even though they do not depend on `Alpine`. If we use a however use BuildStream to build Docker images, we can avoid unnecessary dependencies which in turn means we can avoid unnecessary rebuilds.




(a) Dockerfile Dependency Representation

(b) True Dependency Graph

Figure 1: Implied Dependency of Dockerfile

The second motivation of having native Docker support is that we can target a previous build and produce a functionally equivalent artifact which can be run on a container orchestration platform. This has the effect of decoupling builds with the target deployment platform. Once teams are using BuildStream, teams can easily containerize their builds and as a result easily migrate to deployments on container orchestration platforms. There are also benefits of bringing container builds into the same build ecosystem. If a container has an internally-developed dependency that changes, we can automatically trigger a rebuild for that container. As such, it would be possible to now use BuildStream to create an automated work-flow that would ensure that all containers used within the organizations are kept up-to-date and are synchronized with their non-container counterparts.

### 2.2.2 Design Criteria

Foremost, it was important to outline the solution characteristics to firstly judge possible designs, and secondly to prevent over-engineering. As such, *in decreasing importance*, the criteria agreed upon were:

1. Create images that the Docker daemon can run.
2. Create reproducible images. The digests of images from identical builds should be equivalent.
3. Maximize layer reuse. This maximizes efficiency of Docker pull by maximizing space efficiency.
4. Minimize file duplication
5. Maximize probability of BuildStream cache hit
6. Maximize ease of use for user. This is achieved by minimizing number of elements that need to be created, and through minimizing number of dependencies between elements.

### 2.2.3 Assumptions

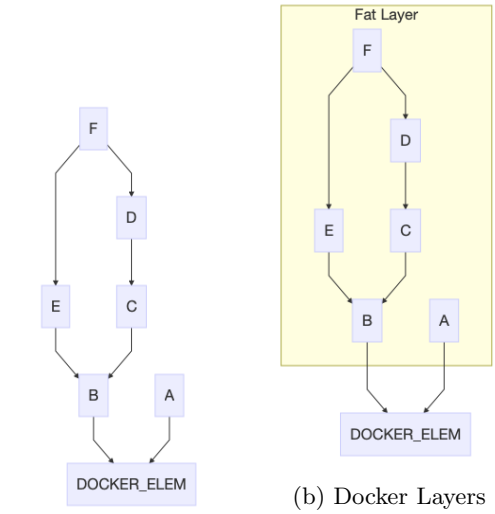
As this was my first exposure to BuildStream, understanding BuildStream's design choices and subtleties were important in effectively completing this task. The following summarize the applicable guarantees BuildStream provided.

- Each element must produce one one artifact

- An element can access its transitive dependencies
- Circular dependencies are not allowed
- When an element gets a list of it's dependencies, order is depended upon:
  $E_1 > E_2 \iff (E_1 \to E_2) \vee (\neg(E_2 \to E_1) \wedge E_1.name > E_2.name)$ (2.1)
  where $x \to y$ means `x` depends on `y`, and the relation is transitively closed.

#### 2.2.4 Designs

**1. Fat Layer (No Layering)**




(a) Dependency Graph

(b) Docker Layers Created from Dependency Graph

Figure 2: Fat Layering

Given a dependency graph, build a single layer Docker image which consists of the union of the artifacts of the immediate dependencies of the Docker image element. For example, in figure 2, the single layer would comprise of a file system that corresponds to the union of the artifacts produced by element B and A. This is the trivial way of ensuring the image is valid as all elements are in the same layer.

