May15-17

Lighthouse Project Plan

Version 2.0

Caleb Brose, Chris Fogerty, Nick Miller, Rob Sheehy, Zach Taylor
November 11, 2014

CONTENTS

1 Problem Statement

Within the past year, Docker has emerged as an execution and deployment environment for large-scale, distributed applications. In its current state, the tooling surrounding Docker is still rapidly evolving and many systems are immature and not production ready. For organizations hosting many applications across multiple cloud platforms, the manual management of Docker hosts and applications has proved to be challenging and time consuming, especially as the number of hosts to monitor grows. The goal for Lighthouse is to provide an open source application that centralizes the management and monitoring of Docker hosts across a variety of cloud platforms, including the Docker containers executed within them.

2 Definitions and Common Abbreviations

Term	Definition
Docker	An open source platform used for creating, packaging, and shipping, and running applications.
Image	A read only file system used by Docker which contains applications and other files.
Container	A "running" image which is writeable. Containers are primarily used to run the applications that are being developed.
Registry	An image storage service.
GCE ***Google Compute Engine***	A virtual machine (VM) hosting service by Google which may host Docker instances.
AWS ***Amazon Web Service***	A virtual machine hosting service by Amazon which may host Docker instances.
JSON ***JavaScript Object Notation***	A data exchange format which is the primary mode of communication between the frontend and backend applications.

Table 1 Common terms and their definitions

3 Deliverables

Lighthouse consists of the following modules:

3.1 Backend

The backend system will be capable of interfacing with an existing instance of Docker. It will communicate with Docker instances via Docker's REST API, and it should be capable of:

- Routing all valid Docker API calls through itself to the appropriate Docker instance.
- Authenticating users, using per-instance permissions, and blocking unauthorized requests.
- Extending Docker's functionality by intercepting requests to Docker and calling internal functions, such as logging and history.

3.2 Frontend

The frontend will be a web application capable of utilizing the backend to manage Docker instances. It will use JavaScript and JavaScript libraries such as Angular, and should be able to:

- Perform basic container management actions such as Create, Stop, Remove, Update, and Pause
- View past containers run in the Docker instance. That is, view a history of each application running on the instance
- Log in and out of the backend application to authenticate frontend users.

3.3 Documentation

The backend API will be well documented in order to accommodate users that wish to automate actions, create their own frontend applications, or extend our frontend with new functionality.

4 SPECIFICATIONS

Specification	Definition	Value	Verification Method
Easy to Setup & Learn	We want the server software to be easy to understand and setup. This is supposed to be a tool that not only helps with managing Docker, but can be a tool that helps teach the potential and importance of Docker. This means that we should have a simple/efficient setup process with high quality documentation.	med	The server shouldn't take more than 10 minutes to set up locally and play around with, given that the user has all the server dependencies preinstalled.
Multi-Platform & "Docker-izable"	Our server software should be highly compatible with running inside a Docker container, seeing as our project is all about managing Docker containers. Allowing our project to run inside Docker also should give us the advantage of running across multiple platforms as well.	high	Our Docker-ized server software should have 99% of the functionality as our standalone server software. We should also support 100% of platforms that support running Docker.
Docker Image Efficiency	The Docker image to run our server should be extremely efficient and lightweight to store inside any Docker registry.	low	Our server Docker image should not be any bigger than 700 MBs.
Hosted Site Uptime	The hosted site must be consistently available to users, such that anyone hosting our site should be able to keep a reasonable uptime with our server software.	med	Over the span of one month our server software should be up for at least 95% of that time, without any intervention of from system admins.

Specification	Definition	Value	Verification Method
Modern Project	The project should be using the latest and greatest software tools and languages. Our project should include and practice new and growing trends in the software industry that will make our project and software more desirable to emerging startups, but should still catch the attention to larger corporate software companies.	low	Our project should include at least two new emerging languages or libraries in the software industry.

