

Compliance Questionnaire Assistant

AI-Powered Web App Enabling Quick Completion of Security
Questionnaires using RAG

David O'Connor

Higher Diploma in Computer Science, SETU Waterford

Student Number: 08566496

Supervisor: Richard Lacey

Declaration of Authenticity

I declare that the work which follows is my own, and that any quotations from any sources (e.g., books, journals, the internet) are clearly identified as such by the use of 'single quotation marks', for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by (date, author) in the text and a fuller citation is the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Student: David O'Connor	Initialed: DOC	Date: 27/03/2026
-------------------------	----------------	------------------

Acknowledgements

Thank you to my family, who have always supported me, particularly over the course of this degree.

Thank you to all the lecturers and people involved with the Higher Diploma. It has been a fantastic experience and I have learnt a lot over the past two years.

Thank you to my supervisor, Richard Lacey, for his guidance during the project.

Last but not least, thank you to everyone at ServisBOT for the project idea and the time to complete it. Thank you in particular to Barry, Steve, and Shane for their oversight, Dara for his invaluable advice, and TJ for his time reviewing my PRs. Thank you also to Steve for his advice. I am very grateful for the experience and I hope the project proves to be very useful to the business.

Table of Contents

Declaration of Authenticity	2
Acknowledgements	2
Table of Figures	5
1. Introduction	6
1.1. Abstract.....	6
1.2. Background.....	6
1.3. Problem Statement.....	6
1.4. Proposed Solution.....	7
1.5. Objective.....	8
1.6. Report Structure.....	8
2. Research and Analysis	9
2.1. Initial Investigation.....	9
2.2. Market Analysis.....	9
2.3. Repository Analysis.....	10
2.4. Technical Research.....	11
2.5. Proof of Concept Deployment.....	11
2.6. Technology Selection:.....	12
3. Design and Modelling	13
3.1. Requirements.....	13
3.2. User Personas.....	14
3.3. Feature Analysis.....	15
3.4. Use Case Scenarios.....	15
3.5. Data Model.....	16
3.6. System Architecture.....	18
3.7. UI Design.....	19
4. Methodology and Iteration Plans	21
4.1. Development Methodology.....	21
4.2. Sprint Overview.....	21
4.3. Sprint 1 - Foundation and MVP.....	22
4.4. Sprint 2 - Document Management.....	22
4.5. Sprint 3 - UI Enhancements and Rejection Reason.....	24
4.6. Sprint 4 - Document Tagging and VPC Logic.....	26
4.7. Sprint 5 - Conversation History, Prompt Editing, and Answer Format.....	26
4.8. Sprint 6 - Questionnaire Mode, Finalisation, Documentation.....	28
4.9. Deviations from Plan.....	30
5. Implementation details	32
5.1. Overview.....	32
5.2. Infrastructure.....	33

5.3. Authentication.....	33
5.4. Document Upload and Approval.....	34
5.5. Answer Generation.....	34
5.6. Conversation History.....	35
5.7. Prompt Management.....	35
5.8. Questionnaire Mode.....	35
5.9. Backend.....	37
6. Reflection.....	38
6.1. What was achieved?.....	38
6.2. What was learned?.....	38
6.3. Problems encountered and solutions.....	39
6.4. Future development.....	42
6.5. Personal Reflection.....	43
7. Use of AI.....	44
7.1. Code Reviews.....	44
7.2. Backend Development.....	44
7.3. Architectural Decisions.....	45
7.4. Frontend Development.....	45
7.5. Other Uses.....	46
8. Glossary of Terms and Abbreviations.....	47
9. Bibliography.....	53

Table of Figures

- Figure 1. Start of Gemini Conversation..... 16
- Figure 2. Proposed schema from Gemini..... 17
- Figure 3. Overall application architecture..... 18
- Figure 4. Lovable Chat page..... 19
- Figure 5. Lovable Policy page..... 20
- Figure 6. Proposed timeline 21
- Figure 7. Policy upload process.....23
- Figure 8. Policy approval process.....25
- Figure 9. Chat in progress..... 28
- Figure 10. Questionnaire mode architecture..... 29
- Figure 11. Questionnaire options..... 30
- Figure 12. Outstanding items at end of project.....31
- Figure 13. Abandoned items.....31
- Figure 14. Policy management page..... 32
- Figure 15. CQA Login Screen.....33
- Figure 16. Prompt management..... 35
- Figure 17. Questionnaire in progress..... 36
- Figure 18. Example debugging conversation..... 41
- Figure 19. Generating JSDoc with Claude..... 46

1. Introduction

1.1. Abstract

The Compliance Questionnaire Assistant (CQA) is a work-based project which uses Retrieval Augmented Generation (RAG) to enable the quick completion of security questionnaires.

Security questionnaires are an essential part of the vendor assessment process. When a company wishes to engage a third-party vendor, they will typically provide a questionnaire to assess their security posture. Completing questionnaires is an effort-intensive task, yet doing so quickly and satisfactorily is essential for attracting new business.

1.2. Background

ServisBOT is an American company operating in the artificial intelligence (AI) sector, with its primary engineering office located in Waterford, Ireland. The company provides AI-powered solutions, including agents, assistants, and copilots, to clients across highly regulated industries such as financial services, mortgage, and insurance.

One of ServisBOT's flagship products is a low-code platform that enables non-technical users to develop and deploy AI bots. A bot developed for a mortgage company, for example, might handle inbound customer calls, identify the nature of each enquiry, and provide relevant information where possible. Where the bot cannot address a customer's query, the system may transfer the call to a human customer service representative. This approach frees up time and resources for human agents, allowing them to focus on more complex queries that may require human expertise and judgment.

1.3. Problem Statement

As ServisBOT primarily operates in highly regulated industries, onboarding new clients typically requires completion of a comprehensive vendor review process. This process verifies that ServisBOT adheres to appropriate data protection standards, follows secure development processes, and meets industry compliance standards.

A critical component of this process is the security questionnaire. This is a detailed assessment of a company's security and compliance posture, comprising complex technical questions intended to evaluate practices across areas such as data handling, access control, and incident response (Chipeta, 2025). A typical questionnaire may contain hundreds of questions, with a standard turnaround time of two weeks. However, in exceptional cases the entire security team has been required to expedite responses to questionnaires within as little as two days.

ServisBOT identified this process as effort-intensive and time-consuming. As a company that leverages automation and AI to reduce manual toil, and with prior company experience with RAG, ServisBOT recognised that a RAG-based solution could substantially streamline the security questionnaire process and save valuable engineering time, and therefore money.

1.4. Proposed Solution

ServisBOT proposed the development of an internal web application to address this problem. The solution would be the Compliance Questionnaire Assistant, a RAG app which would enable ServisBOT staff to upload security and compliance documentation and generate AI-powered responses to questions. Each answer would reference the source documents used to produce it.

RAG is an AI process which improves the output of Large Language Models (LLMs) by allowing them to reference trusted sources, e.g. company documentation, before generating a response to a question. While LLMs are trained on large datasets, they lack access to company-specific documentation and may contain outdated information. RAG addresses these limitations by storing documents in a searchable vector database, where sections or chunks of these documentations are stored as vectors (Amazon Web Services, Inc., 2024). Vectors are essentially long numbers that represent the meaning of data.

When a query is submitted to a RAG application, the system will first search the database, retrieve the most relevant content, and then provide this content along with the user query and a prompt to the LLM. This enables responses which are grounded in the company's actual documentation. RAG is a cost-effective approach as it does not require model retraining, and additionally reduces the occurrence of hallucinations by grounding responses to retrieved source material.

1.5. Objective

The primary objectives of the CQA are as follows:

- Achieve a 50% reduction in time to complete questionnaires
- Consistent, professional, and appropriately referenced responses
- Clear audit trail suitable for compliance reporting purposes

ServisBOT anticipates that meeting these objectives will reduce the burden on the security team, improve response quality and consistency, and make the company more competitive as a vendor.

1.6. Report Structure

This report is structured as follows.

- The **Research and Analysis** section covers the investigation of existing solutions, relevant technologies, and the proof of concept that informed the final design.
- The **Design and Modelling** section describes the system architecture, user personas, data model, and UI design.
- The **Methodology and Iteration Plans** section outlines the development approach and sprint structure.
- The **Implementation Details** section describes the technical implementation of each major feature.

The report concludes with a **Reflection** covering what was achieved, lessons learned, problems encountered, and opportunities for future development.

2. Research and Analysis

2.1. Initial Investigation

The initial investigation phase began with a review of ServisBOT's requirements and the proposed technologies. My primary area of investigation at this stage was RAG, as the process was unfamiliar to me and I was concerned that developing a custom RAG pipeline from scratch could prove a complex task. This prompted research into RAG fundamentals, associated concepts such as vectors and semantic search, and frameworks such as LlamaIndex.

Following this, I moved on to Amazon Bedrock, AWS's managed AI service which provides access to foundation models (FMs). Reading through the Bedrock documentation, I discovered that part of the platform was a managed RAG solution called Bedrock Knowledge Base. Reading further, I realised that this could potentially handle the entire RAG pipeline, significantly reducing the amount of custom code and infrastructure required. This became a central assumption of my planned technical approach.

2.2. Market Analysis

I also performed Market Analysis to assess whether existing solutions could address this problem, and to understand how they approached RAG.

[ChatGPT](#) supports a form of RAG through file uploads, stored in-memory for the duration of a conversation. GPT-5 offers a 128k token context window, which is approximately two to four hundred pages. However, files must be re-uploaded for each new conversation, and exceeding the context window increases the risk of hallucinations. This is a significant concern in the security and compliance context.

[Ragie](#) is a fully managed RAG-as-a-Service platform that supports direct uploads and synchronisation with sources such as Google Drive, Confluence, and Notion. It offers hybrid search and reranking capabilities. However, business-level usage is costly - connectors cost \$250 monthly, with Starter and Pro plans at \$100 and \$500 per month respectively. By comparison, the estimated running cost of a custom solution was under \$100 per month.

[LLMWare](#) is a unified framework for enterprise RAG pipelines using small, specialised models, capable of running without a GPU. It consists of two main components: a RAG pipeline, and over fifty models for tasks such as question answering, classification, summarisation, and extraction. However, this solution would likely require EC2-based hosting, which presents a risk for ServisBOT.

2.3. Repository Analysis

I examined several open-source repositories to inform my design.

- The [Vercel ChatBot](#) repository provided a useful reference for structuring a conversational frontend. However, this application uses Next.js, which is more typically used for server-side applications, whereas my frontend would be client-side only.
- [Chatbot UI](#) offered an example of a multi-provider chat interface, allowing users to interact with ChatGPT, Claude, or Gemini using their own subscriptions.
- [PageIndex](#) demonstrated an interesting vectorless approach to RAG systems by building hierarchical tree indices for document reasoning.
- The most directly relevant repository I found was the [RAG Travel Support Agent](#), which used Bedrock Knowledge Base with Aurora PostgreSQL, API Gateway, and Lambda. This stack closely aligned with my own planned architecture.
- [Docling](#) was also examined as a document parsing library, should Bedrock Knowledge Base prove unsuitable.

2.4. Technical Research

Vector database options considered included S3 Vectors, OpenSearch Serverless, and Pinecone. S3 Vectors were ruled out at the time of research due to the lack of CloudFormation/AWS Cloud Development Kit (CDK) support. OpenSearch Serverless was estimated to cost between \$350-700 per month, which exceeded the project's cost targets. Although Pinecone looked promising, I found reports of pricing uncertainty and longevity concerns on community forums.

