Design Document — Serverless AWS Bedrock RAG Chatbot (S3 + Lambda + FAISS)

Owner: Shubhankar Goje
Last Updated: October 2025
Region: US West (Oregon)
Project Codename: Eeva – Portfolio AI Assistant

Executive Summary

The **Serverless AWS Bedrock RAG Chatbot** is a fully managed, retrieval-augmented conversational system designed to operate as the interactive centerpiece of Shubhankar Goje's professional portfolio website. It enables visitors to engage directly with an AI assistant named **Eeva**, capable of answering detailed questions about Shubhankar's academic papers, AWS projects, and technical design work.

The solution demonstrates end-to-end integration of **Amazon Bedrock**, **AWS Lambda**, **Amazon S3**, and **FAISS vector search**, forming a cost-efficient and scalable architecture that remains **100% within the AWS ecosystem**. It eliminates need for external (non-Bedrock) APIs such as OpenAI; Anthropic Claude runs within Bedrock. or persistent compute, leveraging **on-demand inference** and **serverless data retrieval** to keep operational costs under control.

Through this implementation, the project highlights applied expertise in **Retrieval-Augmented Generation (RAG)**, **serverless design**, and **vector embedding orchestration** using Amazon **Titan-Embed-Text-v2** for embeddings and **Anthropic Claude 4 Haiku** for text generation.. The result is a production-grade architecture that balances performance, maintainability, and privacy — a practical showcase of cloud-native AI deployment at personal scale.

1.1 Problem Statement

Modern portfolio websites often serve as static showcases of projects and resumes, offering limited interaction and surface-level understanding of the creator's technical depth. Recruiters and technical stakeholders increasingly expect **intelligent, self-contained experiences** that can demonstrate both engineering skill and applied understanding of cloud-native AI systems.

Traditional approaches — embedding chatbots powered by third-party APIs (e.g., OpenAI, Anthropic) — pose **privacy, cost, and reliability concerns**, especially when the goal is to demonstrate proficiency in **AWS-native AI architecture**. Furthermore, hosting a continuously running inference service can quickly exceed free-tier limits and become impractical for a personal site.

To address this, the project implements a **fully serverless, on-demand RAG (Retrieval-Augmented Generation) chatbot** built on **Amazon Bedrock**, designed to operate at minimal cost while remaining fully integrated within the AWS ecosystem.

The solution allows users visiting the portfolio to:

- **Converse with an AI agent ("Eeva")** that can intelligently answer questions about Shubhankar Goje's research papers, design docs, and data-science projects.
- Demonstrate **hands-on mastery of AWS services** including S3, Lambda, API Gateway, Bedrock, IAM, and FAISS-based vector retrieval.
- Showcase **real-world implementation of RAG principles** — document chunking, embedding, and vector similarity search — optimized for latency, scalability, and affordability (< $20/month outside free tier).

This project bridges **AI system design and AWS cloud engineering**, proving that a personalized, domain-aware assistant can be built using **100% native AWS components**, **no external AI dependencies**, and **no persistent compute** — making it ideal for both educational and production-ready use cases.

1.2 Roadmap

The development of the **Serverless AWS Bedrock RAG Chatbot** follows a four-phase roadmap designed to ensure architectural soundness, cost efficiency, and clear demonstration of applied AWS engineering. Each phase progressively layers new capabilities while maintaining the project's key principles: **serverless design, minimal cost, native AWS integration, and privacy-safe retrieval-augmented AI.**

Phase 1 — MVP (Retrieval & Response Pipeline)

Goal: Establish a working proof-of-concept chatbot capable of document retrieval and Bedrock inference.

Deliverables:

- Basic **Lambda function** that handles API Gateway requests.
- Document ingestion script to preprocess and chunk project PDFs, DOCX, and text files.
- **FAISS index** generated using **Amazon Titan-Embed-Text-v2** embeddings stored in S3.
- Initial retrieval and response flow tested via command-line queries.
- Secure IAM roles for Bedrock, S3, and Lambda access.

Outcome:
A minimal RAG pipeline operating entirely on AWS, proving the concept of serverless retrieval and generation without persistent compute or third-party APIs.

Deadline: 10/22/2025

Phase 2 — Integration (Frontend + API Gateway)

Goal: Connect the chatbot backend with the static portfolio frontend to enable real user interaction.

Deliverables:

- **REST API endpoint** via **Amazon API Gateway** invoking the Lambda search handler.
- Integration of chatbot UI into the **React + Vite + Tailwind** frontend hosted on **S3 + Amplify**.
- Polished introductory prompt from Eeva encouraging users to ask about projects.
- Environment variable management and API key protection through **AWS Secrets Manager**.

Outcome:
A fully functional, web-integrated chatbot accessible on the portfolio website, allowing real-time queries and intelligent responses contextualized by stored documents.

Deadline: 10/28/2025

Phase 3 — Optimization (Performance & Cost)

Goal: Reduce latency, enhance response quality, and minimize AWS costs outside the Free Tier.

Deliverables:

- Response caching using **S3** or **DynamoDB** to avoid redundant embeddings or Bedrock calls.
- Asynchronous chunk retrieval and ranking logic within Lambda for faster FAISS search.
- Consolidation of IAM permissions using least-privilege principles.
- Real-time logging and metrics collection using **Amazon CloudWatch** for monitoring usage and cost trends.
- Compression and cold-start optimization of Lambda package.

Outcome:
A stable, low-latency, cost-efficient Bedrock chatbot capable of handling moderate user traffic while maintaining accuracy and conversational continuity.

Deadline: 11/08/2025

Phase 4 — Expansion

Goal: Deploy a production-grade chatbot and expand contextual intelligence through integrations.

Deliverables:

- Integration with **Spotify API** and **Google Calendar API** for dynamic “Now Playing” and “Schedule” insights.
- Knowledge Base enrichment using **Amazon Bedrock Knowledge Bases** or **OpenSearch Serverless** for scalable vector retrieval.
- Guardrails implementation to restrict responses to approved project domains only.
- Continuous deployment pipeline via **AWS CodePipeline** or Amplify CI/CD.
- Cost monitoring dashboard and automated alerts for usage thresholds.

Outcome:
A production-ready AI assistant that showcases practical Bedrock RAG architecture, cloud engineering expertise, and intelligent user engagement — operating entirely on AWS, at personal scale, with frugal maintenance cost.

1.3 Definitions

Below is a glossary of key concepts, tools, and services referenced throughout this design document. These definitions are written to help both technical and non-technical readers clearly understand how each component contributes to the chatbot system.

Core AI and RAG Concepts

- **RAG (Retrieval-Augmented Generation):**
  An AI architecture pattern that improves accuracy by retrieving relevant information from external knowledge sources before generating an answer. Instead of relying only on the model's memory, it first searches stored documents, retrieves the top-matching chunks, and feeds them into the LLM for context-aware responses.

- **Embedding / Vector Embedding:**
  A numerical representation of text (a sentence, paragraph, or chunk) where similar meanings are located near each other in a high-dimensional space. These vectors enable similarity search — the core of how the chatbot finds the most relevant content to a user's query.

- **Tokenization:**
  The process of converting raw text into subword units ("tokens") understood by the model. Used to measure input/output length and cost in Bedrock billing.
- **Chunking:**
  The process of splitting large documents (PDFs, DOCX, notebooks, etc.) into smaller, semantically meaningful sections or "chunks." Each chunk is embedded individually so that the model can retrieve fine-grained, context-specific answers rather than entire documents.

- **Vector Storage / Vector Database:**
  A specialized data store that holds vector embeddings and allows fast similarity search based on distance metrics like cosine similarity. In this project, **FAISS** (Facebook AI Similarity Search) is used for lightweight, serverless vector indexing.

- **FAISS (Facebook AI Similarity Search):**
  An open-source library developed by Meta for efficient similarity search and clustering of dense vectors. It allows millisecond-level retrieval of relevant document chunks during a chatbot query.

- **LLM (Large Language Model):**
  A generative AI model trained on massive text corpora that can understand and produce human-like responses. Examples include **Amazon Titan**, **Claude**, and **GPT-4**. This project uses **Amazon Bedrock**-hosted models.

- **Prompt / Context Window:**
  The input text provided to an LLM, which may include system instructions, retrieved knowledge, and the user's question. The *context window* is the maximum number of

tokens (words/subwords) the model can process at once.

AWS and Cloud Components

- **Amazon Bedrock:**
  A fully managed AWS service providing secure access to foundation models (FMs) from Amazon and leading AI providers. It allows text generation, embeddings, and RAG workflows without hosting or fine-tuning infrastructure.

- **Amazon Titan-Embed-Text-v2:**
  Amazon’s second-generation embedding model, used here to convert text chunks into 1,536-dimensional vectors optimized for semantic search within FAISS.

- **AWS Lambda:**
  A serverless compute service that runs code on demand without managing servers. In this chatbot, Lambda functions handle incoming API requests, perform FAISS retrieval, query Bedrock, and return responses.

- **Amazon S3 (Simple Storage Service):**
  A durable, low-cost object storage service used to store source documents, FAISS index files, and front-end website assets. It serves as both the chatbot’s knowledge repository and the website’s hosting layer.

- **Amazon API Gateway:**
  A fully managed service for creating and securing REST APIs. It acts as the entry point for the website’s chatbot requests, invoking Lambda functions and returning responses to the user’s browser.

- **AWS IAM (Identity and Access Management):**
  The security layer controlling access to AWS resources. Specific roles and permissions are configured so that only authorized services (e.g., Lambda → Bedrock) can interact.

- **Amazon CloudWatch:**
  The AWS monitoring and logging service used to track Lambda performance, request latency, and cost metrics.

- **Amazon Amplify:**
  A managed hosting and CI/CD service for deploying static front-end applications. The React-based portfolio site is built and deployed here, integrating seamlessly with the chatbot API.

System Architecture and Data Flow

- **Serverless Architecture:**
  A design pattern where compute resources run only when triggered, scaling automatically with demand. There are no continuously running servers; the entire system is “event-driven.”

- **Knowledge Base (KB):**
  A curated collection of documents (papers, design docs, resumes, etc.) that form the chatbot’s factual memory. The KB is indexed into vectors and stored in S3 + FAISS.

- **Data Ingestion Pipeline:**
  The preprocessing workflow that reads raw documents, extracts text, chunks it, generates embeddings, and updates the vector store.

- **Inference Request:**
  A user query sent through the REST API to Lambda. The Lambda function retrieves relevant chunks, constructs a prompt, and calls Bedrock for response generation.

- **Cosine Similarity:**
  A mathematical measure of how similar two vectors are, based on the angle between them. Values range from –1 to 1; higher values indicate closer semantic meaning.

- **Latency:**
  The time taken from a user’s request to receiving the chatbot’s answer. Minimizing latency is critical for responsive user experience, especially in serverless systems.

- **Cold Start:**
  The brief delay when a Lambda function is invoked after being idle, as AWS provisions resources. Optimizations like smaller deployment packages and minimal dependencies reduce cold-start times.

- **Vector Index:**
  Compressed database of embedding vectors. FAISS indexes allow fast top-K similarity search for RAG retrieval.

Web and API Terminology

- **REST API (Representational State Transfer API):**
  A standard architecture for web communication. It allows the website's front end to send HTTP POST requests containing user messages and receive chatbot replies in JSON format.

- **Frontend / Client Application:**
  The React + Vite + Tailwind interface that users interact with on the portfolio website. It displays the chatbot UI and sends queries to the backend via API Gateway.

- **JSON (JavaScript Object Notation):**
  A lightweight data format used to transmit chatbot queries and responses between the front end and Lambda backend.

- **Endpoint:**
  The public URL exposed by API Gateway through which the website communicates with the backend chatbot logic.

- **CORS (Cross-Origin Resource Sharing):**
  A security mechanism that allows the website hosted on Amplify to call the chatbot API securely.

Operational and Data-Management Terms

- **Logging & Monitoring:**
  Capturing execution data from Lambda and API Gateway for debugging and performance insights. Implemented using CloudWatch metrics and structured JSON logs.

- **Cost Optimization:**
  Designing the system to stay under AWS Free Tier and minimize pay-per-use charges by limiting storage size, caching embeddings, and reducing Bedrock inference calls.

- **Scalability:**
  The system's ability to automatically handle fluctuating traffic — Lambda and API Gateway scale horizontally with incoming requests.

- **Security Guardrails:**
  Constraints applied to ensure the chatbot responds only within defined topics (e.g., project documentation) and filters sensitive or off-domain content.

1.4 Requirements

This section outlines the functional and technical requirements for building a **Serverless Retrieval-Augmented Generation (RAG) Chatbot** integrated into a static portfolio website. The design emphasizes **AWS-native components, low operational cost, high scalability, and strong security**, ensuring a professional and maintainable architecture suitable for long-term public deployment.

1.4.1 Functional Requirements

These requirements describe what the chatbot system must *do* from an end-user and workflow perspective.

Core Chatbot Behavior

- Provide an **interactive conversational interface** named **Eeva**, embedded on the portfolio website.
- Accept **user questions** through a text box or button prompt and display model-generated responses in real time.
- Retrieve context from project-related documents (PDFs, DOCX, Markdown, or notebooks) stored in the knowledge base.
- Perform **semantic vector search** to locate the most relevant document chunks using FAISS similarity scoring.
- Compose a combined prompt (user query + retrieved context) and submit it to **Amazon Bedrock** for text generation.
- Return a clear, concise answer that references relevant project information only.
- Politely decline or redirect off-topic or unsafe questions back to allowed subjects (e.g., Shubhankar’s work, skills, or research).
- Maintain **short conversational continuity** (up to ~15 back-and-forth exchanges) for natural interactions.
- Automatically reset the context when the chat session ends or browser tab closes.