Table 2 Specifications

5 Concept Sketch/Mockup

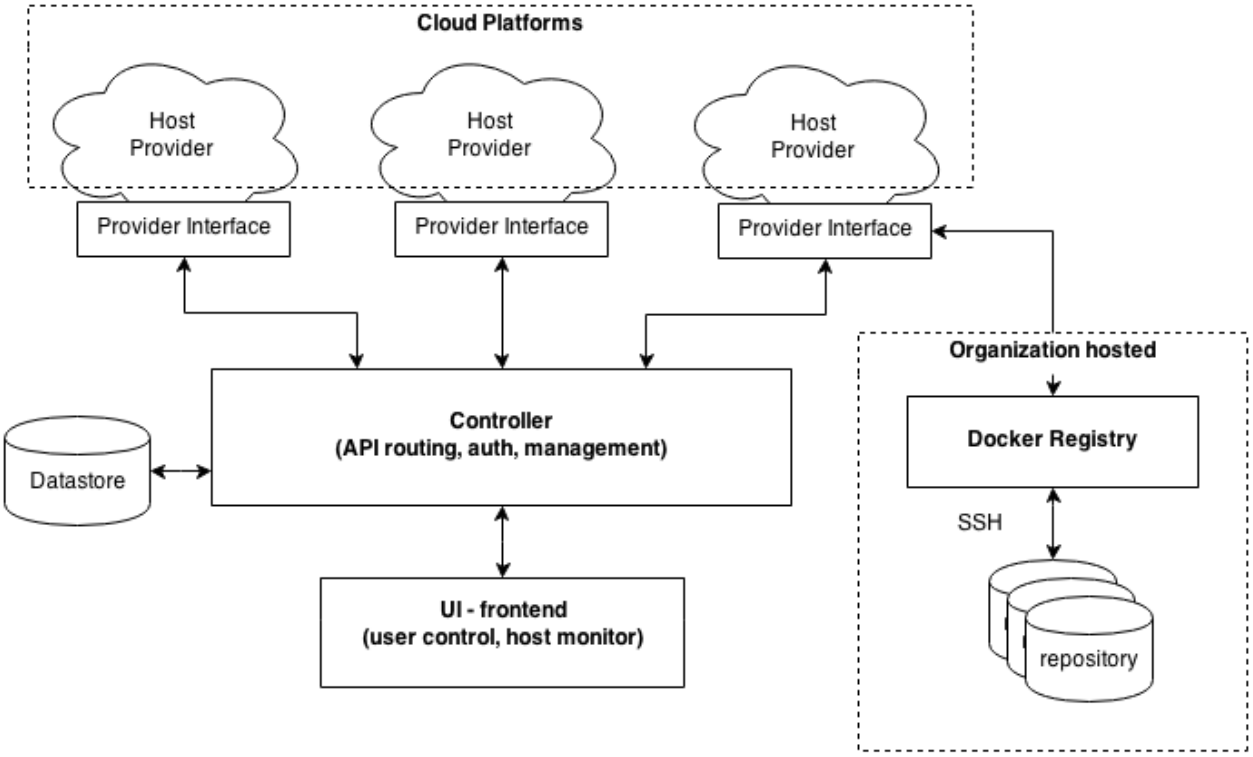

Figure 1 Lighthouse conceptual diagram

The overall architecture of Lighthouse is built upon the common client-server pattern. Users interact with the system via a single-page web application written in JavaScript that is loaded once into their browser, with subsequent system requests made via a REST API over HTTPS, utilizing JSON as the data interchange format.

Requests received from the web interface are routed through authentication and API control. Our API is implemented as a superset of the standard Docker Remote API, allowing users to send specific Docker requests to their target platform, as well as add new platforms, users, and other infrastructure configuration.

Host providers manage the actual startup and application runtime, but are controlled via the Lighthouse main controller. Provider interfaces are installed on each provider as part of the Lighthouse application and define a common interface for communication between the host network and Lighthouse.

Docker host providers are instructed to pull new images from the registry on application deployment. Note that the Docker registry and associated code repositories are defined and configured by the organization and is outside the scope of the Lighthouse system. The registry defined REST API allows Lighthouse to communicate with it.

6 User Interface Description

The primary user interface will be a front-end JavaScript web application. Users will authenticate to the back-end controller module using this web interface. This will provide an easy way for the user to interact with the Lighthouse service.

6.1 Instances

Our goal for the user interface is to allow users to accomplish the success scenarios outlined in the project's requirements.

When a user first navigates to the web application, they will immediately be prompted for a login (username and password. Upon successful authentication with the server, the user will be redirected to the instance index page shown below.