I frequently encountered PostgreSQL and the pgvector extension in my research. A useful article I came across showed little difference in performance between OpenSearch and pgvector, with PostgreSQL offering a significant cost advantage (Tokarev Artem, 2025). As AWS Aurora Serverless v2 offered a managed PostgreSQL option with scale-to-zero capabilities, I realised this could suit ServisBOT's preference for serverless technologies.

RAG pipeline alternatives were also investigated as a risk mitigation measure, including LangChain, LangGraph, and LlamaIndex. I additionally researched chunking strategies, for example semantic, hierarchical, and late chunking techniques.

2.5. Proof of Concept Deployment

Once I had access to the dedicated project AWS account, I manually provisioned a Knowledge Base using S3 for document storage and Aurora PostgreSQL Serverless as the vector store. Initial testing with PDF uploads showed promising results.

When deploying a Knowledge Base through the console, AWS generates a CloudFormation template in the backend. Examining this template in detail allowed me to obtain important information about how Knowledge Base connects to the database, manages secrets, and configures default parameters. The database initialisation script in particular proved extremely valuable when later implementing the CDK infrastructure.

2.6. Technology Selection:

After implementing a proof of concept, I moved to the technology selection phase.

- Frontend: React. I selected this over Svelte and Vue.js because of my prior experience with React, its prevalence in the industry, and its existing use within ServisBOT.
- Backend: Node.js + Express. I chose this over Python + FastAPI because it would allow me to reference my ServisBOT onboarding project. Additionally, using JavaScript for backend and frontend minimised context switching during development.
- Infrastructure: AWS CDK for TypeScript. I selected this over CloudFormation and Terraform. Although Terraform was my preference, ServisBOT indicated a preference against its use. However, as the company is transitioning from CloudFormation to CDK, I decided to try the CDK for my project. One of the key advantages of the CDK are constructs, which are essentially pre-configured resources adhering to AWS best practices. Additionally, using TypeScript/JavaScript throughout the project improves consistency and minimises context switching.
- Database: Aurora PostgreSQL Serverless. I chose this over OpenSearch Serverless and S3 Vector Store due to cost effectiveness and CloudFormation/CDK support respectively. PostgreSQL also ranks as the most popular database engine according to the Stack Overflow survey, and offers true serverless capabilities with scale-to-zero functionality.

3. Design and Modelling

3.1. Requirements

ServisBOT provided an initial requirements document, which was subsequently refined into a Scope of Work. The requirements were categorised as functional, non-functional, and technical. Functional requirements covered three main areas.

Document management:

- Users should be able to upload documents
- Administrators should be able to approve documents for indexing
- Documents should be taggable with compliance frameworks
- Approved documents should be automatically indexed into the RAG pipeline

Answer generation:

- Users should be able to query the system
- Responses should have two possible formats:
 - A detailed answer with references
 - A concise auditor-friendly answer
- Answers should have clickable references to source documents

Other:

- Administrators should be able to edit prompts
- Users should be able to view question/questionnaire history
- Stretch goals included:
 - Questionnaire mode: upload questionnaire CSVs and receive completed responses
 - Slack-based collaboration
 - Admin answer approval (i.e. "this answer is accurate")
 - Explainable AI with ranked query responses

The **Non-Functional Requirements** specified:

- Query responses of approximately 2 seconds
- Questionnaire completion faster than the manual process
- Authentication on both frontend and backend
- Reliability and scalability for approximately 5 concurrent users
- Compatibility with modern browsers (Firefox, Safari, Chrome, Edge)
- Intuitive UI for users familiar with AI tools

The **Technical Requirements** specified:

- Infrastructure: AWS, CDK/CloudFormation, S3, Lambda, CloudFront
- Frameworks: React, Material UI, Tailwind, Express or FastAPI
- Programming Languages: JavaScript or Python
- Integration: AWS Aurora PostgreSQL Serverless for vectors and conversation history

3.2. User Personas

Feature prioritisation was guided by creating two user personas to represent the primary users of the application.

Sean Dolan (Head of Security) is responsible for ServisBOT's security posture. Sean is experienced with AI tools and understands both their capabilities and limitations. He wants to be able to view and approve documents before indexing to prevent the inclusion of irrelevant or outdated material. He also wants assurance against AI hallucinations through clickable source references to source material. Sean also wishes to tag documents with specific compliance frameworks to enable more targeted responses, and to edit system prompts to refine answer quality.

Kirsty Sheehan (SecOps Team Member) manages security incidents and customer enquiries. Kirsty would like to reduce the manual effort involved in completing questionnaires. She is experienced with the major AI platforms and expects a comparable experience. Kirsty would like to be able to upload her compliance documentation independently without requiring Sean's intervention. She would like a streamlined query process, asking questions directly to the application without manually searching through documents. She would also like the ability to upload CSV questionnaires for automated completion.

3.3. Feature Analysis

Core feature	User Persona	Pain Point Addressed
Document upload/staging	Both	Ensures current, relevant documentation
Document tagging	Both	Enables framework-specific filtering of answers
Prompt Editing	Head of Security	Allows refinement of prompts for improved answer quality
CSV Questionnaire Mode	SecOps Team Members	Automates questionnaire completion
Answer references	Both	Verifies documentation relevance
Conversation history	SecOps Team Members	Ability to reference previous answers
Collaboration	SecOps Team Members	Validates answer accuracy with colleagues

3.4. Use Case Scenarios

Query Submission: Kirsty receives a client questionnaire and navigates to the bookmarked CQA application, authenticating via Cognito if her session has expired. She either asks a new question on the new chat screen, or checks for relevant prior conversations in the sidebar. When she submits her query, the system retrieves relevant documentation, generates a response, and displays it with source references. Kirsty copies the answer into the questionnaire and continues to ask questions.

Document Management: Sean receives a Slack message from Kirsty indicating that updated policy documents have been uploaded. He navigates to the document management page, filters for documents awaiting approval, and reviews each new upload against the previous version. When Sean approves a document, the system triggers the backend indexing process, overwrites the previous version where applicable, and updates the Knowledge Base accordingly.

3.5. Data Model

Bedrock Knowledge Base manages the RAG pipeline's internal data model.

To design the policy and upload DynamoDB schema, I consulted relevant AWS documentation, spoke to senior engineers, and asked AI tools such as Gemini for feedback.

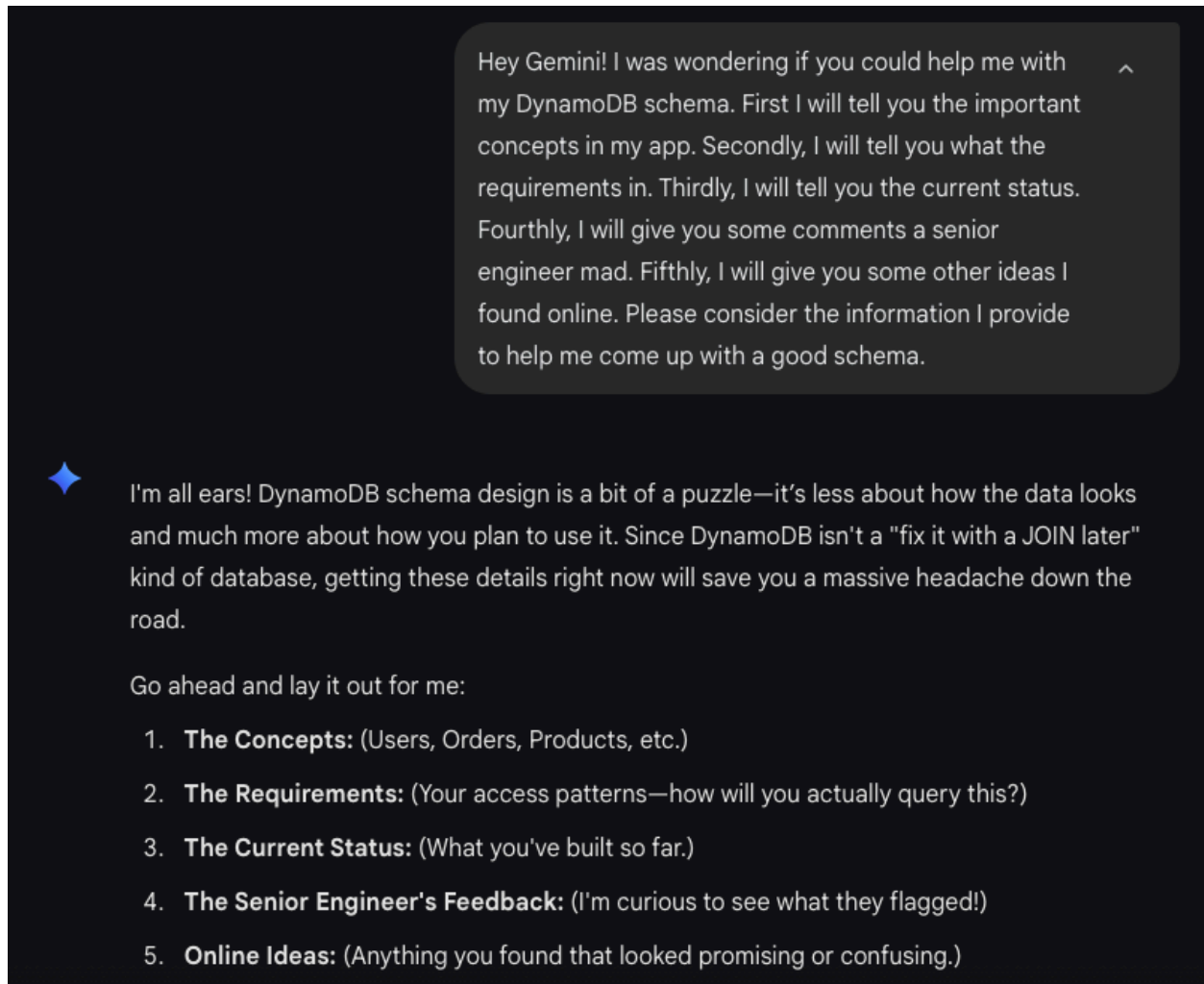


Figure 1. Start of Gemini Conversation



This is a classic DynamoDB modeling challenge. You're balancing **Single Table Design** (getting all related items for one policy in one query) against the need to **list all unique policies** without scanning the whole table.

Your senior engineer is pushing for "locality" (getting everything for one ID at once), but we have to be careful not to make the "List All" view too expensive.

Here is a schema proposal that incorporates your senior's feedback, handles the Bedrock sync requirements, and keeps your metadata manageable.

The Recommended Schema: "The Versioned-First approach"

Instead of a separate `details` record, we treat the **Current Version** as the "Source of Truth" for the policy metadata.

1. The Main Table

Partition Key (PK)	Sort Key (SK)	Attributes (Metadata)
<code>POLICY#<PolicyId></code>	<code>METADATA</code>	policyName, description, tags, syncedStatus, currentVersion, latestStorageKey
<code>POLICY#<PolicyId></code>	<code>VERSION#002</code>	versionDetails, storageKey, createdAt

Figure 2. Proposed schema from Gemini

The schema that was decided upon uses policyId as a partition key, and the sort key may be one of details, upload:::uploadId, or version:::version. This schema allows uploaded documents to be clearly associated with the policies they are intended for, and enables all metadata related to a policy to be retrieved with a single query.

For conversation metadata, a senior engineer provided me with a schema used in other ServisBOT applications. The conversations table uses userId as the partition key and type:::conversationId as the sort key, where type may be either conversation or questionnaire. Messages are then stored in a separate table with conversationId as the partition key and timestamp:::messageId as the sort key, ensuring that messages for a given conversation are retrieved in the order that they were received.

