CSCI 5411: Advanced Cloud Architecting
Mid-Term Project Report (Milestone 2)

Meet Maratha (B00994591), Master of Applied Computer Science, Dalhousie University

Contents

List of Figures

1. Project Overview

1.1. Introduction

New Zig developers tend to struggle in finding precise answers to syntax and best-practices to follow while writing their code in Zig's highly complex documentation. I wanted to tackle this problem for Zig and any other programming language in future by designing a Chatbot that is improved from its base settings with the use of Retrieval Augmented Generation (RAG) pipeline. To design this Chatbot I used various AWS services such as S3, Lambda, EC2, VPC Endpoints, etc. which are discussed in Section ??.

2. Architectural Diagram

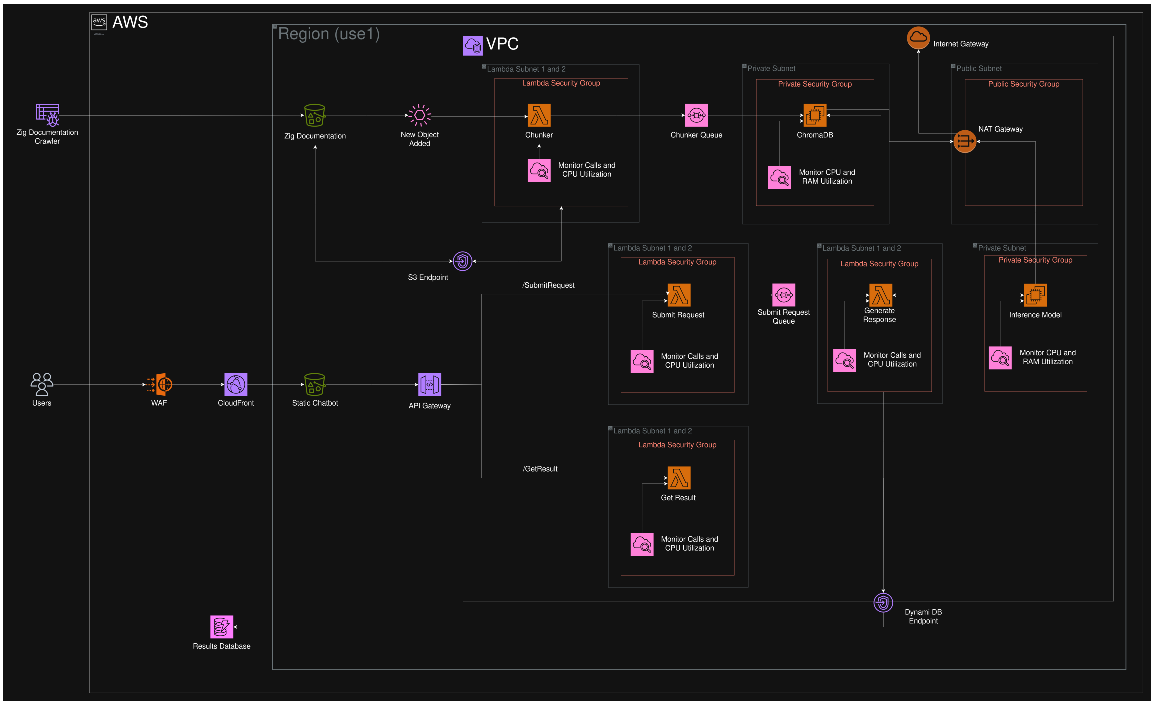

Fig. 1. Architecture Diagram

I updated the original diagram which can be seen in Figure 1, adding more details and increasing the overall security of the architecture. To increase the overall security of the architecture I moved Lambda functions into the Virtual Private Cloud (VPC), and attached a security group to them limiting their access. I also placed them in two subnets following the suggestion from AWS to increase the overall availability of the Lambda functions. The updated design of the Lambda functions can be viewed in Figure 2. This structure is followed by all the Lambda functions in the architecture.

As I had placed the Lambda functions in a VPC I need to make VPC endpoints for the Lambda functions or EC2 instances to connect to other AWS services securely without routing through public internet. An example of the VPC endpoint can be seen in Figure 3.