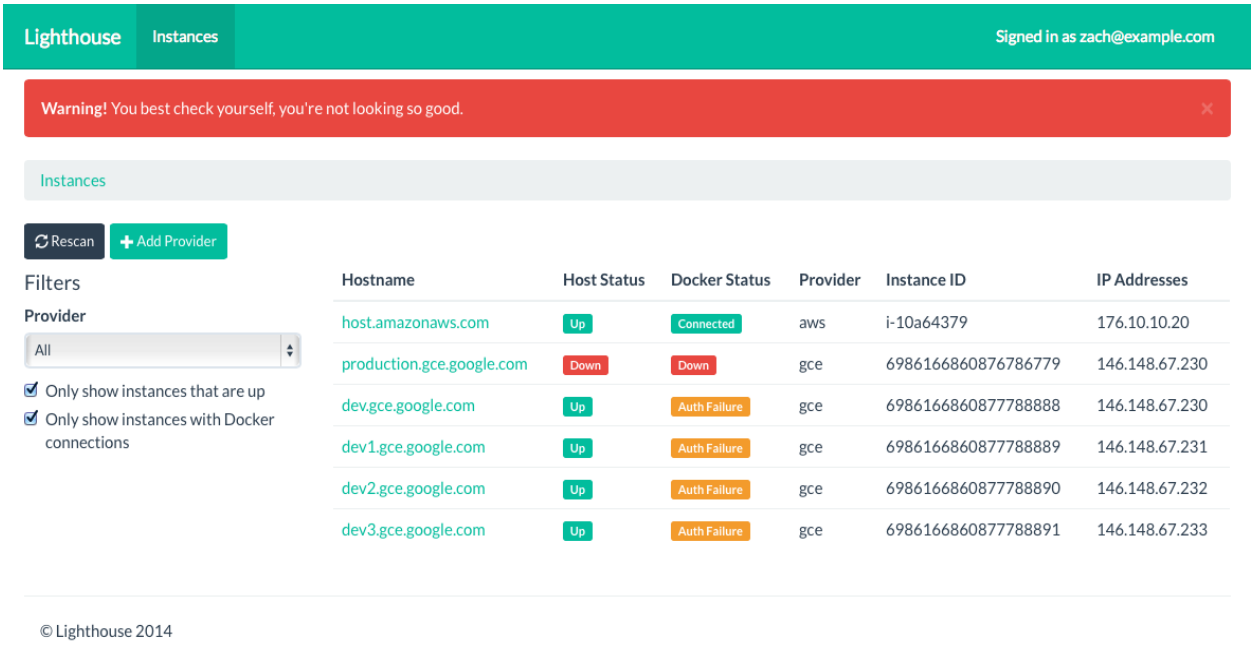

Figure 2 Listing available Docker instances

On this page the user can see a list of the available cloud instances in table format. Relevant data is shown for each individual instance so the user can get an informational overview of their instances. Above the table, there is a list of actions the user can take. *Rescan* will launch an instance discovery task

to find cloud instances that are available. *Add Provider* will take the user to a form where they can add an additional cloud provider to their system.

6.2 Instance Detail

Since the user will potentially have hundreds of instances across several cloud platforms, it is important to make this view filterable, allowing the user to find the exact subset of instances they are looking for.

Once the user has found the instance they are interested in, they can click on its *Hostname* field to drill down to that instance.

From this view, the user can investigate all information regarding this particular host. Some of the information that may be included on this page would be the status of the host, Docker connectivity, the cloud platform it's running on, unique identification, and possibly several other technical data points.