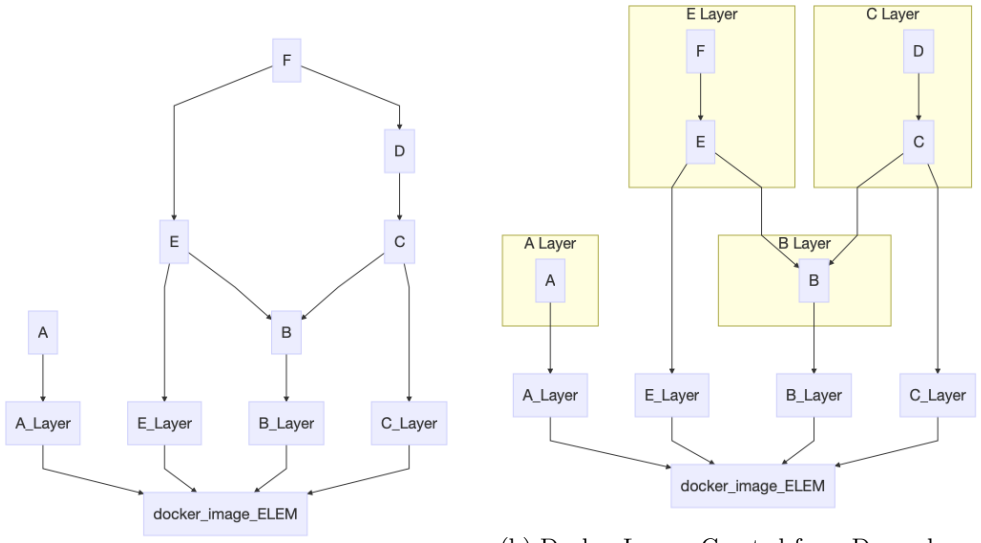
**Advantages**

- Minimal computational overhead, as there is no need to recurse into deeper dependencies.
- Users do not need to add layer Elements
- Minimal Element logic

**Disadvantages**

- Does not decompose images into layers. This is bad as layers could be cached in BuildStream and reused when building other images.
- Lack of layers means that when images are stored on a registry, Docker cannot de-duplicate similar file-systems; leads to space inefficiency.

**2. User Defined Layer Elements (Manual Layering)**




(a) Dependency Graph

(b) Docker Layers Created from Dependency Graph

Figure 3: Manual Layering

In this approach, we introduce a `docker_layer` element. The `docker_image` element can only depend on `docker_layer` elements. Users manually insert `docker_layer` elements where they would like to create layers. There is no need to modify the underlying dependency graph as `docker_layer` elements are tangentially created. Once, `docker_layer` elements are created, we compute the mapping from elements to layers through a breath-first traversal starting from `docker_element`. Figure 3 shows the effect of adding `docker_layer` elements to an underlying dependency graph.

As layers `A, C, E` are topologically equivalent, from assumption 2.1 we know that their layering ordering would depend upon the element names. Furthermore from 2.1, we know that `E, C` will always come before `B`. There are thus 8 possible ordering of layers we could get from clustering elements using a breath-first traversal.

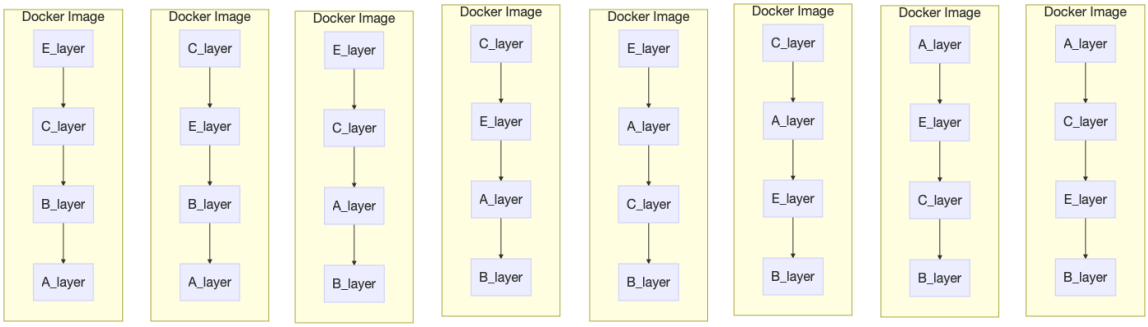



Figure 4: Sample Space of Layer Ordering Generated from Figure 3

Docker reuses layers between images if from the bottom they share the same layers. If two images only have one layer that is different, sandwiched between layers that are identical in both images, layers after the distinguishing layer cannot be shared.[2] Due to this reason, we want a layer ordering with the most commonly used layers at the bottom, and the most frequently changing layers at the top.