I also settled on EC2 instance hosted Vector Database and Inference model as the options available on the SageMaker were not very enticing in my opinion for my software. To optimize the

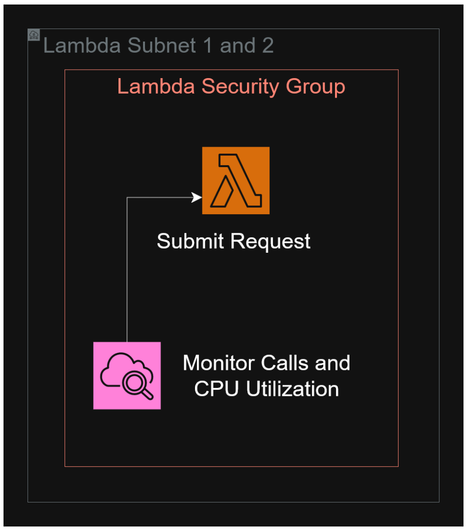

Fig. 2. Updated Lambda Function Configuration

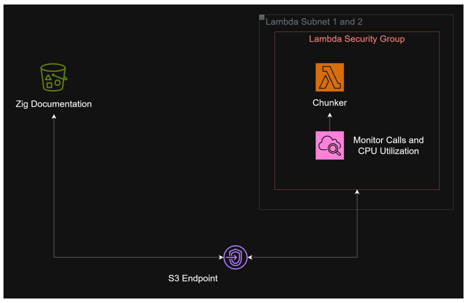

Fig. 3. VPC Endpoints Example

deployed models, I used the quantized models provided by Ollama [1], namely Llama 3.1 8 Billion.

2.1. Data Sequence Diagram

There has been no change the overall data sequence diagram other than some of the things being processed internally in a Lambda functions instead of being passed to another Lambda function .The system employs two distinct data flow pipelines, each visualized in separate sequence diagrams: data ingestion (Figure 4) and data generation (Figure 5).

2.1.1. Data Ingestion Pipeline

This pipeline processes Zig's raw documentation into retrievable embeddings. First, the ”Generate Document Chunks” module splits the raw text into manageable segments. These chunks are forwarded to the ”Generate Chunk Embedding” module, which converts each segment into a numerical embedding vector using a pre-trained model. Finally, the ”Store in Vector Database” module persists these embeddings in the vector database, completing the ingestion cycle.

2.1.2. Data Generation Pipeline

When a user submits a query, the ”Generate Embedding” module converts the input text into an embedding vector. This vector is passed to the ”Get Relevant Vectors” module, which queries