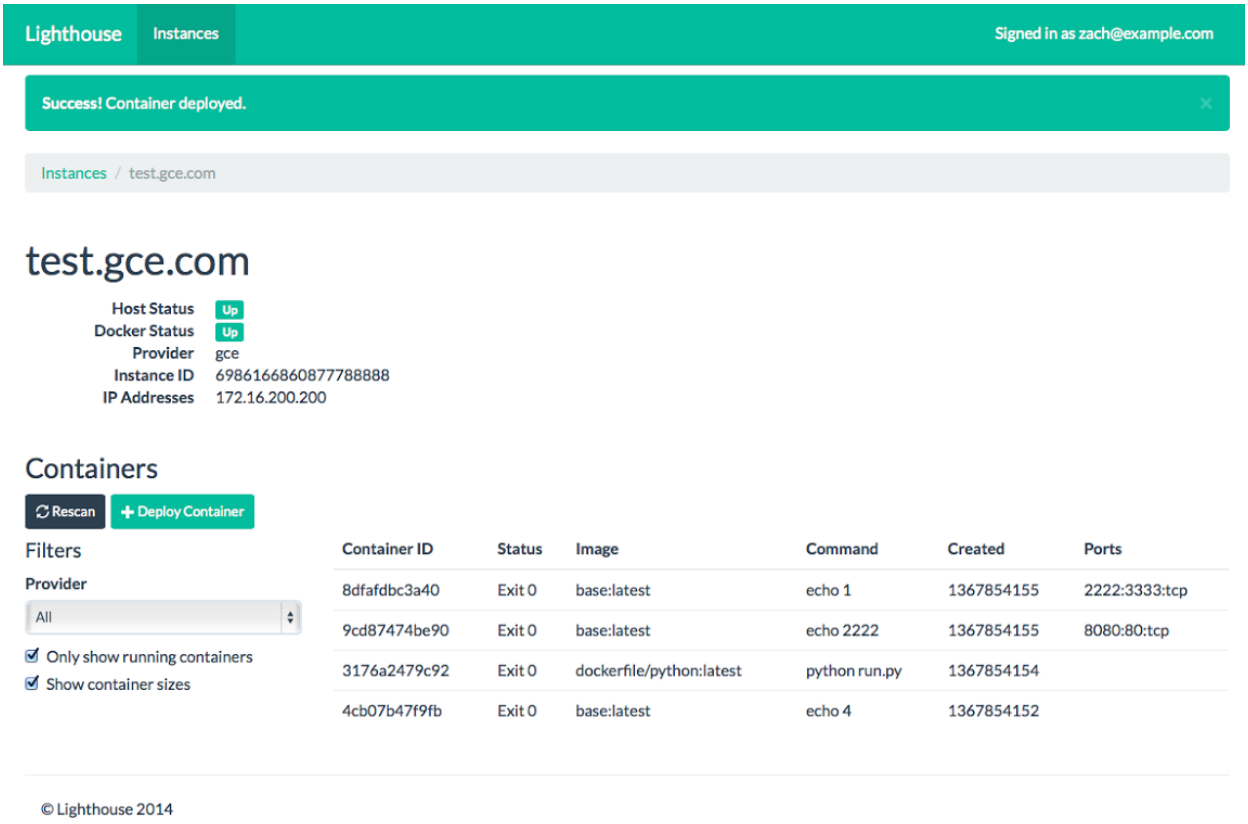

Figure 3 Docker instance detail page

In addition to information about the host, this view will include a list of Docker containers that are running on it. Again the user has options above the table to take an action. *Rescan* will launch a task to refresh the status of Docker on this instance. *Deploy Container* will present the user with a form to deploy a new container to this instance.

6.3 Container Deployment

Deploying a container to an instance is one of the main use cases for Lighthouse. Since this is such an important feature, we will want this feature to be as easy to digest as possible. Upon clicking on *Deploy Container* in the above diagram, the user will be prompted with a form to set up their deployment. This view is shown below.

