Cloud Environment Manager

Final Report

Members:

Adis Osmankic

Zane Seuser

Jet Jacobs

Rishabh Bansal

Gavin Monroe

Team Number: sdmay21-39

Client: PwC

Adviser: Lotfi Ben Othmane

Team Email: sdmay21-39@iastate.edu

Website: https://sdmay21-39.sd.ece.iastate.edu

Last Revised: April, 2021

Executive Summary

Development Standards & Practices Used

- **Development Standards**
  - IEEE 14764-2006 - Standard for Software Engineering - Software Life Cycle Processes - Maintenance
    - This standard explains the life cycle processes of software. This standard provides guidance that applies to planning, execution and control, review and evaluation, and closure of the Maintenance Process.This is where our CI/CD and DevOps practices come into play.
  - IEEE 29119-1:2013 - Software and Systems Engineering — Software Testing
    - This standard is to define an internationally-agreed set of standards for software testing that can be used by any organization when performing any form of software testing.
  - IEEE 15026-1:2019 - Systems and Software Assurance
    - This standard goes over software assurances. We are specifically following part three of this standard which goes over the integrity of a given piece of software. We consider the CIA triad of confidentiality, integrity, and availability of our application with great importance and following this standard is a critical step to achieving this.
- **Practices Used**
  - Agile Development
  - Cloud IAC (Infrastructure As Code)
  - Code Developed with Modularity
  - Well-Documented Code
  - REST
  - OWASP ASVS

Summary of Requirements

- Easily create & manage lab environments on multiple cloud platforms
- Intuitive interface to manage and create lab environments

Applicable Courses from Iowa State University Curriculum

- COM S 309 (Software Development Practices)
- COM S 319 (Construction of User Interfaces)
- COM S 362 (Object-Oriented Analysis & Design)
- COM S 363 (Database Systems)
- SE 329 (Software Project Management)
- SE 339 (Software Architecture & Design)

New Skills/Knowledge Acquired That Was Not Taught In Courses

- Knowledge Of Different Cloud Providers
- Knowledge Of Infrastructure As Code
- React Development
- Python Development
- Ansible Implementations

Table of Contents

1 Introduction

1.1 Acknowledgement

Thank you to our client PwC for the technical support and architectural guidance. We would also like to thank Lotfi Ben Othmane, our advisor for this project, for helping us through the planning and development process.

1.2 Problem and Project Statement

Currently, our sponsor provides lab environments for a variety of reasons such as capture the flag events or to test different tool sets in a simulated environment on a variety of cloud providers manually. The problem they have is that this method is unscalable and it is difficult to recreate these environments, especially on different platforms. They currently have no way to quickly create and manage a large amount of lab environments in a consistent and reliable way.

By developing our project application, the Cloud Environment Manager, we are going to create an open source application that our sponsor and others with similar problems can utilize. We will provide an application that contains a clean and simple user interface that allows them to quickly and easily deploy labs and manage environments on multiple cloud providers.

To summarize, our whole project is to create an user friendly interface that can quickly and easily create and manage virtual machines and their networks in various cloud platforms.

1.3 Reader Context

Throughout this document we will refer to “labs” or “lab environments”. In the context of the Cloud Environment Manager application, a lab or lab environment is a group of one or more virtual machines that are hosted within a cloud provider. Any software can be installed on these machines, but our sponsor’s use case is to host capture the flag training events, create testing environments, or to demonstrate new technologies.

1.4 Operational Environment

This main application will be hosted on a client’s server. The front-end application will be accessible from a web browser and the back-end API it interfaces with will

be able to deploy lab environments to different cloud providers (AWS, Azure, GCP).

1.5 Requirements

- **Functional Requirements**
  - A user should be able to view a list of existing lab environments
  - A user should be able to view all of the attributes of existing lab environments
  - A user should be able to provision lab environments within AWS, GCP, and Azure
  - A user should be able to destroy existing lab environments within AWS, GCP, and Azure
  - A user should be able to deploy the same lab environment on any platform
- **Non-Functional Requirements**
  - The application should be available at all times
  - The application should properly handle errors behind-the-scenes and provide error messages / warnings to the user when necessary
  - The user interface should be visually appealing
  - The user interface should be easy to use and intuitive

1.6 Intended Users and Uses

This project is designed for use by an organization that needs to deploy lab environments to different cloud platforms for a variety of different users. This project is focused on deploying test environments to mainly be used for education and capture the flag events for organizations or individuals that require to create these platforms often and consistently. By using this product, organization admins will be able to deliver training resources to members of the organization quickly and easily.

1.7 Assumptions and Limitations

- **Assumptions**
  - The organization has access to AWS, Azure, and GCP cloud platforms and can provide service account credentials with permissions to manage networks, volumes, and virtual machines.
  - The organization or individual has the resources to host the user interface and back-end with their own login service

    - The organization can configure the lab environments to accept programmatic access to the virtual machine resources in the cloud
- **Limitations**
    - There are some configurations that need to be manually changed by an organization
    - Only administrators can set up the application

1.8 Expected End Product and Deliverables

- Hosted Cloud Environment Manager Application And Source Code
- Project Documentation
- May 2021 Hand-Off

1.9 Related Works

There are related works to this product in the sense of monitoring and controlling virtual machines in individual platforms, but none that are able to manage multiple platforms and allow for cross platform deployments. Similar products, such as Terraform, focus on creating deployments from a given script and are able to create deployments on multiple platforms, but is not a user friendly tool as the user will need to create and manage scripts while running these scripts on some device. Other related works, such as Spot Cloud Analyzer and Nutanix beam, are capable of analyzing deployments on multiple cloud platforms in a user friendly way, but focus more on monitoring the platforms rather than deploying to these platforms in a modular and consistent manner. These works also do not have free models and are missing key requirements for the users we are trying to serve, while the Cloud Environment Manager is open source and focuses on deploying modular environments to separate cloud environments.

2 Design & Implementation

2.1 Engineering Constraints

- **Minimal Constraints Applied From Sponsor**
  PwC gave us a ton of flexibility on this project in regards to which technologies to implement the application with. The main “constraint” they provided to us was more of a recommendation, and that was to make use of Ansible for Infrastructure As Code (IAC) as our cloud orchestrator.

- **No Monetary Resources**
  Our team was not given access to any monetary resources to complete this project. This is not a major issue as we could make use of our ISU-provided virtual machine for hosting as well as the free levels of the various cloud providers, but this did prevent testing scalability and testing more complex environments.

- **Time**
  As with all senior design projects, we are strictly held to an application design and development time-frame of 2 semesters.

- **Cross-Compatibility**
  The whole idea of our project being able to work across multiple cloud provider platforms is perhaps the most important engineering constraint and guide post. The templates for the labs must be able to be deployed across all of the supported cloud platforms.

2.2 Design Thinking

A major requirement for our product is that it needs to be intuitive for the user to utilize and powerful and modular enough to perform multiple deployments across multiple platforms. The product needs to be able to take information from the user and use it to deploy environments. This ultimately led us to deciding to develop a full stack web application. To develop this application, we would need several components that all work in conjunction with one another. The basic components of this application would be a front-end, a back-end, and a cloud orchestrator all utilizing a concept of templates to operate as seen in **Figure 2.2.1**.

- **Front-end**

  To make the service accessible to the user in an intuitive way, we decided that we needed a front-end that is separate from the complex logic and processing necessary for the product. This could be done through several different means such as a command line interface or through a pipeline that is hosted on some platform, but we ultimately decided to develop a graphical user interface that is hosted on the web. This allowed for the user to easily and intuitively view and manage information while allowing us to format requests into a format that could be interpreted properly by our service. Being hosted on the web also allowed for greater availability for the user as it would allow the user to access the application from anywhere on a given network.

- **Back-end**

  Since our service will be hosted separately from our user interface, we needed to have a component that could handle requests from the front-end, perform complex operations, and communicate with our cloud orchestrator. We considered performing this on a pipeline that could be kicked from the front-end, but ultimately decided against this due to the limited resources we were offered and the limited modularity of a pipeline given our use cases. We then decided to host a separate application to perform these operations. Since the front-end needed to interface with this service, we decided to expose an API on this service to interface with the user interface. This back-end then would need to be structured enough to take in specific input from the user interface, yet dynamic enough to turn that input into operations to provision lab environments. With this in mind, we needed to create components within this service that would handle each type of request for each given platform.

  Since we figured we would likely have to store information about how to deploy to specific accounts and information on deployed instances, we decided we needed to implement some storage mechanism. Initially, we decided to go for a filesystem store that would contain configuration information along with a file to store information on deployed instances, but to improve efficiency and to allow for easy modification of data, we decided to go with a database to host this information.

