

Project Title: LLM-Driven Cloud Architecture Automation

Team Name: SysSavvyAI

Team Members: Nishant Nagururu (Backend developer), Aditya Jithesh (Frontend developer)

CIS 4914 Senior Project

Advisor: Dr. Ashish Aggarwal

Advisor Email: ashishjuit@ufl.edu

Presentation Link: <https://youtu.be/xHrxO3EJD8>

Keywords	3
Project Overview	3
Project Name	3
Project Description	3
Project Goals and Objectives	4
Project Scope	4
Deliverables	5
Overview	5
Backlog	6
Technical Details	6
Codebase Overview	6
Key dependencies and libraries	8
Deployment instructions	8
Database schema and data migration details	10
Software Architecture	11
Testing Information	11
User Documentation	12
Transition Plan	17
Conclusion & Learnings	17
Acknowledgements	19
Appendices	19
Appendix A - Assistants API Prompt	19
Biographies	20

Keywords

Prototyping, Cloud Architecture, AI, LLMs, Software Deployment, Automation, AWS

Project Overview

Project Name

LLM-Driven Cloud Architecture Automation

Project Description

Our platform transforms high-level system requirements into production-ready AWS architectures through an end-to-end, LLM-driven workflow. Users simply supply a concise description of their goals—and, optionally, a GitHub repository containing relevant code—and our system automatically analyzes both inputs to generate an interactive flowchart of the AWS resources best suited to their needs. A built-in chatbot converses with users, answering questions and adjusting the flow chart as the user desires. Once the user understands and approves the design, the platform leverages LLMs to produce Terraform (Infrastructure-as-Code) configurations, orchestrates deployment, and monitors the process in real time. Should any deployment errors arise, our feedback-loop mechanism feeds logs back into the LLM for iterative corrections. Throughout, users can review a complete history of Terraform runs, compare cost estimates for each iteration, inspect deployment logs, and even re-execute prior configurations with a single click, ensuring full visibility, control, and cost optimization at every step.

Project Goals and Objectives

Our primary goal was to dramatically reduce the manual overhead and human error traditionally associated with designing and deploying cloud infrastructures. We aimed to automate the end-to-end workflow—from requirement elicitation through deployment—so that both seasoned cloud architects and smaller teams without deep DevOps expertise can consistently produce cost-optimized, secure, and scalable environments. Specific objectives included: (1) understanding high level requirements and turning it into an actionable plan using AWS resources; (2) automating the creation and validation of Terraform scripts; (3) polling and reporting on deployment status and logs in real time; (4) establishing feedback mechanisms where the LLM analyzes deployment errors to iteratively correct IaC artifacts; and (5) querying comprehensive information about run history to give users a good understanding of their cloud architecture. Ultimately, the platform seeks to accelerate time-to-market, reduce overspending, and enforce best practices across diverse cloud configurations.

Project Scope

This project, in its current state, encompasses all stages from user requirement capture to the **first** successful deployment of the generated infrastructure. In scope are: high-level requirement intake, LLM-driven clarification dialogs, architecture diagram and proposal generation, real-time cost estimation against major cloud providers, Terraform code synthesis, automated deployment orchestration to AWS with status polling, error handling via iterative LLM feedback, and maintaining Terraform code history. Out of scope for our project at this stage are cloud providers other than AWS, LLM-driven questioning to get details about the architecture, extremely

complex architectures, and integration into CI/CD pipelines. Overall, the project currently only supports one-time deployments for architectures on the simpler side. This current scope supports one part of our initial target audience - students and small developer teams who are somewhat unfamiliar with the cloud and Terraform, who are looking to deploy a simple project as quickly as possible. Although the current scope fulfills our initial objectives, this scope can be increased by gathering more high-quality and complete Terraform samples for the vector store and integrating with CI/CD to allow for continuous and automated deployment. This would allow our tool to support experienced engineers at an enterprise level to manage and deploy complex production-stage cloud architecture.

Deliverables

Overview

Our primary deliverable is a unified codebase split into a React-powered frontend and a Python Flask backend. The React app communicates directly with a Llama-based LLM to dynamically generate interactive AWS flowcharts and power the conversational chatbot interface. Behind the scenes, the Flask API handles all Terraform operations—invoking the Terraform Cloud/API to apply configurations, retrieving historical run data, and exposing deployment logs. It also integrates with the OpenAI Assistants API (backed by vector storage) to author Terraform code from user intents and to iteratively diagnose and fix any failed deployments. Since we are unable to share our API keys, Terraform configuration, and OpenAI configuration, setup instructions for these will be available on GitHub along with our code.

Backlog

The major backlog item that separates our product where it is from an easily usable end product is making it so users can have their accounts and configure their own Terraform organization directly from the dashboard. Although this was mostly out of scope from the beginning of the semester, this would need to be done before the website and backend are hosted. As is, it is not reasonable to use this as a hosted website. Also, I think we wasted a lot of time this semester trying to find quality sources of Terraform data.