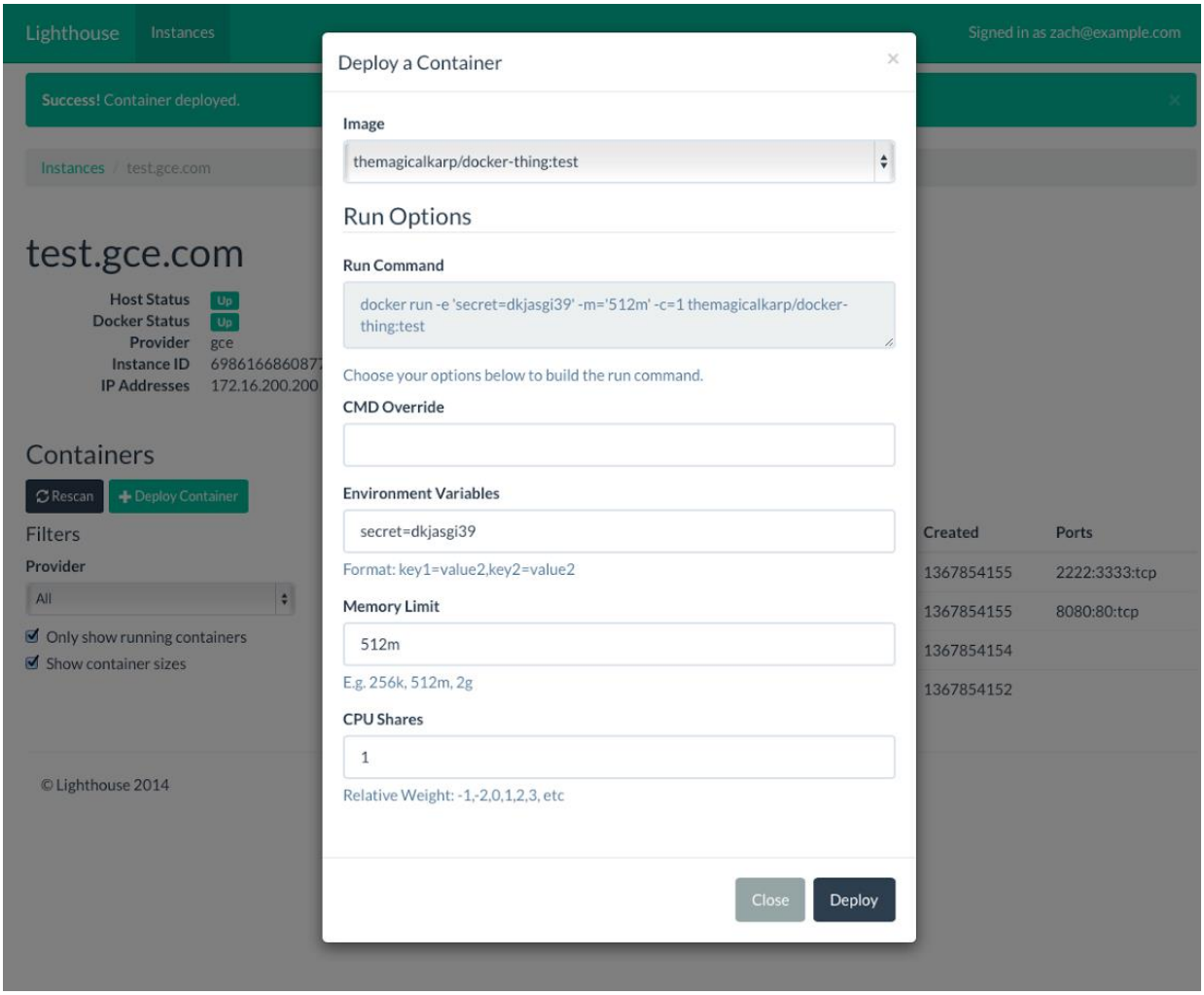

Figure 4 Container deployment page

The user will select the Docker image that they want to deploy. Then they will build their run command in the form. The Lighthouse UI will automatically build and validate the run command as the user is customizing their options. This provides instant feedback for the user and lets them see exactly what they will be deploying.

7 Requirements

7.1 Use Cases

For a successful implementation, we've identified two major actors in the Lighthouse system: release manager and IT admin.

7.1.1 Release Manager Functions

1. Log in/Authenticate
2. View currently deployed containers
3. Deploy and start a new container
4. Rollback a container/deploy a previous version

7.1.2 IT Administrator Functions

1. Log in/Authenticate
2. Initialize provider with authentication credentials
3. Create and delete system users
4. View provider statistics and analytics

7.2 Functional Requirements

- Docker addresses
    - Description
        - The IP address that are hosting Docker daemon publicly on port 2375.
    - Users
        - Administrators
    - Actions
        - add
        - remove
        - edit
- Docker Images
    - Description
        - Each Docker daemon has a set of images referenced by name or unique ID.
    - Users
        - Developers/Administrators
    - Actions
        - add
        - remove
        - edit
- Docker Containers
    - Description
        - Each Docker daemon has a set of running containers referenced by name or unique ID.
        - Containers are spawned from images pre-existing inside a Docker daemon.
    - Users
        - Developers/Administrators
    - Actions
        - start
            - Arguments
                - exposed port

        - command
        - environment variables
    - stop
- Docker Analysis
    - Description
        - Should be able to view running containers in real time.
    - Users
        - Developers/Administrators
    - Actions
        - view
            - logs
            - CPU usage
            - memory consumption
- Authentication
    - Description
        - Given an email and password all users can be denied or granted access to the webapp.
    - Users
        - Everyone
    - Actions
        - login
            - Arguments
                - Email
                - Password
        - logout
- Authentication Accounts
    - Description
        - Admins should be able to control user accounts.
    - Users
        - Administrators
    - Actions
        - add
        - remove
        - edit

7.3 Non-Functional Requirements

Security System (should be secure and not allow unauthorized control)

- Protect against
    - XSS attacks
    - session hijacking
    - unauthenticated requests

    - exposure of sensitive container/server data
- Only use HTTPS to mitigate various communication security exploits.