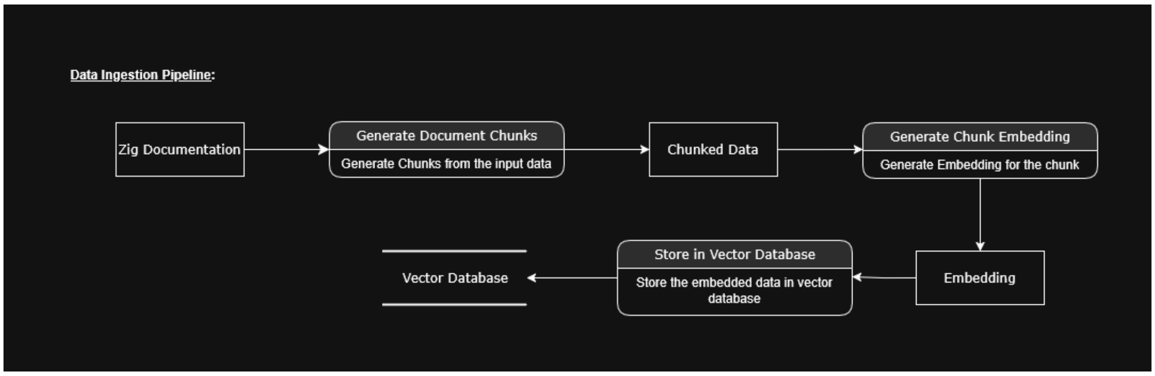

Fig. 4. Data Sequence Diagram for Data Ingestion

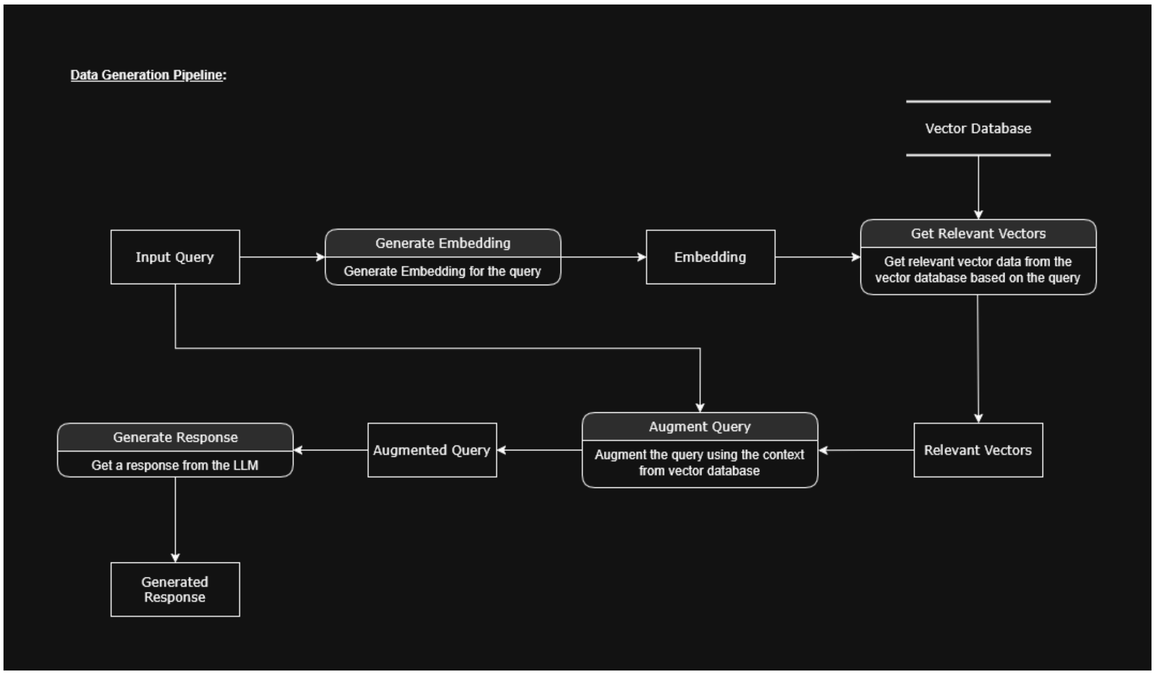
Data Generation Pipeline:
Vector Database
Input Query
Generate Embedding
Generate Embedding for the query
Embedding
Get Relevant Vectors
Get relevant vector data from the vector database based on the query
Generate Response
Get a response from the LLM
Augmented Query
Augment Query
Augment the query using the context from vector database
Relevant Vectors
Generated Response

Fig. 5. Data Sequence Diagram for Data Generation

the vector database for semantically similar documentation snippets. The retrieved vectors and original query are then fed into the ”Augment Query” module, which synthesizes a context-rich prompt. This augmented prompt is processed by the ”Generate Response” module, which leverages the LLM to produce a final answer grounded in Zig’s official documentation.

3. Implementation Details

3.1. Service Configurations

There are two S3 buckets present in the final architecture. The first S3 bucket is called ”Zig Documentation” and is used to store the crawled ZIG documentation from their website. This S3 bucket is not accessible to public and has Server Side Encryption (SSE) - S3 enable on it to encrypt all the data stored on it. This Bucket is filled using the crawler function running on the local machine.

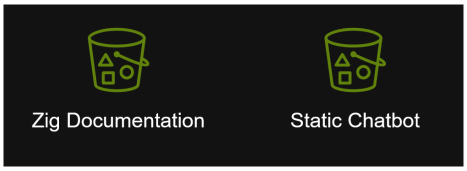

Fig. 6. S3 Buckets

The second S3 bucket is called ”Static Chat bot” and is used to store the static assets required by the chat bot website. This bucket is built based on the teachings learned in the Challenge Lab 1, where we deployed a static website. This website has CORS policies enabled to be properly accessed through a web browser without any errors.

For the Lambda functions I attached them using SQS wherever possible to decouple the architecture and make it more reliable. All the Lambda functions are optimized to complete their execution in as less time as possible, other than the ”Generate Response” function which is used to augment the query, generate the response, and store it into the DynamoDB database. This whole process takes around 4 minutes, so this is the only function whose execution time exceeds other functions.

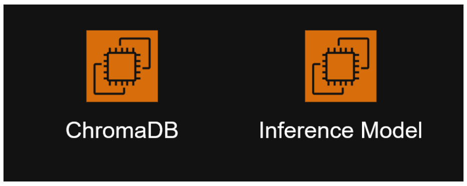

Fig. 7. EC2 Instances

For optimizing the overall runtime and requirements of the program I chose T3 small and R7a large for the Chroma Database and Inference server based on the recommendations found online and in the documentation. All these EC2 instances are in their private subnet and connected to other AWS services using VPC endpoints and IAM Role. The Chroma Database instance uses the ”EMR_EC2_LabRole” which gives the EC2 instance permission to access SQS queue information, which is necessary for the programs running to work properly.

