May15-17

Lighthouse Design Document

Version 1.0

Caleb Brose, Chris Fogerty, Nick Miller, Rob Sheehy, Zach Taylor
October 26, 2014

CONTENTS

1. Introduction

1.1. Project Definition

Within the past year, Docker has emerged as an execution and deployment environment for distributed applications. It supports standardized application installation and execution via “containers” built from a common template 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. Docker has achieved near native performance on application startup and shutdown, meaning new applications can be deployed and scaled as quickly as possible.

1.2. Project Goals

Our goal with Lighthouse is to provide a Docker host management system, capable of interfacing with a variety of cloud providers including, but not limited to, Google Compute Engine, and Amazon Web Services. Lighthouse will allow users to authenticate with their target Docker host platform, monitor the state of their application instances, and deploy new containers from one centralized interface, significantly reducing the time it requires to deploy and monitor applications across multiple cloud providers. By the project completion, our goal is to have built a system that is efficient, modular, and secure.

1.3. Definitions of Common Terms and 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

2. System Level Design

2.1. System Requirements

2.1.1. Use Cases

For a successful implementation of our **minimal viable product**, we've identified two major actors in the Lighthouse system: **release manager**, and **IT admin**.

2.1.1.1. Release Manager Functions

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

2.1.1.2. 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

2.1.2. 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 uid.
    - Users
        - Developers/Administrators
    - Actions
        - add
        - remove
        - edit
- Docker Containers
    - Description
        - Each docker daemon has a set of running containers referenced by name or uid.
        - 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

2.1.3. Non-Functional Requirements

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

- Protect agains
    - 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.