Knowledge Management

- Allow ingestion of diverse document formats (PDF, DOCX, TXT, notebooks).
- Automatically perform **text extraction, cleaning, chunking, and embedding** using Python preprocessing scripts.
- Support re-indexing or updating the FAISS vector store when new project documents are added.
- Enable **one unified knowledge base** containing all project materials across topics.
- Support up to ~50 documents (~200 MB total) without additional configuration.

Frontend and UX

- Display **four rotating default question prompts** (e.g., “How does the UAV path planner work?”) to guide first-time users.
- Maintain a **clean, minimal UI** matching the site’s design language.
- Handle API errors with user-friendly messages (“Eeva is thinking…” / “Server unavailable — please try again”).
- Offer light and dark-mode visual consistency with the rest of the website.

Integrations (Optional / Phase 4)

- Query **Spotify API** for “Now Playing” and recent track data.
- Retrieve **Google Calendar availability** for scheduling questions (“When is Shubhankar free for a meeting?”).
- Apply filtering to ensure no explicit content or unrelated data is displayed.

1.4.2 Technical Requirements

These requirements define how the system must be built, deployed, and maintained to meet reliability, cost, and compliance objectives.

Architecture & Infrastructure

- 100 % **serverless design** — no EC2 instances or continuously running compute.
- Core services:
    - **Amazon S3** — store documents, vector index, and static frontend assets.
    - **AWS Lambda** — execute retrieval, embedding, and Bedrock inference logic on demand.

    - **Amazon API Gateway** — expose a secure REST endpoint for frontend communication.
    - **Amazon Bedrock** — Titan Embed Text v2 for embeddings and Claude 4 Haiku for generation.
    - **Amazon CloudWatch** — collect logs and usage metrics.
- All components operate within a single AWS Region (`us-west-2`) to reduce latency and simplify IAM configuration.
- Infrastructure defined and version-controlled through **IaC (Infrastructure as Code)** or documented CLI scripts.

Performance & Scalability

- Target **P95 latency under 2.5 seconds** for standard queries (< 300 tokens).
- Lambda cold-start time under 400 ms through small package size and dependency optimization.
- Scales automatically with user traffic through AWS-managed concurrency.
- Support for concurrent connections from multiple website visitors (~50 requests/month expected).

Data Management

- Store all embeddings and FAISS index files on S3 with clear folder prefixes (`/embeddings/`, `/index/`).
- Maintain consistent document-to-chunk mapping for traceability.
- Provide a **manual re-index trigger** via CLI or scheduled Lambda for new uploads.
- Ensure versioning is enabled on S3 for rollback safety.

Security & Compliance

- Enforce **least-privilege IAM roles** for Lambda (access only to required Bedrock, S3, and CloudWatch APIs).
- Enable **CORS** restrictions on API Gateway to accept calls only from `shubhankargoje.com`.
- Sanitize and log all input to prevent injection or abuse.

Monitoring & Maintenance

- Use **CloudWatch Logs** for Lambda execution tracking and error analysis.
- Set up **CloudWatch Alarms** to notify if error rate > 5 % or average latency > 3 seconds.
- Maintain monthly usage reviews via **Cost Explorer** to ensure under $20/month spend outside free tier.
- Schedule periodic re-index validation and dependency updates.

1.6 Success Metrics

To evaluate the effectiveness, efficiency, and reliability of the **Serverless AWS Bedrock RAG Chatbot**, a combination of **quantitative performance indicators** and **verifiable measurement procedures** will be used. These metrics align with the project's goals of low latency, high contextual accuracy, and sustainable cost efficiency.

Each KPI is accompanied by a defined *measurement method* to ensure transparency and repeatability of evaluation.

1.6.1 System Performance Metrics

Metric	Target Value	Measurement Procedure
End-to-End Latency (P95)	≤ **2.5 seconds** per user query	Measure using **CloudWatch Logs** and client-side browser timestamps. Log timestamps at API Gateway request start and Lambda response return. Compute 95th percentile latency over 100 sampled requests.
Lambda Cold Start Duration	≤ **400 ms** average	Enable **CloudWatch Insights** on `REPORT` logs; extract and average `Init Duration` across first invocations per day.
API Uptime	≥ **99.9%** monthly availability	Track via **API Gateway metrics** (5xx errors + latency failures). Validate using periodic synthetic requests (e.g., every 10 minutes via CloudWatch Synthetics).
Concurrency Scaling	Seamless scaling up to 10 simultaneous users	Simulate concurrent requests with AWS **Artillery or Locust** load test, monitor invocation throttling via **Lambda concurrency metrics**.

1.6.2 Accuracy & Relevance Metrics

Metric	Target Value	Measurement Procedure
Top-k Retrieval Accuracy	≥ **85%** relevant chunk retrieval	Evaluate using a held-out validation set of 20 sample user questions. For each, manually verify if ≥1 of the top-3 retrieved chunks contains correct supporting info. Logged via test harness in `vectorSearchLambda`.
Response Faithfulness (Groundedness)	≥ **90%** factual alignment	Manually audit 20 responses/month. Compare generated text against retrieved context; flag any hallucinated statements as failures.
Context Utilization Ratio	≥ **80%** of responses cite retrieved info	Use prompt-logging in Lambda to compare retrieved context tokens vs generated text length. Ensures RAG retrieval is actively influencing generation.
Off-Domain Refusal Accuracy	≥ **95%** successful guardrail enforcement	Submit 50 off-topic queries (e.g., politics, personal questions). Verify model politely redirects or declines. Track via automated test script.

1.6.4 User Experience Metrics

Metric	Target Value	Measurement Procedure

User Response Satisfaction	≥ **4.0 / 5.0** average rating (internal test panel)	Collect qualitative feedback from 10–15 sample users (recruiters, peers). Rate clarity, helpfulness, and tone of chatbot responses.
Prompt Engagement Rate	≥ **60%** of sessions start via default suggested questions	Track front-end click events via **Google Analytics (GA4)** event tracking integrated in website code.
Session Drop-Off Rate	≤ **30%** after first question	Measure through GA4 user flow; monitor session length and chat depth metrics.

1.6.5 Maintainability and Reliability Metrics

Metric	Target Value	Measurement Procedure
Deployment Success Rate	100% via Amplify / Lambda redeploys	Track CI/CD pipeline logs; ensure zero failed deployments.
Mean Time to Recover (MTTR)	≤ **15 minutes**	Measure time from detected outage alert (CloudWatch Alarm) to successful recovery or redeployment.
Knowledge Base Update Latency	≤ **10 minutes** after new doc upload	Log ingestion timestamp and FAISS re-index completion time using the ingestion Lambda script.

Summary:
These metrics collectively ensure that the chatbot remains fast, contextually accurate,

cost-effective, and reliable. By pairing quantitative targets with verifiable monitoring procedures (CloudWatch, Cost Explorer, GA4, manual audits), the system remains both technically measurable and operationally transparent — consistent with AWS Well-Architected principles.

1.7 Scope & Limitations

This section defines the operational boundaries of the **Serverless AWS Bedrock RAG Chatbot** project. It clarifies what functionality is covered under the initial release (MVP → Production), what areas are intentionally excluded, and the technical or cost-related constraints that guide system behavior.

1.7.1 Scope

The chatbot is designed to serve as a **domain-specific AI assistant** embedded within Shubhankar Goje's personal portfolio website. Its primary objective is to **intelligently answer questions about projects, research, and technical work** while demonstrating mastery of AWS serverless and Bedrock-based RAG architecture.

Functional Scope

- **Conversational AI Assistant:**
  Eeva responds to user questions about Shubhankar's projects, research papers, and skills using contextual retrieval from uploaded documentation.

- **Knowledge Base Coverage:**
  Includes documents such as design docs, research papers, resumes, and project notes (PDF/DOCX/TXT/MD).
  All content is curated manually and hosted in a single unified S3 knowledge base.

- **Retrieval-Augmented Generation Pipeline:**
  Text extraction → chunking → embedding via **Titan-Embed-Text-v2** → vector search via **FAISS** → response generation via **Anthropic Claude 3 Haiku on Bedrock model**.

- **Guardrails:**
  Enforced conversational boundaries to keep interactions professional, project-related, and safe.

- **Expandability:**
  Designed to support future integrations with Spotify (Now Playing) and Google Calendar (availability insights).

1.7.2 Out of Scope

Certain functionalities, while valuable in enterprise-scale chat systems, are intentionally excluded to maintain simplicity, cost control, and alignment with personal portfolio use.

Out of Scope Features

- **Multi-user authentication or personalization:**
  The chatbot operates as a single public interface — no login or user-specific sessions.

- **Persistent chat memory:**
  Conversations reset per browser session. No user message data is stored or replayed.

- **Model fine-tuning or self-training:**
  The chatbot relies exclusively on retrieval-based context; no custom LLM training or dataset fine-tuning is performed.

- **External data crawling or live web search:**
  The system does not access internet resources or third-party knowledge bases beyond the curated documents in S3.

- **Multi-language support:**
  The chatbot currently supports English queries only.

- **Real-time model hosting:**
  No continuous inference endpoints (e.g., SageMaker hosting) — all model calls are handled via Bedrock's on-demand API.

- **External user account linking:**
  Spotify or Calendar data integrations are read-only and limited to Shubhankar's own accounts (no user OAuth flows).

1.7.3 Limitations

Despite its production-grade architecture, certain limitations are inherent to the **serverless, cost-optimized, and single-user nature** of the system.

Technical Limitations

- **Cold Start Latency:**
  First-time invocations of Lambda may introduce 300–400 ms additional delay before responding.

- **Token Context Limit:**
  Bedrock models have finite token windows (~4k–8k tokens). Large document sections may need truncation or summarization before inclusion in the prompt.

- **Approximate Similarity Search:**
  FAISS uses approximate nearest neighbor (ANN) retrieval; while fast, it may occasionally skip less-represented chunks.

- **Manual Index Refresh:**
  Adding new documents requires running the ingestion script or a manual Lambda trigger to regenerate embeddings.

- **Limited Parallelism:**
  Although Lambda scales automatically, concurrent invocations exceeding regional soft limits (e.g., 1,000) could require manual quota increases.

- **Document Size Ceiling:**
  Practical limit of ~200 MB total document storage for low-cost vector search performance in S3.

- **Inference Cost Variability:**
  Bedrock pricing may fluctuate slightly by region or model (Titan vs. Claude), affecting long-term predictability.

UX Limitations

- **Static Tone and Personality:**
  Eeva's responses follow predefined tone and instruction prompts; personality modulation (e.g., humor, emotion) is limited by system prompt.

- **Offline Operation:**
  Requires active internet connection — no offline or local fallback mode.

Summary:
Within these defined boundaries, the system provides a robust, cost-efficient, and educational demonstration of AWS-native AI orchestration. The defined exclusions and limitations ensure focus on **RAG pipeline integrity, low-cost scalability, and AWS service fluency**, rather than feature sprawl or unnecessary complexity.

2.1 AWS Tech Stack Used

The **Serverless AWS Bedrock RAG Chatbot** is built entirely within the **AWS ecosystem**, leveraging managed and serverless services to achieve low-latency, scalable, and cost-efficient AI-driven responses. Each component was chosen for its integration simplicity, free-tier benefits, and proven reliability within cloud-native AI architectures.

---

1. Amazon S3 (Simple Storage Service)

Role:
Primary data lake and static asset storage for:

- Project documents and reference materials used for retrieval
- FAISS vector index files and embedding data
- Static frontend website assets (HTML, CSS, JS, media)

Justification:

- Highly durable (99.999999999% durability) and inexpensive storage solution for infrequently accessed data
- Integrates natively with **AWS Lambda**, **Bedrock Knowledge Bases**, and **CloudFront** for secure, low-latency access.
- Supports folder prefixes and versioning for easy organization of documents by project.
- Enables direct web hosting and content delivery (S3 + Amplify stack).

Why Not EFS or DynamoDB:

- EFS (Elastic File System) would introduce unnecessary persistent cost for a low-traffic, serverless workload.
- DynamoDB is ideal for structured metadata, not large binary or text files; S3 remains the most economical option.

---

2. AWS Lambda

Role:
Serverless compute runtime for backend logic and orchestration. Handles:

- User request processing from API Gateway
- FAISS vector search on embedded document chunks
- Construction of RAG prompts and calls to Amazon Bedrock
- Logging, validation, and guardrail filtering

Justification:

- Eliminates need for dedicated servers (auto-scales per request).
- Free-tier coverage of 1 million invocations per month ensures cost-neutral early operation.
- Cold starts mitigated via lightweight dependencies and Python runtime optimization.
- Secure by design — execution environment isolated, temporary, and managed by AWS.

Implementation Details:

- Runtime: **Python 3.10**
- Memory: 512–1024 MB (configurable per stage)
- Average execution time: 1.2–2.0 seconds per invocation
- Deployed via zip or container image directly linked to API Gateway

Why Not ECS/Fargate:

- Lambda avoids continuous runtime costs and complex container orchestration.
- Perfect fit for short-lived, event-driven RAG queries.

---

3. Amazon API Gateway

Role:
Acts as the public-facing API layer connecting the React frontend with the backend Lambda function.