For sake of argument, let us assume that we want `E_layer` and `C_layer` at the bottom as they are reused between images, and `A_layer` at the top as it consists of say only configuration files that differs between images. Looking at figure 4, we see that only half of these orders are ideal, let us call the other four orders inefficient layer ordering. We would probably need to use some combination of depth, number of reverse dependencies, and size of layer as a metric to gauge which layers would probably be reused between images and prune the sample space accordingly.

**Advantages**

- Decomposes images into layers. Layers are cached by BuildStream's cache and can be reused between builds. Layers could be tagged for revision purpose which allows for individual layers to be rebuilt upon expiry instead of having to rebuilding an entire image after expiry.
- Gives users fine-grain control over layers which can allows users to optimize based on needs.
- No duplication of elements between layers.
- Layer elements are created tangentially so there would be no need to alter the underlying dependency graph. Any pre-existing BuildStream project could be "annotated" to generate docker images with specified layers.

**Disadvantages**

- Gives users fine-grain control over layers which may be giving users too much control
- Users need to create `docker_layer` elements
- Larger number of elements to keep track of
- Need to recurse into transitive-dependencies to build the BFS tree. This will worsen the complexity of this build step.
- `layer_element` artifacts will be a tarball of the artifact produced by its direct element. This leads to BuildStream cache duplication.
- Can end up with non-ideal layering .

**Additional Notes**

This is how `Bazel`, Google's open source build system, implements layers for python and java rules. In addition, each '`layer` element' has a filter attribute. Lastly, users have a finer control of layering as they also define the ordering of layers. [1]

### 2.5. Shallow Layering (Depth=1 Layering)

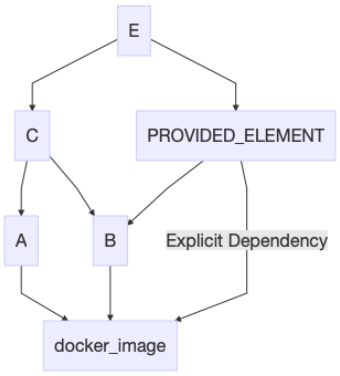



Figure 5: Dependency Graph

In this design, we create layers only based on immediate dependencies. For elements users want to explicitly create a layer out of, we shall call them provided elements, users would have to manually add the

provided element as a dependency of the `docker_image` element in order to create a layer for it. Shared Sub-dependencies are resolved by clustering the dependency tree through a DFS traversal with the `docker_image` element as the root. This will generate valid images as deeper provided-elements will be traversed first due to property 2.1.