- **Cloud Orchestrator**

  By this point, we already have a service planned out that can perform logic and data processing with a user interface that allows the user to make requests to perform separate use cases, but the service still needs to communicate with the

cloud. There were many options on how to perform these actions, such as utilizing each cloud platform's command line interface, utilizing each platform's SDK/API, or utilizing some third-party automation software. We decided to go with the latter since we found that creating libraries for our back-end to use each cloud platform would be cumbersome and overly complex while attempting to programmatically use a command line interface introduces extra unnecessary complexities. Utilizing third-party software would allow for consistent data to be used for each platform and would be quicker to implement, which is critical given the time constraint.

- **Template**

  Since a major requirement of this project is to develop code that is modular, we needed to create a way that would allow for a user to create an environment that contained basic information, such as network information, images of given machines, and volume information of those machines all of which would create a “lab” or “lab environment.” Since each platform varied greatly, we needed to find a way to keep this information generic enough to parse and create these lab environments on different platforms.

  Initially we were thinking of specifying this information on deployment requests, but figured that the user may want to store this information and redeploy another instance of this lab, even potentially on another platform, so we decided to create the concept of a template. The user would be allowed to create a template by providing basic information, such as a list of images for machines to be deployed in a single environment and what those machines can access, then we would turn this information into a generic, parseable collection of information and store it in some data store. The user could then reference this template during deployment giving consistent deployments of the same lab no matter the platform. Since some cloud platforms contain options that are platform specific, such as AWS security groups or Azure resource groups, we decided that during deployment, the user must supply a template, but the user should also be given a list of optional, platform specific configurations they could modify during deployment.

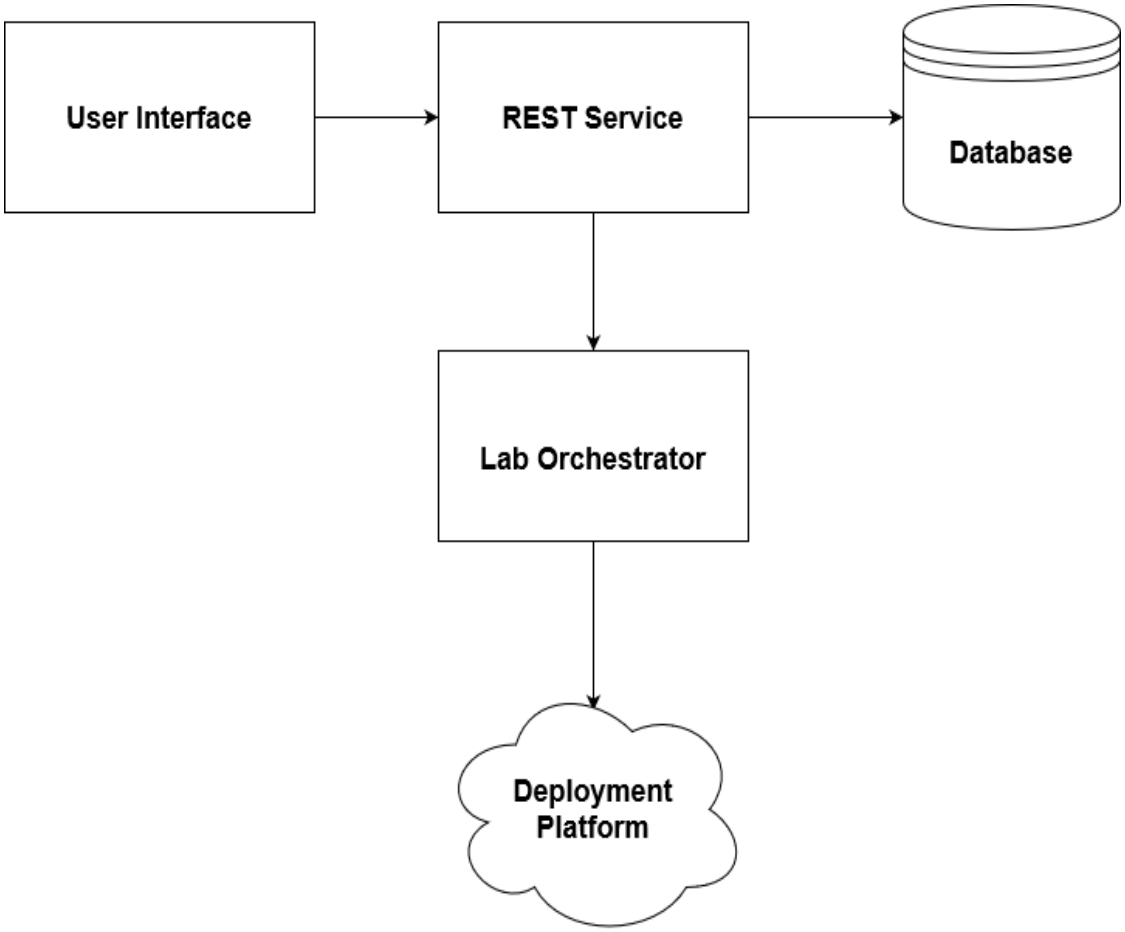

Figure 2.2.1: Design Overview

- **Use Cases**

  To help illustrate the flow of how this design would work, we will go over several use cases.

  User wants to create a new lab environment template

  Due to the nature of our product, the user will need to create custom lab environments and save them to be referenced later. To accomplish this, a page on the user interface will be created that requests information from the user about the new lab environment. This information will be structured in such a way that it can be deployed to any platform needed by the end user. We utilize a “template” to structure this data and save it for later referencing. The template is a generic JSON object that contains fields such as what the name of this template should be, what machine images should be used, and what connectivity the machine should have (can it access other machines in its environment, can it

access the internet, can it be exposed to the internet). The user data is transformed into a template and sent to the back-end using a POST request to an exposed API. The back-end then validates the data and if the template is valid, stores it in a mongo database, otherwise responds with an error. An illustration of this can be seen in **Figure 2.2.2**.

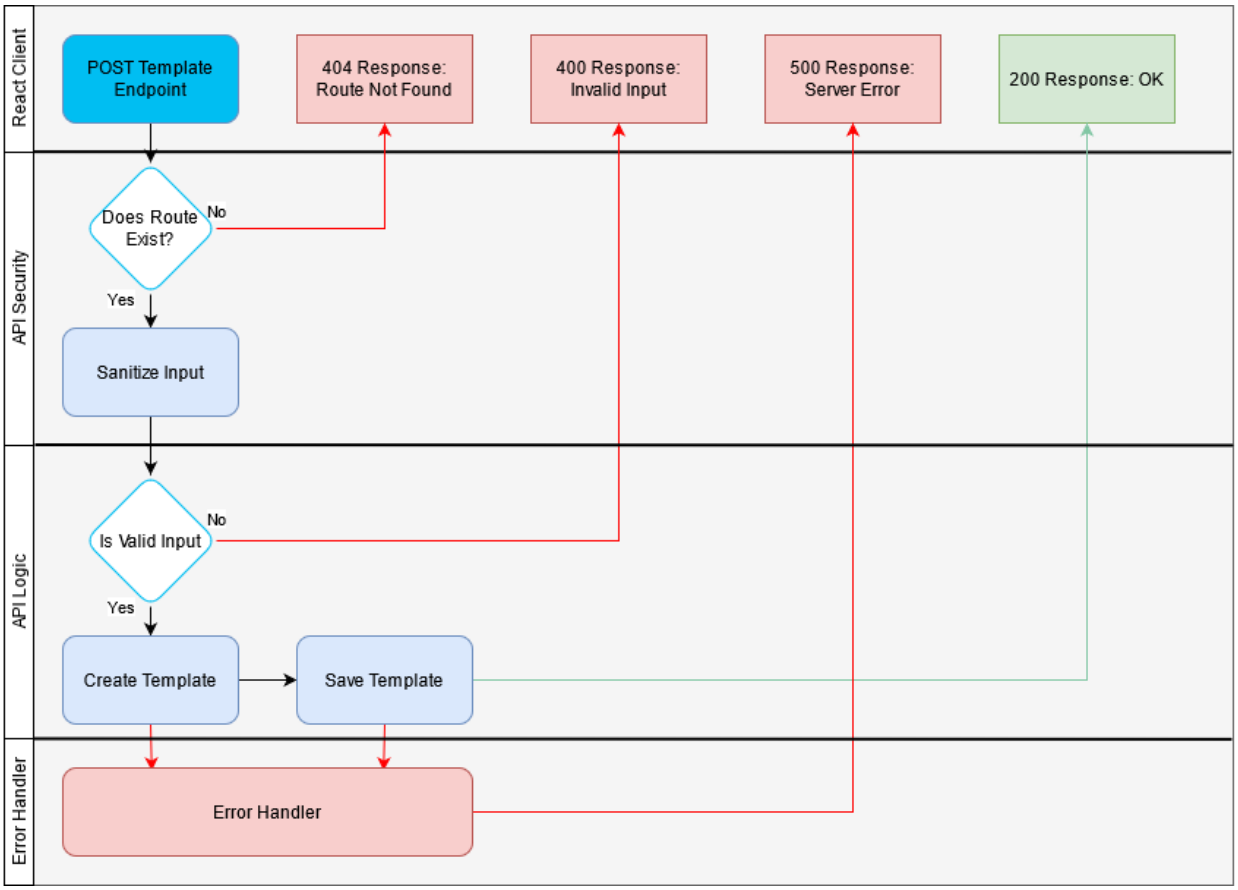

Figure 2.2.2: Create Template Workflow

Flow diagrams for each use case will be similar to the above with different API logic.