3.6. System Architecture

Consistent with ServisBOT's typical architecture approach, the application is designed using serverless infrastructure throughout. The frontend is a React single-page application hosted in S3 and distributed using CloudFront. API requests are routed through API Gateway to a single Lambda function hosting the Express backend. Both frontend and backend are protected by Amazon Cognito for authentication.

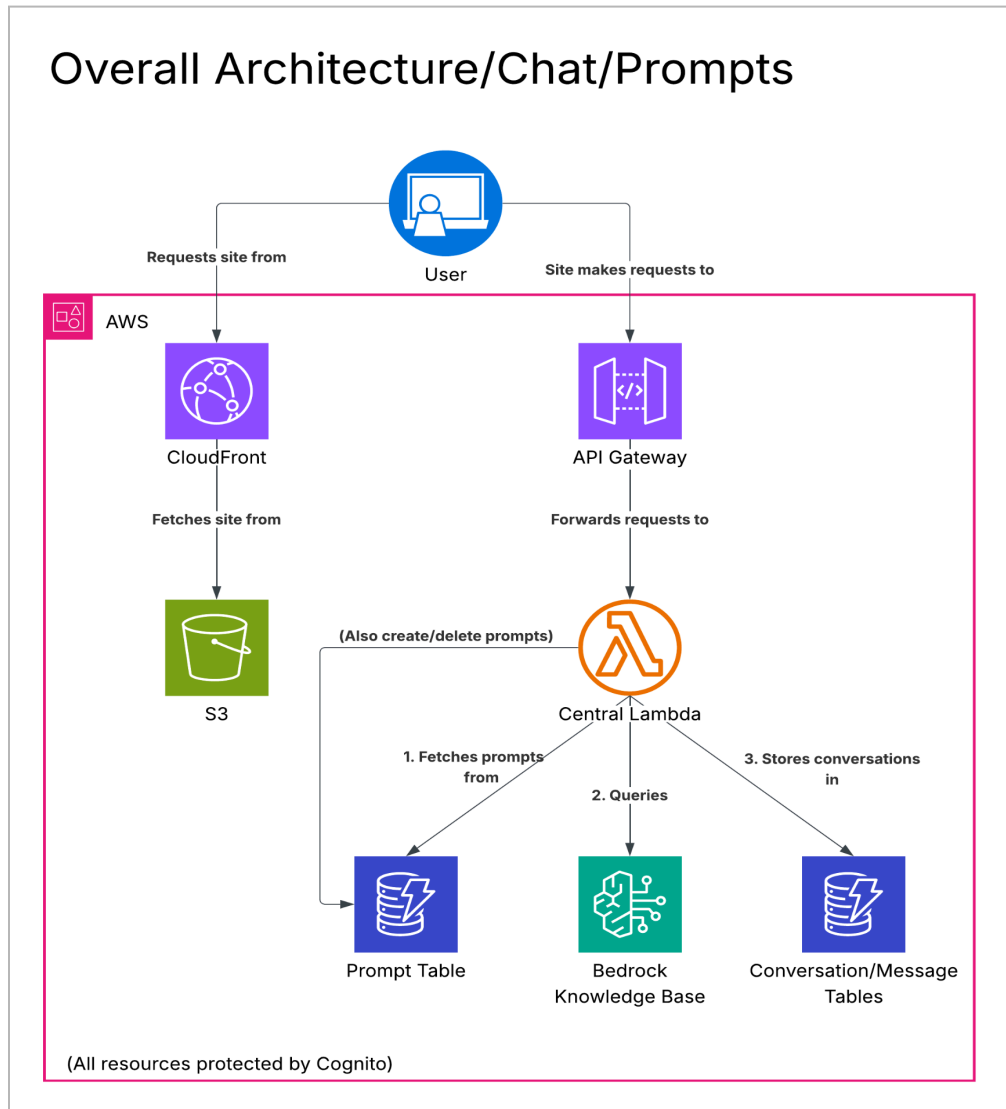


Figure 3. Overall application architecture

Document storage uses S3, with access provided to authenticated users via presigned URLs to ensure buckets remain private. The RAG pipeline is managed by Bedrock Knowledge Base, using Aurora PostgreSQL Serverless as the vector store.

3.7. UI Design

A UI mockup was produced by a member of the ServisBOT SecOps team using Lovable.

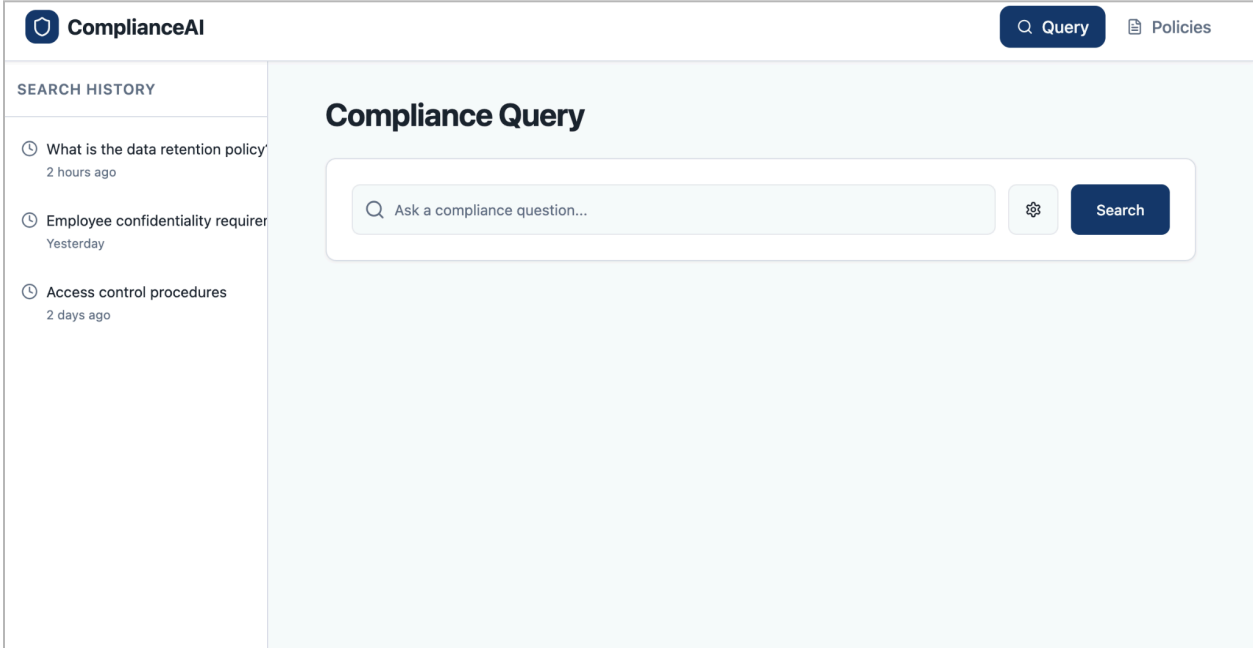


Figure 4. Lovable Chat page

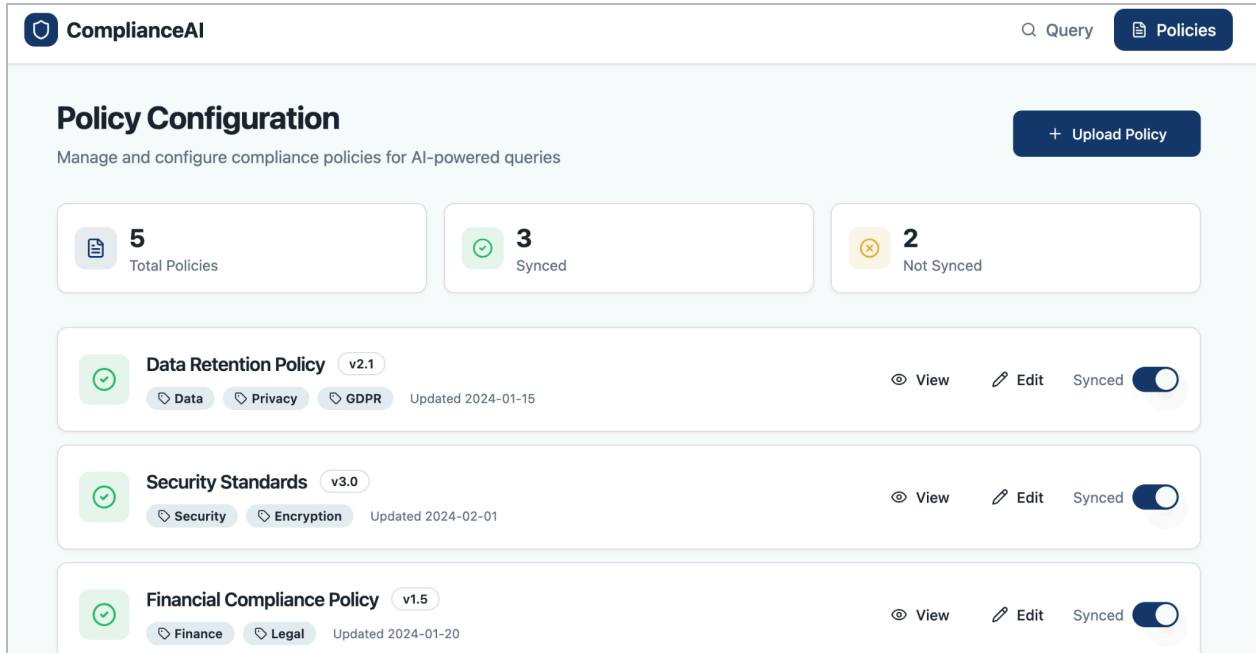


Figure 5. Lovable Policy page

This served as a reference point during frontend development, alongside popular AI interfaces such as ChatGPT and Gemini, informing layout and interaction decisions given that target users will already be familiar with these tools.

4. Methodology and Iteration Plans

4.1. Development Methodology

I chose an Agile/Scrum approach with two-week sprints, allowing time for unexpectedly complex tasks while also enabling additional backlog tasks to be pulled forward when ahead of schedule. I chose Asana as the project management tool over Jira, as it is more lightweight and better suited to individual projects.

ServisBOT provided a high-level timeline at the outset of the project.

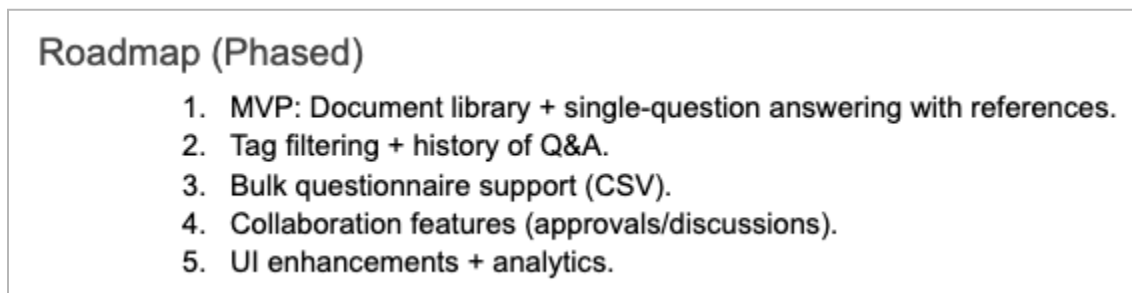


Figure 6. Proposed timeline

The opportunity to begin working on the project early allowed me to validate my planned architecture against ServisBOT's requirements before development started in earnest.