2.2. Functional Decomposition

2.2.1. High Level Decomposition

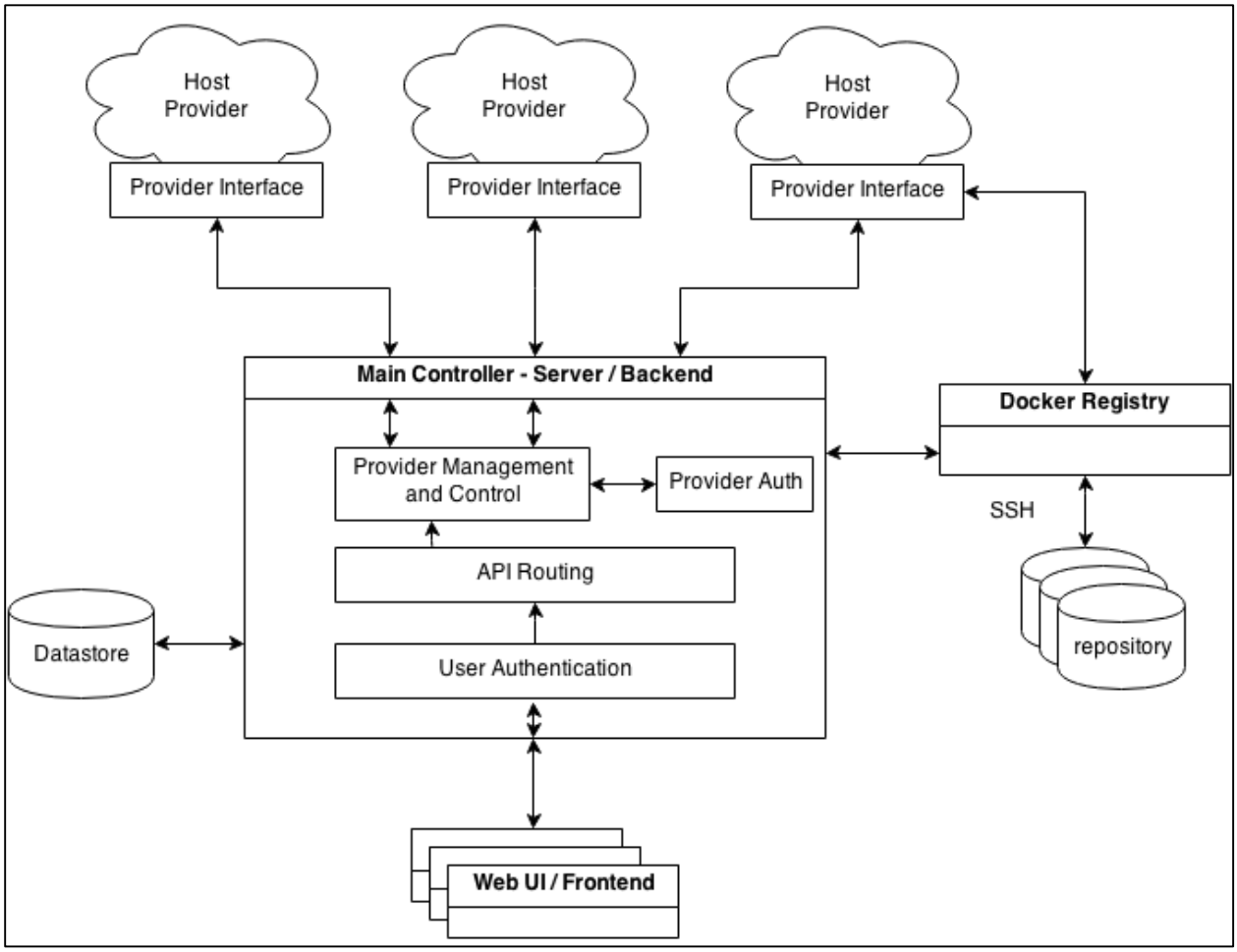

Figure 1: System Block Diagram

The overall architecture of Lighthouse is built upon the common client-server pattern. Users interact with the system via a “single page” web application that’s 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 images are hosted on the Docker registry provided by the organization using Lighthouse. Host providers are instructed to pull new images from the registry on application deployment.

2.2.2. Front-End

The goal of the front-end of the system is to present the user with information and provide the user with a simple UI to manage their array of Docker installations. Using the API exposed by the back-end, the front-end renders the response from the back-end into HTML and displays it to the user. The front-end must know how to handle any errors in communicating with the back-end server, manage user sessions (for logging in and out), and generate API requests to the back-end.

2.2.3. Back-End

It is the back-end's job to authenticate users, cache Docker hosts from the provider interface, and provide features on top of what Docker already provides, such as rolling back deployments. The backend exposes a REST API (described below) that anyone can build off. While we use it as a way to communicate with the front-end, the API allows for users to build easily build automation into their systems. The back-end communicates with provider interface(s) in order to discover any cloud instance that is owned by the user and running Docker. It must also elegantly handle errors by returning a sane response to the client, and log any errors or events that pop up.

2.2.4. Provider Interface

The provider interface is installed within each cloud segment the user wants to access. It allows us to use a single API on the back-end to locate relevant cloud instances across all providers. By being in the relevant cloud, the provider interface is granted access to privileged APIs, allowing it to discover other instances owned by the same account.

2.3. System Analysis

2.3.1. 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 demultiplexer 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.

2.3.2. 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.

2.3.3. Logical Design

Lighthouse needs to be easy to deploy and customize. It should work right out of the box, with the installer only providing Cloud authentication details and user accounts. To accomplish this goal, we have decided to "Docker-ize" Lighthouse so a container can be created based on Lighthouse with virtually no hassle on the part of the installer. As this is a webapp, all interaction will be performed through a browser pointed at a server running the Lighthouse software.

3. Detailed Description

3.1. Interface Specifications

3.1.1. Lighthouse API version 0.1

Note: prepend all endpoints with `/api/v0.1`

3.1.1.1. Requests to Docker Instance

These endpoints forward a request to a Docker instance. Note that:

- The *hostAlias* parameter serves as a lookup to the instance's address.
- The *dockerEndpoint* parameter is exactly the request which is sent to the Docker instance.
- The *payload* JSON field is exactly the HTTP request body which is sent to the Docker instance.

```
GET /{hostAlias}/d/{dockerEndpoint} HTTP/1.1
```

```
POST /{hostAlias}/d/{dockerEndpoint} HTTP/1.1
Content-Type: application/json

{
      "payload": <Docker Request Body>
}
```

```
PUT /{hostAlias}/d/{dockerEndpoint} HTTP/1.1
Content-Type: application/json

{
      "payload": <Docker Request Body>
}
```

```
DELETE /{hostAlias}/d/{dockerEndpoint} HTTP/1.1
```

Aliases will be client specific. For a full list of available Docker endpoints, see http://docs.docker.com/reference/api/docker_remote_api_v1.14/

Note: The request types exactly mimic the Docker request types.

3.1.1.2. Responses to Docker Requests

The server will return one of two categories of responses:

- Status code in the 200 range – these are success responses and return exactly the Docker response.
- Status code in the 300-500 range – these are error responses and return a JSON error message.

Examples can be found below.