User wants to deploy a saved lab environment

In order for a user to deploy an environment, they had to have created a template first. To deploy an environment, the user must specify the platform they want to deploy to and additional platform specific information. Once the user selects a template and a platform, the request is converted to a JSON format and sent to the back-end via an exposed POST API. The API then validates and sanitizes

the request and parses the data to perform whatever deployment actions are necessary to have the lab deployed to a given platform. The deployed lab information is then stored in a database that is used to track all deployments.

User wants to view all deployed lab environments

If the user wishes to view all lab environments and their status, they navigate to the deployed labs page which sends a GET request to an exposed API on our back-end service. The back-end Service then queries a deployment database that contains information on all deployed labs and queries all cloud platforms to get all running machines. The back-end then uses the information from the deployment database to filter the information gathered from all cloud platforms and group the individual machines into corresponding lab environments. This information is then sent back to the client to be displayed in an intuitive way.

User wants to modify state of lab environments

For the user to modify the state of a lab (on/off/restart/terminate) they must view all deployed lab environments, select the lab environment they wish to modify and select either individual machines to modify or the entire environment. They will then confirm their choice and send a PUT request to the exposed API on the back-end service with all machines they wish to modify along with their choice of modification (on/off/restart/terminate) and the service will make the appropriate calls to modify the state of a machine on each given platform.

User wants to view all templates

For the user to view all the saved templates, they must either view the templates page or the deploy lab page. Either page will make a GET request to the exposed API on the back-end service and retrieve all the templates from the template database. The UI will then display the templates to the end user.

User wants to remove a template

For the user to delete a saved template, they must view the templates page and select the template they wish to remove. After confirming their choice, the interface will send a DELETE request to the exposed API on the back-end service. The back-end will then verify the request and delete the template from the database.

2.3 Technology Decisions

- **Hosting**

  The Iowa State virtual machines were used as the infrastructure that hosts our user interface, Flask API, Ansible nodes, and our MongoDB database during the development of the application. Initially we were planning to host this on AWS to utilize many of the great resources on AWS, such as CodePipeline and API Gateway, but given that we had no budget, we decided to utilize the resources given to us and host on Iowa State's servers. On the virtual server provisioned for use, we utilize Docker to host each component and utilize docker compose to allow for each component to communicate with one another.

- **User Interface (UI) Framework**

  We decided that the best way for a user to interact with our service would be through a graphical web user interface. To implement this, we utilized React, a Javascript library for building user interfaces, to build the Cloud Environment Manager's user interface. We chose to go this route because React allowed us to build a highly functional and dynamic UI extremely quickly while also being a framework that was familiar with a majority of the team. The componentization aspect of React and all of the external libraries that can be easily imported into a React allowed us to quickly develop an interface within the strict time constraint while also giving us flexibility to work with our API. We considered developing a basic HTML/JS web front-end, but we figured that React had many libraries to simplify how we create our user interface.

- **API Framework**

  To access and control our service, we needed to have some interface that could be accessed by the outside world. We decided to implement an API to account for this and we decided to utilize Flask, a Python API framework, to accomplish our API for this project. We chose to use Flask, because it allowed for us to get an API running in a timely manner, and allowed us to use any package that is available to Python via pip. Having access to the Python libraries is extremely important for the Cloud Environment Manager application, because there are a multitude of libraries that exist and they allow us to interact with all the major cloud providers, Ansible, MongoDB, and various other useful utilities. We also considered utilizing other languages and frameworks, such as express.js to serve as our API, but given that our cloud orchestrator tool only offered

a python interface and we were all familiar with python and Flask, we ultimately decided to utilize Flask.

- **Database Technology**

  The database technology of choice for this project was MongoDB. We chose to go this route because of the NoSQL flexibility. Additionally, the back-end framework, Python Flask integrates very nicely with it.

- **Cloud Orchestrator**

  Ansible, specifically the Ansible Playbooks, is the technology that we are utilizing for the IAC aspect of the Cloud Environment Manager application. We chose this route because Ansible integrates with all of the major cloud providers that we support relatively easily. Additionally, our sponsor nudged us to use this technology during our initial design conversations with them. An important driving factor in our design is that it is supposed to be open source and modular. Utilizing playbooks allows for a user to quickly add or modify functionality for the application without any major programming, making it a clear choice for this application. We were looking into other orchestrator tools such as Chef and Puppet, but we found that they required a more complex setup and required purchasing their tool, unlike ansible that fit our budget of free.

2.4 Design Components

The overall design of our product can be summarized as 5 major components as seen in **Figure 2.4.1** and **2.4.2**.

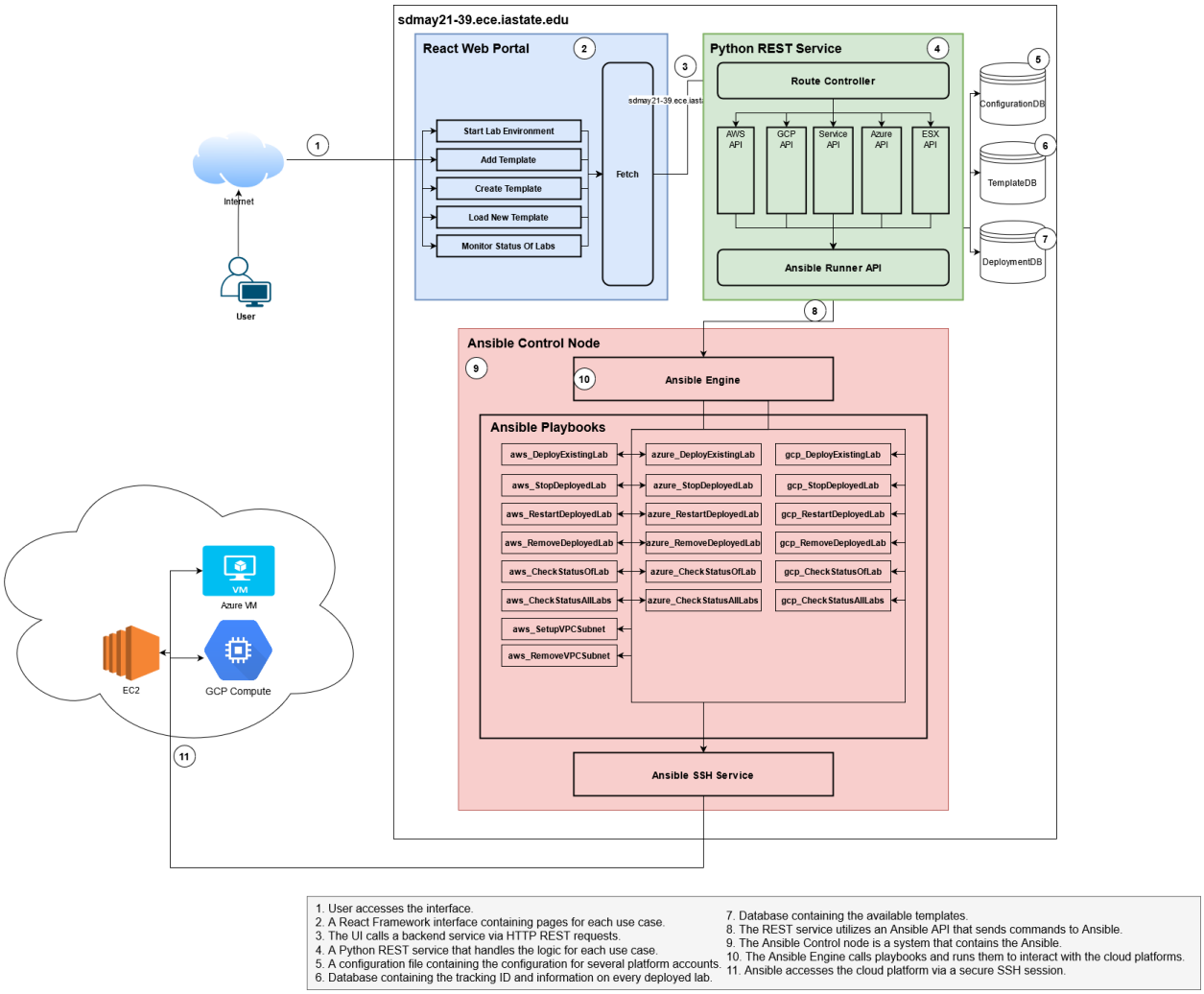

Figure 2.4.1: Architecture Diagram

User Interface (React Web Portal):

A React application was created for our project's user interface. This application presents users with the ability to define new lab templates, provision labs into various cloud providers with said templates, and view and destroy existing lab environments that were previously created. All of this data and functionality is available to our React application via the API.