Technical Details

Codebase Overview

As mentioned previously, the codebase is organized into two top-level directories—frontend and backend. The frontend directory contains a `src/` folder with the core React application, while the backend directory houses a Flask server and schemas that are used to interface with MongoDB. Here's the current structure for each:

```
frontend/
├── src/
│   ├── App.css
│   ├── App.js
│   ├── DesignArchitecture.tsx
│   ├── NewRun.tsx
│   ├── PreviousRuns.tsx
│   ├── components/
│   │   ├── FormattedApplyLogs.tsx
│   │   ├── RunCard.tsx
│   │   └── Wizard/
│   │       ├── ChatBot.tsx
│   │       ├── RequirementsForm.tsx
│   │       └── RequirementsWizard.tsx
```

```
|
|   |— ServiceSelection.tsx
|   |— VisualizationPreview.tsx
|   |— index.ts
|— index.css
|— index.js
|— services/
|   |— iconMatcher.ts
|   |— llmService.ts
|— types/
|   |— react-aws-icons.d.ts
|   |— types.ts
```

Within frontend/src, `App.js` serves as the application's entry point—wiring up routing and global style. Our website has three main tabs, which are set up in `App.js`. The Design Architecture page, New Run page, and Previous Runs page. The `DesignArchitecture.tsx` component renders the interactive AWS flowchart, while `NewRun.tsx` and `PreviousRuns.tsx` handle the UI for kicking off fresh Terraform runs and browsing historical executions, respectively. Reusable pieces live under `components/`: `FormattedApplyLogs.tsx` prettifies raw Terraform output, `RunCard.tsx` is the modal that appears with more details when a user clicks on an individual run, and the `Wizard/` subfolder encapsulates the multi-step requirements wizard (from chat-based clarification in `ChatBot.tsx` to service selection and visualization preview) that is on the Design Architecture page. The `services/` directory abstracts external integrations—`iconMatcher.ts` maps service identifiers to SVG icons, and `llmService.ts` manages prompts and responses with the LLM. Finally, the `types/` folder defines TypeScript interfaces for AWS icons and shared data models, ensuring type safety throughout the UI.

```
backend/
|— app.py
|— schemas/
|   |— runModel.py
```

The backend is a Python Flask application (`app.py`) that exposes REST endpoints to receive user requirements, invoke Terraform operations via the Terraform Cloud/API, fetch logs and historical run data, and orchestrate LLM-driven code generation and error remediation through the OpenAI Assistants API. The `schemas/` directory defines data models. Currently, it only contains the `runModel.py`, which defines the schema for `Run` documents stored in MongoDB. These `Run` documents are used to store the Terraform code for a particular run because Terraform does not store it.

Key dependencies and libraries

The dependencies for the backend and frontend are in the `requirements.txt` file and the `package.json` file, respectively. The key dependencies of the backend are `Flask`, `mongoengine` for interfacing with MongoDB, and `openai` for interfacing with the OpenAI Assistants API. The frontend, on the other hand, has no major dependencies other than Node because it gets all its info from the Llama/Groq API or the backend API.

Deployment instructions

The backend and frontend folders both have .env files that are needed for the website to function. The backend .env file has the following keys:

```
HCPT_TOKEN={TERRAFORM TEAM TOKEN}
HCPT_ORG={NAME OF TERRAFORM ORGANIZATION}
HCPT_WORKSPACE={NAME OF TERRAFORM WORKSPACE}
MONGODB_URI={MONGODB CONNECTION URI}
OPENAI_API_KEY={OPENAI API KEY}
GITHUB_TOKEN={GITHUB PERSONAL ACCESS TOKEN}
VECTOR_STORE_ID={OPENAI VECTOR DB ID}
ASSISTANT_ID={OPENAI ASSISTANT ID}
```


Getting all the values for this .env requires setting up five things: Terraform, AWS, MongoDB, GitHub, and OpenAI. Starting with Terraform, make an account, make an organization, and within that organization, make a workspace. Fill in the names for the organization and workspace in the .env. In your AWS account, go to the IAM section and make a user. Assign it permissions for all resources that you want your Terraform scripts to be able to target. Make sure to copy the access key and secret access key. Then, go back to the Terraform console and make two Workspace Variables. They should be called AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY. The permission level for both should be sensitive. Next, in the Terraform console, if you then go to the Settings > API Tokens page, you can make a team API Token, which will be the HCPT_TOKEN in your .env. Next, go to MongoDB and make an Atlas instance. When it prompts you, make a username and password and fill them into the connection URI template it gives you to get the MONGODB_URI. For the GitHub token in the .env, go to Settings > Developer Settings > Personal Access Tokens. Create a personal access token and put it in the .env. Finally, for the OpenAI setup, make an account, go to the OpenAI Playground settings, and make an API Key. Next, go to the Assistants Tab in the console and make a new assistant. Set the temperature to 0.25, the model to 4o, and the output type to 'text'. See Appendix A for the prompt. You will need to change this slightly to reflect the name of your organization and workspace. Copy the ID of the assistant for the .env. Finally, create a vector store, attach it to the assistant in the console, and copy the ID of the vector store into the .env. To initialize the vector store with the documents from the Terraform Registry, run the 'http://localhost:4000/store-readmes' route in the backend.