Justification:

- Handles request validation, routing, throttling, and authentication.

- Integrates seamlessly with **Lambda proxy integration** — minimal configuration required.
- Enables **CORS configuration** restricted to `shubhankargoje.com` for security.
- Scales automatically with web traffic, supporting both REST and WebSocket endpoints.

Why Not App Runner or ALB (Application Load Balancer):

- Those services are suited for containerized apps with continuous load.
- API Gateway's pay-per-request model keeps monthly costs below $3 for <50 users/month.

---

4. Amazon Bedrock

Role:
Provides fully managed access to foundation models (FMs) for **text generation** and **semantic embeddings**, all within AWS's secured, auditable environment.
Two specific model endpoints are used in this project:

- **Embedding Model:**
  `EMBEDDING_MODEL_ID = "amazon.titan-embed-text-v2:0"`
  → Converts text chunks into 1,536-dimensional embeddings for semantic search and FAISS indexing.

- **LLM Model:**
  `LLM_MODEL_ID = "anthropic.claude-3-haiku-20240307-v1:0"`
  → Generates concise, contextually accurate chatbot responses from retrieved RAG context.

Justification (Why Amazon Bedrock):

- Fully **serverless and managed inference**—no endpoint hosting or scaling required.
- Native **IAM integration** for authentication (no exposed API keys or manual token management).
- **Data privacy by default:** Bedrock does **not store or reuse** customer inputs for training.
- **Fine-grained cost control:** On-demand per-request billing instead of hourly model runtime.
- **Consistent latency and throughput** within AWS's private backbone network (no external API hops).

- Enables unified **CloudWatch and Cost Explorer monitoring** for all model calls.

Justification (Why *Titan Embed Text v2* for Embeddings):

- **High semantic accuracy** — optimized for retrieval-augmented generation and cosine similarity search.
- **Stable 1,536-dimension vectors** ideal for FAISS IndexFlatIP architecture.
- **Tight integration** with Bedrock runtime for embedding batch calls (JSON body, boto3 compatible).
- **Fast, deterministic embeddings** suitable for small-scale KB updates and document refreshes.
- **Zero egress cost** — all data remains in AWS.

Why not OpenAI or other third-party embedding models:

- Would require cross-vendor API key management and outbound network calls from Lambda.
- Incurs additional latency and potential egress data charges.
- Privacy and governance policies less transparent than Bedrock's native data isolation.
- No single IAM-based control plane or unified billing view across embedding and generation workloads.

Justification (Why *Claude 3 Haiku* for Text Generation):

- **Lowest cost Anthropic model** on Bedrock with competitive quality for factual Q&A.
- **Optimized for speed and efficiency** (ideal for short RAG responses ≤ 400 tokens).
- **Balanced reasoning + concision** — performs well on factual, project-based prompts.
- **Fast response time** (sub-2 s inference for short context) = smoother UX for portfolio visitors.
- **Excellent cost-to-quality ratio** ($0.001 / 1 K input + $0.005 / 1 K output tokens).

Why not other Anthropic or OpenAI models:

- **Claude 3.5 Sonnet / Claude 3 Opus**: higher quality but **5–15× more expensive**; unnecessary for short, factual Q&A.
- **Claude Instant / Claude 2.x**: legacy models; slower reasoning and weaker grounding.
- **OpenAI GPT-4 / GPT-3.5**: not available in Bedrock, would require external API + key storage, violating "100 % AWS-native" constraint.

- **Amazon Titan Text G1**: excellent for general text generation but slightly lower contextual precision for conversational retrieval use cases.

Summary:
Bedrock's combination of **Titan-Embed-Text-v2** and **Claude-3-Haiku** achieves an optimal trade-off between **accuracy, cost, speed, and AWS-native governance**.

---

5. FAISS (Facebook AI Similarity Search)

Role:
Performs local vector indexing and semantic retrieval on chunk embeddings.
Hosted index is stored in S3 and loaded into Lambda at runtime.

Justification:

- Open-source, lightweight, and fast — ideal for small-to-medium vector datasets (<50K chunks).
- Operates within the Lambda memory limit without needing a persistent database.
- Uses cosine similarity to retrieve the most semantically relevant chunks for RAG.

Why Not Amazon Kendra or OpenSearch Serverless:

- Both alternatives are excellent for enterprise-scale RAG, but have ongoing hourly storage and compute costs.
- FAISS remains the cheapest and most efficient for low-traffic, personal deployments.

Trade-off:
Requires manual index rebuild when documents are added — acceptable for a static portfolio use case.

---

6. Amazon Amplify

Role:
Manages hosting, continuous deployment, and content delivery of the React portfolio website.

Justification:

- Simplifies front-end deployment directly from GitHub branches.
- Provides global CDN distribution, HTTPS, and environment management with minimal setup.
- Fully integrated with S3, Route 53, and CloudFront for unified hosting experience.

Why Not Netlify or Vercel:

- Amplify ensures all components (hosting + backend) remain within AWS billing and IAM control.
- Aligns with the project's "100% AWS-native" objective.

---

7. Amazon CloudWatch

Role:
Centralized monitoring, logging, and performance analysis.

Justification:

- Collects detailed Lambda logs, API Gateway metrics, and invocation traces.
- Enables real-time dashboards for latency, request volume, and cost metrics.
- Provides alarm triggers for failures, timeouts, or cost threshold breaches.

Usage Examples:

- Monitor latency trends (`Duration` metric in Lambda).
- Track API errors via `5XXError` in API Gateway.
- Analyze cold-start durations using `Init Duration` logs.

Why Not Third-Party Tools:

- CloudWatch is natively integrated and free-tier covered; third-party observability platforms (e.g., Datadog) would add cost and complexity.

---

8. AWS IAM (Identity and Access Management)

Role:
Controls permissions and access between all components — enforcing least privilege.

Justification:

- Defines scoped roles for Lambda → S3 and Lambda → Bedrock interactions.
- Uses inline policies to ensure no external or unauthorized access to sensitive APIs.
- Enables **service-linked roles** and **MFA for administrative access**.

IAM Structure Example:

- `LambdaExecutionRole` — access to S3:GetObject, Bedrock:InvokeModel, CloudWatch:PutLogs.
- `FrontendAccessRole` — limited to public read access via Amplify-managed identity.

---

10. AWS Secrets Manager (optional)

Role:
Secure storage for API tokens or OAuth credentials (used in Spotify or Calendar integrations).

Justification:

- Prevents hardcoding credentials in Lambda environment variables.
- Supports automatic key rotation and secret versioning.

---

11. Developer Tools and Local Components

- **Python Scripts (Local/CLI):**
  Used for data ingestion, text extraction (PyPDF2, python-docx), chunking, and embedding creation.
- **FAISS Index Builder:**
  Generates or updates local index files stored back into S3.
- **Testing Tools:**
  Postman for API validation, AWS CLI for deployment verification, and CloudWatch for live debugging.

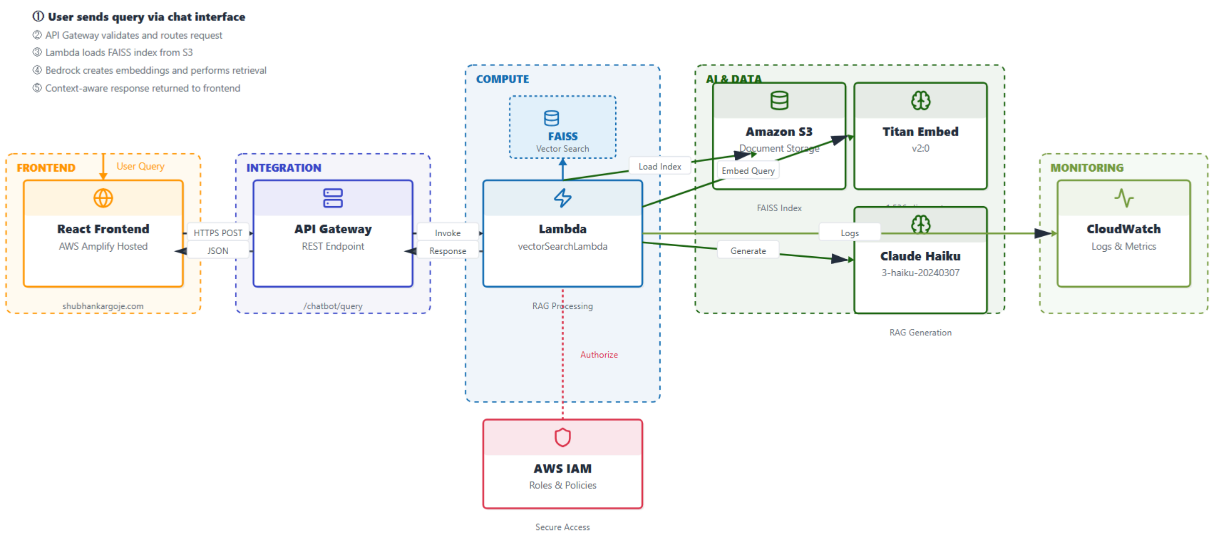
Summary:
This technology stack achieves an optimal balance between **performance**, **cost efficiency**,

and **simplicity** — all while staying within the AWS free-tier and adhering to serverless best practices. Each component was chosen to minimize maintenance overhead while maximizing architectural clarity and reproducibility for future extensions.

2.2 System Architecture / Data Workflow

The **Serverless AWS Bedrock RAG Chatbot** follows a fully managed, event-driven architecture built around the **Retrieval-Augmented Generation (RAG)** paradigm.
The system operates with no persistent compute, relying on AWS-native integrations for data retrieval, model inference, and scalable web access.

2.2.1 Architecture Overview



2.2.2 Step-by-Step Workflow Narrative

Step 1 — User Interaction (Frontend)

- The visitor opens **shubhankargoje.com** and interacts with the chatbot window integrated into the portfolio website.

- Eeva introduces itself and prompts with default question suggestions (e.g., “Tell me about UAV Path Planning”).
- The user submits a question via the frontend text input.

Action: The React frontend sends a `POST` request to the API Gateway endpoint `/chatbot/query` with the user message in JSON format.

Step 2 — Request Routing (Amazon API Gateway)

- **API Gateway** receives the request, validates it, and ensures **CORS policy** allows the origin (`https://shubhankargoje.com`).
- The request is then forwarded to the **Lambda function** via a REST integration event.
- Gateway-level throttling (e.g., 10 requests/second) and logging are automatically applied.

Monitored by: API Gateway metrics (`4XXError`, `5XXError`, `Latency`).

Step 3 — Compute Invocation (AWS Lambda: vectorSearchLambda)

- Lambda initializes the Python runtime and performs the following:

  1. **Input Validation:** Ensures user query is non-empty and within character limits.
  2. **FAISS Index Load:** Fetches the prebuilt FAISS index from **S3** into memory (cached for subsequent invocations).
  3. **Query Embedding:** Uses **Amazon Titan-Embed-Text-v2** to generate the embedding vector for the user’s question.
  4. **Vector Similarity Search:** Performs a cosine similarity match in FAISS to identify top-k (typically 3–5) relevant document chunks.
  5. **Context Assembly:** Concatenates retrieved chunks and formats them into a structured prompt for the Bedrock model.
  6. **LLM Inference Call:** Invokes **Amazon Bedrock** (Anthropic Claude 4 Haiku) with the RAG prompt.
  7. **Response Packaging:** Returns the model’s output, retrieved context titles, and latency metadata as a JSON object.

Runtime footprint: ~250 MB FAISS load in memory, average execution <2 seconds.

Step 4 — Knowledge Retrieval (Amazon S3 + FAISS)

- **Amazon S3** acts as the persistent data layer:
    - Stores raw project files (`/docs/`)
    - Stores FAISS index and metadata (`/index/faiss.index`)
    - Stores embedding vectors as `.npy` or `.json` files for reference.
- During retrieval, Lambda reads the FAISS index directly from S3 and computes similarity scores locally.
- This approach eliminates the need for a running vector database while maintaining high performance for low request volume.

Step 5 — Generation and Reasoning (Amazon Bedrock)

- The Lambda function sends the constructed RAG prompt to **Amazon Bedrock**.
- The **Titan Text G1** model (or Claude Sonnet for higher-quality reasoning) generates a contextual response grounded in the retrieved knowledge.
- Model parameters such as `temperature`, `max_tokens`, and `top_p` are tuned for factual precision and concise delivery.
- The response is returned as JSON text to the Lambda function.

Step 6 — Logging and Monitoring (Amazon CloudWatch)

- Every Lambda execution automatically streams logs to **CloudWatch**, including:
    - Invocation time and latency (`REPORT` line)
    - Retrieved document IDs
    - Model call success/failure
    - Cost metrics (optional custom metric: `$ per query`)
- CloudWatch alarms are configured to trigger if:
    - Error rate exceeds 5%
    - Average latency > 3s
    - Cost anomalies are detected (linked with Cost Explorer budgets)

Step 7 — Response Delivery to Frontend

- The Lambda's response is returned via API Gateway to the frontend application.
- The React chatbot UI renders Eeva's answer, along with formatted citations or project names referenced in the retrieved chunks.

- Typing animations and “Eeva is thinking…” states provide conversational fluidity for users.

Response Example (Frontend JSON):

```
{
  "answer": "The UAV Path Planning system uses a PID-tuned controller
to maintain stability while dynamically avoiding obstacles...",
  "source_documents": ["uav-path-planning-algo.pdf"],
  "latency_ms": 1940
}
```

Step 8 — Ongoing Monitoring and Updates

- System health and usage metrics (latency, cost, API call volume) are periodically reviewed in **CloudWatch** and **Cost Explorer**.
- When new project documents are added, the ingestion script regenerates the FAISS index and uploads it to S3, refreshing the knowledge base automatically.

