Use Case: RAG with PDF - AWS Cloud Reference Architecture

Let’s create a reference architecture for a document query system (RAG-based GenAI system) on the AWS Cloud platform. The system, which currently processes and analyses a PDF about the impact of the Indian Premier League on Test cricket in India, will be reimagined using AWS services to improve scalability, performance, and cost-effectiveness.

Notebook of RAG with PDF standalone Use Case

Key AWS Services and Migration Strategy:

1. Document Storage:

- Migrate from local PDF storage to Amazon S3 for secure, scalable document storage.

2. Text Processing and Chunking:

- Utilise **AWS Lambda** functions to handle PDF parsing and text chunking, triggered by S3 events when new documents are uploaded.

3. Text Embedding:

- Deploy the embedding model on **Amazon SageMaker**, using endpoints for real-time inference to generate text embeddings.

4. Vector Database:

- Replace the local FAISS implementation with **Amazon OpenSearch Service** (with k-NN plugin) for efficient similarity search at scale.

5. Large Language Model (LLM):

- API based LLM using **Amazon Bedrock**, a fully managed service providing a single API to access foundation models from top AI companies, enabling secure and responsible generative AI applications.

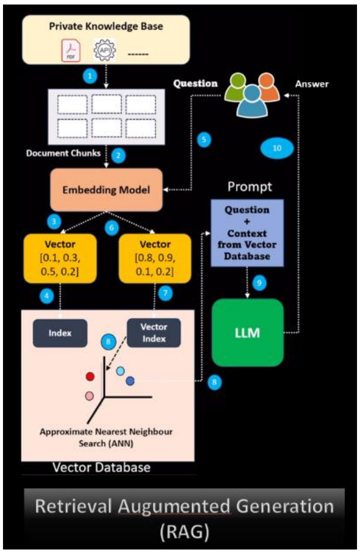
6. Question & Answer Pipeline:

- Implement the retrieval-augmented generation process using Python, LangChain / LlamaIndex using AWS Lambda to orchestrate the workflow between OpenSearch, SageMaker endpoints, API based LLMs using Amazon Bedrock and other AWS services.

7. API Layer:

- Create an API using **Amazon API Gateway** and AWS Lambda to handle user requests and responses.