Code Quality (should be easy to maintain and understand)

- Git
    - Commit comments should be clear and concise, standard commit conventions
- Style Guides
    - GO
        - style guide
        - effective tips
    - JavaScript
        - style guide
- Code reviews must include at least 2 other team members to be verified for master merge.

8 Work Breakdown Structure

Each member has been assigned a role on the team. In addition, individuals have been assigned separate pieces of the project, mainly divided by the project's architectural modules.

Team Member	Role	Role Responsibilities	Project Responsibilities
Caleb Brose	Project Lead	• Project scheduling • Communication with Workiva • Communication with advisor	• Development of Lighthouse controller module
Chris Fogerty	Communication Lead	• Document compilation • Weekly report generation	• Development of Lighthouse controller module
Nick Miller	Web Developer	• Development of project website	• Development of Lighthouse front-end user interface module
Rob Sheehy	Key Concept Holder	• Handling and documentation of project concepts	• Development of Lighthouse hosting provider interface module(s)
Zach Taylor	Key Concept Holder	• Handling and documentation of project concepts	• Development of Lighthouse front-end user interface module

Table 3 Work breakdown by member

9 Resource Requirements

Resource	Purpose	Means of acquiring	Estimated cost
Shared Google Cloud Services developer account	Used to test and deploy our application	Provided by Workiva	Approx. $10/month
Amazon Web Services developer account	Used in late development to test multi-provider functionality	Provided by Workiva	Free for low usage

Table 4 Resource requirements and cost

10 Project Schedule

Our goal for the development lifecycle of Lighthouse is to utilize Agile practices as much as possible. Agile development defines a loose set of constraints focused on small iteration cycles. These cycles are split into (in our case) two-week sprints.

The beginning of the sprint involves a planning meeting where the team decides on which work tickets in the backlog are highest priority for the upcoming sprint. Tickets should be as small and well-defined as possible and efforts are made to break potentially large tickets down into smaller units of work, with the end goal being feasibility and testability once the work is complete.

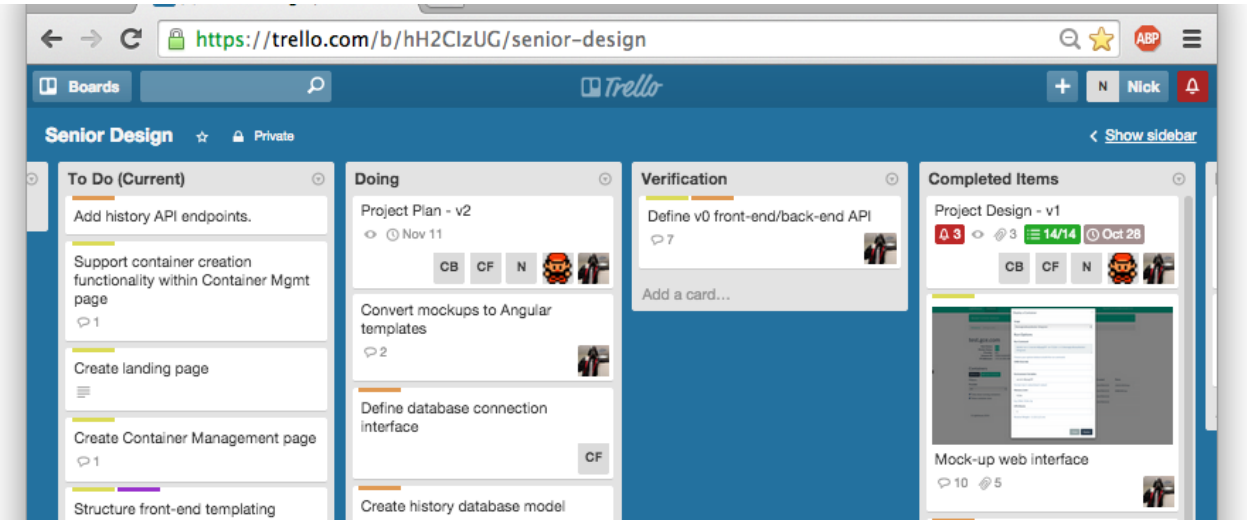

Figure 5 Sprint ticket board hosted by Trello.