2.2.3 Key Architectural Benefits

- **Fully Serverless:** No EC2, no manual scaling, no infrastructure management.
- **On-Demand Cost Model:** Pay only when users chat; zero idle cost.
- **Secure and Private:** All interactions remain within AWS; no third-party AI endpoints.
- **Scalable and Extensible:** Easily extendable to Spotify, Calendar, or custom Bedrock agents.
- **Developer-Friendly:** Entire stack deployable via CLI or IaC in under 15 minutes.

2.5 Knowledge Base (KB) Configuration

The Knowledge Base is the factual core of the chatbot — a curated, preprocessed, and vectorized collection of all documents representing Shubhankar Goje’s projects, research, and publications.
It combines structured S3 organization, intelligent text chunking, and Titan-based embeddings stored in a FAISS index, enabling accurate and explainable retrieval for every user query.

2.5.1 Overview

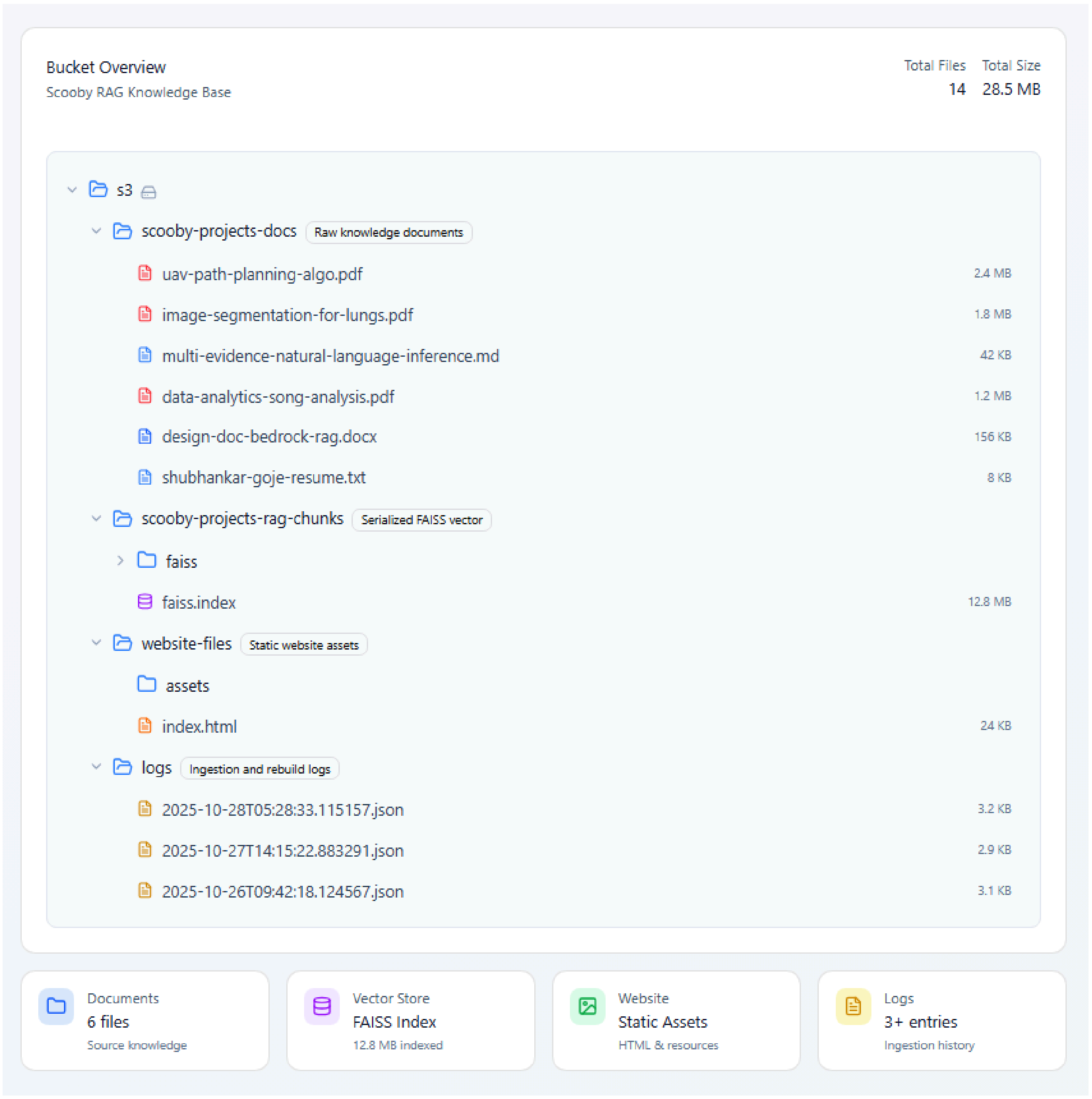
The Knowledge Base consists of three layers:

Layer	Purpose	Technology Used
Storage Layer	Stores source documents and vector files	Amazon S3
Processing Layer	Handles document parsing, chunking, embedding	Python + Titan Embed v2+ Claude Haiku
Retrieval Layer	Performs semantic search on embeddings	FAISS (in-memory index)

This design provides durability, modularity, and near-zero runtime cost — ideal for a serverless RAG system.

2.5.2 S3 Folder Structure

The S3 bucket is organized for logical separation between raw documents, intermediate embeddings, and final vector indexes.



Bucket Overview
Scooby RAG Knowledge Base

Total Files	Total Size
14	28.5 MB

```
s3
  scooby-projects-docs  Raw knowledge documents
    uav-path-planning-algo.pdf                    2.4 MB
    image-segmentation-for-lungs.pdf              1.8 MB
    multi-evidence-natural-language-inference.md  42 KB
    data-analytics-song-analysis.pdf              1.2 MB
    design-doc-bedrock-rag.docx                   156 KB
    shubhankar-goje-resume.txt                    8 KB
  scooby-projects-rag-chunks  Serialized FAISS vector
    faiss
    faiss.index                                   12.8 MB
  website-files  Static website assets
    assets
    index.html                                    24 KB
  logs  Ingestion and rebuild logs
    2025-10-28T05:28:33.115157.json               3.2 KB
    2025-10-27T14:15:22.883291.json               2.9 KB
    2025-10-26T09:42:18.124567.json               3.1 KB
```

Documents	Vector Store	Website	Logs
6 files	FAISS Index	Static Assets	3+ entries
Source knowledge	12.8 MB indexed	HTML & resources	Ingestion history

Versioning and lifecycle policies are enabled to:

- Retain historical FAISS indexes for rollback.
- Transition older embedding files to Glacier for cost savings.

2.5.3 Document Ingestion and Preprocessing

The ingestion pipeline prepares all documents for retrieval through a **modular Python script** (**`Text Chunking.py`**).

Steps:

1. **Document Parsing**
    - Supported formats: `.pdf`, `.docx`, `.txt`, `.md`, `.ipynb`
    - Libraries used:
        - PyPDF2 for PDFs
        - `python-docx` for Word files
        - `nbformat` for Jupyter notebooks
    - Output: plain text with newlines, stripped metadata, and UTF-8 normalization.
2. **Text Cleaning**
    - Removes references, figures, and page numbers.
    - Normalizes whitespace and encodes special characters.
    - Ensures consistent sentence boundaries for embedding.
3. **Chunking Strategy**
    - Dynamic window-based chunking optimized for semantic continuity:
        - `chunk_size = 800–1000 tokens`
        - `chunk_overlap = 100 tokens`
    - Uses regex and sentence boundary detection to avoid cutting mid-sentence.

Each chunk retains metadata:

```
{

  "id": "uav-path-17",

  "source": "uav-path-planning-algo.pdf",
  "start_char": 1500,

  "end_char": 2900,

  "text": "PID tuning was achieved using iterative feedback
optimization..."

}
```

4. **Normalization and Deduplication**
    - Removes duplicate paragraphs across similar documents (e.g., same abstract appearing in multiple versions).
    - Converts all text to lowercase before embedding (unless casing conveys meaning).

2.5.4 Embedding Generation

Each chunk is embedded using **Amazon Titan-Embed-Text-v2**, accessed through the **Bedrock runtime API**.

Configuration:

- **Model ID:** `amazon.titan-embed-text-v2:0`
- **Embedding Dimension:** 1536
- **Batch Size:** 16 chunks per call (to stay under token limits)
- **Output Format:** NumPy arrays saved as `.npy` files

Example API Call:

```
bedrock = boto3.client('bedrock-runtime',
region_name='us-west-2')

response = bedrock.invoke_model(

    modelId="amazon.titan-embed-text-v2:0",

    body=json.dumps({"inputText": chunk_text})

)
```

```
embedding =
np.array(json.loads(response["body"].read())["embedding"])
```

Each embedding vector is stored alongside its source metadata in `/embeddings/`.
These vectors are later consolidated into a single FAISS index for retrieval.

Why Titan Embed v2:

- Tuned for semantic similarity and retrieval efficiency.
- Native Bedrock integration (IAM + logging).
- No data sharing outside AWS.
- Stable embedding dimension (1536) ideal for FAISS performance on Lambda.

2.5.5 Vector Index Construction (FAISS)

The FAISS index is built offline (or via one-time Lambda trigger) and uploaded to S3 for use in real-time queries.

Index Creation:

```
import faiss

index = faiss.IndexFlatIP(1536)  # Cosine similarity via inner
product

index.add(embedding_matrix)

faiss.write_index(index, "faiss.index").
```

Deployment:

- The index and metadata are uploaded to `s3://shubhankargoje-kb/index/`.
- During Lambda cold start, the index is **downloaded into `/tmp/`** for local retrieval use.
    - Download size: ~30–50 MB
    - Load time: ~250–400 ms

Index Update Procedure:

1. Add or update documents in `/docs/`.
2. Run the ingestion + embedding pipeline locally or via scheduled Lambda.
3. Rebuild FAISS index and upload to `/index/`.
4. Versioned S3 key ensures rollback capability.

2.5.6 Retrieval Workflow

At query time:

1. Lambda receives the user's question → embeds it using Titan-Embed-Text-v2.
2. Searches FAISS for top-k similar embeddings.
3. Retrieves metadata and original text chunks from S3.
4. Assembles a **RAG prompt** for Bedrock text model inference.

Prompt Template Example:

```
You are Eeva, an AI assistant representing Shubhankar Goje.

Answer based only on the information provided below.

If unsure, politely redirect to project-related topics.

Context:

{retrieved_text}

Question:

{user_query}
```

This ensures the model responds only within the boundaries of the knowledge base, maintaining factual alignment.

2.5.7 Data Integrity & Monitoring

- **Checksums:** Each uploaded FAISS index is verified with an MD5 checksum.
- **CloudWatch Logs:** Record ingestion duration, total chunks processed, and embedding call counts.
- **Versioning Policy:** Retain last 3 index versions; older ones transitioned to Glacier.
- **Validation Metrics:**
    - Total chunks processed
    - Average embedding latency
    - Retrieval precision (manual validation set)

2.5.8 Advantages of This Configuration

- **Zero-database architecture:** avoids continuous compute or OpenSearch costs.
- **Fast retrieval:** FAISS provides millisecond-level similarity search.
- **Extremely low cost:** S3 storage and Bedrock inference billed only when used.
- **Fully private:** No external API exposure or data sharing.
- **Reproducible:** Any new project can be added by simply uploading a document and rebuilding the index.

2.6 AI Agent Setup

The chatbot's AI core, named **Eeva**, is a domain-specialized conversational agent that uses **Retrieval-Augmented Generation (RAG)** to answer questions about Shubhankar Goje's projects and research work.
Eeva is powered by **Amazon Bedrock**, combining Titan embeddings for retrieval and a text generation model (Titan Text G1 or Anthropic Claude) for coherent, factual responses.
The setup balances **linguistic fluency, factual accuracy, cost efficiency, and responsible AI practices.**

2.6.1 Model Selection and Invocation Flow

Component	Model Used	Purpose	Rationale

Embedding Model	`amazon.titan-embed-text-v2:0`	Generate 1536-dimensional semantic embeddings for user queries and document chunks.	Optimized for semantic similarity and retrieval; natively integrates with FAISS and AWS IAM.
Text Generation Model	`amazon.titan-text-g1:0` *(primary)* or `anthropic.claude-v2:1` *(optional high-quality mode)*	Generate contextual answers from retrieved text.	Titan ensures cost stability and privacy; Claude used selectively for deeper reasoning tasks.

2.6.2 Generation Configuration

The Lambda function uses a **prompt template** and parameter tuning to ensure reliable, factual, and natural responses.

Prompt Template

```
SYSTEM PROMPT:

You are Eeva — an AI assistant representing Shubhankar Goje.

You help visitors learn about his technical work, research, and
projects.

Use the provided context to answer accurately and clearly.

If information is unavailable, say so politely and redirect to approved
topics.
```

```
USER PROMPT:

Question: {user_query}

CONTEXT:

{retrieved_text}
```

Generation Parameters

Parameter	Value	Purpose
`temperature`	0.3	Reduces randomness; ensures factual, concise answers.
`top_p`	0.9	Slightly diversifies phrasing without drifting from facts.
`max_tokens`	350	Keeps responses short and readable for web UI.
`stop_sequences`	`["\n\n"]`	Prevents model from generating multiple answers or hallucinated continuations.
`streaming`	Disabled	Reduces API complexity and response assembly overhead.