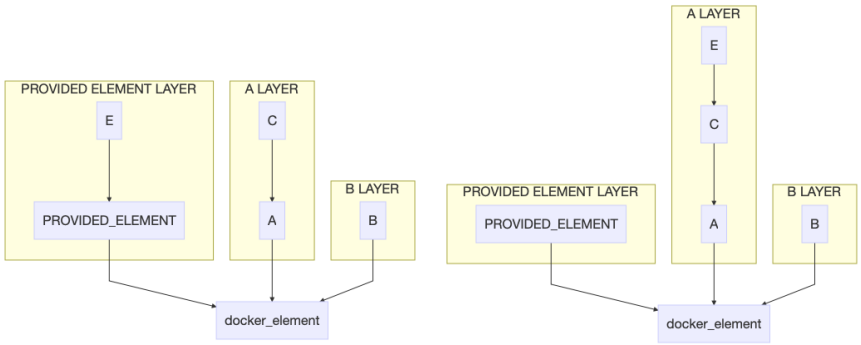



Figure 6: Docker Layers Created from Clustering Dependency Graph

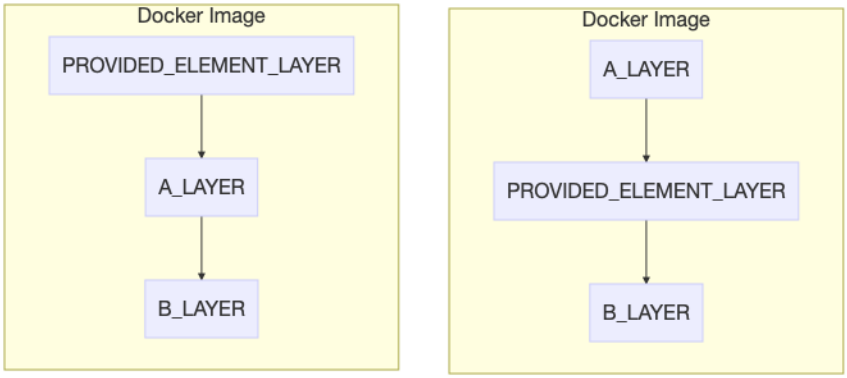



Figure 7: Docker Layer Ordering

Even though valid images would be generated, looking at figure 7, we see that there were two ways of possibly ordering the layers. This presents the same issue of inefficient layering ordering suggested previously.

**Advantages**

- Decompose images into layers, which could hypothetically be globally cached. Global layers could be tagged for revision purposes and reused when building other images.
- Gives users fine-grain control over complexity of layers
- Allows users to use meta-elements such as 'stack' / 'compose'/ 'filter' elements to either squash or merge elements to form a single layer. (Unix brownie points)
- By creating explicit dependencies between leaf `provided elements` and `docker_image` elements, you can ensure that if the same `provided elements` are used between projects, the same layer can be used.
- Additional elements will not be needed.

**Disadvantages**

- Need to recurse into sub dependencies to build DFS Tree. This will worsen the complexity of this build step.

- Layers will not be cached within BuildStream as layers are not generated as artifacts
- In some cases where there are multiple independent dependencies (e.g. $B \rightarrow A, C \rightarrow A, D \rightarrow A$), the ordering may cause most frequently changed layers, e.g. configuration files, to be the base layer of the image, which is not ideal. Ideally we would like to have layers that change the most frequently to at the top.
- Users need to add additional dependencies if they want to explicitly make a layer.

**3. Popularity Contest Winner Layers (Automatic Layering)**
This approach is based on earlier work by Graham Christensen on Nix.[10] The idea here is to automatically generate layers based on weighting every element in the dependency tree. In this approach the user can pass in `max_layers` as a parameter in `docker_element`'s configuration. Weights are allocated so that deeper, and more referred nodes in build graph have higher weights, and we create layers out the top `max_layers` weighted elements. We then create `max_layers - 1` layers for the highest weighted elements, and merge the rest into a single layer.

**Advantages**

- Creates layers automatically and to maximize layer re-use without need for user definition.
- Minimal / optional user input to manage complexity of image.
- If `max_layers = 1` then this approach is the same as the fat-layer approach, hence in a way this is a generalization of the fat-layer design.

**Disadvantages**

- Calculating weights add to build time
- Users have no control over order of images or which elements will belong to which layer

#### 2.2.5 Evaluation

After evaluating the design choices, the element was finally implemented using the 'Shallow Layering' approach. This was for several reasons. For one, it provided a deterministic way, assuming element names do not change, of creating a layered Docker image. Secondly, it gave users complete control over the number of layers, and gave users control over elements they wanted to explicitly make a layer out of. This is particularly useful if we are using an element that generates a base platform, e.g. Alpine, and we then run build instructions on this platform. In such a case it would be important to create a layer out of the element that generates, say Alpine, as other build processes might require the same base.