4.2. Sprint Overview

Development was organised into the following high-level areas, each one addressed over one two-week sprint. However, as is the nature of a greenfield project, I often tackled minor bugs or improvements in other areas while working on major features.

4.3. Sprint 1 - Foundation and MVP

The first sprint consisted of the initial setup of the AWS environment, base CDK infrastructure, and a minimal working application. A Bedrock Knowledge Base with Aurora PostgreSQL Serverless as the vector store was provisioned alongside an API Gateway and Lambda-hosted Express backend, along with a basic React Router single page application (SPA) frontend capable of submitting a query and displaying a response. At this stage in development, the frontend was running on localhost only.

4.4. Sprint 2 - Document Management

This sprint covered document uploads, metadata storage, and the document approval workflow. ServisBOT provided a list of required metadata fields for each upload, which necessitated a two-step process. Upon receiving a request to upload a document, the backend first stores preliminary metadata in DynamoDB, such as `userId` and `policyId`. Upon successful upload, an S3 event triggers a Lambda function to update the metadata entry with additional information, such as file size.

The document approval workflow proved more complex than initially anticipated. After a discussion with the security team, I built a pending uploads page which would allow administrators to review all pending uploads for a policy. There, they would be able to approve or reject uploads, with approved uploads automatically syncing to the Knowledge Base.

A call with a senior engineer proved particularly useful here. He suggested simplifying the logic by placing each synced policy at a fixed S3 path, `production/<policyId>/policy`, rather than including the original filename as I had previously been doing. After some initial tests, I found that this approach worked well. Bedrock does not require a specific filename or file extensions, and it detects when a file has changed and resyncs it accordingly. This change significantly reduced the complexity of the approval workflow.

This sprint also included the ability to view policies and uploaded documents through the frontend, supporting both PDF and Markdown file types. This required additional backend logic to return a fileType field alongside presigned URLs when requests to view documents were made. Due to the complexity of these features, work extended beyond the two week sprint window.

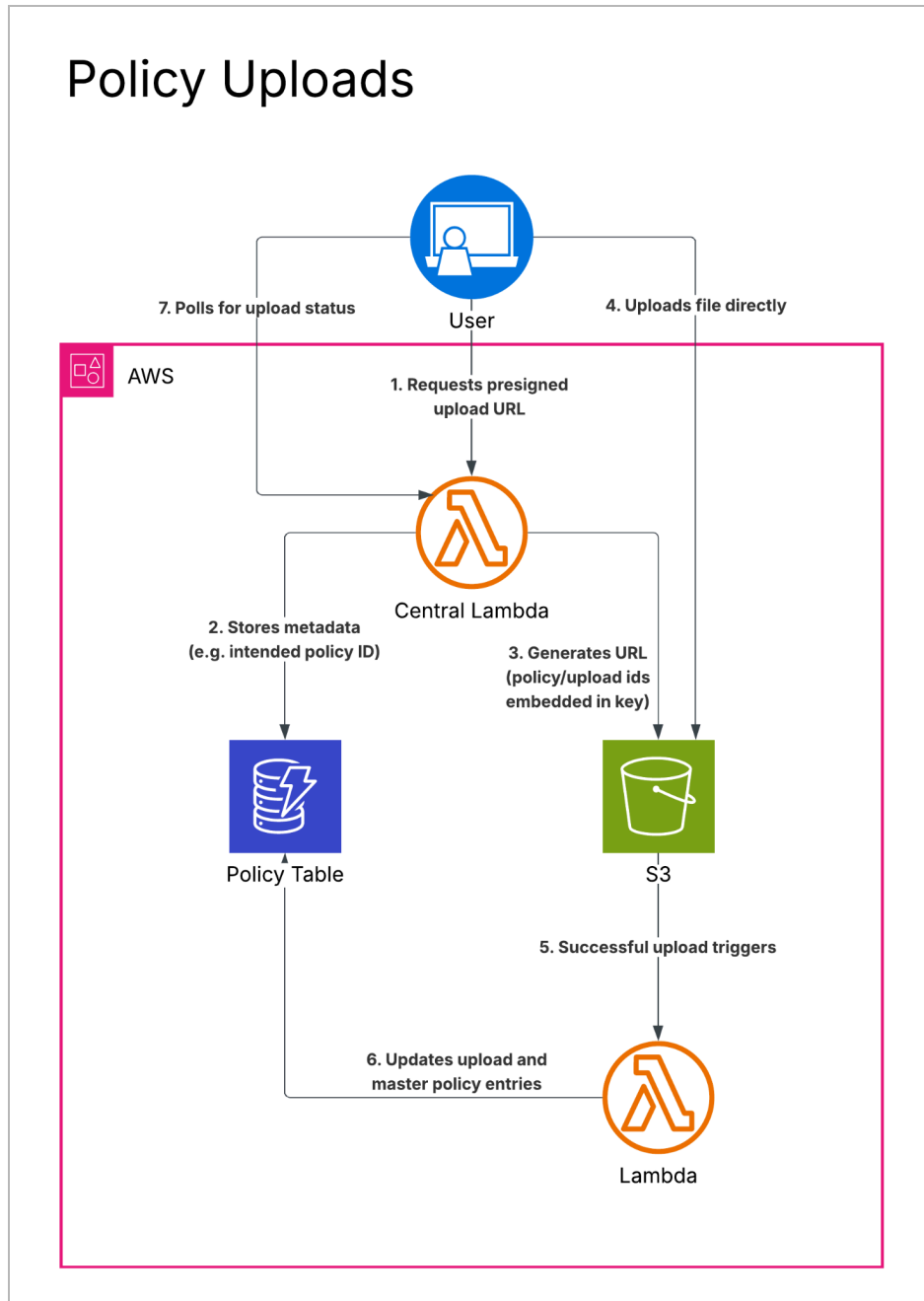


Figure 7. Policy upload process

4.5. Sprint 3 - UI Enhancements and Rejection Reason

At this point in the project, much of the core functionality had been implemented, but the frontend appearance had not received much attention. I decided to address this before continuing with new features. The first task was improving the dark mode. The app had been set up using React Router, which uses Tailwind by default, while the rest of the app used Material UI. I simplified the setup by removing Tailwind and using Material UI throughout, which made dark mode straightforward to implement and therefore improved the overall appearance of the application.

The security team also requested a rejection reason feature, which would allow administrators to provide a reason when rejecting uploads. This was implemented as an extension to the pending uploads page.

Policy Approval

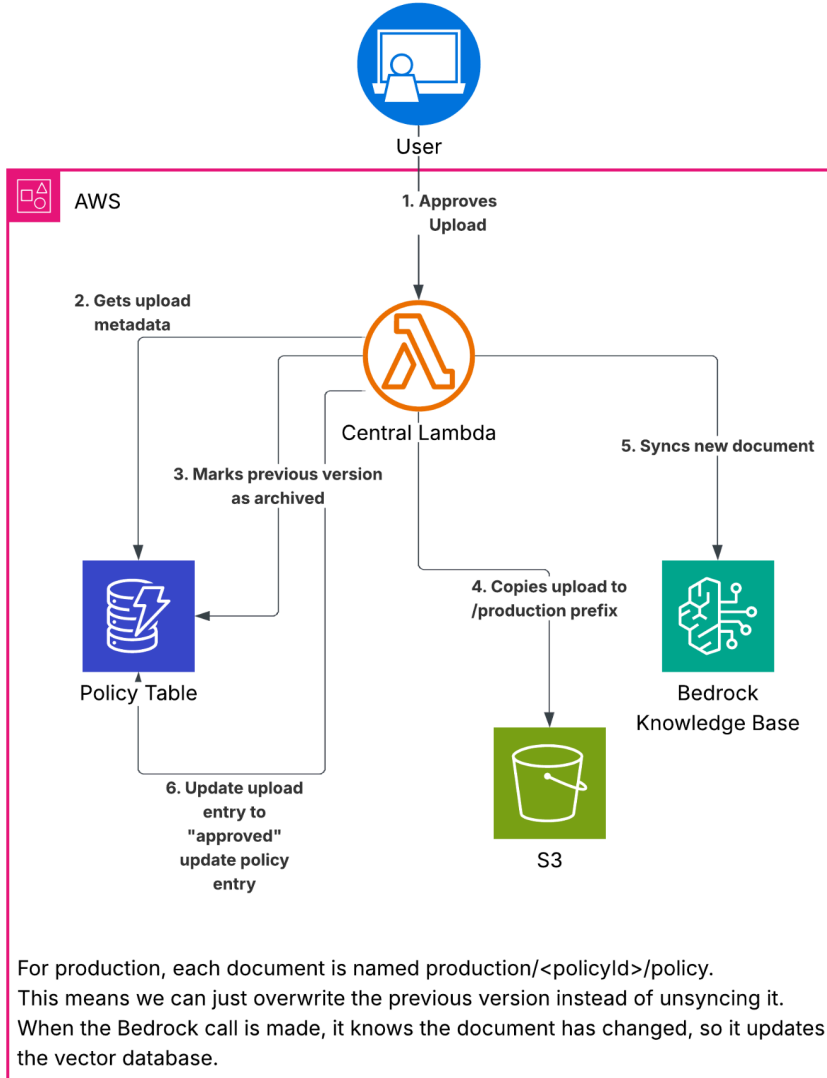


Figure 8. Policy approval process

4.6. Sprint 4 - Document Tagging and VPC Logic

This sprint introduced a document tagging feature, allowing documents to be associated with specific compliance frameworks for use in filtered queries. Bedrock Knowledge Bases supports metadata filtering, which was well-suited to ServisBOT's requirements. The implementation involved generating a metadata.json file for each document which included its tags.

I encountered a problem when implementing this logic. I found that Bedrock Knowledge Bases does not support all data types across all vector data stores, and Aurora PostgreSQL does not support string arrays. I resolved this by converting tag arrays to comma-delimited strings and using Knowledge Base's stringContains and andAll filters, which together check whether a document's metadata contains all of the specified tags.

VPC logic was also added in this sprint. The CDK stack had initially been configured to deploy a new VPC for the Aurora database, but a senior engineer mentioned that it would be useful to also support deployment into an existing ServisBOT VPC. I added logic to check for a provided VPC ID at time of deployment, using that VPC if present but creating a new one otherwise.

4.7. Sprint 5 - Conversation History, Prompt Editing, and Answer Format

Prior to this sprint, conversations were not persisted and navigating away from the page would result in the conversation being lost. This sprint addressed that by implementing conversation history in DynamoDB.

My original plan had been to also use the Aurora PostgreSQL database for conversation storage, and I began to modify the database initialisation script accordingly. However, a conversation with a senior engineer prompted a change of approach. He pointed out that storing conversations in PostgreSQL would couple the application tightly to that database. As it stood, changing the backing vector store would be relatively easy, but storing conversations in the database alongside vectors would make future migration difficult.

He suggested using DynamoDB instead, and provided a conversation schema based on patterns used in other ServisBOT applications. This advice was particularly useful as one future development may be migrating to S3 Vector Store for cost and maintainability.

The backend implementation was relatively straightforward. However, the frontend was considerably more complex as it needed to handle three distinct states: starting a conversation, continuing a conversation, and loading a pre-existing conversation. I used TanStack Query to manage the cache, which would allow conversations to be continued without reloading from the backend on each message.

However, it was difficult to ensure that messages were cleared when navigating to a new conversation, and that there were no visible screen flashes during transitions. This required several days of troubleshooting, but was ultimately resolved with the help of Google and Claude.