The frontend .env has the following keys:

```
REACT_APP_GROQ_API_KEY={GROQ_KEY}
```

This is as simple as making a Groq account and making an API key from the console. There is no additional configuration necessary.

The backend dependencies can be installed by starting a virtual environment in Python 3.12 and running `pip install -r requirements.txt` from inside the backend folder. To install the frontend dependencies, run `npm install` from the frontend folder. To start the backend, execute `python app.py`, and to start the frontend, in a separate terminal, in the frontend folder, run `npm start`.

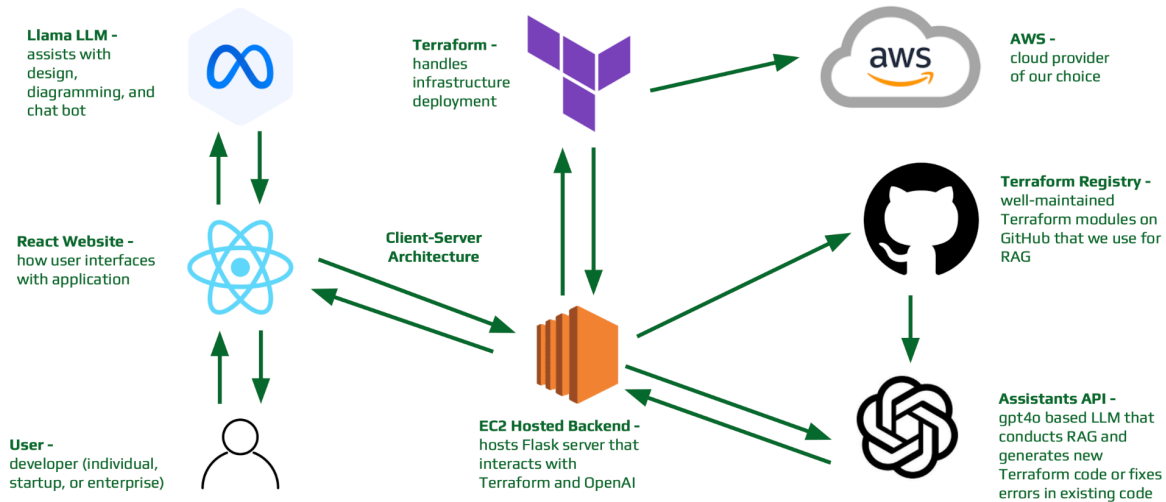
Database schema and data migration details

```
class TerraformFile(EmbeddedDocument):
    file_name = StringField(required=True)
    file_content = StringField(required=True)

class Run(Document):
    run_id = StringField(required=True)
    tf_files = ListField(EmbeddedDocumentField(TerraformFile), required=True)
    workspace_id = StringField(required=True)
    organization_name = StringField(required=True).
```

The only documents stored in MongoDB are the Run documents. The schema is quite self-explanatory from the code above, which is also in the backend/schemas/runModel.py file. This is needed mostly to store the Terraform files that trigger each run, since Terraform doesn't store them. No migration is necessary because if you choose to configure this project with your Terraform account and AWS account, our run history isn't relevant or necessary.