API (Python REST Service):

A Python Flask API has been created to be the connection to all of our back-end services and provide all the functionality to our React front-end. Depending on what a specific route or path is doing, it interfaces with Ansible or MongoDB. Additionally, Flask makes it super easy to import all the various libraries needed, sanitize API requests, and provide usable API responses.

Ansible (Playbooks):

Ansible is utilized within the Cloud Environment Manager application to do the actual interfacing with the various cloud providers. In short, the Ansible Playbooks that we have written are able to use our encrypted cloud provider credentials to retrieve information about existing labs, provision new labs, and de-provision existing labs. These actions can be done on Amazon Web Services, Microsoft Azure, and Google Cloud Provider.

MongoDB (Database):

MongoDB is the chosen database technology for this project. We utilized it to store our re-usable lab templates, deployed lab environments, and various configuration parameters.

Cloud Infrastructure (Labs / Virtual Machines):

We are giving users the ability to create labs within various different cloud providers such as AWS, GCP, and Azure. A lab is created with the user-selected template, and placed into a network on the selected cloud provider that makes it accessible to those trying to utilize the lab.

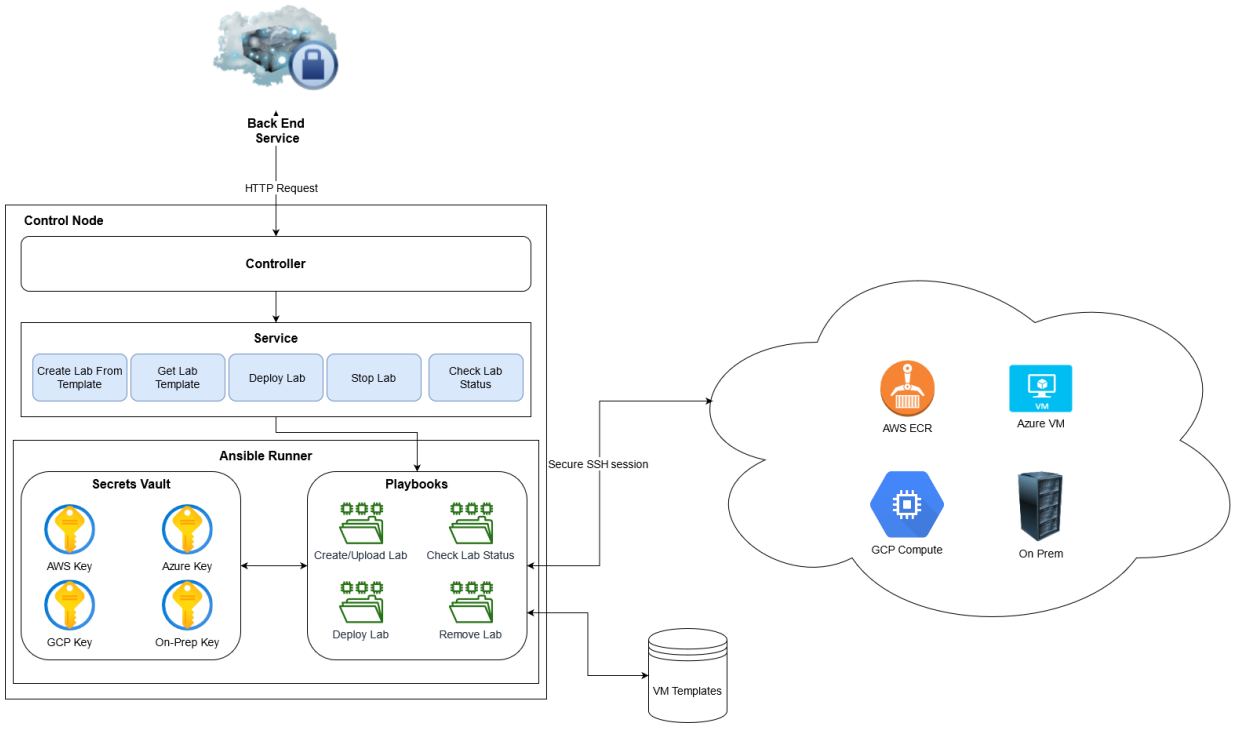

Figure 2.4.2: General Design Diagram

2.5 Design Implementation

Front-end

For our front-end, we utilized the react framework to create a high functioning graphical user interface. We created several components within our React project that all attempt to handle each use case of our application. To accomplish this, we utilize the React Redux for our state bindings, React Router to handle our routes, and React Bootstrap to handle our webpage formatting. To account for our use cases, our front-end contains three routes. A home route that displays generic information, a manage route to handle all requests to create and manage templates and to deploy any environments on any selected cloud platform, and a status route to display the statuses of deployed labs.

To receive templates and deployed labs, we make a fetch request to our API to collect an array of templates and deployed labs respectively (in addition to other information like subnets). To display this to the user, we utilize the react table package to organize this information into a table that sorts the information into easy to read rows. To modify or delete a template or lab, we request the user to select the template or lab from their respective row and press the modify or

delete button to send a put and delete request to our API respectively. To then deploy a template to a cloud platform, we request the user select a template, then request them to input some optional data related to the platform that they must select, such as select a subnet ID from a list of IDs associated with a given platform that are received from the API or “create one for me.” The user then submits this information and the front-end creates a post request to the API by placing the provided information in the request body.

Back-end

For our back-end, we implemented our design by utilizing the python flask framework. Within flask, we created a controller for receiving requests from the web application by exposing the application to the internet and had the control route to 5 main services. An AWS service that handles the the creation and management of AWS lab deployments, an Azure service that handles the creation and management of our Azure based lab environment deployments, a GCP service that handles the creation and management of the GCP lab deployments, an auth service to handle authentication of the API, and a mongoDB service to handle the retrieval and modification of saved lab deployments and templates. We also have an ansible service to run our ansible playbooks and a utility service to handle some general sanitization and validation.

We utilize the pymongo python library to connect to our flask API to our mongo database and we utilize the python library ansible runner to execute our ansible playbooks. We utilize the PyJWT library to help manage our authentication by requiring an authentication token for each request and generating the token for the requests.

The general flow of our API is that it first receives a request and our authentication service validates that the request is authenticated. If it is validated, the request sanitizes all input and begins to parse the request. If, for example, the user requests to deploy an environment to AWS, the request will contain a template ID on information on customization options, such as whether to create a subnet or not. The API will receive the template from the template mongo collection and use the information from that template and the request to gather information to deploy the virtual machines in the template. The back-end will then attempt to deploy each of those machines, along with the subnet if specified by calling the ansible runner. If the deployment is successful, it stores the information into the database. This flow is similar to every route but utilizing

different functions to achieve functionality after the request has been validated and sanitized. To view all API routes, see Appendix IV.

MongoDB

For our mongo database, we implemented our design by creating 3 collections, a configuration collection for storing our account information, a deployment collection for storing information on the deployed labs, and a template database for storing information on the template.

Overall implementation

Each component is separate from one another, but they utilize HTTP traffic to communicate with each other. Each component is hosted on its own docker container and they communicate with one another by a specified port. The front-end is hosted on a container with a public facing port (80) such that any user on the network can access it from their web browser. The back-end is hosted on a container also with a public facing port (5000). This needed to be public in order for the front-end to be able to communicate with it. Since the mongoDB will only ever communicate with the back-end, it is hosted on a container that is only exposed to the docker host and can be routed to the back-end service. An example communication of one of our use cases can be seen in **Figure 2.5.1**.