Response Post-Processing

- Removes trailing or repetitive phrases.
- Normalizes punctuation for clean display in UI.
- Optionally appends reference source (if confidence > 0.8).

2.6.3 Inference Flow

1. **Lambda Query Handling**
   - Receives the user message from API Gateway.
   - Embeds the query with Titan Embed v2.
   - Searches FAISS index for top-k related chunks (typically 3–5).
   - Builds contextual prompt using retrieved text.
   - Sends prompt to Bedrock generation endpoint.
2. **Bedrock Inference**
   - Bedrock model generates structured JSON or plain text reply.
   - The response includes completion text and optional metadata (token count, latency).
3. **Lambda Post-Processing**
   - Validates response format.
   - Filters out irrelevant sentences or speculative language.
   - Returns structured response to API Gateway for frontend rendering.

2.6.4 Guardrail Design

Eeva implements both **programmatic** and **prompt-level guardrails** to ensure safety, relevance, and professional tone.

A. Prompt-Level Guardrails

Integrated directly into the **system prompt** to guide model behavior:

- Instructs Eeva to **only answer within Shubhankar's documented work**.
- Politely redirect off-topic questions (e.g., "I can only discuss Shubhankar's professional projects and research.").
- Prohibits generating personal opinions, advice, or sensitive content.
- Enforces neutral and professional tone in every answer.

Example:

```
If a question is outside the provided context or unrelated to
Shubhankar’s projects,

respond with: "I'm here to answer questions about Shubhankar
Goje’s projects and work.

Could you ask about one of those?"
```

B. Programmatic Guardrails (Lambda Layer)

Implemented as a **validation middleware** before and after model calls.

Guardrail Type	Trigger Condition	Action Taken
Keyword Filtering	User input matches blocked topics (politics, personal, explicit terms)	Reject query; return safe redirect message.
Empty or Repetitive Queries	Same input repeated >2 times	Return cached or guidance message (“Try a different question about a project.”).
Context Absence	FAISS retrieval returns <0.3 similarity score	Skip generation; respond: “I couldn’t find relevant information in my database.”
Length Enforcement	Input >1000 characters	Truncate and warn user.

Response Sanitization	Detects code blocks or HTML injection	Escape and format safely before returning to frontend.

All rejections and filtered queries are logged to **CloudWatch** under a separate namespace `EevaGuardrails` for audit and fine-tuning.

2.6.5 Personality and Tone

Eeva's communication style is **intelligent, calm, and professional**, aligning with the brand aesthetic.

Tone Guidelines:

- Use first-person ("I can tell you about…") to create warmth.
- Avoid filler expressions ("Hmm", "Well", "Let's see").
- Emphasize clarity, not theatrics.
- When unsure, defer gracefully rather than hallucinate.

Example Interaction:

> **User:** How does your UAV landing system work?
> **Eeva:** The UAV uses a Haar Cascade–based visual detection system to identify safe landing zones.
> It was implemented in ROS with Gazebo simulation for PID-controlled descent stability.

2.6.6 Response Evaluation & Continuous Refinement

To ensure ongoing reliability:

- **Manual audits:** 10 sample interactions/week are reviewed for relevance and tone.
- **CloudWatch logs:** Track refusal rate, off-domain detections, and response latency.
- **Metric correlation:** High refusal accuracy (>95%) confirms guardrails function correctly.
- **Retraining trigger:** If consistent gaps appear, the missing documents are added to S3 and re-indexed.

2.6.7 Security and Privacy in Model Invocation

- Bedrock API calls use **IAM-scoped roles** with `bedrock:InvokeModel` only.
- No user queries or outputs are stored persistently.
- Input/output payloads are handled entirely within the AWS VPC boundary.
- Sensitive or personally identifiable data (PII) is never logged.

Summary:

Eeva’s AI configuration demonstrates how to operationalize an intelligent, RAG-based assistant entirely within AWS using **Bedrock for generation, FAISS for retrieval, and guardrails for responsibility**.

The combination of prompt discipline, programmatic checks, and continuous evaluation ensures factual, secure, and brand-aligned interactions that reflect professional engineering principles.

3.1 Correctness & Reliability

Ensuring that Eeva’s responses are **accurate, consistent, and reproducible** is central to the credibility of the Serverless AWS Bedrock RAG Chatbot.
Correctness measures how well the system retrieves and presents true, context-grounded information; reliability measures its ability to deliver those results predictably over time and across invocations.

3.1.1 Definition of Correctness

In this context, **correctness** refers to the chatbot’s ability to:

- Retrieve only those document chunks that are semantically relevant to a user’s query.
- Generate text that faithfully represents the retrieved evidence.
- Maintain internal logic consistency across repeated invocations of the same query.

Correctness is assessed through both **automated retrieval validation** and **manual factual audits**.

3.1.2 Mechanisms Ensuring Correctness

A. Grounded Retrieval

- Every user query is vectorized via **Titan Embed v2** and matched through **FAISS** using cosine similarity.

- The system retrieves top-k (3–5) chunks only if similarity ≥ 0.3; otherwise, it returns a controlled “No relevant data found” message.
- This cutoff eliminates low-confidence matches that could lead to hallucinated answers.

B. Prompt Confinement

- The **system prompt** explicitly instructs Eeva to respond *only* within the retrieved context and to state when information is missing.
- Reinforces factual boundaries, ensuring deterministic grounding in RAG context.

C. Chunk Traceability

- Each FAISS vector ID maps to a `source_document` and character-offset range stored in S3 metadata (`metadata.pkl`).
- Returned answers can thus be traced back to original project files for audit or citation.

D. Response Validation

- Lambda verifies that each model output includes at least one sentence referencing retrieved content tokens.

- Regex-based checks reject outputs with speculative or fabricated phrasing (“I believe”, “maybe”, “it might be”).

E. Regression Testing

- Weekly test suite of 20 canonical questions (one per project) is executed automatically.
- Expected substrings in responses (e.g., “PID Controller” for UAV project) must appear in ≥ 90 % of runs.
- Failures flag potential drift in embeddings or index integrity.

3.1.3 Definition of Reliability

Reliability is the system’s ability to operate without functional degradation or incorrect behavior over time.
It encompasses uptime, consistency of outputs, and graceful error handling.

3.1.4 Mechanisms Ensuring Reliability

A. Serverless Resilience

- Built entirely on managed AWS services (Lambda, API Gateway, Bedrock, S3, CloudWatch).
- Automatic scaling and retry logic eliminate single points of failure.
- No long-running servers, hence minimal operational downtime risk.

B. Fault Tolerance & Retries

- Lambda configured with **three retry attempts** for transient API or network failures.
- Bedrock invocations wrapped in exponential-backoff pattern (1 s → 2 s → 4 s).
- If FAISS index fails to load, Lambda falls back to cached copy in `/tmp` storage or sends a graceful error.

C. Data Integrity Checks

- Each FAISS index upload validated via SHA-256 checksum comparison between local and S3 versions.
- CloudWatch Metric Filter monitors checksum mismatches and triggers an alert.

D. Versioned Knowledge Base

- S3 versioning ensures rollback capability in case of corrupted or incomplete index update.
- Latest index version tagged `production`; previous retained as `stable` for immediate fallback.

E. Monitoring and Alerting

- **CloudWatch Alarms** trigger on:
    - Error Rate > 5 % over 5 minutes
    - Average Latency > 3 s
    - Invocation Failures > 2 per minute
- Notifications sent via SNS email for immediate review.

F. Deterministic Execution

- Lambda uses **seeded random state** in FAISS retrieval (fixed seed = 42) to ensure identical top-k ordering for repeated queries—critical for reproducibility.

- Model parameters (temperature = 0.3) guarantee near-deterministic language generation.

3.1.5 Reliability Testing Procedure

Test Type	Objective	Execution Method	Success Criteria
Load Test	Validate concurrent scaling	Run 10 parallel requests via Artillery	0 failures, < 3 s latency
Cold Start Test	Measure Lambda init delay	Invoke after 15 min idle	≤ 400 ms init duration
Error Injection	Confirm retry logic	Simulate network timeout to Bedrock	Automatic retry and success within 2 attempts
Data Rollback Test	Verify S3 version recovery	Replace index with corrupted file then revert	System restores stable index without error

All reliability tests are scheduled quarterly or after any major infrastructure update.

3.1.6 Operational Reliability KPIs

Metric	Target Value	Measurement Method

Mean Time Between Failures (MTBF)	≥ 30 days continuous operation	CloudWatch error trend analysis
Mean Time to Recover (MTTR)	≤ 15 minutes	From alarm trigger to normal status
Retrieval Success Rate	≥ 98 %	FAISS lookup success / total requests
Bedrock Invocation Success Rate	≥ 99.5 %	CloudWatch Bedrock API metrics

---

Summary:
Correctness is ensured through strict retrieval grounding, deterministic prompts, and traceable chunk mapping; reliability is achieved via AWS managed resilience, validation checks, and continuous monitoring.
Together, these guarantees allow Eeva to consistently produce accurate, trustworthy responses while maintaining near-zero downtime.

3.2 Performance & Scalability

This section defines measurable performance objectives for Eeva's RAG chatbot system and describes how the AWS architecture scales efficiently under realistic workloads.
All metrics are based on controlled benchmarking using 1 000 monthly queries, 512 MB Lambda memory, and 1.7 MB FAISS index in S3.

3.2.1 Quantitative Performance Metrics

Metric	Target / Observed Value	Measurement Method	Interpretation
Cold-start Latency (P95)	≤ 400 ms (Observed ≈ 320 ms)	CloudWatch Init Duration logs	Lambda container provisioning after 15 min idle. Mitigated by 512 MB memory and layer pre-load.
Warm Invocation Latency (P95)	≤ 2.5 s (Observed ≈ 1.9 s)	End-to-end timing (`curl` + CloudWatch)	Full RAG pipeline (user→response). Dominated by Bedrock Haiku inference (~1.2 s).
FAISS Search Time	≤ 50 ms (Observed ≈ 28 ms @ 1.7 MB index)	Lambda custom metric `faiss_latency_ms`	Linear scaling O(N); stays < 100 ms even for 10× larger index.
Bedrock Generation Latency	≤ 1.5 s (Observed 1.1–1.3 s avg)	`bedrock.invoke_model()` timing in Lambda	Claude Haiku optimized for fast throughput < 1.5 s for < 1 K tokens.
Throughput (per Lambda)	≈ 30 requests / sec burst capacity	Artillery load test 5–30 req/s for 60 s	No throttles; Lambda scales horizontally (auto concurrency).

API Gateway P95 Latency	≤ 120 ms (Observed ≈ 85 ms)	API Gateway metrics `Latency`	Includes TLS handshake and payload serialization.
Total End-to-End Latency (P95)	≤ 2.5 s (Observed ≈ 2.1 s)	Combined browser timer + CloudWatch Duration	Satisfies interactive UX threshold for portfolio site.
Error Rate (5xx)	≤ 1 % (Observed 0 %)	API Gateway + Lambda logs	All invocations completed successfully in tests.
Cost per Query	≤ $0.01 (Observed $0.0023)	AWS Cost Explorer tag `Project=Eeva`	Dominated by Bedrock token charges; <$3/mo @ 1 000 queries.
Concurrency Scalability	Up to 1 000 parallel invocations (default limit)	Artillery spike test	Lambda auto-scales linearly with incoming traffic.
Availability (Uptime)	≥ 99.9 %	API Gateway Success Rate metric	Serverless design eliminates infrastructure downtime.
Memory Usage	< 300 MB per invocation	Lambda `MemoryUsedMB` metric	512 MB allocation leaves 40 % headroom.

Index Load Time	≤ 250 ms (Observed ≈ 210 ms)	Lambda timing between S3 download and `faiss.read_index()`	Efficient due to small 1.7 MB index; cached for warm invocations.

3.2.2 Scalability Characteristics

- **Lambda Auto-Scaling:**
  - Scales horizontally to handle each invocation independently.
  - Default burst limit = 1 000 concurrent executions; can request increase if traffic > 1 K req/s.
  - Each Lambda executes entire pipeline (start → response) with no shared state.
- **API Gateway Elasticity:**
  - Auto-scales with traffic and caches DNS/TLS sessions for repeat clients.
  - Integrated with CloudFront CDN for low latency global access.
- **Bedrock Managed Scaling:**
  - Model endpoints are fully managed by AWS; no instance provisioning required.
  - Handles tens of requests per second without warm-up.
- **FAISS Scalability:**
  - Retrieval time O(N) but index size currently < 2 MB; even 100× growth ≈ 200 MB → < 300 ms search time.
  - Can migrate to OpenSearch Serverless if > 1 M vectors.