It is important to address the short-comings of this approach. As mentioned, this approach does not allow users to specify the order of the layers specified. This could potentially lead to layering that is not space-optimal. One way to artificially dictate ordering would be to add redundant dependencies between elements. This is so that based on the guarantee of 2.1, one element will strictly come before another. This is not ideal from a user's perspective, the sixth point of the solution characteristic, but allows for potential inefficiencies to be corrected. Lastly, this design is still extensible, and the ordering defined in 2.1, can be augmented in the future to incorporate some of the ideas in the 'Popularity Contest Winner Layers' approach. That being said, given how this was a first implementation of the element, it was important to not over-engineer and this approach struck a good balance between complexity and covering the design criteria.

# 3 Reflection

## 3.1 Relevance to Courses in the Department of Computing

### 3.1.1 211 Operating Systems

As my project was concerned with a sandboxed build-system and Docker, I often had to recall OS concepts that were taught during 211. A few of the concepts include namespaces, cgroups, kernel-user separation, and file-systems. At one point I even brought my OS notes in as I needed them to understand how Docker leverages namespsaces and cgroups for virtualization.

### 3.1.2 220 Software Engineering Design

The sections on automated testing and continuous integration / deployment of this course were crucial in giving me context of the engineering tools and work-flows at Bloomberg.

### 3.1.3 150 Graphs and Algorithms

As shown in section 2, I had to design algorithms that clustered direct acyclic graphs. The tutorials and exercises that accompanied the course were particularly important in building my intuition for designing the described algorithms.

## 3.2 Challenges

Getting to grips with BuildStream's medium-to-large code-base was an initial challenge.[7] Taking the time to ask colleagues and the reading the open source mailing lists, helped me understand the rationale behind some of the design choices.[6] Occasionally I would be asked to compare my implementation of a feature against a similar feature of another open source project. An example of this would be when I had to implement white-out file semantics for the element for loading a Docker image into BuildStream's sandbox and was then asked to compare my implementation with the Docker deamon's implementation. As such, a challenge I had to frequently overcome was the challenge of comprehending other large open source code-bases.

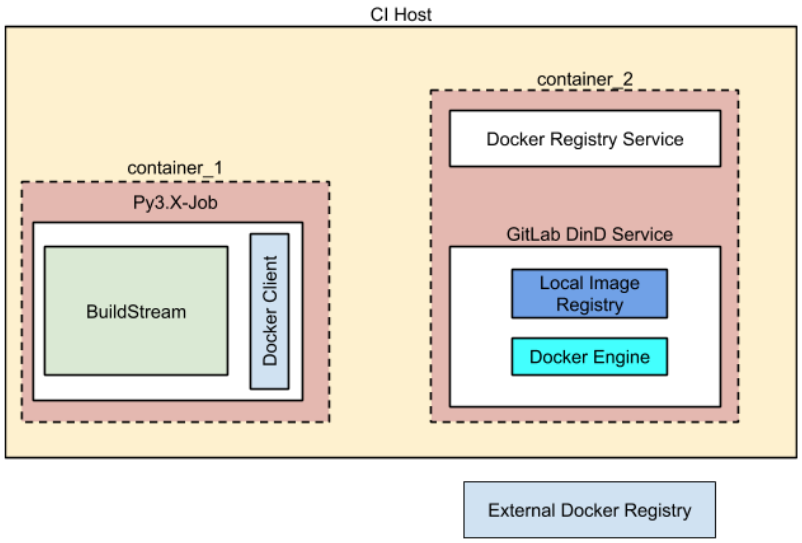



Figure 8: CI Setup

Another challenge I faced was with continuous integration configuration. Given how the element I was building involved generating Docker images, the integration tests I wrote required a Docker deamon to

check whether the image generated would be acceptable. Additionally it required a test Docker registry so that images could be pulled and pushed. This environment was easy to reproduce when running the test locally. However as our CI pipeline tests also ran in Docker containers, it was a challenge to setup both a Docker deamon and a Docker registry while also running our `pytest` suite from a container. After some experimentation, we were finally able to achieve reproducible and consistent environments locally and in CI using the architecture shown in figure 8. Given how this would have been an easy task if I had previous CI configurations and inter-container networking experience, it would have been nice if the Department of Computing gave a seminar regarding CI configuration.