All the services present in the private subnet does not have any connection to the internet, other than EC2 instances, which have connection through NAT Gateway to get required packages. This helps us to promote cloud least access privilege.

3.2. Model Integration

There are 2 models present in the overall architecture, one for encoding the data and other for generating the response from the Large Language Model. For the encoding we are using all-MiniLM-L6-v2 which is provided directly by Chroma database. To generate inferences required for the output we are using Llama 3.1 8 Billion parameters quantized model provided by Ollama. This quantized model makes it easier for our machine to compute results.

For the chunking the data to be stored in the vector database we are using the chunking function provided by Adam Lucek, Brandom Starxel, and Crawl4AI [2][3][4]. These sources taught me to how to efficiently crawl and properly chunk the data to be stored in the vector database. We have stored the data as separated by the markdown specific delimiters and a maximum length of 500.

3.3. Monitoring and Alarms

I have to monitor the Lambda functions and EC2 instances for their CPU utilization, with monitoring memory usage in the case of the Inference model EC2 instance. I have setup CloudWatch alarms for all these specific services to notify me about any excess usage of resources. I have also attached a cloud watch alarm if the number invocations and error rate increase for the Lambda functions.

3.4. Security Measures

For securing the overall architecture from attacks such as DDoS and SQL Injection I applied AWS WAF to the CloudFront so that we are protected from most of the malicious attacks. To further increase the security I made it so that we can only access all the lambda functions and other processes through the API gateway where we can further trim suspicious calls from some IP addresses in the future. For the data storage services such as S3 and DynamoDB we have encryption enabled at rest and during transit.

4. Cost Analysis

I computed the overall AWS cost of deploying the overall architecture and it came out to be around $50 per month, at the high end. Some of the computation in it such as S3 bucket for documentation data storage can be made more cheaper by using proper life cycles for the objects as we do not need them much once they are vectorized and stored in the database. Also the overall cost of EC2 instance can also be saved by getting the savings plan discounts and booking the machines for 1 year. The computation of cost for EC2 instances was made by assuming that they would be utilized for 8 hours per day, i.e. during normal working time.

5. Lessons Learned and Improvements

5.1. Challenges

I faced multiple challenges during the creation of this product such as an EC2 instance present in a private subnet not being able to access SQS queue, not being able to use managed services such as OpenSearch Database and SageMaker AI to save cost, Learning how to configure and use VPC endpoints, and the increase of overall security being highly restrictive to development. I found solutions for each of them through learning about them through YouTube tutorials and services mentioned in in-class lectures.

5.2. Optimizations

I optimized the EC2 instances used for storing and getting relevant context from the vector database and the inference model, so that they can perform efficiently without getting hanged. I also optimized the cold start of lambda functions by using containerized versions of the code so that they are easy to package and easy to load from the containers instead of being initialized each time manually.

5.3. Future Road map

This software can further be improved by implementing Infrastructure as Code so we can deploy the services in a proper fashion. It will also be really helpful to make AMI images from the EC2 instances for the vector database and inference model so that they can be launched as quickly as possible without any manual setup requirement. It would be highly beneficial if we can use the managed services such as OpenSearch database and SageMaker AI endpoints instead of hosting these models to be ran the whole time, even when they are not required.

6. Conclusion

I deployed an end-to-end GenAI project supporting developers in their search of better understanding of Zig programming language. The overall architecture is highly efficient and cost effective, costing around $50 per month. The architecture also allows easy swapping of the inference model by just loading some other Ollama quantized model instead of Llama 3.1 in future.

References

[1] Ollama, "Ollama," 2025, accessed: 2025-05-25. [Online]. Available: https://ollama.com/

[2] Adam Lucek, "The best way to chunk text for rag," 2024, accessed: 2025-05-25. [Online]. Available: https://youtu.be/Pk2BeaGbcTE?feature=shared

[3] Brandon Starxel, "Chunking evaluation," 2025, accessed: 2025-05-25. [Online]. Available: https://github.com/brandonstarxel/chunking_evaluation

[4] Uncle Code, "Crawl4ai," 2024, accessed: 2025-05-25. [Online]. Available: https://github.com/unclecode/crawl4ai