Capstone Project on Generative AI for Cloud Architecture Diagram Generation from Natural Language

---

Project Title:

AI-Powered Cloud Architecture Diagram Generator Using NLP

---

1. Objective

Develop a **Generative AI system** capable of transforming **natural language inputs** into **cloud architecture diagrams**, automatically mapping user requirements to relevant cloud services and visualizing the complete infrastructure layout. The system should support **multiple cloud providers** including **AWS, Azure, and GCP**.

---

2. Problem Statement

Architecting cloud solutions typically requires deep domain knowledge of cloud platforms, services, and design best practices. Non-technical users or business teams often struggle to visualize infrastructure requirements when they describe them verbally or textually.

This project aims to build a system that: - Accepts **natural language descriptions** of cloud requirements (e.g., "Build a scalable web app with a load balancer, database, and autoscaling on AWS"). - Automatically maps requirements to **provider-specific services** (e.g., EC2, RDS, ELB in AWS). - Dynamically generates an **architecture diagram** using visualization tools like **Graphviz**, **Mermaid**, or **Draw.io API**.

---

3. Project Scenario

Scenario A – AWS Cloud Example

User Input: "Create a scalable e-commerce application on AWS with a load balancer, EC2 instances, and an auto-scaling group connected to an RDS MySQL database."

Expected Output: - Components: ELB, EC2, Auto Scaling Group, RDS. - Connections: Load Balancer → EC2 Cluster → RDS. - Diagram auto-generated showing architecture flow.

Scenario B – Azure Cloud Example

User Input: "Design a data analytics solution using Azure Data Lake, Synapse Analytics, and Power BI."

Expected Output: - Components: Data Lake Storage, Synapse Workspace, Power BI. - Connections: Data Flow from ingestion to visualization. - Diagram created and exportable as a PNG or SVG file.

Scenario C – GCP Cloud Example

User Input: "Build a GCP solution for real-time analytics using Pub/Sub, Dataflow, and BigQuery."

Expected Output: - Components: Pub/Sub, Dataflow, BigQuery. - Data pipeline visually represented.

---

4. Technical Architecture

Components:

1. **Frontend Interface:** Streamlit or Flask app for user input (text query) and visualization.
2. **NLP Engine:** Generative AI model (Gemini / GPT-4) to parse user intent and extract infrastructure components.
3. **Mapping Engine:** Translates extracted requirements into cloud-specific service mappings.
4. Example: "Load Balancer" → AWS ELB / Azure Front Door / GCP Load Balancer.
5. **Diagram Generator:** Uses Graphviz, Mermaid, or Draw.io API to generate dynamic visual diagrams.
6. **Storage:** Saves diagrams and mappings for reuse.
7. **Deployment:** Google Cloud Run / AWS Lambda for hosting.

Architecture Flow:

1. User enters textual query.
2. NLP model processes text and identifies components and relationships.
3. Mapped cloud services are generated based on provider selection.
4. Diagram engine renders architecture diagram.
5. Output visualized on UI and downloadable as image or PDF.

---

5. Key Features

- **Multi-Cloud Support:** AWS, Azure, and GCP component mapping.
- **Dynamic Visualization:** Automatically renders cloud architecture diagrams.
- **Customizable Diagrams:** Users can modify, save, or export generated diagrams.
- **Context-Aware NLP:** Understands synonyms and functional equivalents (e.g., "database" → RDS, SQL Database, BigQuery).
- **Interactive Output:** Supports real-time diagram editing.

---

6. Evaluation Metrics

Metric	Description	Target
Component Accuracy	Correct mapping of text to cloud services	> 90%
Relationship Accuracy	Correctness of interconnections	> 85%
Visualization Quality	Clarity and correctness of generated diagrams	> 90%

Metric	Description	Target
Response Time	Query-to-diagram generation time	< 10 sec
User Satisfaction	Feedback from test users	>= 8/10

---

7. Deliverables

- Functional web app for text-to-diagram conversion.
- Multi-cloud component mapping database.
- Sample input-output pairs for AWS, Azure, and GCP.
- Generated diagrams (PNG, SVG formats).
- Technical documentation (architecture and setup guide).

---

8. Hands-On Activities

1. **Dataset Preparation:**
2. Collect text-based architecture descriptions from blogs, whitepapers, or cloud documentation.
3. Annotate key cloud components and their relationships.
4. **Model Development:**
5. Fine-tune or prompt-tune Gemini/GPT models for cloud terminology extraction.
6. Test prompt examples like:
    - "Generate an AWS architecture diagram for a 3-tier application with RDS and S3."
    - "Create a GCP architecture for real-time analytics."
7. **Mapping Engine Implementation:**
8. Build JSON mapping files linking abstract components to provider-specific services.
9. **Visualization:**
10. Use Graphviz to convert mappings into diagrams.
11. Example: Generate .dot file dynamically and render output.
12. **Output Testing:**
13. Validate component accuracy and connectivity.

---

9. Supportive Guide for Participants (with Hints)

Step 1: Environment Setup

- Install dependencies: `pip install google-generativeai graphviz streamlit`
- Set API key: `genai.configure(api_key="YOUR_GEMINI_KEY")`
- **Hint:** Ensure Graphviz is installed on your system (`sudo apt install graphviz`).

Step 2: NLP Query Parsing

- Use Gemini/GPT to identify nouns and actions related to infrastructure (e.g., "web server," "database," "load balancing").

- **Hint:** Apply Named Entity Recognition (NER) to detect cloud-related keywords.

Step 3: Component Mapping

- Create dictionaries for cloud provider mappings:

```
aws_map = {"database": "RDS", "compute": "EC2", "storage": "S3"}
azure_map = {"database": "SQL Database", "compute": "VM", "storage": "Blob Storage"}
gcp_map = {"database": "BigQuery", "compute": "Compute Engine", "storage": "Cloud Storage"}
```

- **Hint:** Use synonym expansion to handle varied input terms.

Step 4: Diagram Generation

- Use Graphviz to create visual layouts:

```
from graphviz import Digraph
dot = Digraph()
dot.node('A', 'Load Balancer')
dot.node('B', 'Web Server (EC2)')
dot.node('C', 'Database (RDS)')
dot.edges(['AB', 'BC'])
dot.render('architecture_diagram', format='png')
```

- **Hint:** Define consistent color codes for each provider (AWS=Orange, Azure=Blue, GCP=Green).

Step 5: User Interface

- Build an input form in Streamlit for query and cloud provider selection.
- Display generated diagram instantly.
- **Hint:** Allow users to download diagram outputs.

Step 6: Evaluation

- Compare generated diagrams against standard reference architectures.
- Evaluate accuracy of service mappings.
- **Hint:** Create a validation dataset of 20–30 sample queries for testing.

Step 7: Advanced Extensions

- Integrate with **Draw.io API** for editable diagrams.
- Add feedback loop to improve NLP mapping accuracy.
- Implement a knowledge graph for relationship inference.

---

10. Expected Outcome

A functional prototype that automatically converts **natural language cloud architecture requirements** into **accurate, provider-specific diagrams**. The output will include: - Visual representation of infrastructure. - Accurate component mapping across multiple clouds. - Exportable, editable diagrams suitable for design documentation and proposal workflows.