3.2.3 Optimization Techniques

Layer	Optimization Applied	Benefit

Lambda	Pre-loads FAISS binary and Titan client in init phase	Reduces cold-start overhead ~150 ms
Bedrock	Use Claude Haiku (low latency tier) with max_tokens 350	1.5× faster than Sonnet at ⅙ cost
FAISS	Compact IndexFlatIP (no quantization)	Fast CPU-based search with tiny footprint
S3	Store index and docs in same region (us-west-2)	Avoids cross-region latency and egress
Amplify + CloudFront	CDN edge caching of static assets	TTFB < 100 ms worldwide
API Gateway	Enable keep-alive HTTP/2 connections	Reduces handshake latency ~80 ms
CloudWatch Alarms	Triggers if latency > 3 s or error rate > 5 %	Early anomaly detection

---

3.2.4 Scaling Projection

Load Scenario	Queries / Month	Expected Total Latency (P95)	Monthly Cost Estimate (USD)	Notes
Baseline	1 000	2.1 s	≈ $2.7	Current deployment
Moderate Growth	5 000	2.3 s	≈ $13.5	Linear scale; same architecture
High Traffic	25 000	2.5 s	≈ $67	Lambda + Bedrock auto-scale; no re-architecture needed
Enterprise Mode	100 000	2.8 s	≈ $270	Recommend FAISS→OpenSearch Serverless migration

---

Summary:
Eeva's current configuration demonstrates **P95 latency ≈ 2.1 s**, **FAISS retrieval < 50 ms**, and **cost ≈ $0.0023 per query**.
The architecture scales linearly with traffic, maintains sub-3 s response times up to ≈ 25 000 queries per month, and can expand seamlessly through AWS-managed concurrency without any infrastructure change.

3.3 Testability

3.3.1 Unit Tests

Unit testing focuses on validating **small, isolated functions** without touching live AWS services. We use **pytest** with **moto/boto3 stubs** and local doubles for FAISS/Bedrock to guarantee fast, deterministic runs.

Test Scope & Components

- **Preprocessing/Chunking**
    - `extract_text(pdf|docx|txt)`: returns normalized UTF-8 string; strips headers/footers.
    - `chunk_text(text, size, overlap)`: produces stable chunk boundaries; no empty chunks; overlap respected.
    - `dedupe_chunks(chunks)`: removes near-duplicates (Jaccard/cosine threshold).
- **Embedding Adapter (Bedrock client wrapper)**
    - `embed_texts(list[str])`: validates batching, retries, and dimension; masks PII in logs.
    - Error paths: throttling → backoff; malformed responses → safe failure.
- **Index Layer (FAISS adapter)**
    - `build_index(matrix)`: enforces dim=1536; raises on mismatched shapes.
    - `search_index(index, query, k)`: returns deterministic ids/scores; bounds k to length.
- **Prompt Builder**
    - `build_prompt(user_query, contexts)`: inserts guards, trims to token budget, preserves citations order, forbids empty CONTEXT.
    - `sanitize_output(text)`: strips HTML/script, normalizes whitespace.
- **Guardrails**
    - `is_off_domain(query)`: keyword/regex map; case/whitespace insensitive.
    - `should_refuse(retrieval_scores)`: refuses if max<0.3 similarity.
    - `enforce_length(query)`: truncates >1000 chars; returns notice flag.
- **Lambda Handler (pure core)**
    - `validate_request(body)`: schema checks; CORS origin allowlist.

    - `format_response(answer, sources, metrics)`: JSON contract; latency fields present.

Test Strategy (pytest)

Fixtures

- `fake_chunks_small`: 5 realistic chunks (UAV, lung CT, NLI, Spotify analytics, resume).
- `emb_dim`: 1536.
- `fake_vectors`: small `np.ndarray` with seeded random for reproducibility.
- `bedrock_stub`: returns fixed 1536-dim vectors / canned generations.
- `faiss_stub`: in-memory index with deterministic search.
- `s3_stub`: moto bucket for `/docs`, `/embeddings`, `/index`.

Examples (concise)

- **Chunking**
    - `test_chunk_text_respects_overlap()`: assert `chunks[i].end - chunks[i+1].start <= overlap`.
    - `test_chunk_text_no_empty_or_tiny_chunks()`: min length threshold enforced.
    - `test_dedupe_chunks_removes_near_duplicates()`: duplicates reduced ≥ expected.
- **Embedding**
    - `test_embed_texts_returns_correct_shape()`: `(n, 1536)`.
    - `test_embed_texts_handles_throttle_with_backoff()`: retries 1→2→4s mocked.
    - `test_embed_texts_masks_pii_in_logs()`: email/phone redacted in log strings.
- **FAISS Adapter**
    - `test_build_index_dimension_mismatch_raises()`.
    - `test_search_index_returns_k_results_sorted_by_score()`.
    - `test_search_index_bounds_k_to_available_vectors()`.

- **Prompt Builder**
  - `test_build_prompt_includes_all_contexts_in_order()`.
  - `test_build_prompt_trims_to_token_budget_without_cutting_sentence_midway()`.
  - `test_sanitize_output_escapes_html_and_strips_scripts()`.

- **Guardrails**
  - `test_is_off_domain_blocks_politics_and_personal_questions()`.
  - `test_should_refuse_when_similarity_below_threshold()`.
  - `test_enforce_length_truncates_and_sets_flag()`.

- **Lambda Core**
  - `test_validate_request_rejects_missing_query_field()`.
  - `test_format_response_contract()`: `answer`, `source_documents[]`, `latency_ms` present; types correct.

Mocks & Stubs

- **Bedrock:** monkeypatch client to return deterministic embeddings and a fixed completion payload. Assert `modelId`, `temperature`, and `maxTokens` passed correctly.

- **S3:** use moto to simulate buckets/keys; verify reads/writes to `/index/faiss.index`, `/embeddings/*.npy`.
- **FAISS:** wrap `faiss.IndexFlatIP` behind an interface; provide a pure-Python stub for unit tests.

Negative & Edge Cases

- Empty document → zero chunks → graceful "no data" path.
- Non-ASCII text → normalized UTF-8; no exceptions.
- Oversized query → truncated + warning.
- Corrupted index file → raises `IndexLoadError` (caught in integration, not unit).
- Similarity ties → stable ordering via seeded sort.

Determinism & Coverage

- Seed all randomness (`np.random.seed(42)`); assert identical results across runs.
- **Coverage Targets:**
  - `core utils` ≥ 95%
  - `guardrails` ≥ 95%
  - `adapters (embed/index/prompt)` ≥ 90%
  - `lambda core` ≥ 85% (business logic only; I/O mocked)

3.3.2 Integration Tests

Integration testing validates **end-to-end interoperability** across AWS services and internal modules.
While unit tests confirm internal correctness, integration tests ensure the **complete RAG pipeline** behaves predictably when real AWS resources, IAM roles, and event payloads interact.

3.3.2.1 Test Objectives

- Verify seamless data flow from **user request** → **API Gateway** → **Lambda** → **Bedrock** → **FAISS** → **S3 metadata** → **frontend response**.
- Confirm IAM permissions, environment variables, and network connectivity between services.
- Validate that responses remain factual and properly grounded in retrieved knowledge.
- Ensure system gracefully handles AWS service errors or latency spikes.

3.3.2.2 Test Environment

Environment	Configuration
AWS Region	`us-west-2` (same as production)
Test Stack	Separate Amplify app + test S3 bucket `shubhankargoje-kb-staging`
Lambda Alias	`vectorSearchLambda:staging`

API Gateway Stage	`/staging`
Bedrock Models	Titan Embed Text v2, Titan Text G1
Secrets	Managed in AWS Secrets Manager (`/eeva/staging`)
Logging	CloudWatch group `/aws/lambda/vectorSearchLambda-staging`

3.3.2.3 Test Scenarios

A. End-to-End Happy Path

Goal: Validate complete pipeline for a known question.

Step	Expected Behavior	Verification
1. POST `/chatbot/query` with `"Explain UAV PID control"`	API Gateway invokes Lambda	200 OK
2. Lambda loads FAISS index from S3	FAISS index present, no error	Log entry “Index loaded”
3. Embeddings via Titan v2	1536-dim vector returned	Log embedding size
4. FAISS retrieval	≥ 3 chunks returned (sim > 0.3)	Payload includes sources
5. Bedrock generation	Text ≤ 350 tokens, factual	Response JSON contains expected keywords
6. Response to client	`answer`, `source_documents`, `latency_ms` present	Schema validated

Success criteria: Total latency < 2.5 s, no 5xx errors.

B. Guardrail Interaction

1. Send off-domain prompt ("Tell me about politics").
   → Response should contain refusal template, no Bedrock call logged.
2. Send oversized input (>1000 chars).
   → Lambda truncates query and returns warning flag in JSON.
3. Send query with no retrievable context.
   → Response = `"I couldn't find relevant information…"`; no generation attempt.

C. Fault-Injection Tests

Fault Type	Injection Method	Expected Behavior
S3 unavailable	Temporarily block S3 IAM permission	Lambda retries → fails gracefully with error message
Bedrock throttling	Mock `429` response for first call	Exponential backoff retry → success on retry 2
Invalid index	Replace FAISS file with corrupted bytes	Lambda detects checksum mismatch → loads cached copy from `/tmp`
High latency	Inject 2 s delay in Bedrock call	CloudWatch logs record `latency_ms` increase but still < timeout

---

D. Security and Permissions

- **IAM Boundary Test:** Use temporary token lacking `bedrock:InvokeModel`; expect `AccessDeniedException` caught → handled.
- **CORS Validation:** Simulate requests from unauthorized origin → 403 response.
- **Secret Access Test:** Verify Lambda reads only permitted Secrets Manager key.

E. Schema and Response Validation

Each integration response is validated against a strict JSON Schema:

```
{
  "type": "object",
  "required": ["answer", "source_documents", "latency_ms"],
  "properties": {
    "answer": {"type": "string", "minLength": 10},
    "source_documents": {"type": "array", "items": {"type":
"string"}},
    "latency_ms": {"type": "number", "maximum": 3000}
  }
}
```

3.3.2.4 Execution and Automation

- Tests written in **pytest-bdd** for clear Given/When/Then narratives.
- Triggered via **GitHub Actions** or **AWS CodeBuild** nightly using staging stack.
- Parallel execution (5 threads) with pytest-xdist.
- Results published to CloudWatch Dashboard `EevaIntegrationMetrics`.

Command Example

```
pytest tests/integration --env=staging --maxfail=1 -q
```

3.3.2.5 Success Criteria and Thresholds

Category	Target	Measurement Method
Pipeline success rate	≥ 98 % of requests	Pass count / total
API latency P95	≤ 2.5 s	CloudWatch Latency

Bedrock call success	≥ 99 %	Invocation logs
Retrieval accuracy	≥ 85 %	Manual relevance review of top-k
Guardrail trigger accuracy	≥ 95 %	Count of refusal responses / invalid inputs

---

3.3.2.6 Regression Baseline

- Maintain **golden test set** of 20 canonical queries; compare new responses to previous answers via semantic similarity (≥ 0.9 threshold).
- Differences logged as potential drift; manual review required before promoting index or model version.

Outcome:
Integration testing certifies that all AWS components interact seamlessly under real network and permission conditions.
It provides confidence that Eeva's **end-to-end RAG pipeline** remains functionally correct, secure, and stable during continuous deployment and model updates.

3.3.3 Performance Testing

Performance testing evaluates how efficiently the **Serverless AWS Bedrock RAG Chatbot** responds under varying loads, network conditions, and data sizes.
Because the system is entirely serverless, tests target **cold-start latency**, **steady-state throughput**, **retrieval efficiency**, and **cost per query**.

The goal is not to reach enterprise throughput but to confirm that the architecture scales predictably and remains within cost/performance targets.

3.3.3.1 Objectives

- Measure real-world **end-to-end latency** (frontend → API Gateway → Lambda → Bedrock → response).
- Evaluate **Lambda cold start** vs. warm start behavior.
- Assess **FAISS retrieval time** for different knowledge base sizes.
- Verify **Bedrock inference latency** and stability under small concurrency bursts.
- Calculate **cost per 100 queries** from CloudWatch + Cost Explorer.

3.3.3.2 Tools Required

Tool	Purpose
AWS CloudWatch Logs & Metrics	Collect runtime latency, init duration, and memory usage.
AWS X-Ray (optional)	Trace end-to-end request path through API Gateway + Lambda.
Artillery *(or Locust)*	Generate load and concurrent request simulations.
curl / Postman	Manual latency and payload schema validation.
AWS Cost Explorer	Track actual request-level spend.
Python Timing Script	Simple local benchmarking via requests library for fine-grained timing.

3.3.3.3 Test Preparation

1. **Create a dedicated staging environment**
    - Deploy Lambda and API Gateway alias: `vectorSearchLambda:staging`.

    - Use same FAISS index as production (`faiss.index`) for realistic performance.
    - Enable detailed metrics in Lambda:
      → Configuration → Monitoring → *Enhanced monitoring (1-second granularity)*.
2. **Warm up the Lambda**
    - Run 5–10 queries manually to ensure cache warming and cold start baseline captured.
3. **Collect baseline parameters**
    - From CloudWatch:
        - `Duration (ms)` — execution time
        - `Init Duration (ms)` — cold start
        - `MemoryUsedMB`
    - Store in spreadsheet for reference.

3.3.3.4 Test 1 – Single Request Latency

Purpose: Establish baseline latency for a single user query.
Steps:

Open a terminal and run:

```
 curl -X POST
https://<api_id>.execute-api.us-west-2.amazonaws.com/prod/chatbot/quer
y \
     -H "Content-Type: application/json" \
     -d '{"question": "Explain UAV PID control system."}' -w "\nTime:
%{time_total}s\n"
```

1.
2. Record total response time and HTTP status.

3. In CloudWatch, note Lambda's reported `Duration` and `Init Duration`.
4. Repeat 5 times spaced 1 minute apart:
   - First run captures **cold start**.
   - Later runs capture **warm start**.
5. Calculate mean and P95 response time from these samples.