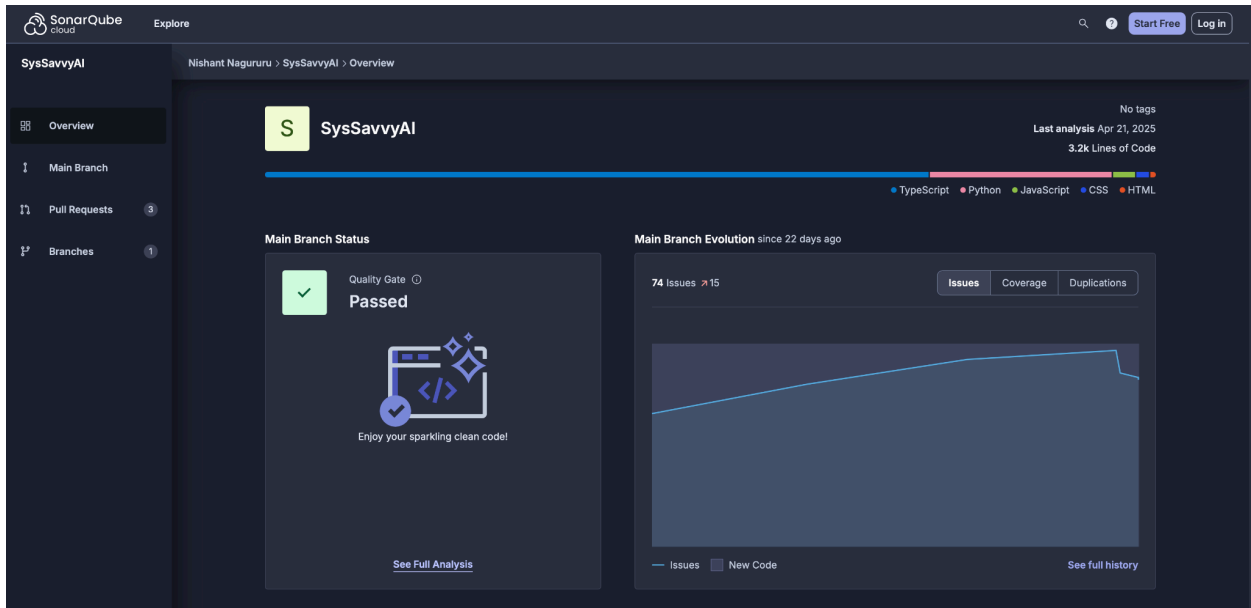
Software Architecture Diagram



The above diagram sums up our architecture well. The only clarification is that the React Website interfaces with the Llama LLM via Groq. Also, the backend interfaces with GitHub for the Terraform modules and gets context on your GitHub when it is relevant to deploying.

Testing Information

We opted for static code analysis rather than unit tests or integration tests. This is because most of our tools are API dependent, so the tests we could write would be relatively simple. We stuck to thorough API testing through Postman and blackbox testing of the entire website. This has been sufficient for our project. The static code analysis has helped us maintain good coding practices and write secure code. As you can see below, SonarQube was configured to scan the main branch after each pull request. As of the most recent pull request, all major security and readability issues have been fixed, and the Quality Gate was passed.



As the repository expands, it may be helpful to add unit tests to the frontend and mock API tests to the backend, but we do not have any at the moment.

The only outstanding bug is that sometimes, on the New Run page, error messages do not show up in a pretty format. There needs to be improvements to the parsing logic in the backend or the frontend.

User Documentation

Step 1: Enter a description of the final product you want and any specifics you can provide on the architecture you want to achieve. If you are deploying a website, API, or anything else that has accompanying code, attach a link to a public GitHub with your code so that the LLM can get context on your code and better understand how to deploy it.