During the sprint, tickets are moved across the board as shown above. **To Do** marks tickets moved from the backlog during the spring planning meeting, **Doing** marks tickets in progress, **Verification** marks tickets in the code review/testing process, and **Completed** marks tickets merged into the master development branch.

The sprint development cycle is outlined as follows:

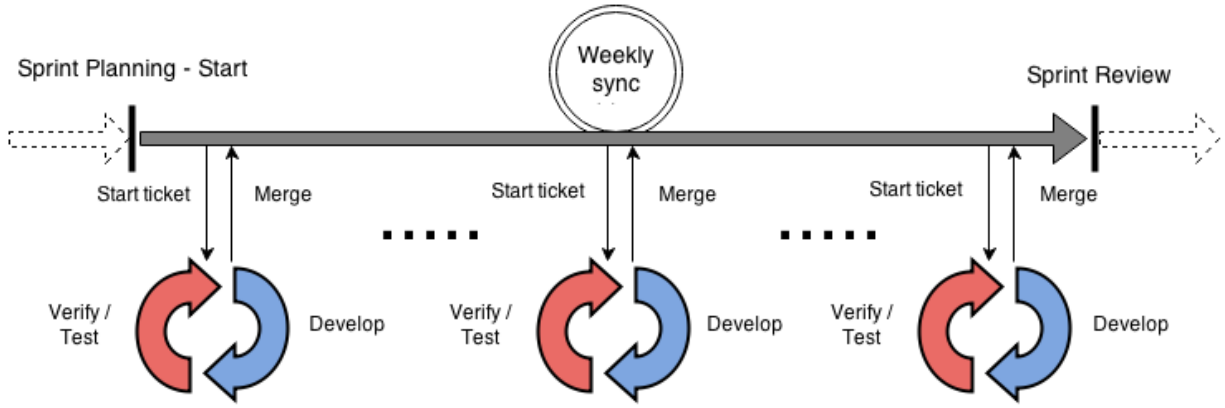

Figure 6 Sprint development cycle

Our intention is to meet with our project stakeholder and client, Workiva, at the end of every two-week sprint cycle to relay the work that has been done and discuss work for the upcoming sprint. The first week of the upcoming semester will focus on re-grouping on a team and outlining the major features left to implement. After that, we will start our two-week sprint cycles, with the last two weeks of the semester wrapping up any work left and preparing for our final demonstration to Workiva and the review committee.

11 Risks

11.1 Technical Feasibility

There should not be any breaking issues with technical feasibility in terms of the base idea of the project. Docker already provides a standard web interface. At its core, the Lighthouse project acts as a dashboard for Docker containers, providing a single interface to interact with many Docker containers.

We have run into some issues with clean design, however. Our initial goal was to provide a standard routing API for routing all Docker API calls. This would allow us to provide a no-maintenance tunnel for all Docker API calls right out of the box. Unfortunately, this became more complicated with the addition of some features such as deployment undo and logging.

11.2 Current Environment

Docker v1.0 was released a little over a year ago, so it is just now starting to be adopted by large companies. Because of this, there are only a few projects that accomplish what Lighthouse is looking to accomplish. That being said, the two big players right now are Kubernetes and Panamax, and neither does exactly what we're looking to do. Kubernetes applies more to managing a cluster of containers on a few machines, and Panamax is more unwieldy with less features than we're looking to build. The existence of these projects prove that something like Lighthouse is needed. Lighthouse may be a bit behind them in terms of development, but because Kubernetes and Panamax are currently in a rough/beta state, it is possible for Lighthouse to catch up. That is, Lighthouse will not be immediately irrelevant.

Appendix A: Table of Figures

APPENDIX B: WHAT IS DOCKER?

Essentially, Docker supports standardized application installation and execution via "containers" built from common templates that can run on a variety of host platforms. In the past, the efforts to standardize distributed and scalable applications were focused on the virtual machine, which was replicated across the desired host platforms. Docker removes the overhead of hosting and running the virtual machine, and instead utilizes the native system resources, while maintaining the desired isolation one would like to see between multiple applications running on one system.

In practice, the Docker host responsible for running the application container does not need to know of or explicitly install the application dependencies, as the container is built with its own layered file system to store dependencies and executables. Because of this, a Docker container can be built and run on a development machine, a testing server, and a production server with no difference in configuration of that container. A container can access its own file system, unless explicitly given permission otherwise, which maintains isolation between multiple containers on the same host.

Docker has achieved near native performance on application startup and shutdown, meaning new applications can be deployed and scaled as quickly as possible.