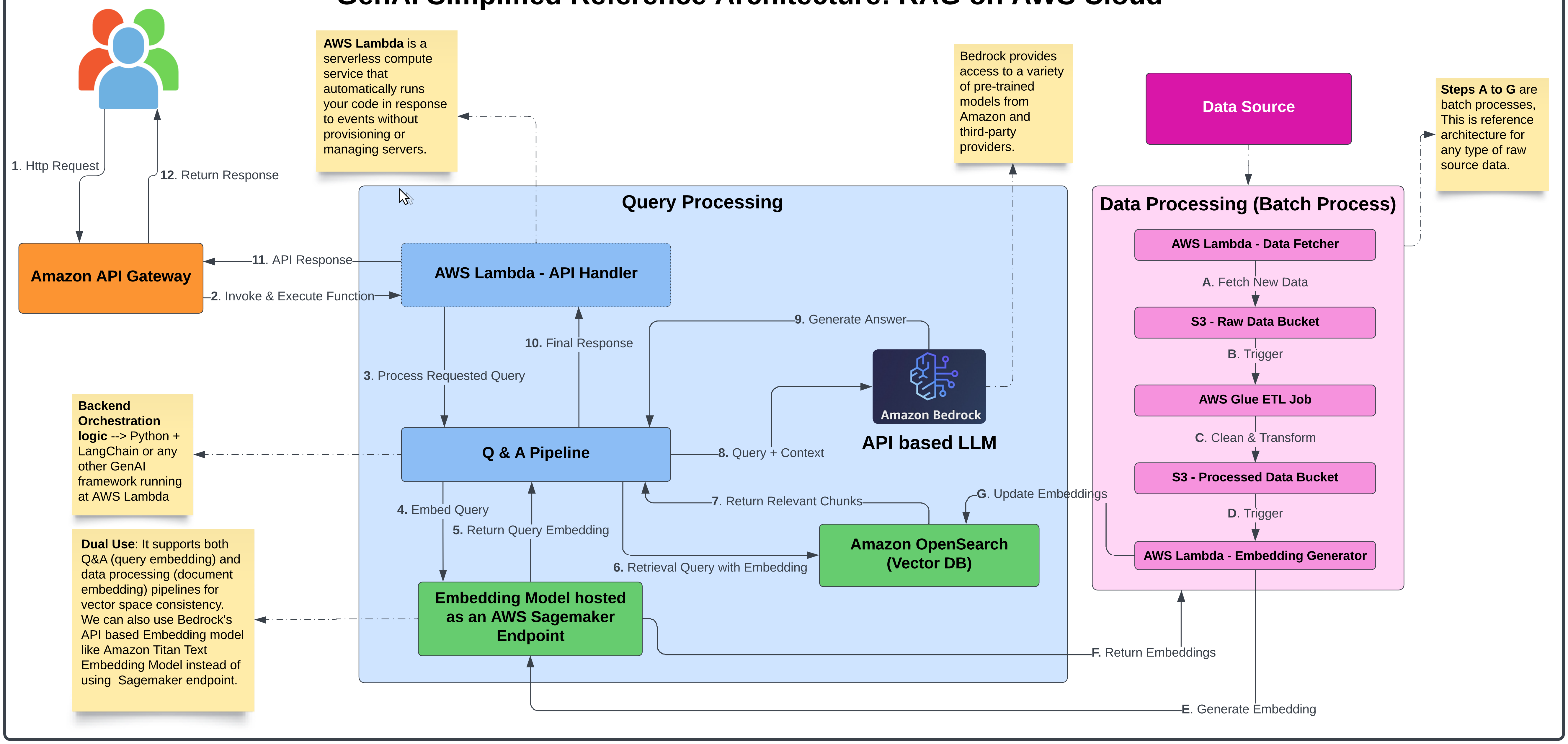
This AWS migration will transform the solution into a cloud-native, serverless architecture, offering better scalability, enhanced security, and cost savings through pay-per-use pricing. It reduces operational overhead, letting the team focus on improving the document query system.


Private Knowledge Base
PDF
API

1
Question
Answer
Document Chunks
2
5
10
Embedding Model
Prompt
3
6
Question + Context from Vector Database
Vector [0.1, 0.3, 0.5, 0.2]
Vector [0.8, 0.9, 0.1, 0.2]
4
7
9
Index
Vector Index
8
LLM
8
Approximate Nearest Neighbour Search (ANN)
Vector Database
Retrieval Augumented Generation (RAG)

Prepared By → @genieincodebottle

GenAI Simplified Reference Architecture: RAG on AWS Cloud



AWS Lambda is a serverless compute service that automatically runs your code in response to events without provisioning or managing servers.

Bedrock provides access to a variety of pre-trained models from Amazon and third-party providers.

Steps A to G are batch processes, This is reference architecture for any type of raw source data.

Backend Orchestration logic --> Python + LangChain or any other GenAI framework running at AWS Lambda

Dual Use: It supports both Q&A (query embedding) and data processing (document embedding) pipelines for vector space consistency. We can also use Bedrock's API based Embedding model like Amazon Titan Text Embedding Model instead of using Sagemaker endpoint.

Amazon API Gateway

1. Http Request

12. Return Response

11. API Response

2. Invoke & Execute Function

Query Processing

AWS Lambda - API Handler

3. Process Requested Query

10. Final Response

Q & A Pipeline

9. Generate Answer

8. Query + Context

Amazon Bedrock

API based LLM

4. Embed Query

5. Return Query Embedding

7. Return Relevant Chunks

6. Retrieval Query with Embedding

Amazon OpenSearch (Vector DB)

Embedding Model hosted as an AWS Sagemaker Endpoint

Data Source

Data Processing (Batch Process)