Prompt editing proved easier to implement. I built a simple Admin page with prompt creation and list capabilities in the backend, storing prompts in DynamoDB. While testing, I realised that the RetrieveAndGenerate command I was using in the backend required a specific prompt format. I implemented a rollback feature, which would allow administrators to quickly roll back to a previous version if required.

One of ServisBOT's requirements was answer references. After a trial and error process, I decided to implement backend logic that would take the citations array returned by Bedrock and filter it so that a simple list of page numbers and locations was included in each answer. Here, I also decided to include metadata such as policyName and version in the metadata.json file. This would allow users to see which specific version of a policy was used to answer a question.

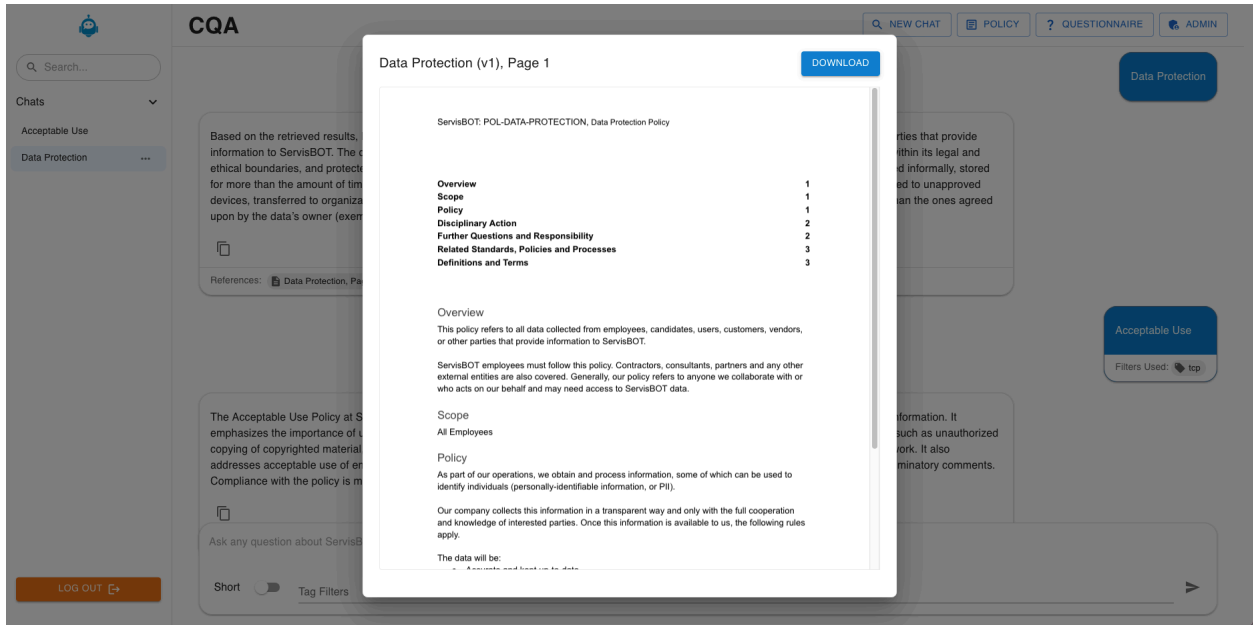


Figure 9. Chat in progress

4.8. Sprint 6 - Questionnaire Mode, Finalisation, Documentation

I assessed the time remaining to complete the project, and decided to attempt implementing the questionnaire stretch goal. A conversation with a senior engineer was again helpful in this process. He noted that a questionnaire could be modelled as simply an automated conversation, and I could likely use the preexisting DynamoDB tables. He also advised using SQS over Step Functions, as although this would be a suitable use case for the technology, they would be quite complex to implement in the timeframe I had available.

I first built an endpoint which would allow users to upload a questionnaire CSV. Upon upload, a Lambda is triggered, parses the CSV, and sends each question as a message to an SQS queue. Initially, I used a FIFO queue to preserve question order. However, after a conversation with a senior engineer, I realised that FIFO was probably unneeded in this situation.

When sending messages to SQS, you must provide a messageID. I realised that by using this ID, plus a timestamp, I could generate the sort key of each message before it was sent to Bedrock and additionally preserve order when reconstructing the CSV. Because I was using DynamoDB transactions, I could also prevent duplicate questions or answers being stored in the database.

Switching to a standard queue allowed multiple Lambda instances to process questions concurrently, reducing the time to complete a 100 question questionnaire to approximately 30 seconds. The last step of this process was getting the answers from the table, adding them to the CSV rows, and writing the completed questionnaire back to the bucket. A polling job in the frontend checks whether the question has been marked as complete, and if so, shows a download link to the user.

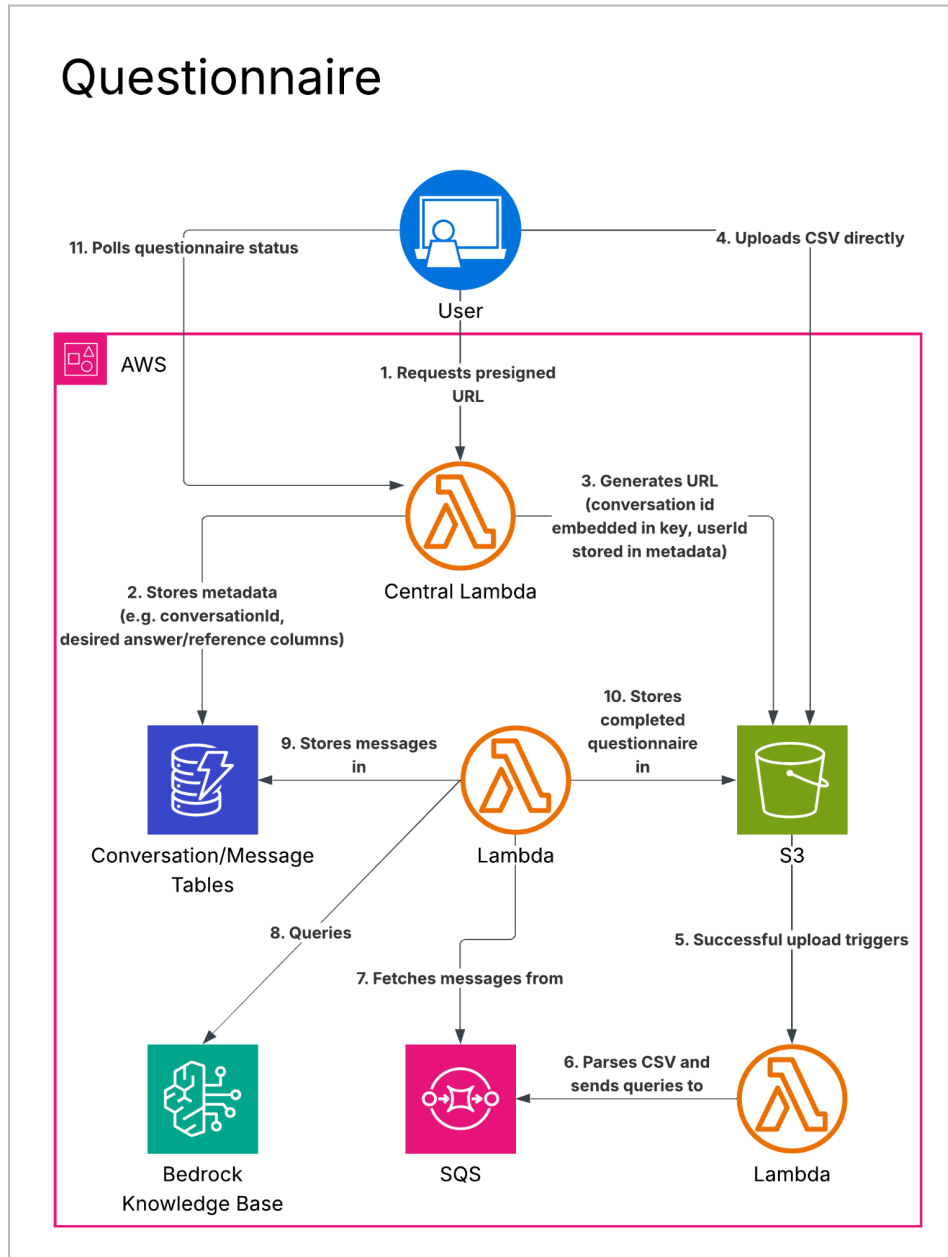


Figure 10. Questionnaire mode architecture

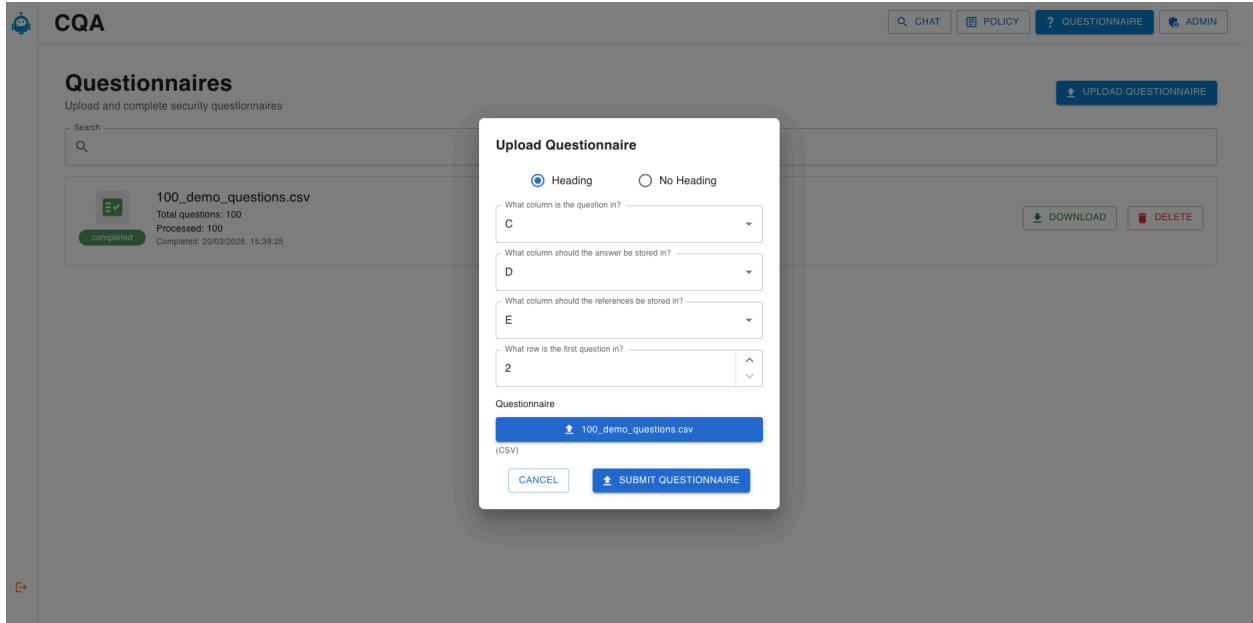


Figure 11. Questionnaire options

After this last feature was completed, the remaining sprint time was used to clean up the codebase. My main controller had grown quite large, so I decided to split it into separate controllers for easier testing. I also added JSDoc, improved README files, and added further documentation.

4.9. Deviations from Plan

The project did not follow ServisBOT's original timeline precisely. The document approval workflow and chat features in particular proved more complex than estimated, extending beyond their planned sprints. However, these delays did not impact the final deliverables, in part due to the choice of Bedrock Knowledge Base as a managed RAG solution and the early start on the project.