DESIGN ARCHITECTURE
NEW RUN
PREVIOUS RUNS

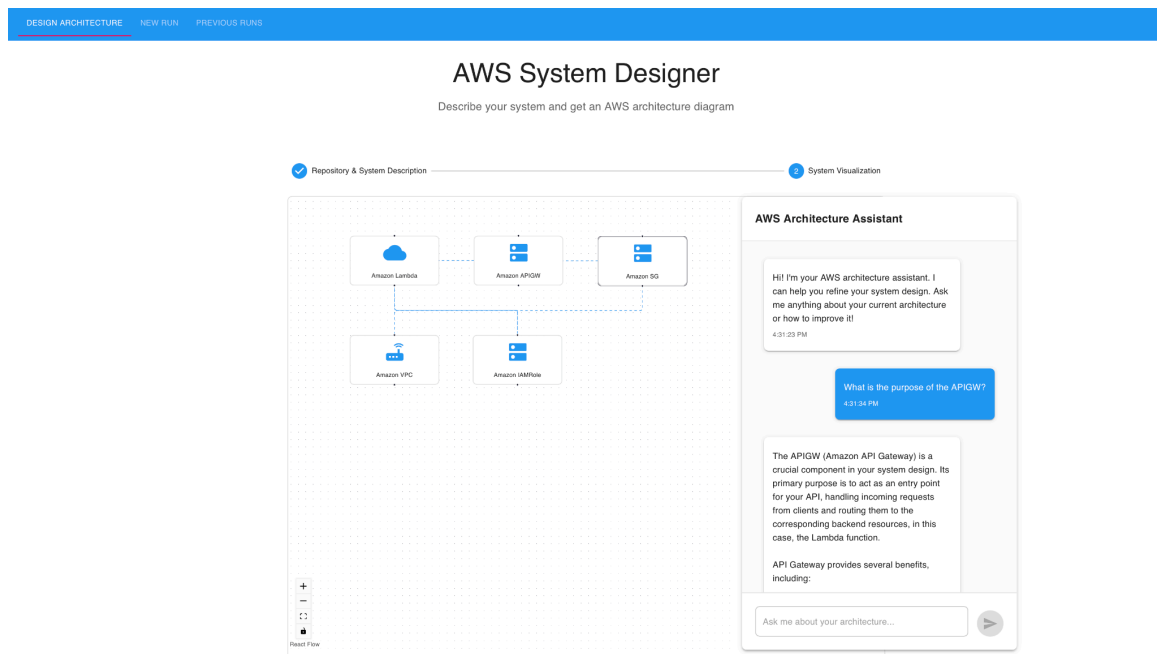
AWS System Designer

Describe your system and get an AWS architecture diagram

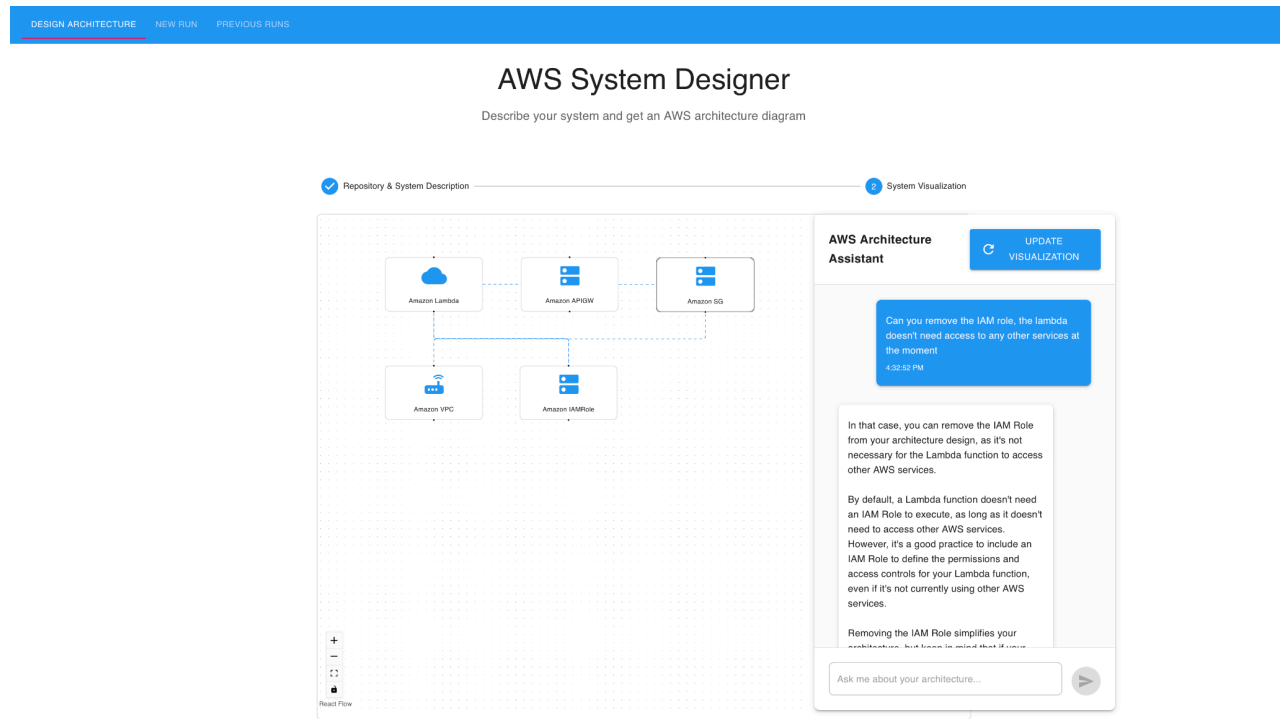
1 Repository & System Description
2 System Visualization

BACK
NEXT

Step 2: Ask any questions about the initial architecture that the AI comes up with. Click on the icons in the flow chart to get a better idea what each component is for.



Step 3 (Optional): If there is any component that is unnecessary or out of scope, or a component you want to add, bring it up in the chat. The chatbot will then provide you the option to update your visualization to request the update you asked for.



Step 4: After you are satisfied with your architecture design, click the ‘Finish’ button at the bottom of the page. This will prompt Llama with the analysis of your GitHub, if given, along with the updated flow chart. It will create a detailed description of what your Terraform file should look like and take you to the New Run page, filling in said description into the text box. Make any edits you feel are necessary.

DESIGN ARCHITECTURENEW RUNPREVIOUS RUNS

New Run

UPLOAD FILESGENERATE FROM DESCRIPTION

Enter Architecture Description

The Terraform script will create an AWS Lambda function using the Python code from the GitHub repository at https://github.com/Nishant-Nagururu/test_lambda. The script will first download the 'lambda_function.py' file from the repository and package it as a Lambda function. It will then create an IAM role with the necessary execution permissions and assign it to the Lambda function.

Next, the script will create an API Gateway REST API with a single resource and method, which will trigger the Lambda function. The API Gateway will be configured to use the Lambda function as its integration target.



To allow incoming traffic to the API Gateway, the script will create a security group with a rule allowing incoming traffic on port 443 (HTTPS). The API Gateway will be associated with this security group.

Finally, the script will create a VPC and associate the security group with it, ensuring that the API Gateway is deployed within the VPC.