AWS Lambda - Data Fetcher

A. Fetch New Data

S3 - Raw Data Bucket

B. Trigger

AWS Glue ETL Job

C. Clean & Transform

S3 - Processed Data Bucket

D. Trigger

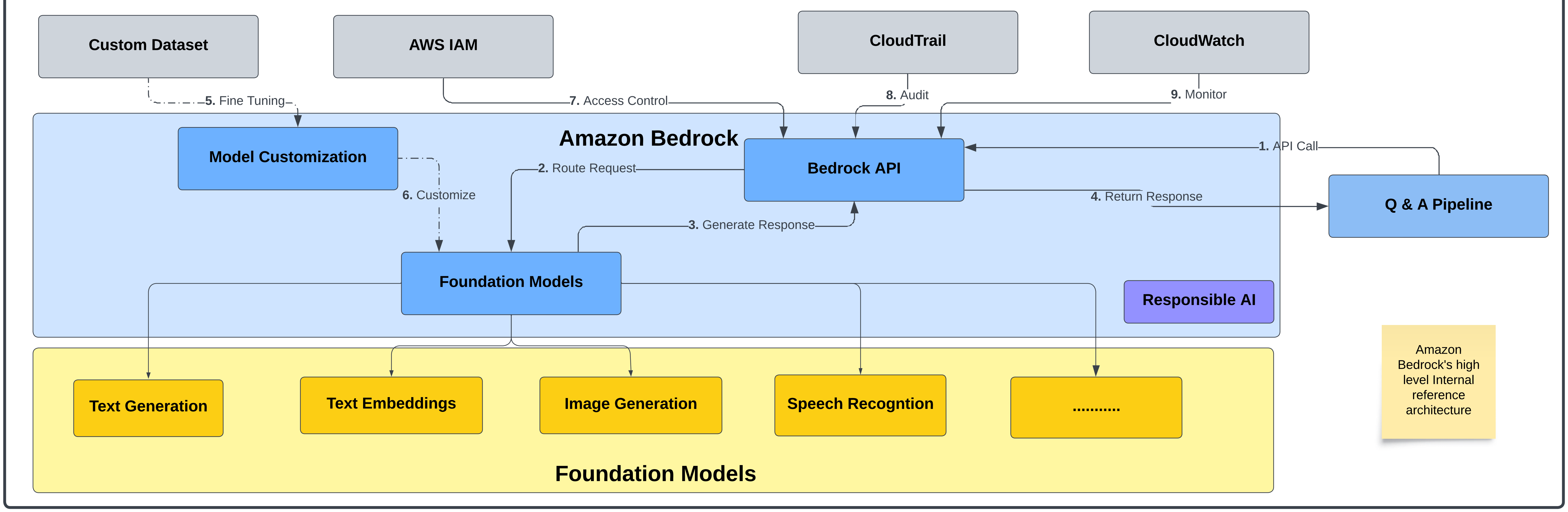
AWS Lambda - Embedding Generator

G. Update Embeddings

F. Return Embeddings

E. Generate Embedding

Amazon Bedrock: Reference Architecture



Custom Dataset

AWS IAM

CloudTrail

CloudWatch

5. Fine Tuning

7. Access Control

8. Audit

9. Monitor

Amazon Bedrock

Model Customization

Bedrock API

1. API Call

2. Route Request

6. Customize

4. Return Response

Q & A Pipeline

3. Generate Response

Foundation Models

Responsible AI

Text Generation

Text Embeddings

Image Generation

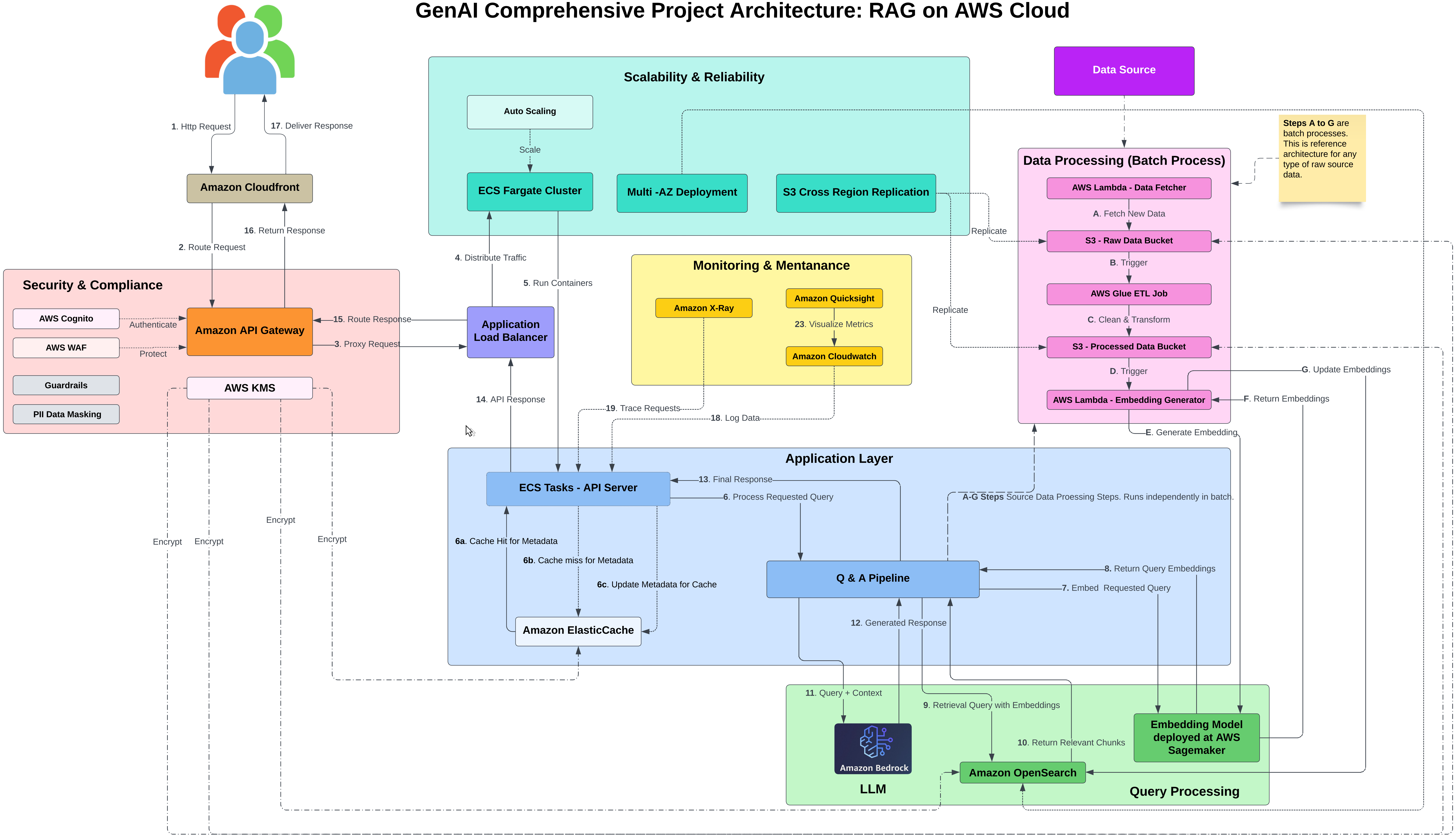
Speech Recogntion

....

Foundation Models

Amazon Bedrock's high level Internal reference architecture

GenAI Comprehensive Project Architecture: RAG on AWS Cloud



With ECS Fargate Cluster

Comprehensive AWS Cloud-Based Architecture Flow Details

1. User Interaction

- The process begins with a user sending an HTTPS request to the system through User Interface.

2. Amazon CloudFront

Amazon CloudFront is a content delivery network (CDN) that:

- Caches content at global edge locations
- Serves as the entry point for web requests
- Reduces latency by delivering content from the nearest server
- Provides DDoS protection by absorbing and dispersing attack traffic
- Integrates with other AWS services for enhanced performance and security

CloudFront essentially improves website speed, security, and reliability by distributing content delivery across a worldwide network.