## 3.3 Benefits of the Placement

Working at Bloomberg, a company that has a very strong engineering culture, has made me greatly appreciate the processes and operations behind shipping software for production. In addition to learning about the technologies associated with my project, I've also learned the importance of a clean commit history, good integration tests, thorough code-reviewing, logging, and monitoring. Bloomberg emphasizes inter-team collaboration, and as such I frequently took part in show-and-tells, hackathons, and guild talks. This broadened my perspective on engineering at Bloomberg, and was able to learn a bit about the overall technology architecture at Bloomberg, and Bloomberg's unified effort to phase out legacy infrastructure.

## 3.4 Ethical Issues

### 3.4.1 Respect for Others

The team was from all over the world, and culturally very knowledgeable. This made discussions, albeit not always about the work at hand, incredible interesting, and I always learned something new. I felt mutual respect within the team, felt comfortable expressing my opinions, and felt that criticism was always constructive.

### 3.4.2 Competence

Quite often the task given initially seemed straightforward, until a blocker was uncovered that brought about a week-long detour. As such tasks have often unearthed and solved previously unknown issues, but at the expense of taking longer to complete. 1-1s with my team leader were often centered around the topic of balancing thoroughly solving the secondary task without losing focus of the primary task. An example of this is when I had to package our system using the element I wrote. This task unearthed the fact that an existing Element lacked critical features, and the main system needed some permissioning logic to be changed. This detour was long, but yielded a more complete solution. Throughout the tasks, I used a scrum board to plan my work and give my mentor and team leader transparency on the state of my work.

### 3.4.3 Integrity

Given how the project is something that will eventually impact the system through which most code in Bloomberg will be packaged to production, writing resilient, efficient, and future-proof code is paramount. One example of this is writing clear documentation so that the work I've done can timelessly be utilized across the organization. In addition it reflects Bloomberg's commitment to open source and support for creating tools for even external entities. In addition, given the reach of this project, I have given equal importance to automated tests and feature implementation. This is so that a future developer, either internal or external, has the confidence of editing the source code.

### 3.4.4 Environmental

The Bloomberg office is the world's most sustainable office building.[3] There is heavy emphasis on recycling.

# References


[1] Bazel Image Build Rules
https://github.com/bazelbuild/rules_docker

[2] Best Practices for Writing Dockerfiles, Leverage Build Cache
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#leverage-build-cache

[3] Bloomberg's New European Headquarters Rated World's Most Sustainable Office Building
https://www.bloomberg.com/company/announcements/bloomberg-most-sustainable-office-building/

[4] Bloomberg Philanthropies Annual Report 2019
https://annualreport.bloomberg.org

[5] Build systems a la carte - Microsoft Reseach
https://www.microsoft.com/en-us/research/publication/build-systems-la-carte/

[6] BuildStream Mailing List
https://mail.gnome.org/archives/buildstream-list/

[7] BuildStream Source
https://gitlab.com/buildStream/buildstream/

[8] Doctors Without Borders
https://www.msf.org.uk/

[9] OpenStreetMap
https://www.openstreetmap.org

[10] Optimising Docker Layers for Better Caching with Nix
https://grahamc.com/blog/nix-and-layered-docker-images

[11] What is BulidStream
https://docs.buildstream.build/main_about.html

[12] Unix Philosophy
https://en.wikipedia.org/wiki/Unix_philosophy

[13] Zen of Python
https://www.python.org/dev/peps/pep-0020/