The Slack integration feature had been identified as unlikely from the outset and was not delivered. The questionnaire mode stretch feature, however, was successfully implemented and is considered one of the more significant features of the project.

▼ Backlog			
✓ Update to Node 24			
✓ Prompt Versioning/Etc	↵ >		
✓ Pagination			
⌚ (?) Model Choice/Options			(Stretch) Re...
✓ (Stretch) AI generate conversation title			
✓ (Stretch) Rework RetrieveAndGenerate			
✓ (Stretch) Dashboard			
✓ (Stretch) Flag/Rate Questions/Answers			
✓ (Stretch) Collaboration			✓ Question ...
✓ (Stretch) Okta SSO			✓ Authentic...
✓ (Stretch) Cache			✓ Question ...
✓ (Stretch) Cognito Cookies			
✓ (Stretch) Streaming Responses			
✓ (Stretch) Share Conversations			

Figure 12. Outstanding items at end of project

▼ Abandoned	
✓ S3 Vector Store (potentially revisit)	DC David O' Co...
✓ Split backend stack	DC David O' Co...

Figure 13. Abandoned items

5. Implementation details

5.1. Overview

The CQA is a serverless web application built on AWS. It comprises a React SPA frontend written in TypeScript, using React Router for navigation, Material UI for components and styling, and TanStack Query for cache management. The frontend is hosted in S3, distributed using CloudFront, and communicates with the backend through an API Gateway endpoint. This endpoint then forwards these requests to a Node.js/Express backend which runs on AWS Lambda.

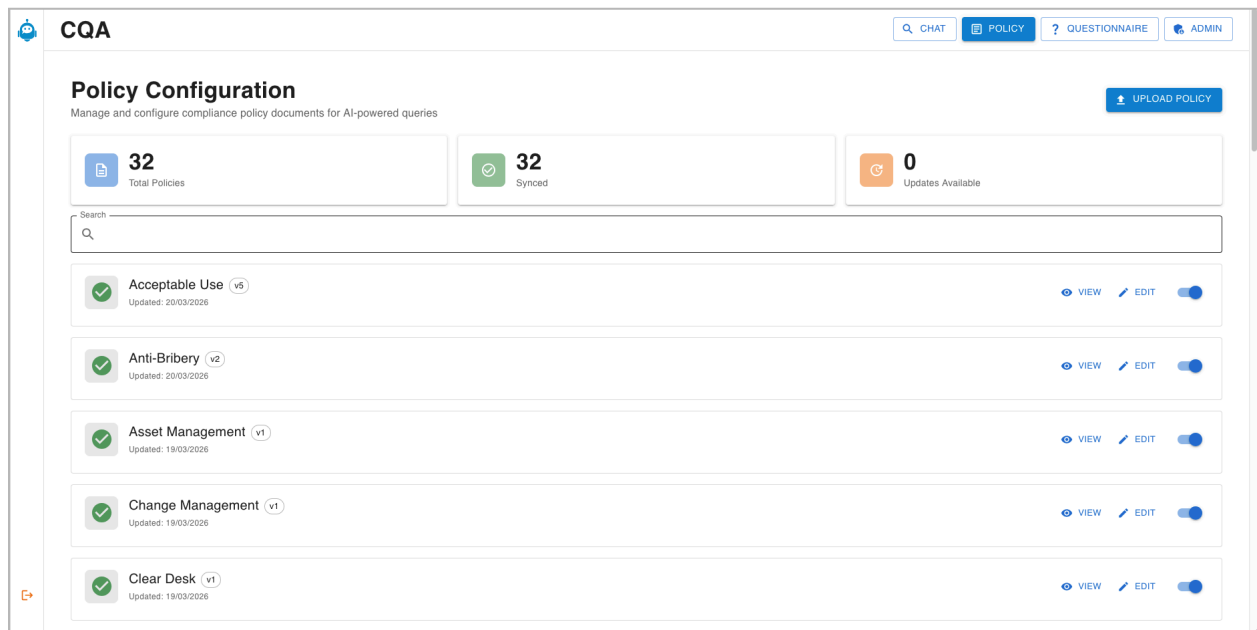


Figure 14. Policy management page

5.2. Infrastructure

All resources are provisioned as code using the AWS CDK for TypeScript. CDK constructs were used whenever possible, as they provide resources pre-configured to AWS best practices. Due to limited CDK support for Bedrock, low-level CloudFormation and custom resources were required for the Knowledge Base and Aurora PostgreSQL integration. The database initialisation script was derived from the CloudFormation template used by AWS when deploying a Knowledge Base manually through the console.

5.3. Authentication

Both frontend and backend are protected by Amazon Cognito, configured using a Google identity provider which restricts access to ServisBOT Google accounts. Once authenticated, the application sends a Cognito identity token with each request. The backend is also configured to use Cognito authentication. Role-based access is enforced in both the frontend and backend. This consists of checking whether the user is a member of the Cognito admin group before displaying admin features to users or approving backend admin requests.

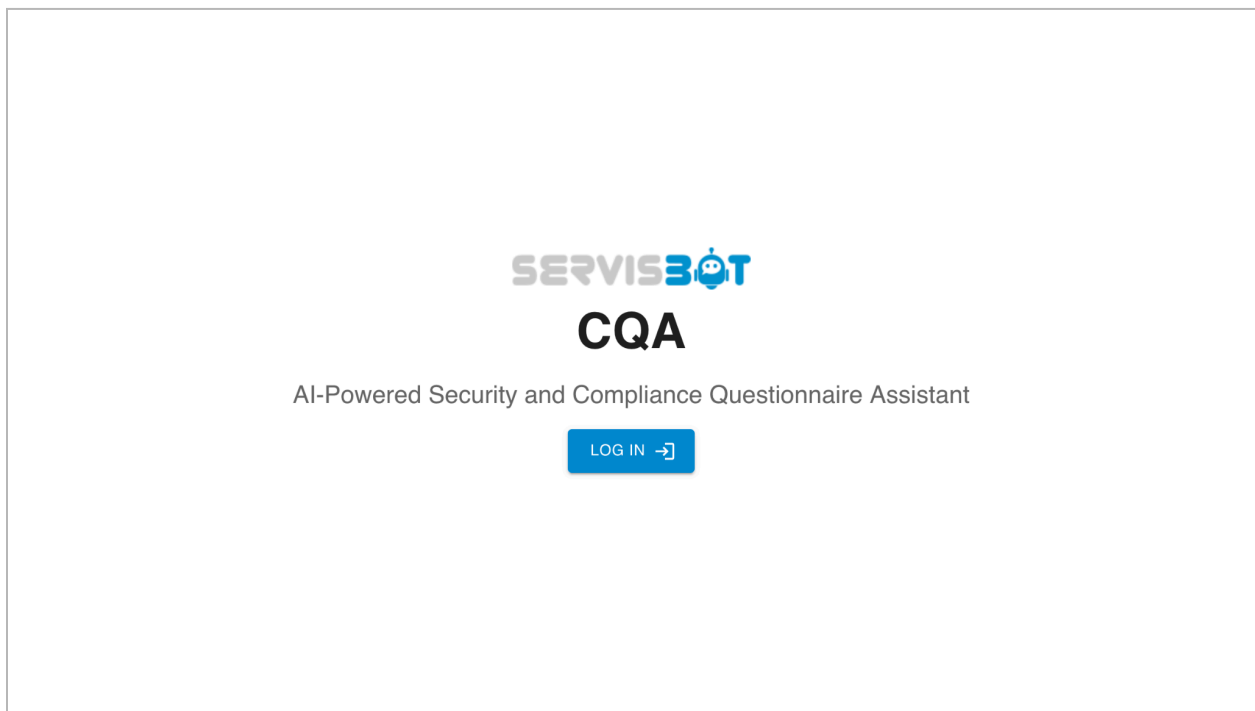


Figure 15. CQA Login Screen

5.4. Document Upload and Approval

Document upload was implemented as a two-step process. When a presigned upload URL is requested, the backend stores preliminary metadata in DynamoDB. Once the file is successfully uploaded to S3, this event triggers another Lambda to update the metadata record with additional attributes such as file size.

Uploaded files are stored at the path `storage/<policyId>/<uploadId>/<fileName>`. This structure means that with the location of an upload, the `policyId` and `uploadId` can be derived, in turn allowing the corresponding DynamoDB metadata entry to be located. These paths are used in the application to view policies, uploads, and versions.

When syncing a policy to the Knowledge Base, we use a `production/<policyId>/policy` path. This path is exclusively used by the Knowledge Base. Keeping storage and production paths separate means that we can view policies regardless of whether they have been synced to the Knowledge Base.

5.5. Answer Generation

Answer generation uses the Bedrock Knowledge Base `RetrieveAndGenerate` command, which combines retrieval and response generation into a single operation. Generation prompts are retrieved from DynamoDB and passed alongside the query in each request. If no prompts are present in DynamoDB, the application falls back to the default Bedrock prompt.

The application supports two prompt formats, short and long. These are controlled by a `promptType` variable passed with each request. Only the short format is used in questionnaire mode.

The complete generated answer is taken from `output.text` in the `RetrieveAndGenerate` response. The `citations` array is used separately to populate a list of source references displayed beneath each answer. These references are clickable, and will open a modal on the specific page that was used in generating that response.

5.6. Conversation History

Conversation history is stored in two DynamoDB tables. The conversations table uses `userId` as the partition key, and `type:::conversationId` as the sort key. Type distinguishes between standard conversations and questionnaires. The messages table uses `conversationId` as the partition key and `timestamp:::messageId` as the sort key. This ensures that messages are returned in order without requiring a separate sort operation.

5.7. Prompt Management

Prompts are stored in DynamoDB and retrieved at query time. To avoid a DynamoDB read on every Lambda invocation, an in-memory cache is used to store prompts between invocations. The most recently created prompt is always the active version, but administrators can roll back to previous prompts via the Admin page if required.

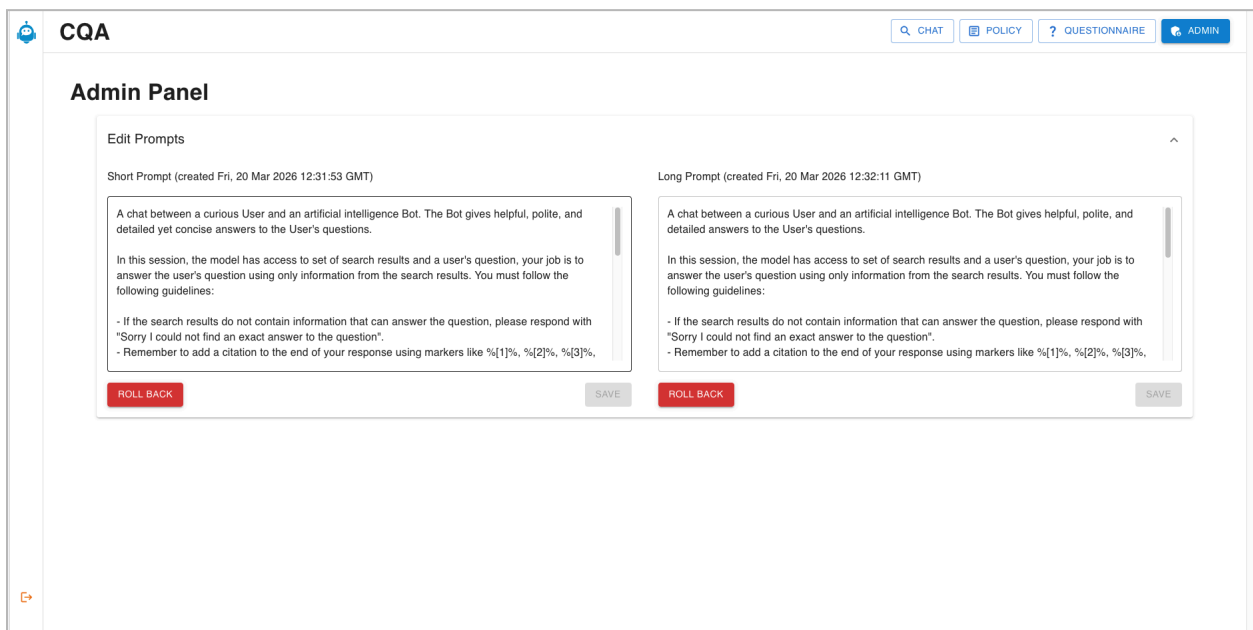


Figure 16. Prompt management

5.8. Questionnaire Mode

Similar to document uploads, when the frontend requests a presigned URL to upload a questionnaire, the backend stores preliminary metadata in DynamoDB. Once the CSV is

successfully uploaded, a Lambda function is triggered to parse the file, send each query to SQS, and update the questionnaire metadata record with the total number of questions.