3. Amazon API Gateway

Amazon API Gateway is a fully managed service that:

- Manages APIs for web applications
- Routes incoming API requests to appropriate backend services
- Implements throttling to prevent API abuse
- Handles authentication and authorization for API access

API Gateway simplifies API creation, security, and management, allowing developers to focus on building their core application logic rather than API infrastructure.

4. Application Load Balancer

Application Load Balancer (ALB) is an AWS service that:

- Distributes incoming application traffic across multiple targets
- Specifically balances load for ECS Fargate tasks in this architecture
- Operates at the application layer (Layer 7) of the OSI model
- Can route requests based on content type, headers, or URL paths
- Supports advanced routing features like host-based and path-based routing

ALB intelligently distributes incoming web traffic to ensure optimal performance and availability of the application running on ECS Fargate.

5. ECS Fargate Cluster

ECS Fargate Cluster is an AWS service that:

- Provides serverless compute for running containers
- Deploys and manages Docker containers without requiring server management
- Automatically scales compute resources based on demand
- Eliminates the need to provision or manage EC2 instances
- Integrates with other AWS services for networking, storage, and security

ECS Fargate allows developers to focus on building and running containerized applications without worrying about the underlying infrastructure management.

6. ECS Tasks - API Servers

ECS Tasks - API Servers are:

- Containerized applications (Like backend Python + LangChain etc) running as tasks within ECS Fargate
- Designed to handle incoming API requests
- Responsible for managing the **Question-Answering** process
- Scalable units that can be increased or decreased based on demand
- Isolated environments for running the API server code

These ECS tasks contain the core API logic for processing requests and coordinating the question-answering functionality of the GenAI application.

7. Question-Answering Pipeline

- The core logic for processing user queries and generating responses.

8. Amazon ElastiCache

Amazon ElastiCache is a service that:

- Provides in-memory caching for fast data retrieval
- Stores metadata to reduce database queries
- Improves application performance by caching frequent requests
- Reduces load on the primary database

- Supports popular caching engines like Redis or Memcached

ElastiCache enhances system speed and efficiency by storing frequently accessed data in memory, allowing quicker access than repeatedly querying the database.

9. Amazon OpenSearch

Amazon OpenSearch is a service that:

- Provides a managed search and analytics engine
- Enables fast and efficient retrieval of relevant document chunks
- Indexes and searches through large volumes of text data
- Supports advanced query capabilities for precise information retrieval
- Scales easily to handle growing data and query loads

OpenSearch allows the system to quickly find and return the most relevant information from a large corpus of documents based on user queries.

10. AWS Bedrock

AWS Bedrock is a service that:

- Provides managed access to large language models (LLMs)
- Generates human-like text responses based on input
- Processes user queries and context to produce relevant answers
- Offers various pre-trained models for different tasks
- Allows integration of AI capabilities without managing complex infrastructure

Bedrock powers the system's ability to understand queries and generate coherent, contextually appropriate responses using advanced language models.

11. Response Flow

- The generated answer flows back through the system components to the user.

Data Processing Pipeline

12. AWS Lambda - Data Fetcher

AWS Lambda - Data Fetcher is:

- A serverless function running on AWS Lambda
- Designed to fetch new data from external sources

- Triggered automatically or on-demand to update the system's data
- Scalable and cost-effective, as it only runs when needed
- Integrated with other AWS services for data processing and storage

This Lambda function acts as an automated data collector, keeping the system's information up-to-date by retrieving fresh data from external sources.

13. S3 Raw Data Bucket

- Storage for raw, unprocessed data.

14. AWS Glue ETL Job

- Managed ETL (Extract, Transform, Load) service.
- Cleans and transforms raw data into a processed format.

15. S3 Processed Data Bucket

- Storage for cleaned and processed data.

16. AWS Lambda - Embedding Generator

- Generates vector embeddings for the processed data.
- Updates the OpenSearch index with new embeddings.

Monitoring & Maintenance

17. Amazon CloudWatch

- Monitoring and observability service.
- Collects and tracks metrics, logs, and events.