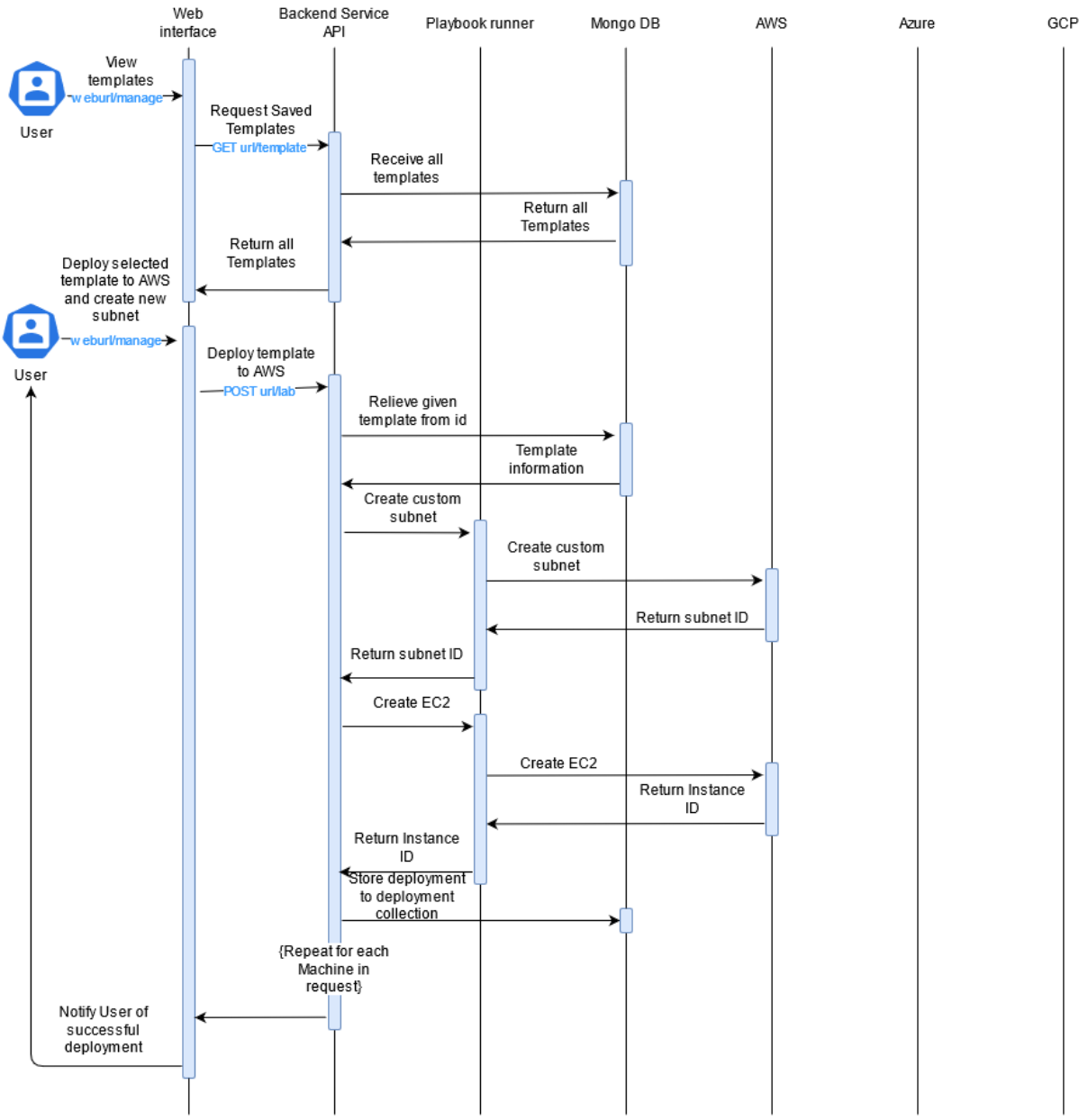

Figure 2.5.1: AWS Lab Creation Sequence Diagram

2.6 Design Evolution (From SE491)

The initial design that we considered implementing was a React front-end user-interface hosted on AWS S3 interfacing with an AWS API Gateway that was integrated with AWS Lambda functions for the back-end API.

There were 3 main reasons we didn't go this route. These reasons were, in short, more familiarity, increased simplicity, and host-independence.

The first reason is that not everyone on the team was familiar with the various AWS services that would have been required. This project was already on a steep learning curve, and since we had our ISU-provided virtual machine available to us, it just made more sense to go that route.

Second, architecting and developing an API with API Gateway and AWS Lambda(s) is pretty simple, but involved. Various AWS Lambdas would have been required, and, in a Flask API, all we need to do is add another route to the code to in order provide another API service (versus writing & provisioning another Lambda). Simplicity and ease of development was very important to us, and so, again, it made more sense to host our Flask API on our virtual machine.

The final reason is that we wanted our sponsor to be able to host the Cloud Environment Manager application anywhere, and not just on AWS. Using AWS S3, AWS API Gateway and AWS Lambda would have made that impossible as, obviously, the application would have been completely tied to the AWS cloud provider.

2.7 Security Concerns & Countermeasures

There are really 3 main components of the design that we had to ensure were secure. These components are the UI, the API, and the cloud providers. Protection of these 3 components was crucial, otherwise it would be very easy to run up an unintended and extremely large bill on all the connected cloud providers by creating endless amounts of heavily-specced labs, create major privacy issues by exposing user information, or create potential information exposure of sensitive information by exposing cloud configurations or structures.

For the development process, the user interface is hosted within the ISU network, which is important as the rest of the internet does not have access to the Cloud Environment Manager application. Since the project is planned to be open source, we expect the user to put some emphasis on security by hosting this software in a controlled network and by offering certificates for https traffic, aswell. If they wanted to expand the application and add additional security, then they could take the time to add an Identity and Access Management layer on top of the application to further protect from users they don’t want to have access.

Since the front-end generates output based on input received from the server, we sanitize the input that is received from any API call to assure that in the case of a malicious user being able to fabricate an API request/response, the user interface cannot render anything harmful and cannot be redirected to another site. We also planned to add the capabilities to have a cross site request forgery token to help prevent cross site forgery attacks.

During the development, the API and the services it provides are protected again by the ISU network and additionally by a static access key which must be present in all requests, so again it is up to the user to host this application in a secured network. If given more time, we would implement a login system using JWT tokens to access both the front-end and back-end. It is up to the user to decide whether to implement this or implement some type of API gateway to protect this API.

Since we have a database that we are interacting with and we are using a runner that executes commands on the host, we perform validation and sanitization of all inputs to the back-end. We do this by escaping potentially malicious strings that would be placed in the database and confirming that the input that will be used to run is sanitized and conforms to the general format of what is expected to be executed. To also assure that all functionality is as expected, we log many of the functions of the API and serve this to the user so that they may implement monitoring of these logs.

The cloud providers that we are integrating with have credentials that give access to them stored on our ISU-provided virtual machine. These credentials are used by our Flask API and Ansible Playbooks to retrieve, provision, and de-provision resources within the cloud providers. We are protecting these credentials by encrypting them within an Ansible Vault and giving only the account associated with our service access to it. This prevents credentials from being exposed to unnecessary actors.

Since this is a web application, we followed OWASP's security guidelines for securing a web application during development, including preventing the OWASP top 10. The above information addresses how we prevent several of these vulnerabilities and why it is important to consider them in a web application.

3 Testing

3.1 Unit Testing

Testing

During the course of the semester this was under utilized in favor of the other testing. Largely the components we were building were either to display received data, or to retrieve the same data. Because of this it is highly state based when testing, which made this much higher effort than the value we would get out of it.

Since this was shifted away from, we changed to testing logical components interfaces rather than effects.

3.2 Integration/Interface Testing

Interface Testing

It is important for each of our web facing APIs to be well tested and well defined. For this project we have a good procedure for testing both our web server, as well as the engine endpoints.

To do so each interface has been appropriately tested along all routes, and operations. To do this we used a REST service test utility(hoppscotch.io or postman). From here we insured the functionality of each interface as a whole unit.

Interface List

- Front end server web interface
- Back end server REST API

We were able to establish dummy routes to test functionality of the front-ends requests. This is to ensure that the front-end is sending valid requests in response to a given action.

Integration Testing

For this we have utilized the tools from the interface testing plan, but now tracking effects across the environment.

Additionally we recruited a targeted form of blitz testing to hit an integrated environment prior to moving into the well defined stage of acceptance testing.

3.3 Acceptance Testing

Acceptance Testing is the final stage of testing and was used after the unit and integration testing. The entire application should be fully developed and should be able to function as expected. When working on the acceptance tests, we made sure that assumptions and the constraints were considered ahead of the time. For instance, the execution of a certain use case of the application can be affected by different operating systems and web-browsers; the acceptance tests were performed on each operating system and web browser that would be specified ahead of time. An example of the acceptance testing: if the user would be successfully able to select/design template by following steps:

Create Template Acceptance Testing:

Need 1: User is able to add the template by doing the following:

1. By creating a new template
2. Navigating to the Create Template Page
3. Filling out the template creation form
4. Submit the template creation form

Acceptance Criteria:

- The user was able to successfully create the template.
- The template shows in the templates list.

Delete Template Acceptance Testing:

Need 1: User is able to remove selected template by doing the following:

1. Selecting a template from templates list
2. Selecting the “Delete Template” button
3. Selecting the “Confirm Delete” button

Acceptance Criteria:

- The template is removed from the templates list.