A successful GET request:

Request:

```
GET /a_valid_host/d/version HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
      "ApiVersion":"1.12",
      "Arch":"amd64",
      "GitCommit":"990021a",
      "GoVersion":"go1.2.1",
      "KernelVersion":"3.13.0-37-generic",
      "Os":"linux",
      "Version":"1.0.1"
}
```

A successful POST request:

Request:

```
POST /a_valid_host/d/containers/create HTTP/1.1

{
      "payload": <Docker Create Body>
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
      "Id":"e90e34656806",
      "Warnings":[]
}
```

A failed request to an invalid Docker endpoint:

Request:

```
GET /valid_host/d/bad_endpoint HTTP/1.1
```

Response:

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
      "error":"docker",
      "message":"404 Not Found"
}
```

A failed request to an invalid host:

Request:

```
GET /bad_host/d/valid_endpoint HTTP/1.1
```

Response:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
      "error":"control",
      "message":"GET request failed"
}
```

A failed request to a valid endpoint with a server error:

Request:

```
GET /valid_host/d/valid_endpoint HTTP/1.1
```

Response:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json

{
      "error":"postgres",
      "message":"Database lookup failed"
}
```

3.1.1.3. Authentication

Log in to the application:

```
POST /login HTTP/1.1
Content-Type: application/json

{
      "Email":<Email>,
      "Password":<Password>
}
```

Example request and response:

Request:

```
POST /login HTTP/1.1

{
      "Email":"my_email@example.com",
      "Password":"my_password"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

true
```

Log out of the application:

```
GET /logout HTTP/1.1
```

Example request and response:

Request:

```
GET /logout HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

3.1.1.4. GCE VM Discovery

List all known, authorized, GCE VMs:

```
GET /plugins/gce/vms HTTP/1.1
```

Example request and response:

Request:

```
GET /plugins/gce/vms HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
      {
            "Name":"My Project Name",
            "Address":"example.gce.com",
            "CanAccessDocker":"true"
      }
]
```

Request a VM search:

This endpoint repopulates the `/plugins/gce/vms` and redirects to it.

```
GET /plugins/gce/vms/find HTTP/1.1
```

Example request and response

Request:

```
GET /plugins/gce/vms/find HTTP/1.1
```

Response:

```
HTTP/1.1 302 Found
Content-Type: text/html

<a href=”/plugins/gce/vms”>Found</a>.
```

Authorize a new GCE VM:

This endpoint redirects to a Google Authentication page.

```
GET /plugins/gce/authorize HTTP/1.1
```

Example request and response:

Request:

```
GET /plugins/gce/authorize HTTP/1.1
```

Response:

```
HTTP/1.1 302 Found
Content-Type: text/html

<a href=[Google Authorization Page]>Found</a>.
```

3.1.1.5. History

Get data of image's last run container:
This endpoint retrieves the data used to create the container which was last run on the given image.
Note that:

- The *hostAlias* parameter serves as a lookup to the instance's address.
- The *imageName* parameter is the Docker image whose history is requested.

```
GET /{hostAlias}/history/{imageName} HTTP/1.1
```

Example request and response:

Request:

```
GET /a_valid_host/history/my_image HTTP/1.1
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
      "History": [
            {
                  "CreateTime":"10-25-2014 11:53",
                  "Name":"MyContainer",
                  "Payload":<Docker Create Body>
            }
      ]
}
```

3.1.2. UI Specification

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

3.1.2.1. Instances

Our goal for the user interface is to allow users to accomplish the success scenarios outlined in our Use Cases section in a timely and simplistic fashion.

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.