The Terraform script will include the following elements:

- * An 'aws_lambda_function' resource to create the Lambda function, using the 'lambda_function.py' file from the GitHub repository.
- * An 'aws_iam_role' resource to create the IAM role with the necessary execution permissions.
- * An 'aws_api_gateway_rest_api' resource to create the API Gateway REST API.
- * An 'aws_api_gateway_resource' and 'aws_api_gateway_method' resource to create the API Gateway resource and method.
- * An 'aws_api_gateway_integration' resource to integrate the Lambda function with the API Gateway.
- * An 'aws_security_group' resource to create the security group with the incoming traffic rule.
- * An 'aws_vpc' resource to create the VPC and associate the security group with it.

The script will use the GitHub repository URL to download the 'lambda_function.py' file and package it as a Lambda function.



GENERATE

Step 5: After you click ‘Generate’, it will load for a while and then produce the Terraform code to meet your needs. Once again, make any necessary edits, fill in any information like port numbers that are missing, and click the ‘Upload’ button.

DESIGN ARCHITECTURENEW RUNPREVIOUS RUNS

New Run

UPLOAD FILESGENERATE FROM DESCRIPTION

Terraform Configuration


```
variable "AWS_ACCESS_KEY_ID" {
  description = "AWS Access Key ID"
  type      = string
}

variable "AWS_SECRET_ACCESS_KEY" {
  description = "AWS Secret Access Key"
  type      = string
  sensitive = true
}


variable "lambda_function_name" {
  description = "Name of the Lambda function"
  type      = string
  default   = "my_lambda_function"
}

variable "lambda_repo_url" {
  description = "The GitHub repository URL for the Lambda source"
  type      = string
  default   = "https://github.com/Nishant-Nagururu/test_lambda"
}

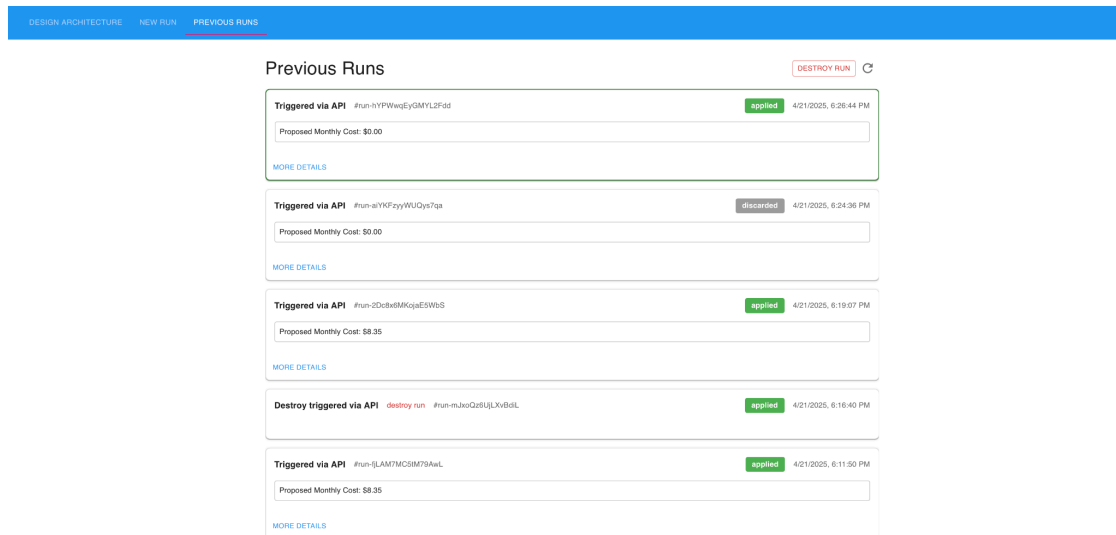
variable "lambda_artifact" {
  description = "Name of the artifact (.zip file) to be generated"
```



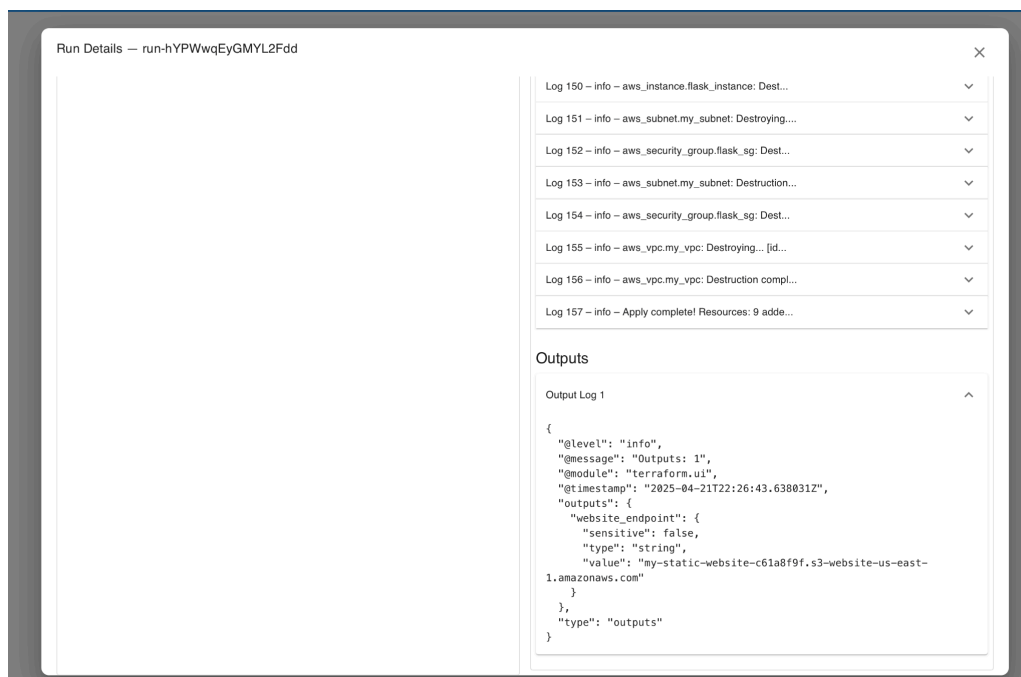
UPLOADING...