Deploy Template Acceptance Testing:

Need 1: User is able to deploy a lab by doing:

1. Selecting a template from templates list
2. Selecting the “Deploy Environment” button
3. Filling out the form with appropriate data
4. Selecting Confirm Deploy button

Acceptance Criteria:

- The lab appears in labs list

- The instance is created inline with the selected values, and values in the template.

More Info Acceptance Testing:

Need 1: User is able view more info on about a lab by doing the following:

1. Selecting the lab from labs list
2. Selecting the “More Info” Button

Acceptance Criteria:

- The user was able to see a json (may have some delay).
- The json reflect lab status

End Lab Acceptance Testing:

Need 1: User is able to terminate a lab by doing the following:

1. Selecting a lab from the lab list
2. Selecting the “End Environment” button
3. Selecting the “Confirm End” button

Acceptance Criteria:

- The lab is removed from lab list
- The lab ended in whatever cloud platform it was deployed.

4 Appendices

Appendix I: Operation Manual

4.1 Setting Up Production

The installation of this application requires a linux machine with docker installed and sudo access.

To install the application, navigate to the git repository: https://github.com/AOsmankic/sdmay21-39 and download the repository to some temporary working directory on the linux machine. From there, you must collect credentials from each cloud provider and add them to the security_keys directory that was downloaded from the repository. To create and gather the credentials for each platform, follow the following steps:

For AWS, log into your AWS console, access the IAM service, and select user. From there you will add a user and set its access type to programmatic. Then attach the EC2FullAccess policy (or select specific EC2 policies to fine grain security access) and finish the creation process. You will be able to generate an access and secret key for the user and save it to file named aws_keys.yml in the format of:

aws_access_key: {access_key}
aws_secret_key: {secret_key}

Save this file into the security_keys folder in the root of the project.

For GCP, access your GCP console and create a service account by accessing the IAM manager and selecting Service Accounts. From there, you must create a new service account by giving it some meaningful name and ID. Add the compute admin role to the service account and finish the account creation. Then you must access this account and generate a new key by selecting keys -> add keys and selecting JSON. This will download a json credentials file that must be stored in that credentials folder.

For Azure, log into your azure portal and collect the subscription and tenant id for the account and follow this guide, https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-create-ser

vice-principal-portal, to create a client and secret id. Collect all this information and save this in a file named azure_keys.yml in the format of:

client_id: {your client id}
secret: {your secret}
tenant_id: {your tenant id}
subscription_id: {your subscription id}

Save this to the security_keys folder as well.

After collecting and storing all the credentials, run the installation script located in the root directory. For usage help, run *./install.sh -h*. After following the above steps, run *sudo ./install.sh -i* to begin the installation process. After the install script has completed, the web interface will be exposed to port 80 of the local machine and the API will be exposed to port 5000. It is up to the user how they want to expose these services to the network.

4.2 Using the Web Application

In a browser navigate to whatever url the web server can be accessed by (this will vary based on the setup)
The web app is intuitive for the user and most of the fields are labeled such that if you were familiar with the platform you are deploying to you would be able to fill the form correctly.

Manage Templates Pages:

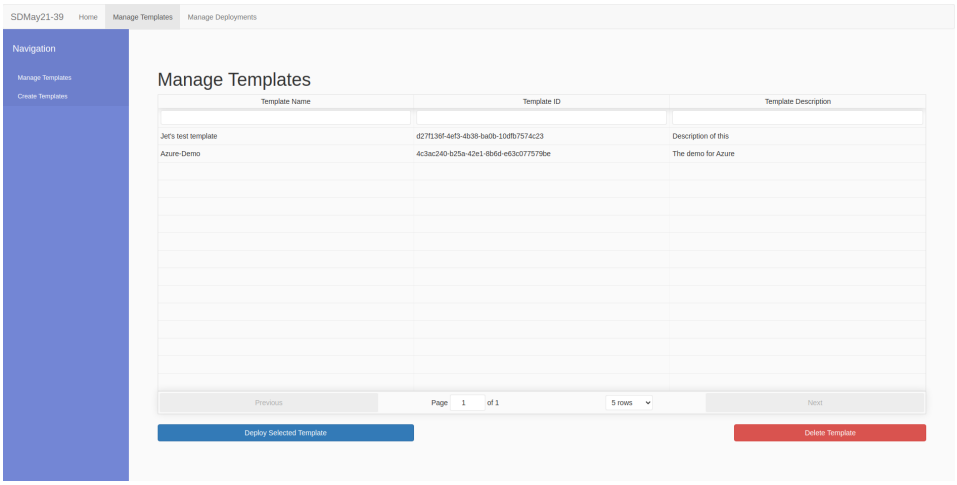

Figure 4.2.1: Manage Template Landing Screen

Figure 4.2.1 shows the page where template actions stem from. Start by selecting the create template from the nav bar on the left.

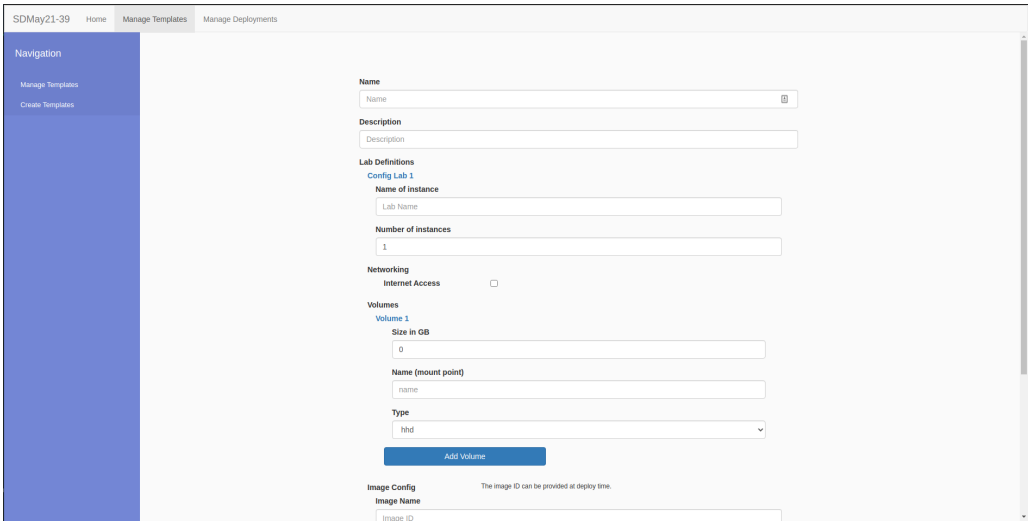

Figure 4.2.2: Manage Template Creation Page

Creating a template:
Fill out the form with appropriate data. The creation page is seen in **Figure 4.2.2.** Take note you are configuring one machine at a time and are only adding that machine to the template if you select the add lab button. Take note of the “Config Lab 1” text in blue as this shows what machine, and how many you have added to the lab.
The images currently need to be specifically the platform you will deploy to.

Adding a volume to the lab:
During lab creation you may need to add an extra volume beyond the base volume. Just fill out the volumes sub form. Note the blue “Volume 1” text which functions much like it does for the lab creation text.

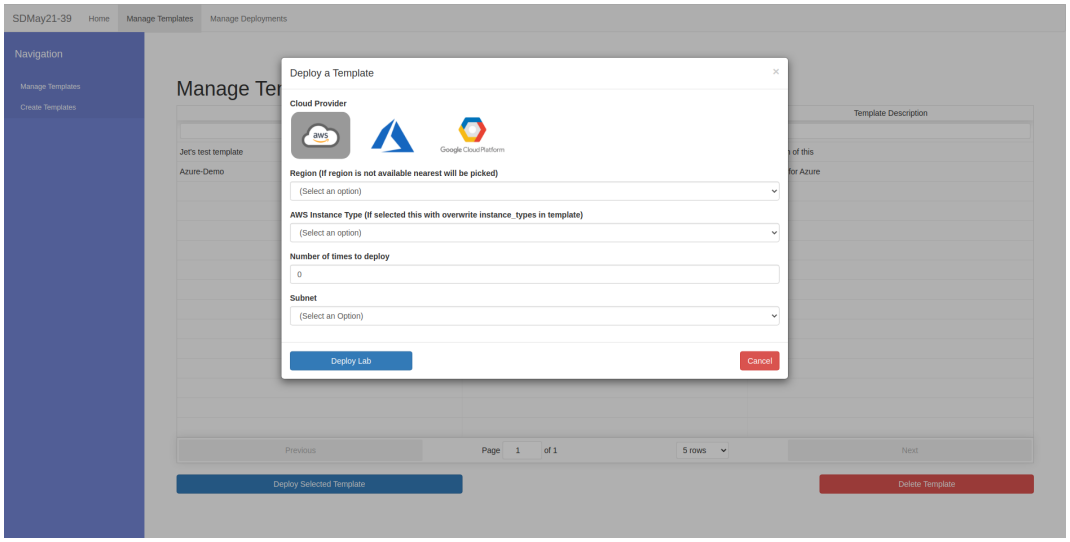

Figure 4.2.3: Deploy Template Modal

Deploy Template:
Deploy template by selecting a template from the table and clicking the “Deploy Selected Template” button.Fill out the form with appropriate data for whatever Cloud Provider you wish to deploy with. Select “Deploy Lab” button.