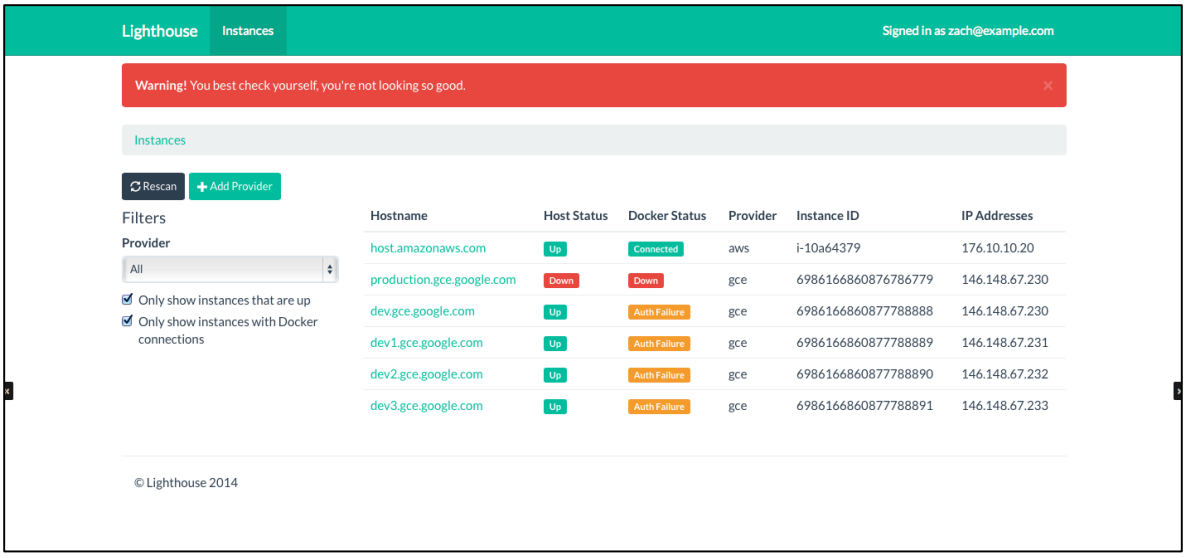

Figure 2: List of Cloud Instances running Docker

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.

3.1.2.2. 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 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.

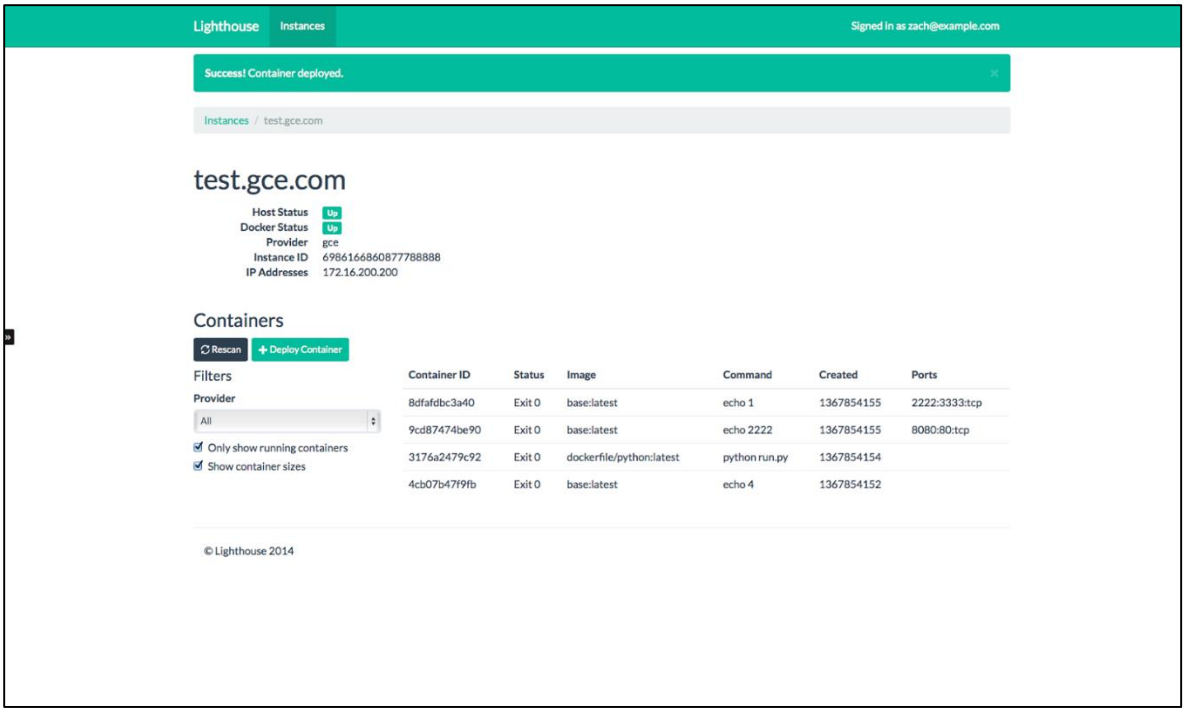

Figure 3: Instance Detail page

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.

3.1.2.3. 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.