Each SQS message is assigned a messageId and timestamp before it is processed, which are used to assemble the DynamoDB sort key. This ensures correct ordering and, combined with DynamoDB transactions, prevents duplicate entries in the event of a retry. Using a standard rather than FIFO queue allows multiple Lambda instances to process questions concurrently, reducing the time to complete a 100-question questionnaire to approximately 30 seconds.

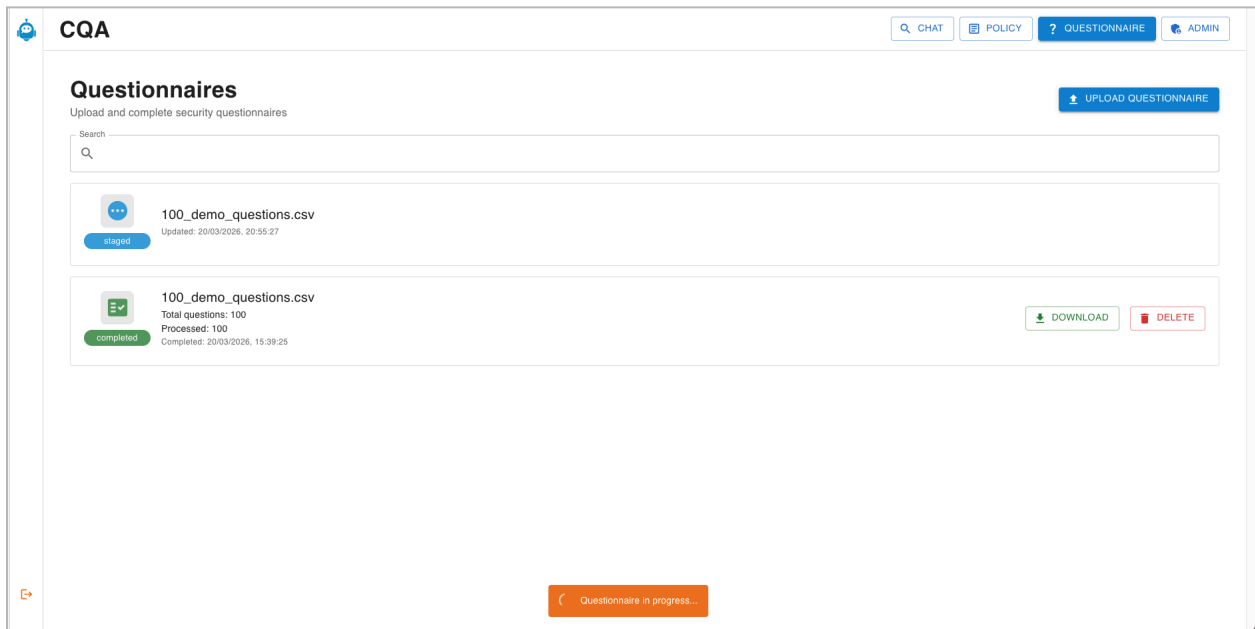


Figure 17. Questionnaire in progress

5.9. Backend

The backend consists of five Lambda functions. The central Lambda hosts the Express application and handles all API requests, interacting with DynamoDB, S3, and Bedrock via the AWS Software Development Kit (SDK). The remaining four Lambdas handle specific event-driven tasks. This includes database initialisation upon first deploy, post-upload metadata fetching for uploads, post-upload parsing for questionnaire CSVs, and processing of SQS messages from questionnaire mode. All Lambdas were written in Node.js.

6. Reflection

6.1. What was achieved?

I was able to achieve the majority of the project goals, with most of the features requested by ServisBOT successfully implemented. The only items which were incomplete by submission were those categorised as “nice to haves”. The application is production-ready and also serves as a solid foundation for future development.

I believe I have met the primary objective of saving the company time and money. The running cost of the application is approximately €50 a month, with the potential to reduce this to as little as €10-20 if it is eventually moved to S3 Vectors. In either case, the time saved in engineer hours should comfortably offset this cost.

I have always had an interest in cyber security, and as this is a security-focused app, security was an area of focus throughout the project. Although the application is publicly accessible, both the frontend and backend are protected by Amazon Cognito. Additionally, the serverless architecture eliminates the operational burden of maintaining and patching servers. Access to S3 resources is handled using presigned URLs, ensuring that storage buckets remain private at all times.

A highlight was the successful implementation of the questionnaire mode stretch goal in the final weeks of the project. This feature allows a hundred-question questionnaire to be completed in as little as thirty seconds - a reduction of several orders of magnitude compared to the typical two-week turnaround.

6.2. What was learned?

This project provided significant learning opportunities. From a software engineering perspective, the value of object-oriented programming became increasingly clear to me as the project grew in complexity. The experience of having built a similar application during the ServisBOT onboarding process proved very valuable. The further the project progressed, the more apparent it became why this earlier application had been structured the way it was. Good

architectural decisions made early compound over time, while shortcuts tend to surface as problems early, and having that earlier application to reference was very useful.

If I were to start the project again, I would likely choose to use TypeScript over JavaScript. As TypeScript surfaces type errors early, directly in the IDE, this would have saved time spent debugging. Failing that, I would have implemented JSDoc more thoroughly earlier in the process. ServisBOT typically use dependency injection, which allows easier testing of individual components but tends to cause the loss of type hinting in IDEs. I did not realise until close to the end of the project that JSDoc could be used to add type hinting for things like AWS SDK clients, and having this from the start would have made development easier.

Similarly, I would also have taken a more test-driven development approach to the project. Writing tests ahead of implementation would likely have caught a number of issues earlier in the development cycle.

6.3. Problems encountered and solutions

The first major issue I encountered was when implementing tag filtering. Although Bedrock Knowledge Base supports metadata filtering, not all data types are supported across every vector store. I found that Aurora PostgreSQL vector stores did not support array types, which presented a problem when associating documents with multiple tags. I was able to resolve this by converting tag arrays into comma-delimited strings and using Knowledge Base's "andAll" retrieval filter combined with "stringContains". This achieved the desired filtering behaviour without a change to the underlying infrastructure of the application.

One of the main requirements of the app was that answers would be accompanied by references to the source documents used to generate them. The RetrieveAndGenerate command returns a citations array containing both the references and the segments of the response they relate to. At first I intended to use this citations array to build inline citations throughout the response text. However, I discovered that Knowledge Base intermittently returns an empty array of citations, which is a known issue with this command.

My solution was to decouple the displayed answer from the citations array. The complete generated answer is taken from `output.text`, and the citations array is used separately to create

a list of references displayed under each answer. This ensures the answer is always presented to the user, even in the rare case that the citations array is empty. In future, as the application matures and potentially moves to a different architecture, it may be possible to reliably implement inline citations.

The Chat feature was perhaps the most technically challenging aspect of the project. The backend logic was relatively straightforward to implement, but delivering a smooth user experience in the frontend proved considerably more difficult, particularly as users will inevitably compare this aspect of the application to established AI products such as ChatGPT or Gemini.

The specific challenge was enabling smooth transitions between conversations, scrolling to the latest message on new input, loading historical chats from the backend, and navigating between chats without visible screen flashes or stale state. A particularly stubborn issue caused messages from a previously viewed conversation to persist in the cache when navigating to a new one.

After researching this issue without success, I used Claude to help diagnose and resolve this behaviour, and a satisfactory solution was reached after several days of iteration. However, I think there are still potential improvements to be made, particularly from other ServisBOT engineers who have more expertise in frontend and React development.

CHAT + v ⚙️ ... | {} ×

← ISSUES WITH APP CONVERSATION FEATURE {}

Hey Copilot! I'm experiencing a lot of issues with the new conversation feature of my app. Right now, I can see a message send, but then it disappears. I can't view other conversations either. Could you help?

TS routes.ts chat.tsx chat.tsx useConversation.tsx

Workspace
Used 49 references
Collecting workspace information

I'll help you debug the conversation feature issues. Looking at your code, I can identify a few problems in the `useConversation` hook that are causing messages to disappear and conversations not to load properly.

Issues Found

- Missing `conversationId` in `useEffect` dependency array**
In `useConversation.tsx`, the `useEffect` that loads conversations is missing `conversationId` in its dependency array, which can cause stale closure issues:
- Race condition with navigation and state**
When sending a message, you navigate before the state is fully updated, and the component may remount with a fresh state. The `navigate` call happens after `continueConversation` but the new `conversationId` from `resolveConversationId` isn't being set properly before navigation:

```
// ...existing code...
const handleSend = async ({ query, promptType, tags }: SendMessageProp
  if (!token) return;

  setError(null);
  setIsLoading(true);

  const tempId = generateMessageId();
  appendMessage({ id: tempId, role: "user", content: query, tags });

  const activeConversationId = await resolveConversationId(query);
  if (!activeConversationId) {
    setIsLoading(false);
```

Figure 18. Example debugging conversation

Debugging a serverless Lambda-based application typically presents challenges compared to traditional server-based programs. Although there are ways to debug serverless applications locally, I typically used CloudWatch logs and console statements instead. Most errors I encountered were related to misconfigured permissions or human error, and the error messages produced were generally sufficient information to identify the cause. I occasionally added my terminal output to GitHub Copilot when investigating less obvious issues.

6.4. Future development

There are several opportunities for further development. When the project started, Aurora PostgreSQL was selected as the vector store due to its balance between performance and cost. Although I am happy with this choice, it did introduce additional complexity to the application in the form of VPC logic, initialisation scripts, and occasional issues with the chat feature due to Aurora's autopause functionality.

At the time of development, S3 Vectors were available in preview but could not be deployed through the AWS CDK, which ruled them out. They have since become generally available and have been adopted in other ServisBOT projects. I attempted to migrate to S3 Vectors approximately a month before the final submission of the project, but CDK support for Bedrock Knowledge Base backed by S3 Vectors seems to be incomplete. Once CDK support matures, migrating to S3 Vectors would reduce both complexity and cost while delivering similar results.

The Prompt feature of the app currently allows creation and rollback via deletion. It is likely that ServisBOT would benefit from a more robust versioning system, similar to the system used for policies. This would allow prompts to be retained and compared historically rather than simply deleted upon rollback. It would also enable users to identify which prompt performed best for a given type of question, and revert to this prompt without losing subsequent versions.

The application uses the RetrieveAndGenerate command which combines data retrieval and answer generation into a single operation. Separating these two processes into distinct steps would provide greater control over the process, allow the use of a wider range of models, and allow more flexible prompt structures. Currently, prompts must be in a specific format for the RetrieveAndGenerate command.