18. AWS X-Ray

- Distributed tracing system.
- Helps analyse and debug production, distributed applications.

19. Amazon QuickSight

- Business intelligence and data visualization service.
- Creates dashboards and visualizations from CloudWatch metrics.

Security & Compliance

20. Amazon Cognito

- Provides user authentication, authorization, and user management.

21. AWS WAF

- Web Application Firewall that protects against common web exploits.

22. AWS KMS

- Key Management Service for creating and managing encryption keys.
- Ensures data encryption at rest for various components (OpenSearch, ElastiCache, S3 buckets).

Scalability & Reliability

23. Auto Scaling

- Automatically adjusts the number of ECS tasks based on demand.

24. Multi-AZ Deployment

- Replicates data across multiple Availability Zones for high availability.

25. S3 Cross-Region Replication

- Replicates S3 bucket contents across different AWS regions for disaster recovery and reduced latency.

Interview Questions on AWS Architecture for GenAI RAG Project

Q: How does this architecture ensure low latency and high availability for global users?

A: The architecture uses **Amazon CloudFront** as a **CDN** to reduce latency for global users. It also employs **Multi-AZ deployments** for critical services like **OpenSearch** and **S3 Cross-Region Replication** for data redundancy. The **Application Load Balancer** distributes traffic across multiple ECS Fargate instances, ensuring high availability.

Q: Explain the caching strategy used in this architecture. Why are two different caching mechanisms employed?

A: The architecture uses caching layers: Amazon ElastiCache is used for fast retrieval of frequently accessed metadata, reducing database load.

Q: How does the system handle data processing and keep the question-answering capabilities up-to-date?

A: The system uses a data processing pipeline that starts with an AWS Lambda function fetching new data. This data is stored in an S3 Raw Data Bucket, which triggers an AWS Glue ETL job for cleaning and transformation. The processed data is then stored in another S3 bucket, which triggers another Lambda function to generate embeddings. These embeddings are then used to update the Amazon OpenSearch index, ensuring that the question-answering system has access to the latest information.

Q: Describe the security measures implemented in this architecture.

A: The architecture implements several security measures:

- HTTPS for all client-server communications
- Amazon Cognito for user authentication and authorization
- AWS WAF to protect against web exploits
- AWS KMS for encryption key management, ensuring data encryption at rest for various components
- API Gateway for API management and throttling

- CloudFront for DDoS protection

Q: How does this architecture handle scaling under increased load?

A: The architecture uses Auto Scaling for the ECS Fargate cluster, which automatically adjusts the number of tasks based on demand. The Application Load Balancer distributes traffic across these tasks. Additionally, serverless components like Lambda and Fargate inherently scale based on demand. ElastiCache and OpenSearch can also be scaled horizontally to handle increased load.

Q: Explain the role of AWS Bedrock in this architecture. What are its advantages?

A: AWS Bedrock is used as the managed service for large language models. It generates answers based on the user's query and the context retrieved from OpenSearch. The advantages include:

- Simplified integration of state-of-the-art language models
- Managed infrastructure, reducing operational overhead
- Scalability to handle varying loads
- Regular updates to models without requiring system changes

Q: How does the system ensure efficient retrieval of relevant information for user queries?
A: The system uses Amazon OpenSearch for efficient retrieval. When a query comes in, it's sent to OpenSearch, which uses vector embeddings to find the most relevant document chunks. This semantic search capability allows for more accurate and context-aware information retrieval compared to traditional keyword-based search.

Q: Describe the observability and monitoring setup in this architecture.

A: The architecture uses Amazon CloudWatch for collecting metrics, logs, and events from various components. AWS X-Ray is employed for distributed tracing, helping to analyse and debug the application. Amazon QuickSight is used to create dashboards and visualizations from the collected metrics, providing insights into system performance and behaviour.

Q: How does this architecture handle potential failures or outages in a single AWS region?