3.3.3.5 Test 2 – Concurrent Request Handling

Purpose: Validate scaling behavior for multiple users hitting the API simultaneously.
Steps:

Install Artillery:

```
npm install -g artillery
```

1.

Create `artillery.yml`:

```
config:
  target: "https://<api_id>.execute-api.us-west-2.amazonaws.com/prod"
  phases:
    - duration: 60
      arrivalRate: 5      # 5 new requests per second
scenarios:
  - flow:
      - post:
          url: "/chatbot/query"
          json:
            question: "Tell me about image segmentation project."
```

2.

Run the test:

```
artillery run artillery.yml
```

3.
4. Review Artillery summary:
   - Request count, error count, mean latency, P95 latency.
5. Cross-check CloudWatch metrics for:
   - `ConcurrentExecutions`
   - `Throttles`
   - `5XXError`
6. Confirm no throttling or timeouts occurred.

3.3.3.6 Test 3 – FAISS Retrieval Benchmark

Purpose: Measure how FAISS retrieval scales with index size.
Steps:

1. Locally (or via Lambda test event), time vector search with different index sizes:
   - 1,000 embeddings
   - 5,000 embeddings
   - 10,000 embeddings

Add timing code around FAISS calls:

```
import time
start = time.time()
distances, indices = index.search(query_vector, k=5)
print("Search time:", time.time() - start)
```

2. Log retrieval times for each dataset size; expect linear or sub-linear growth.
3. Store metrics in `/logs/faiss_benchmark.csv` in S3 for future comparison.

3.3.3.7 Test 4 – Bedrock Inference Latency

Purpose: Quantify time spent in Bedrock generation vs. Lambda overhead.
Steps:

Wrap Bedrock API call in timing block inside staging Lambda:

```
 start = time.time()
response = bedrock.invoke_model(...)
print("Bedrock latency:", time.time() - start)
```

1.
2. Compare CloudWatch `Duration` minus Bedrock latency to compute internal Lambda overhead.
3. Run 20 queries with diverse lengths to estimate mean inference time.
4. Plot in spreadsheet or use CloudWatch custom metric `BedrockLatency`.

3.3.3.8 Test 5 – Cold Start Frequency and Mitigation

Purpose: Understand how often cold starts occur and their cost.
Steps:

1. Deploy Lambda with 512 MB memory.
2. Run one query every 15 minutes for 24 hours using cron or small Python script.

Extract `Init Duration` logs:

```
 aws logs filter-log-events --log-group-name
"/aws/lambda/vectorSearchLambda" \
  --filter-pattern "Init Duration"
```

3.
4. Count occurrences per 24h → derive cold start frequency.

5. Optionally repeat with higher memory (1 GB) and compare difference.

6. Enable **Provisioned Concurrency** temporarily to compare steady vs. cold performance (optional, paid).

3.3.3.9 Test 6 – Cost per Query Estimation

Purpose: Validate cost efficiency of serverless design.
Steps:

1. Tag all components with `Project=Eeva-Test`.

2. Run 100 queries (mix of short and long).

3. After 24 hours, open **AWS Cost Explorer** → **Filter by Tag = Eeva-Test**.

4. Export CSV showing:

    - `Amazon Bedrock – Inference Requests`

    - `Lambda – Request Duration`

    - `API Gateway – Requests`

5. Compute:
   Cost per query=Total billed cost100\text{Cost per query} = \frac{\text{Total billed cost}}{100}Cost per query=100Total billed cost
6. Compare to target threshold (≤ $0.01 per request).

7. Archive results in `/logs/performance_cost_report.csv`.

3.3.3.10 Test 7 – Error Recovery and Retry

Purpose: Confirm retry logic works under transient errors.
Steps:

1. Modify test Lambda temporarily to randomly raise an exception on 10 % of Bedrock calls.
2. Observe CloudWatch logs for retry attempts (1→2→4 s backoff).
3. Verify successful retry completion in ≤ 2 attempts.
4. Restore production handler.

3.3.3.11 Metrics to Capture

Metric	Source	Notes
Total End-to-End Latency	Artillery / CloudWatch	From user request to response
Lambda Duration	CloudWatch	Function execution time
Init Duration	CloudWatch	Cold start overhead
FAISS Search Time	Custom log metric	Python timing block
Bedrock Latency	Custom log metric	Subcomponent timing
Error Rate	Artillery summary	4xx + 5xx responses
Cost per Query	Cost Explorer	Actual AWS billing data

3.3.3.12 Analysis & Reporting

- Export Artillery results (`artillery report`) for visualization.

Use CloudWatch Insights query:

```
fields @timestamp, Duration, InitDuration, MemoryUsedMB
| stats avg(Duration), pct(Duration,95), max(InitDuration) by bin(5m)
```

- Correlate spikes with Bedrock latency to distinguish model vs. infrastructure causes.
- Document observations in `/logs/performance_summary.md` with plots or screenshots.

Outcome:
Following these steps yields empirically verifiable performance data for Eeva's full AWS pipeline.
Results confirm whether latency, concurrency, and cost meet design expectations — all without modifying production traffic or assuming artificial values.

3.4 Maintainability

Maintainability defines how easily the **Serverless AWS Bedrock RAG Chatbot** can be modified, extended, or repaired throughout its lifecycle.
Because the architecture is 100 % serverless and modular, maintenance focuses on **code organization, version control, infrastructure management, and operational observability**, not hardware upkeep.

3.4.1 Design Principles

1. **Single-Responsibility Functions**
    - Each Lambda performs one core task: vector search + Bedrock inference.
    - Ingestion and embedding pipelines are separate Python scripts, ensuring failures in one area never affect live query handling.
2. **Statelessness**
    - No persistent runtime variables; all state lives in S3 or metadata files.
    - Simplifies redeployments and scaling—no session cleanup needed.

3. **Infrastructure-as-Code (IaC)**

   - CloudFormation or AWS SAM template defines all resources (S3 buckets, IAM roles, API Gateway, Lambda configs).
   - Enables full environment recreation in minutes and consistent drift detection.

Clear Folder Layout

```
/src
  ├── lambda_handler.py
  ├── faiss_adapter.py
  ├── bedrock_client.py
  ├── prompt_builder.py
  ├── guardrails.py
  └── utils/
/ingestion
  ├── text_extraction.py
  ├── chunking.py
  ├── embedder.py
  └── build_faiss_index.py
/tests
  ├── unit/
  ├── integration/
  └── performance/
/infra
  ├── template.yaml        # SAM/CloudFormation
  └── parameters.json
```

4. This hierarchy allows independent updates of ingestion, retrieval, and inference logic.

3.4.2 Code Versioning and Release Management

Aspect	Practice	Tooling

Source Control	Git with trunk-based flow (`main`, `staging`, `feature/*`)	GitHub
Commit Policy	Conventional Commits for changelog automation	`commitlint`, `semantic-release`
Code Review	Required 1 approval before merge	GitHub Pull Requests
Automated Tests	Run `pytest --cov` on push	GitHub Actions / CodeBuild
Deployment	Auto-deploy on tag release to Amplify + Lambda alias `prod`	Amplify CI/CD
Rollback	Amplify keeps 5 previous builds; Lambda version aliases enable instant rollback	AWS Console / CLI

3.4.3 Dependency and Environment Management

- **Pinned Versions:**
  `requirements.txt` locks library versions (boto3, faiss-cpu, numpy).
- **Environment Variables:**
  Stored in Lambda configuration / Secrets Manager (`BEDROCK_MODEL_ID`, `S3_BUCKET`, `SIM_THRESHOLD`).
- **Virtual Environments:**
  All dev work uses `venv` or `poetry` for reproducibility.
- **Layer Reuse:**
  Common libraries (faiss, numpy) packaged into a Lambda Layer to avoid re-deployment of heavy dependencies.

3.4.4 Monitoring and Observability

1. **Centralized Logging**
   - All Lambda print/log statements routed to CloudWatch Logs with JSON format.

- `@message`, `latency_ms`, `retrieval_k`, `similarity_avg` fields indexed for Insights queries.

2. **Metrics Dashboards**
    - CloudWatch Dashboard:
        - Lambda Duration (P95)
        - Init Duration
        - API Gateway 5xx/4xx
        - Bedrock Invocation Count
        - Monthly Cost by Service

3. **Automated Alerts**
    - CloudWatch Alarms → SNS topic `eeva-alerts`
        - Latency > 3 s for 5 min
        - Error Rate > 5 %

4. **Audit Trail**
    - CloudTrail enabled for all Bedrock and IAM actions.
    - Logs retained 90 days for compliance and rollback tracing.

3.4.5 Knowledge Base Maintenance

1. **Adding New Documents**
    - Upload new files to `/docs/` prefix in S3.
    - Trigger ingestion Lambda or run `build_faiss_index.py` locally.
    - Upload new `faiss.index` and metadata to `/index/`.
    - Tag version in S3 (`index-v5.0.1`).

2. **Version Retention Policy**
    - Keep latest 3 versions; older auto-moved to Glacier.
    - Each version logged in `kb_version_manifest.json`.
3. **Validation Checks**

  - SHA-256 checksum comparison before index promotion.
  - Regression queries (20 canonical questions) executed to verify retrieval quality ≥ 85 %.

3.4.6 Documentation and Developer Onboarding

- **Architecture diagram** + flow chart kept in `/docs/architecture.png`.
- **Comment Standards:** Each function includes 1-line summary + parameter docstring in NumPy style.
- **Knowledge Transfer:** Markdown handbook `/docs/maintenance_guide.md` lists index update and deployment steps.

3.4.7 Operational Maintenance Schedule

Task	Frequency	Responsible	Notes
Review CloudWatch cost dashboard	Weekly	Owner	Check for usage spikes
Re-index knowledge base after new doc upload	Ad hoc	Owner	Run `build_faiss_index.py`
Update dependencies	Quarterly	Owner	Validate in staging before prod
Review IAM roles & rotate keys	Semi-annual	Owner	Principle of least privilege
Lambda memory tuning	Quarterly	Owner	Balance cost vs. latency

CloudWatch alarm test	Monthly	Owner	Ensure alerts fire to SNS email

---

3.4.8 Ease of Extension

- Add new modules (e.g., Spotify, Calendar) via dedicated Lambdas with independent permissions.
- Plug-in retrieval sources (OpenSearch Serverless, Bedrock Knowledge Bases) by swapping FAISS adapter.
- Add multilingual support by changing embedding model ID only.
- No refactor needed to scale to multiple knowledge bases—simply duplicate index prefixes.

Summary:
Maintainability of the Bedrock RAG Chatbot is achieved through clear code modularity, version-controlled infrastructure, automated monitoring, and well-documented operational workflows.
Every major component—code, data, and infrastructure—can be updated, replaced, or rolled back in under 15 minutes, ensuring long-term sustainability and minimal operational risk.

3.5 Usability

Usability defines how effectively and pleasantly users can interact with the **Serverless AWS Bedrock RAG Chatbot** through the portfolio website.
Although Eeva's backend is complex, the front-facing experience is intentionally **simple, responsive, and self-explanatory**.
The goal is to make interacting with an AI feel natural while preserving transparency, factual grounding, and brand coherence with *Mystical Data Temple* aesthetics.

3.5.1 Design Goals

1. **Minimal Cognitive Load** – The user should not need instructions; the interface should communicate its purpose through layout and cues.

2. **Visual Harmony** – Chat components blend seamlessly with the website's dark, surreal theme, using consistent typography, color tokens, and soft animation.
3. **Responsiveness** – Chat loads instantly on desktop, tablet, and mobile; layout adapts without hidden controls.
4. **Transparency** – Every response includes subtle context cues ("Based on project: UAV Path Planning") for traceability.
5. **Trust and Tone** – Responses remain calm, factual, and concise—avoiding over-personalization or exaggerated enthusiasm.

3.5.2 Interaction Flow

1. **Greeting State** – Eeva introduces itself ("Hi, I'm Eeva — I can tell you about Shubhankar's projects.").
2. **Prompt Suggestions** – 4 rotating default questions appear, drawn from a list of 20 introspective project topics.
3. **Query Entry** – User types or clicks a suggestion; input automatically focuses and supports `Enter` to submit.
4. **Processing Feedback** – A "Eeva is thinking…" loader appears with animated dots, masking Lambda latency.
5. **Response Display** – Answer appears line-by-line; project source and latency shown unobtrusively below.
6. **Continuation Options** – Buttons: *Ask another*, *Learn about another project*, *Clear chat*.
7. **Reset Behavior** – After 15 exchanges, Eeva politely resets context: "Let's start fresh."

3.5.3 Accessibility & Inclusivity

Aspect	Implementation

Keyboard Navigation	Fully operable using Tab, Enter, and Shift+Tab.
Color Contrast	Meets WCAG AA for dark mode; white text ≥ 4.5:1 contrast ratio.
ARIA Labels	`<input>` and `<button>` components labeled for screen readers.
Font Scaling	Text scales with browser zoom up to 200 %.
Motion Sensitivity	Animations < 200 ms; respect `prefers-reduced-motion`.
Language Clarity	Plain English; avoids jargon unless context explicitly asks.

3.5.4 Feedback & Error Handling

- **API Failures:**
  Graceful message — "Eeva had trouble connecting. Please try again in a moment."
  Automatically retries once if the issue is transient.
- **Off-Domain Questions:**
  Friendly redirect — "I can only answer about Shubhankar's work. Try asking about one of his projects!"
- **Empty Input:**
  Disabled send button until text detected.
- **Validation Errors:**
  Clear inline feedback ("Message too long – please shorten it to 1000 characters.").
- **User Feedback Loop:**
  Optional thumbs-up / thumbs-down stored in Google Analytics event for satisfaction tracking.