This may take a few minutes, and if staying on the same managed templates page a notification will show stating the status of the deploy. You can still trigger other actions while this processes, but once you navigate to another page the toast notification won't trigger.

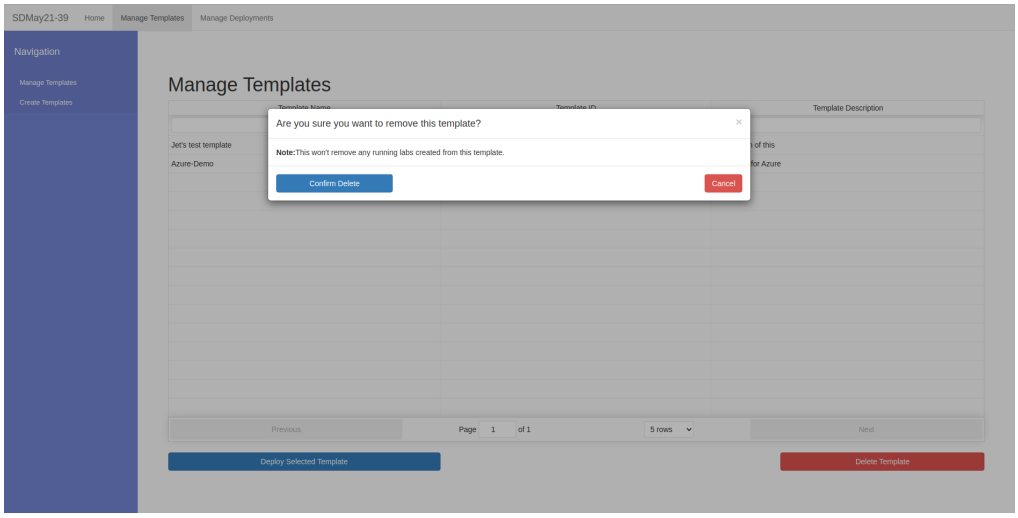

Figure 4.2.4: Delete Template Modal

Delete Template:

Figure 4.2.4 shows the delete modal. It works by selecting the template you want to delete, and selecting the “Delete Template” button. Once on this page select the “Confirm Delete” button to remove the template.

Manage Deployment Page:

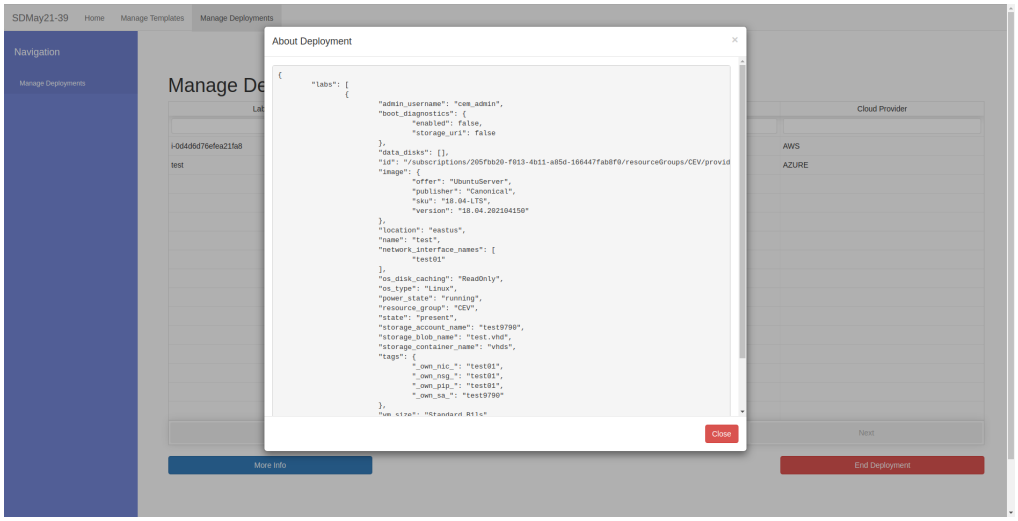

Figure 4.2.5: More Info

More Information:

To get more information about a deployed lab, select the lab and click the “More Info” button. Note this is executing a chain of requests, so it may take a minute to load. This will also display an error, when no deployment is selected.

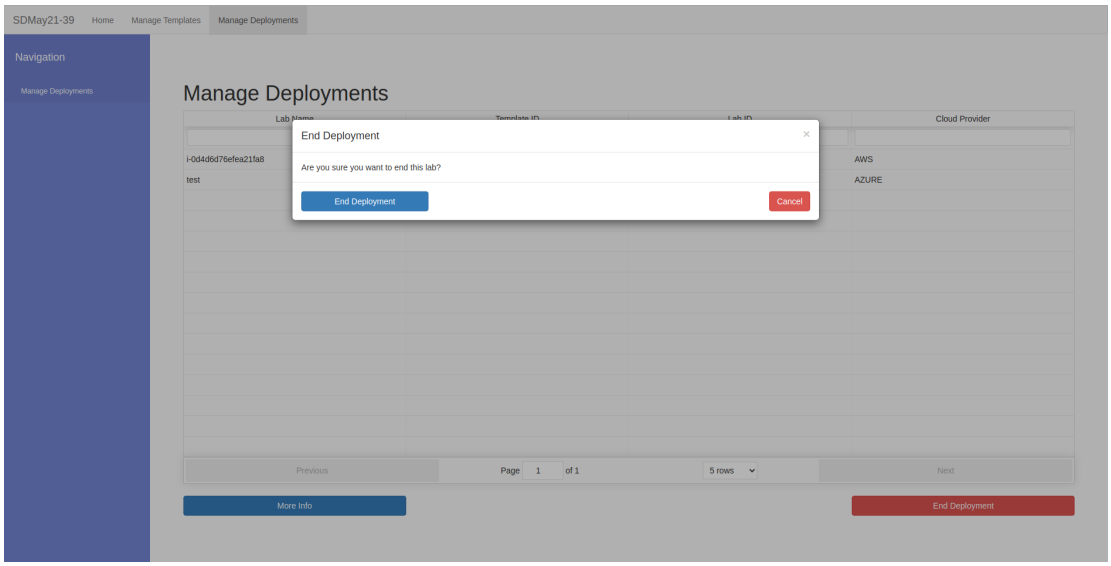

Figure 4.2.7: End Deployment Modal

Ending Deployments:
To end a deployment select the lab, and click the “End Deployment” button. Note this is executing a chain of requests, so it may take a minute to load. This will also display an error, when no deployment is selected. Unlike other areas, you should leave this open until the action is executed, otherwise you may not be able to see the status of the end deployment without refreshing.

Appendix II: Alternative / Other Initial Versions of Design

Our design has gone through many different iterations to try and account for our requirements and use cases. Many of these designs we decided to modify or abandon because of some technical issue, some functionality error, or just because it did not fit what we needed. A list of some of our original designs along with some diagrams are below.

- Our first design involved utilizing Docker and the container managers for each cloud platform to create our lab environments. We initially were going to utilize this for Docker’s simplicity and consistency, but found that the client’s needs required a valid virtual machine utilizing native cloud services for networking and management to allow for controlled environment management and machine access, so we decided to not implement this.

- We initially were considering creating a platform that would deploy individual virtual machines instead of labs to allow for greater control and allow for more configuration options from the user, but found this to not fit a critical requirement of allowing modular “bundles” to be deployed to each environment.

- One of our earlier designs implemented ESX to simulate an on premises platform, but with the given resources and the time constraints, we could not implement a valid ESX platform for development.