A: The architecture implements several measures for resilience:

- Multi-AZ deployments for critical services ensure availability even if one Availability Zone fails
- S3 Cross-Region Replication provides a backup of data in a different region
- CloudFront's global edge network can route traffic to healthy regions in case of regional issues
- The use of managed services like ECS Fargate, Lambda, and Bedrock inherently provides some level of fault tolerance

Q: Explain the purpose of the Application Load Balancer in this architecture. How does it differ from a Network Load Balancer?

A: The Application Load Balancer (ALB) distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. It operates at the application layer (Layer 7) of the OSI model, allowing it to route based on content of the request (e.g. URL path). This differs from a Network Load Balancer, which operates at the transport layer (Layer 4) and is best suited for routing TCP traffic where extreme performance and static IP addresses are needed.

Q: In this architecture, how would you implement a feature to allow users to upload and process their own documents for question-answering?

A: To implement this feature, we could:

- Add an S3 bucket for user uploads with appropriate access controls.
- Create a Lambda function triggered by S3 upload events.
- This Lambda would process the document, generate embeddings, and add them to OpenSearch.
- Implement user-specific data partitioning in OpenSearch to ensure data isolation.
- Modify the question-answering pipeline to include user-specific context in queries.

Q: How would you modify this architecture to support multi-tenancy while ensuring data isolation and performance for each tenant?

A: To support multi-tenancy:

- Use Amazon Cognito for tenant authentication and authorization.
- Implement a tenant identifier in API requests.
- Use separate OpenSearch indices or index aliases for each tenant.
- Employ RLS (Row-Level Security) in OpenSearch for additional isolation.
- Use tenant-specific ElastiCache clusters or implement cache key prefixing.
- Consider using separate ECS task definitions for high-priority tenants.
- Implement tenant-specific rate limiting in API Gateway.

Q: The current setup uses ECS Fargate for the API servers. In what scenarios might you consider using EC2 instances instead, and how would this change the architecture?

A: Consider EC2 instances if:

- You need more control over the underlying infrastructure.
- There are specific hardware requirements (e.g., GPUs for on-premise inference).
- You have predictable, steady workloads where reserved instances could be more cost-effective.

Changes to architecture:

- Replace Fargate with EC2 Auto Scaling groups.
- Implement an EC2 instance management strategy (patching, updates).
- Consider using EC2 Image Builder for maintaining custom AMIs.
- Potentially use EC2 Spot Instances for cost optimization.

Q: The architecture uses AWS Bedrock for language model inference. How would you implement a fallback mechanism if Bedrock experiences downtime or throttling?

A: To implement a fallback mechanism:

- Set up a secondary language model service (e.g., self-hosted open-source model on ECS or EC2).
- Implement circuit breaker pattern in the API servers.

- Use Amazon SQS to queue requests during throttling or downtime.
- Create a Lambda function to process queued requests using the fallback service.
- Implement retry logic with exponential backoff for Bedrock requests.
- Use CloudWatch alarms to detect and alert on Bedrock issues.

Q: How would you implement a robust rate limiting system in this architecture that accounts for different user tiers and prevents abuse?

A: To implement a robust rate limiting system:

- Use API Gateway's built-in throttling features for coarse-grained control.
- Implement custom rate limiting logic in the API servers for fine-grained control.
- Use ElastiCache (Redis) to store rate limiting data with low latency.
- Implement token bucket or leaky bucket algorithms for flexible rate limiting.
- Store user tiers and limits in DynamoDB for easy updates.
- Use Lambda@Edge to implement rate limiting at the CloudFront level for earliest possible throttling.
- Implement retry-after headers and appropriate HTTP status codes (429) for exceeded limits.

Q: The architecture uses S3 for data storage. How would you implement a data lifecycle management policy that optimizes for both cost and performance?

A: To implement an effective data lifecycle management policy:

- Use S3 Lifecycle policies to transition data between storage classes.
- Move infrequently accessed data to S3 Infrequent Access after 30 days.
- Archive data older than 90 days to S3 Glacier.
- Set up S3 Intelligent-Tiering for data with unknown or changing access patterns.
- Use S3 analytics to gain insights into data access patterns.
- Implement S3 Object Lock for data that requires immutability.
- Use S3 Inventory for tracking object metadata and S3 Storage Lens for detailed analytics.