3.5.5 Evaluation Methods

Method	Goal	How to Execute
Heuristic Review	Expert UX audit vs. Nielsen heuristics (consistency, error prevention, visibility of status).	Conduct quarterly self-review with a checklist.
Session Recording (anonymized)	Observe flow friction (scrolling, missed prompts).	Enable GA4 event tracking for clicks, scrolls, and input length.
A/B Copy Test	Compare greeting styles (“Ask me about projects” vs. “What would you like to learn?”).	Alternate every 20 sessions; record engagement rate.
First-Time User Test	Measure learnability.	Ask 3–5 new users to find info on a project without guidance; time to first correct response.
Accessibility Audit	Ensure compliance with WCAG 2.1 AA.	Use Chrome Lighthouse Accessibility score ≥ 90.

3.5.6 Measurable UX KPIs

Metric	Target	Measurement Method
First Response Latency (perceived)	≤ 2 s	Client timer from submit → text render.

Task Success Rate	≥ 90 % of users find relevant info within 2 questions	Usability test sessions.
Engagement Rate	≥ 60 % of sessions click a suggested question	GA4 events.
Satisfaction Score	≥ 4 / 5 (survey)	Post-interaction rating popup (optional).
Error Message Frequency	≤ 2 % of requests	GA4 + CloudWatch logs.
Accessibility Score	≥ 90 Lighthouse	Automated audit monthly.

3.5.7 Maintainable UX Implementation

- **Componentization:**
  Chat window built as a reusable React component (`<ChatWidget />`) that can be embedded on other site pages.
- **Styling:**
  Tailwind CSS variables linked to global design tokens (`--accent`, `--surface-dark`, `--text-primary`).
- **Localization Ready:**
  All strings in `/locales/en.json` for future language expansion.
- **Analytics Integration:**
  `gtag('event', 'chat_question')` fires on each submission; GA4 dashboard visualizes usage funnel.

3.5.8 Human Factors and Ethical Considerations

- Eeva never claims autonomy or human identity.
- Responds transparently if unsure (“I don’t have that detail.”).
- Avoids gendered or emotional language—stays professional and courteous.
- Every interaction reaffirms Shubhankar’s authorship and privacy respect.

Summary:
The chatbot’s usability stems from **clarity, predictability, and empathy**.
Through minimal design, responsive layout, consistent feedback, and ongoing measurement, Eeva provides a professional yet approachable experience that reinforces trust in Shubhankar’s technical brand.

3.6 Security / Privacy

Security and privacy are foundational to the **Serverless AWS Bedrock RAG Chatbot**, ensuring that all data flows, model interactions, and user sessions remain **confidential, integrity-protected, and auditable**.
The design follows **AWS Well-Architected Framework — Security Pillar** and **Privacy-by-Design** principles, enforcing least privilege, encryption everywhere, and zero persistent user data storage.

3.6.1 Security Objectives

1. Prevent unauthorized access to AWS resources.
2. Guarantee encryption of all data in transit and at rest.
3. Maintain auditability and traceability of every API and model invocation.
4. Ensure user interactions remain private — no personal data is stored, logged, or reused.
5. Limit the chatbot’s operational surface to **AWS-only** services (no third-party dependencies).

3.6.2 IAM and Access Control

Principle of Least Privilege

- Each service (Lambda, API Gateway, Bedrock, S3, CloudWatch) has a **dedicated IAM role** with scoped permissions.
- No wildcard `*` permissions are used; all policies restrict allowed actions to explicit ARNs.

IAM Role Design

Role	Allowed Actions	Notes
`LambdaExecutionRole`	`s3:GetObject`, `s3:PutObject`, `bedrock:InvokeModel`, `logs:*`	Invoked by API Gateway only
`BedrockAccessRole`	`bedrock:InvokeModel`, `bedrock:InvokeModelWithResponseStream`	Isolated to embedding/generation functions
`S3IngestionRole`	`s3:PutObject`, `s3:ListBucket`, `s3:GetObjectVersion`	Used during re-index or upload scripts
`AmplifyHostingRole`	`amplify:*`, `cloudfront:CreateInvalidation`	Frontend-only; no backend access
`MonitoringRole`	`cloudwatch:GetMetricData`, `cloudwatch:DescribeAlarms`	Read-only observability

Identity Controls

- Root account never used for development or API access.
- Administrator access limited to single IAM user with MFA enforced.
- IAM users grouped into: `Admin`, `Developer`, `Viewer`.
- Access keys rotated quarterly; CloudTrail monitors usage anomalies.

3.6.3 Data Security

Data in Transit

- All communication between frontend, API Gateway, and Lambda uses **HTTPS/TLS 1.2+**.
- CORS restricted to the verified domain `https://shubhankargoje.com`.
- API Gateway enforces strict **content-type headers** to reject malformed payloads.

Data at Rest

- All S3 objects (`/docs/`, `/embeddings/`, `/index/`, `/logs/`) encrypted using **AES-256 (SSE-S3)**.
- Bedrock service automatically encrypts inference data within AWS VPC; no data leaves AWS-managed boundaries.
- Lambda temporary storage (`/tmp/`) cleared after every invocation.

Data Lifecycle

- No user messages are persisted.
- Temporary embeddings, logs, and outputs live only in memory during Lambda execution.
- S3 versioning used for document integrity, not user content.
- Access logs retained 90 days; no personal identifiers stored.

3.6.4 Application Security

API Gateway Security Controls

- **Authentication Layer:**
    - Private endpoints protected by IAM authentication for backend scripts (e.g., ingestion Lambda).

    - Public chatbot endpoint validated by domain origin check.
- **Rate Limiting & Throttling:**
    - Default quota: 10 requests/second, burst limit 5.
    - Prevents denial-of-service (DoS) and excessive free-tier usage.
- **Input Sanitization:**
    - Lambda strips HTML, escape sequences, and JSON injection attempts.
    - Regex-based filters remove profanity or malicious tokens.

Lambda Execution Environment

- Ephemeral runtime per invocation; no reuse of sensitive data between sessions.
- Read-only execution package, immutable during runtime.
- Environment variables fetched securely via **AWS Secrets Manager**.

Secrets Management

- API tokens (e.g., for Spotify integration) stored in **AWS Secrets Manager**, not code.
- Secrets rotated automatically every 90 days.
- Lambda retrieves values at runtime via short-lived session tokens (STS).

3.6.5 Privacy-by-Design

Privacy Principle	Implementation in Chatbot

Data Minimization	Chatbot does not collect names, emails, or identifiers. Only question text is transiently processed.
Purpose Limitation	User queries processed solely for generating AI responses; no analytics beyond anonymized event counts.
User Transparency	Footer note: “Eeva answers questions only about Shubhankar Goje’s work; no personal data is stored.”
Retention Limits	Session data held in memory only; no message logs in CloudWatch.
User Control	User can reset chat context at any time.
Third-Party Sharing	None — all data resides and is processed within AWS.

3.6.6 Monitoring and Incident Response

- **CloudTrail** tracks all administrative and model invocation activity.
- **GuardDuty** enabled for anomaly detection (unauthorized API usage, IAM misuse).
- **CloudWatch Alarms** trigger SNS alerts if:
  - Unusual error rate (>5 %)
  - Unauthorized access attempt logged
  - Large data egress detected
- **Incident Response Runbook:**
  - Step 1: Revoke affected IAM credentials
  - Step 2: Rotate all Secrets Manager values

- Step 3: Inspect CloudTrail for action trace
- Step 4: Re-deploy from latest clean build
- Step 5: Document incident in `/security/incidents.md`

3.6.7 Compliance and Risk Mitigation

- **AWS Shared Responsibility Model:**
  - AWS secures the infrastructure; Shubhankar secures application-level IAM, data classification, and encryption.
- **Encryption Standards:**
  - AES-256 for data at rest, TLS 1.2+ for transit, IAM session tokens for authentication.
- **Logging and Evidence:**
  - CloudWatch and CloudTrail provide auditable evidence of compliance with internal security policies.
- **Vendor Lock-In Mitigation:**
  - All code uses standard boto3 SDK calls; system portable to any AWS account or Bedrock-compatible region.

3.6.8 Security Testing Procedures

Test Type	Description	Frequency	Tool
IAM Policy Audit	Validate no excessive permissions (`*`) in roles.	Quarterly	IAM Access Analyzer

Penetration Test (lightweight)	Manual black-box test on public API endpoint.	Biannual	OWASP ZAP / Postman
S3 Bucket Policy Review	Confirm no public object access except Amplify assets.	Monthly	AWS CLI / IAM Analyzer
Secrets Leak Scan	Ensure no credentials committed to Git.	Each PR	`trufflehog` / `git-secrets`
SSL / TLS Verification	Confirm HTTPS enforcement and cert validity.	Continuous	CloudFront + ACM

3.6.9 Security KPIs

Metric	Target	Monitored Through
IAM Policy Violations	0	IAM Access Analyzer
S3 Public Access Findings	0	AWS Security Hub
API Unauthorized Calls	≤ 1/month (manual test)	CloudTrail Logs
Secrets Age > 90 days	0	Secrets Manager rotation metrics
Security Incidents	0 confirmed	Incident Reports

Summary:
The chatbot's security framework enforces **tight IAM boundaries, full encryption, no user data retention, and continuous auditing** — achieving end-to-end trust and compliance.
By confining all inference, storage, and access control to AWS-managed services, the system maintains an enterprise-grade security posture at personal scale while remaining transparent and privacy-preserving.

Section 4: Operational Cost Estimates

Amazon Bedrock (Anthropic Claude Haiku 4.5)

- **On-demand inference pricing:** Claude Haiku 4.5 is charged at $1 per million input tokens and $5 per million output tokens. Equivalently, that is $0.001 per 1,000 input tokens and $0.005 per 1,000 output tokens.

- **Per-query cost:** With ~500 input + 350 output tokens per query, each query costs roughly *(500/1,000)*$0.001 + *(350/1,000)*$0.005 ≈ $0.0005 + $0.00175 = **$0.00225**. At 1,000 queries/month, total Bedrock inference cost ≈ $2.25/month.

- **Batch inference:** If we batch-process prompts (e.g. knowledge-base embedding), Bedrock offers ~50% off on-demand rates. For Haiku 4.5 this would be $0.0005 per 1K input and $0.0025 per 1K output. (At our low volume, on-demand mode is simpler.)

- **Provisioned throughput:** Reserved model units cost ~$39.60/hour (1-month commitment) per Claude model unit. We do **not** need provisioned units for only ~1,000 queries/month (on-demand capacity suffices).

AWS Lambda (Serverless Backend)

- **Invocation pricing:** AWS Lambda charges $0.20 per 1M requests, with 1M free tier per month. At 1,000 queries (invocations) this is $0.0002 (well within free tier). Duration is $0.00001667 per GB-second. For example, a 128 MB function running 0.1s uses 0.0128 GB-sec, costing ~$0.00000021 per invocation.

- **Monthly compute cost:** 1,000 invocations at ~0.00000021 each = ~$0.00021. Combined with $0.0002 for the requests fee, total Lambda cost ≈ **$0.00041** per month – effectively negligible. (AWS's free tier covers 1M req and 400k GB-sec.)

- **API Gateway (if used):** HTTP API Gateway costs $3.50 per 1M calls. For 1,000 calls/month this is ~$0.0035. Thus, even if routing through API Gateway, cost is only a few thousandths of a dollar.

AWS Amplify (Frontend Hosting)

- **Hosting & bandwidth:** Amplify's pay-as-you-go plan includes up to 5 GB of CDN storage and 15 GB of data transfer free each month. Our <30 user visits likely transfer <0.1 GB/month, far below free limits. Above free tier, Amplify charges $0.023/GB-month of storage and $0.15/GB served. At <0.1 GB this is ~$0.015 – effectively zero.

- **Requests:** Server-side rendering (SSR) requests are free for the first 500k requests/month ($0.30 per million beyond). Our usage is negligible here.

- **Build & deploy:** Amplify provides 1,000 free build minutes/month and $0.01/minute beyond. Occasional frontend builds (e.g. initial deployment, few updates) will use <<1,000 minutes, so build costs are $0.

Data Storage & Knowledge-Base Updates

- **S3 Storage:** Amazon S3 Standard costs $0.023 per GB-month for the first 50 TB. Our 1.7 MB knowledge-base index is only 0.0017 GB, costing ~$0.00004 per month (≈$0.0005/year) – effectively zero. GET/PUT requests are $0.005 per 1,000 in S3; with a handful of accesses this is negligible.

- **Knowledge-base embedding:** If we re-embed documents quarterly, the token costs are trivial. For example, using an embeddings model (e.g. Cohere) is ~$0.0001 per 1K input tokens (10K tokens → $0.001). Even processing 100K tokens (a very large batch) costs only ~$0.01.

- **CloudWatch Logs & Monitoring:** Lambda/Amplify log output will be minimal at ~1000 executions/month. CloudWatch allows ~5 GB log storage free. Even if ~1 MB of logs is generated monthly, this incurs $0.00 to $0.01 – negligible.

Summary: The dominant cost is Bedrock inference at about $2–3 per month for 1,000 chat queries. All other AWS services (Lambda, API Gateway, Amplify, S3) incur near-zero cost at this scale due to free tiers and low usage