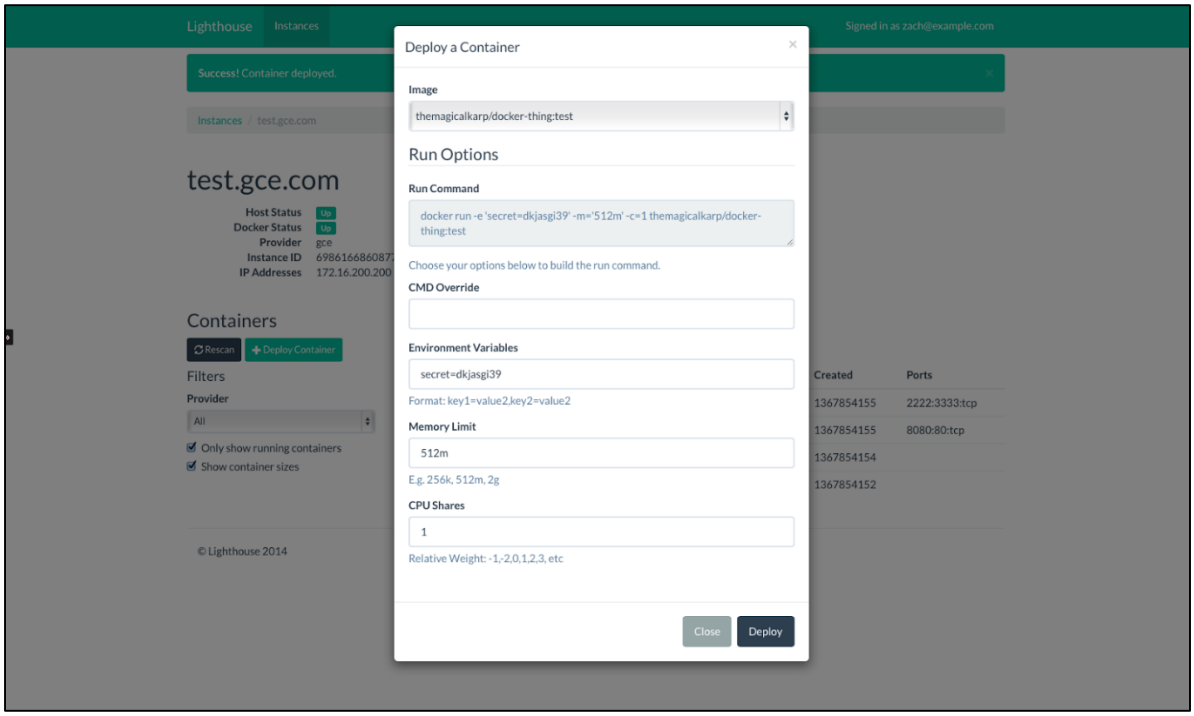

Figure 4: Container Deployment page

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.

3.2. Testing Procedures and Specifications

The plan is for Lighthouse to be developed in an agile fashion. New features will be added over time, with each iteration of the product adding additional features and/or bug fixes. The following rules will apply to all developers.

1. Tests will be written for each commit, covering the code written for that commit or updating previous tests to reflect code changes.
2. If a bug is found, a bug fix commit will be created, and tests will be created to prevent a regression.
3. Tests will not be modified unless an investigation into a test failure determines the test is incorrect.
4. Pull Requests will not be accepted unless all tests are passing in the Pull Request's branch.

4. Implementation Issues and Challenges

4.1.1. Understanding Docker

The Docker platform is a relatively new environment, providing plenty of challenge for us as a team to digest and understand the use and workflow. To ease the learning curve, our plan is to wrap Lighthouse itself within a deployable Docker container. This will allow us to better understand the workflow associated with Docker and use the recent best practices that have come out of the community.

4.1.2. Development coordination

One of the major challenges we've encountered thus far is coordination between development tracks. We are utilizing git for our version control, but the responsibility to keep the design modular and flexible rests on the team as a whole.

4.1.3. Security

At every level of the application, any network transfer must be as secure as possible. Breaches in our security protocol have the potential to significantly harm the target Docker host providers. Unauthorized access of the target host platform gives full control over the Docker instance exposed, meaning applications (i.e. Docker containers) can be stopped and started at will. The two major areas of security concern are the provider interfaces and the user interface. A first step towards strong security for Lighthouse is user authentication and server-side data validation.

5. Conclusion

With the advent of Docker, there has been a torrent of new technical challenges surrounding the technology. Now that Docker has landed and has become an important and accepted industry standard, the foundation has been created. Some of the new challenges will be around building on top of this technology, particularly management, security, and development of best practices.

Lighthouse's main purpose is to tackle the management of the Docker platform. Our main goal is to make management of this platform simple, efficient, and secure. By communicating frequently with potential stakeholders, making the project open, and using standard and modern technologies, we will be able to accomplish the goals outlined in this document, which will allow Lighthouse to be a successful and useful tool for Docker management.

APPENDIX A: TABLE OF FIGURES