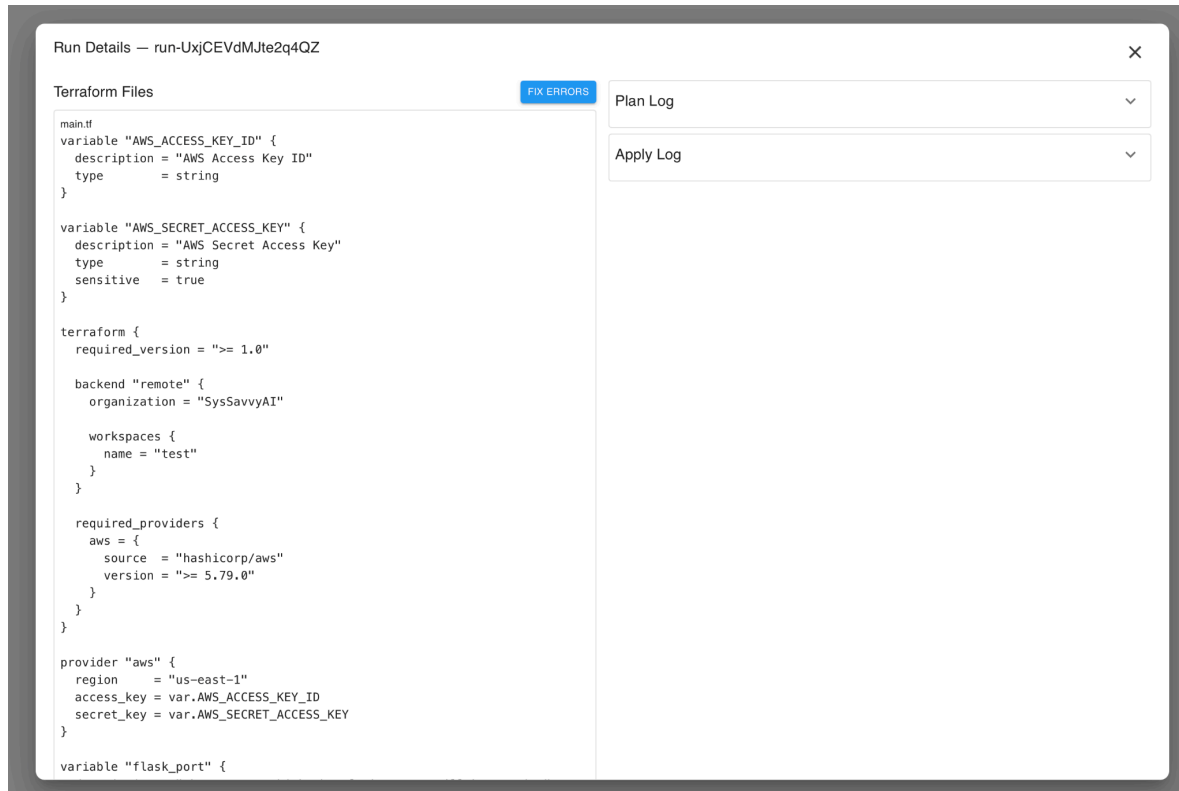
Step 6: Go to the ‘Previous Runs’ screen and Approve the run once it is done planning. If all goes well it will be applied!



Step 7: If you click on ‘MORE DETAILS’, you will see the code that was run and the plan/apply logs. If you scroll to the bottom of the apply logs, you will see relevant outputs, such as the link to your API or website:



Step 8 (If initial Run errored): If the initial run errored and you click on ‘MORE DETAILS’, there will be an option to ‘Fix Errored Run’. It will prompt ChatGPT with the error log and the previous Terraform code so that it can fix it.



Step 9: Once you are done with whatever AWS architecture you have provisioned, you can easily tear it down by simply triggering a ‘destroy run’. This can be done by pressing the ‘Destroy Run’ button. Make sure to approve the run after it is done planning. Make sure to delete your resources this way to ensure you don’t end up paying for unterminated resources in the long term.