Q: Given the sensitive nature of user queries, how would you enhance the architecture to ensure GDPR compliance, especially considering the "right to be forgotten"?

A: To enhance GDPR compliance:

- Implement granular data tagging and classification in S3 and OpenSearch.
- Use AWS Macie for sensitive data discovery and classification.
- Encrypt all data at rest and in transit using AWS KMS.
- Implement a user data deletion workflow:
  - Lambda function to remove user data from S3, OpenSearch, and any databases.
  - Use SQS to ensure reliable processing of deletion requests.
- Implement data retention policies using S3 Lifecycle rules.
- Use AWS CloudTrail for comprehensive auditing of all data access and changes.
- Implement fine-grained access controls using IAM and resource policies.
- Use Amazon Cognito for user consent management.
- Consider using AWS CloudHSM for enhanced key security.

Q: How can you improve the relevance of retrieved passages in a RAG system?

A: To improve passage relevance:

- Fine-tune embeddings on domain-specific data
- Implement re-ranking using cross-encoders
- Use query expansion techniques
- Incorporate semantic similarity alongside keyword matching
- Adjust chunk size and overlap for better context capture

Q: What strategies can be employed to handle queries that require multi-hop reasoning in a RAG system?

A: For multi-hop reasoning:

- Implement iterative retrieval, using initial results to formulate follow-up queries

- Use graph-based knowledge representations
- Employ query decomposition techniques
- Implement a multi-step reasoning pipeline
- Utilize meta-learning approaches to adapt to complex query patterns

Q: How can you address the challenge of retrieving relevant information for queries with little context?

A: For queries with little context:

- Implement query expansion using synonyms and related terms
- Use prompt engineering to get more context from users
- Employ zero-shot classification to infer query intent
- Implement conversational history tracking
- Use hybrid retrieval combining dense and sparse methods

Q: How can you ensure diverse and comprehensive retrieval results in a RAG system?

A: To ensure diverse and comprehensive results:

- Implement maximum marginal relevance (MMR) for diversity
- Use clustering techniques on retrieved passages
- Employ multi-index retrieval from different knowledge sources
- Implement ensemble methods combining different retrieval strategies
- Use query reformulation to capture different aspects of the query

Q: What is the purpose of ElastiCache in the context of a RAG-based system in this architecture?

A: ElastiCache's Role in RAG Systems

- **Metadata Caching**
  - Purpose: Store and quickly retrieve metadata about documents or chunks.

  - Issue Solved: Reduces the need to query the main database (e.g., OpenSearch) for frequently accessed metadata, improving response times.

- **Query Result Caching**
  - Purpose: Cache results of recent queries to OpenSearch.
  - Issue Solved: Reduces load on OpenSearch and improves response times for repeated or similar queries.

- **Embedding Cache**
  - Purpose: Store pre-computed embeddings for frequent queries or document chunks.
  - Issue Solved: Reduces computation time and load on embedding generation services.

- **Session State Management**
  - Purpose: Store temporary session data for multi-turn conversations.
  - Issue Solved: Enables stateful interactions without the need to query a persistent database for every request.

- **Rate Limiting and Request Deduplication**
  - Purpose: Implement counters and timestamps for API rate limiting.
  - Issue Solved: Helps prevent API abuse and reduces unnecessary duplicate requests to the language model.

- **Feature Flags and Configuration**
  - Purpose: Store and quickly retrieve system configuration and feature flags.
  - Issue Solved: Allows for rapid system updates without code deployments or database queries.

- **Caching Intermediate Results**
  - Purpose: Store intermediate results in multi-step RAG pipelines.
  - Issue Solved: Reduces redundant computations in complex query processing workflows.

- **Token Usage Tracking**
  - Purpose: Keep real-time counters of token usage for billing or quota management.
  - Issue Solved: Enables accurate, low-latency tracking without constant database writes.

By addressing these issues, ElastiCache significantly improves the performance, scalability, and cost-effectiveness of the RAG system.