- One of our early designs also included utilizing an image repository that would allow the user to create custom virtual machine images and select those for deployments. We were also planning to have a translator for converting images from one platform to another utilizing a flow similar to **Figure 4.3.1**

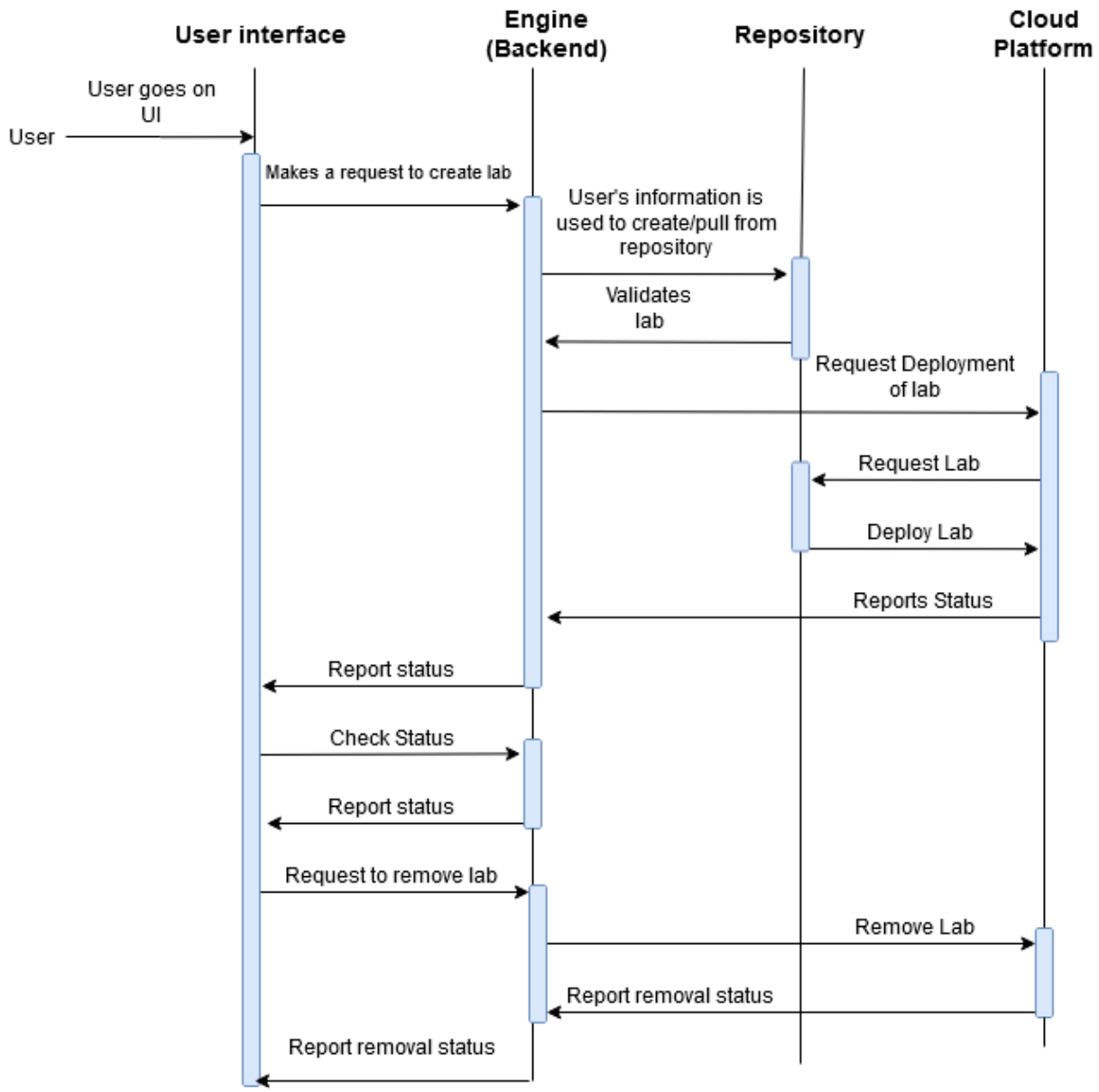

Figure 4.3.1: Initial Sequence Diagram

- Our original design seen in **Figure 4.3.2** was focused on hosting our application on AWS using serverless services. This would allow us to bundle the application on the marketplace for general consumer use and easy installation, but after attempting to secure funding or necessary resources, we found that we could not afford to host this on AWS after several design iterations.

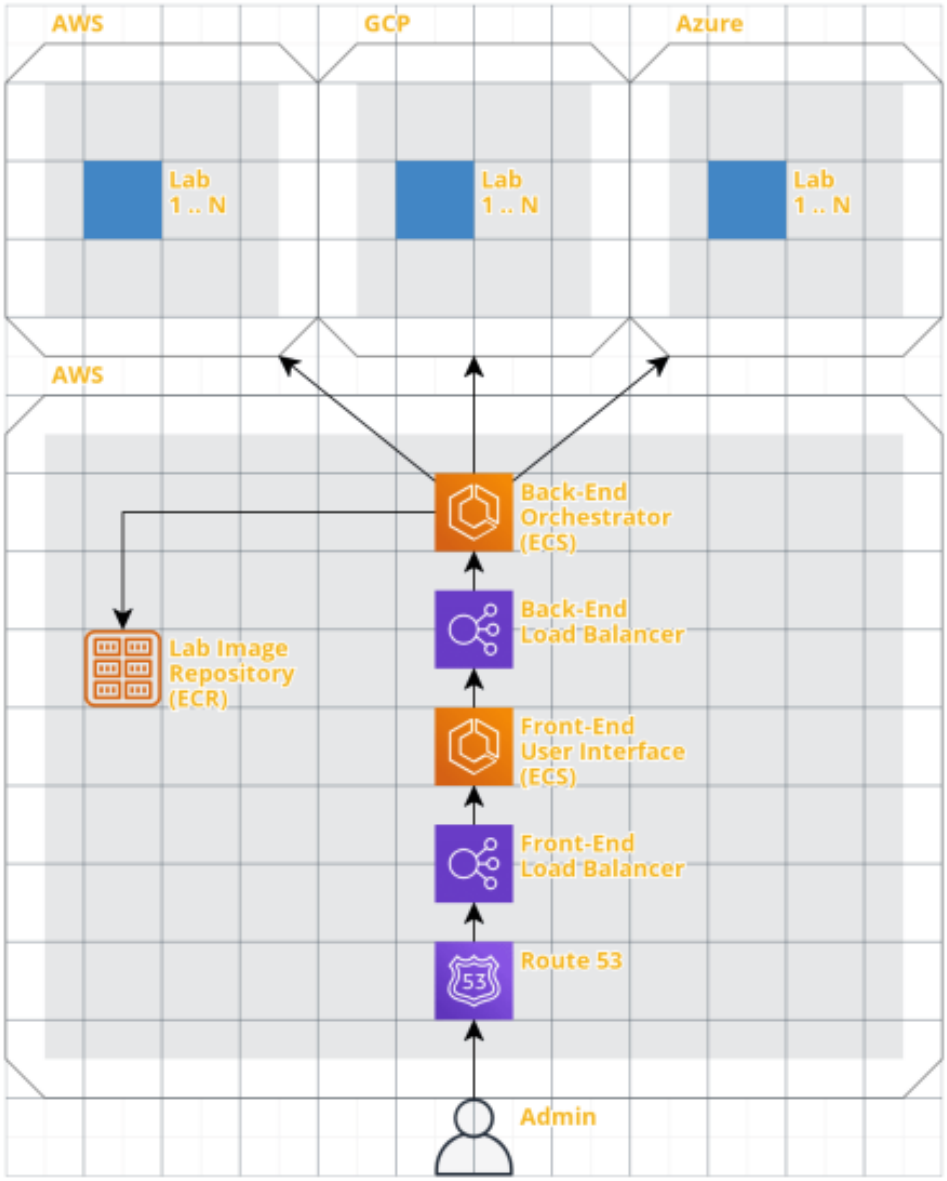

Figure 4.3.2: AWS Block Diagram

Appendix III: Other Considerations

While developing our design, we wanted to implement many other items. This includes features such as a fully functionally login system with specified roles to control user access to the application, a feature to have an image uploader and translator to allow a user to customize and deploy custom images to each platform, and a feature to allow the user to add multiple account configurations to the application to manage multiple

cloud accounts for the same platform with customization for individual users with specific roles to have access to specific features of each account. It is important to note that all these features are valid features that can be added in the future, but given the time constraints, could not be done so during this semester.

Appendix IV: API Documentation

Route	Description	Parameters
DELETE /lab	Destroy a lab within a cloud provider	lab_id OR vm_name platform
GET /lab	Retrieve a lab within a cloud provider	lab_id OR vm_name *optional* platform
POST /lab	Create a lab within a cloud provider	platform region subnet_id template_id vm_name (Azure)
GET /deployed	Provides a list of all deployed labs	lab_id OR vm_name *optional* platform
GET /helper	Provides a list of cloud platform objects	object
DELETE /configuration	Removes a configuration parameter	name
GET /configuration	Provides a list of all configuration settings	N/A
POST /configuration	Creates a new configuration parameter	name value
DELETE /template	Removes a lab template	template_id
GET /template	Retrieves a lab template	N/A
POST /template	Creates a lab template	template (in JSON form via body)
GET /test	Dummy response to verify API is up	N/A

Table 1: API Routes