DESIGN ARCHITECTURE

NEW RUN

PREVIOUS RUNS

Previous Runs

DESTROY RUN

Destroy triggered via API

destroy run

#run-2VVGchtao7UqbtU

cost_estimated

4/23/2025, 6:56:37 PM

APPROVE RUN

DISCARD RUN

Triggered via API

#run-FDuHkwuZ52KE3o1z

applied

4/23/2025, 6:29:05 PM

Proposed Monthly Cost: \$8.35

MORE DETAILS

Destroy triggered via API

destroy run

#run-s8PSHABPqPxSGjRa

applied

4/23/2025, 6:26:08 PM

Triggered via API

#run-2chTRdD7aGgdn5KG

applied

4/23/2025, 6:21:57 PM

Proposed Monthly Cost: \$9.15

MORE DETAILS

Triggered via API

#run-gDqnUNMX7FanuRYQ

errored

4/23/2025, 6:19:43 PM

MORE DETAILS

Triggered via API

#run-jNMWEQJkcStEbvq

applied

4/22/2025, 4:08:16 PM

Proposed Monthly Cost: \$0.00

Transition Plan

All of our code is in GitHub, and instructions on how to set up all of the supporting architecture are detailed above. If any future groups want to continue working on this, it should be relatively easy for them to pick up with this.

Conclusion & Learnings

OpenAI's LLMs are currently not performant out of the box at Terraform generation. Syntax is often wrong or highly deprecated. Our main approach to fixing this was to use Terraform. Registry's modules. However, even though we provided all of these modules, their implementations, and the parameters they took, the LLM was unable to understand how to translate the requirements into the use of these modules. The LLM struggled with this abstraction. Even when we tried uploading the examples from the Terraform registries, they are extremely small examples that use these abstracted modules and also only use one AWS service

at a time. From our testing, these are not enough for the LLM to understand and produce new, accurate Terraform code. The jump in performance came only when we uploaded a couple of high-quality Terraform examples of our own making. Using those, the LLM was then able to produce functioning code examples that built on top of the provided examples. Thus, we believe the key to making this work better would be to find or produce several high-quality and high-complexity Terraform configurations and provide them as context. Another thing that would improve this is getting the LLM to take initiative. Based on its knowledge of deployment, it should be able to ask questions that clarify the user's requirements and get all necessary information without depending on the user to provide everything. These two things would take our deliverables to the next level, and we hope future teams are interested in working on it.

However, in its current state, our tool provides a lot of value to solo developers and students who are just looking to get something online. It does save a lot of time learning about the right way to host things, nitpicking over VPC and network rules, and figuring out bugs with IAM roles. It takes the time to do basic server, API, and web hosting down from couple days to a few minutes. I know that we will be making use of it in our future projects.

Acknowledgements

We would like to acknowledge Dr. Aggarwal for serving as our advisor. We would also like to thank our startups and student organizations, like Dream Team Engineering, that exposed us to the importance of the cloud and all of the difficulties that come with managing it.

Appendices

Appendix A - Assistants API Prompt

You will write and return Terraform .tf files and nothing else. You will either be given some requirements for an AWS architecture, and you must provide the appropriate .tf file. Or you will be given an existing .tf file and error output, and you must fix it. Make sure all of your output is one continuous code file and not split into multiple files.

All of your .tf files should start with something like this:

```
variable "AWS_ACCESS_KEY_ID" {
    description = "AWS Access Key ID"
    type        = string
}
variable "AWS_SECRET_ACCESS_KEY" {
    description = "AWS Secret Access Key"
    type        = string
    sensitive   = true
}
terraform {
    required_version = ">= 1.0"
    backend "remote" {
        organization = "SysSavvyAI"
        workspaces {
            name = "test"
        }
    }
}
required_providers {
    aws = {
        source = "hashicorp/aws"
        version = ">= 5.79.0"
```

```
    }  
  }  
}  
provider "aws" {  
  region    = "us-east-1"  
  access_key = var.AWS_ACCESS_KEY_ID  
  secret_key = var.AWS_SECRET_ACCESS_KEY  
}
```

If you are using other providers, make sure to add it to the required_providers section. Use the code examples and modules from the vector store to inform your Terraform code. Try to use as much as is applicable from the relevant documents. Be careful in specifying the depends_on fields, and make sure you are using up-to-date Terraform syntax.

Biographies



Hi! My name is Nishant Nagururu. I have previously interned at NextEra Energy and Bank of America. This upcoming summer, I am interning at Palantir, and I will be starting an MS in CS at Columbia in the fall. I am excited for both of these upcoming opportunities and can't wait to see where my career takes me!



Hi, my name is Aditya Jithesh. I have previously interned at Texas Instruments and Barclays.

This upcoming summer, I will be interning at Susquehanna International Group (SIG). In the fall,

I will start my MS in CS at UPenn.