One feature I wanted to implement was response streaming, which would improve the user experience by reducing the time to first token. Although API Gateway supports response streaming, the current Lambda proxy integration makes this difficult to achieve without significant architectural changes. In future, it may be possible to enable response streaming by splitting out parts of the Chat feature of the app into its own separate Lambda function.

While Bedrock Knowledge Base saved time and development effort by providing a managed RAG pipeline, there is an opportunity in future to replace it with a custom RAG pipeline using a framework such as as LangChain or LlamaIndex. This would offer greater flexibility and control over document indexing, retrieval, and generation behaviour as the requirements of the application evolve.

6.5. Personal Reflection

This is the largest and most challenging project I have worked on to date, and also the most rewarding. It provided an opportunity to work on a real business problem while also combining areas that I am personally interested in - like cloud architecture, security, and AI - into a product which has the potential to make a time-consuming task much quicker and easier.

I am delighted that I have delivered an application that ServisBOT can use immediately, while also leaving room for the team to built on it further. The project has been a fitting capstone to the Higher Diploma in Computing, and I am very grateful to ServisBOT for providing both the idea and the time to bring it to life.

7. Use of AI

As a company in the AI sector, ServisBOT has a progressive approach to AI adoption and typically stays on the cutting edge of emerging technologies. Given that the CQA itself is an AI-powered application, it is natural that AI tools played a role throughout its development.

7.1. Code Reviews

One of the primary uses of AI during this project was in the code review process. ServisBOT has implemented CodeRabbit to run automatically on each pull request and subsequent commit, providing an initial review layer before human engineers perform their reviews. This typically identifies potential issues or improvements early in the review cycle.

CodeRabbit proved effective at identifying issues such as potential errors, repeated logic, and oversights such as typos. However, as it is configured to run in "nitpick" mode, suggestions required careful evaluation. Each comment was assessed to determine whether it represented a genuine improvement and whether the effort involved was proportional to the benefit.

More recently, ServisBOT began supplementing CodeRabbit and human code reviews with Claude-generated reviews, particularly during periods when engineering resources were constrained due to illness, leave, or competing priorities. The same evaluative approach was applied to Claude's suggestions. As Claude tended to identify different issues than CodeRabbit, the two tools complemented each other well and together served to improve code quality.

7.2. Backend Development

AI assistance was relatively minimal during the development of the backend API. The AWS CDK documentation is comprehensive and well-structured, and my prior AWS experience meant that I was able to troubleshoot most issues I encountered. Additionally, as a similar technology stack had been used during the company onboarding process, I was able to reference that prior work when needed. GitHub Copilot was used selectively to debug issues when a straightforward solution was not immediately apparent.

7.3. Architectural Decisions

AI was consulted at several points during the architectural design. When designing the DynamoDB schema for document metadata storage, I consulted AWS documentation and then provided relevant sections to Gemini, alongside my specific application requirements, to refine the schema design. The resulting design differs somewhat from the referenced AWS documentation and ServisBOT's typical DynamoDB schemas, but has proven effective in practice.

I also consulted Gemini when architecting the document approval process. Based on my requirements, it suggested AWS Step Functions as a potential solution. I started to plan a possible design around this approach. However, after a conversation with a senior ServisBOT engineer I realised that Step Functions did not suit this use case as the process did not require pauses or long- running operations. Lambda was recommended as a simpler and more appropriate alternative. This experience served as a valuable reminder that although AI can be a valuable design aid, it is not a substitute for the expertise of senior engineers.

7.4. Frontend Development

AI was used more extensively during frontend development, primarily due to a stronger personal preference for Infrastructure as Code (IaC) and backend development and the absence of a reference repository for frontend code.

I primarily used Copilot with GPT-4 models in Ask mode, as I found that Agent mode often overcomplicated or even broke code. Over time, I began to use Claude models when required and began to use Agent mode for appropriate tasks. Claude proved particularly effective at scaffolding new pages using React and Material UI. This code was always reviewed afterwards, with an eye to remove unnecessary complexity when present.

AI was also used to make the codebase more DRY (Don't Repeat Yourself). When I identified possible opportunities to reduce repetition, I would choose a Claude model, add the context to the chat, and ask whether it was possible to reduce repetition here. Claude proved particularly effective in this task.

7.5. Other Uses

Towards the end of the project, Claude was used for supplementary tasks such as correcting and simplifying the Swagger file, cleaning up README files, and producing JSDoc documentation for classes. I also used Claude to generate the project website, which I then edited as required. I used either Claude Code or the VSCode Claude integration for these tasks.

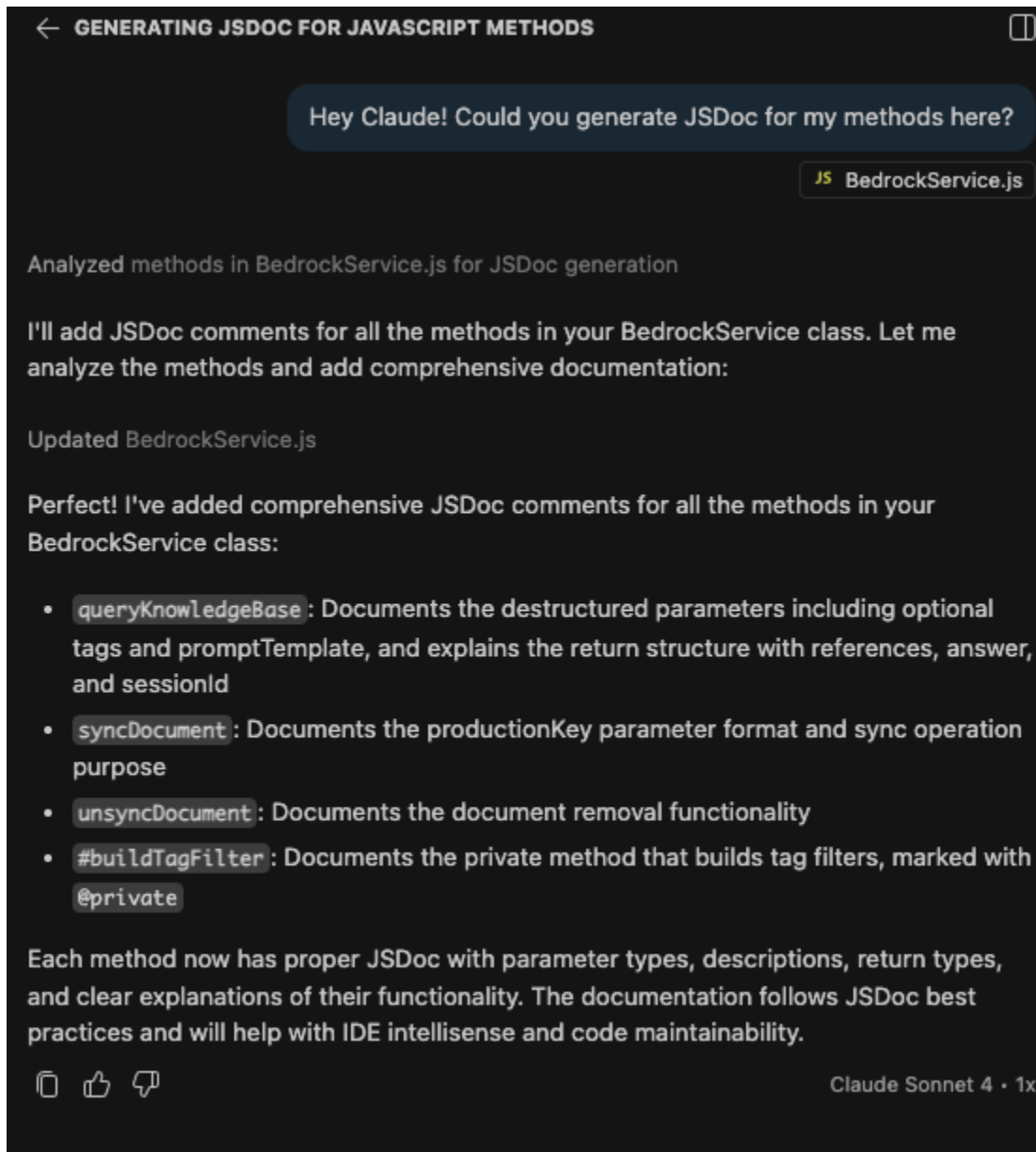


Figure 19. Generating JSDoc with Claude

8. Glossary of Terms and Abbreviations

Access Control: The practice of restricting access to systems, resources, or data to authorised users only.

AI (Artificial Intelligence): Technology that enables computers and machines to simulate human learning, comprehension, and problem solving.

AI Agent: An AI system capable of acting autonomously to achieve a goal.

AI Model: A program trained on a large dataset of information to recognize patterns, make predictions, or generate content.

Amazon Cognito: An AWS service that provides user authentication and authorisation.

AWS Step Functions: An AWS service for orchestrating complex, multi-step workflows using a visual editor.

CDK (Cloud Development Kit): An AWS framework that allows cloud infrastructure to be defined and deployed using programming languages such as TypeScript.

Chunking: The process of splitting documents into smaller segments before storing them in a vector database.

CloudFront: AWS's content delivery network (CDN) service, which is commonly used to distribute websites hosted on S3.

CloudWatch: An AWS monitoring and observability service. Lambda invocation logs are typically delivered to CloudWatch logs.

CodeRabbit: An AI-powered code review tool that analyses pull requests and provides automated feedback.

Context Window: The maximum amount of text an LLM can process in a single interaction, measured in tokens. Exceeding the context window can result in the model losing track of earlier content or producing hallucinations.

Dependency Injection: A software design pattern in which a component's dependencies are provided externally rather than created internally.

DRY (Don't Repeat Yourself): A software development principle which states that logic should not be duplicated across a codebase.

DynamoDB: A fully managed, serverless NoSQL database provided by AWS.

DynamoDB Transactions: A DynamoDB feature that allows multiple write operations to be executed atomically, ensuring that either all operations succeed or none are applied.

EC2: Amazon Elastic Compute Cloud. An AWS service providing virtual servers. Unlike serverless options, the responsibility for maintaining and patching EC2 falls upon the user or business.

FIFO: First In, First Out. A queueing strategy that guarantees messages are processed in the order they were received.

GitHub Copilot: An AI-powered code assistance tool integrated into VSCode.

GPU: Graphics Processing Unit. A processor widely used to train AI models.

Hallucinations: Instances where an LLM generates plausible-sounding but factually incorrect or unsupported content. RAG reduces hallucinations by grounding responses in retrieved source material.

Hierarchical Tree Index: A document indexing approach that organises content into a tree structure, enabling reasoning across large documents without the use of vector embeddings.

Hybrid Search: A search technique that combines vector-based semantic search with traditional keyword search to improve retrieval accuracy and relevance.

Identity Provider: A service that authenticates users and provides identity information to other applications. The CQA uses Google as an identity provider via Amazon Cognito, restricting access to ServisBOT accounts.

IaC (Infrastructure as Code): The practice of managing and provisioning infrastructure through code rather than manual configuration.

In-Memory Cache: A temporary data store held in a running process's memory, enabling fast retrieval without repeated database reads.

Indexing: The process of processing and storing documents in a vector database so that they can be retrieved during a semantic search.

JSDoc: A documentation standard for JavaScript and TypeScript code, using structured comments to describe functions, parameters, and return types.

Large Language Models (LLMs): AI models trained on large volumes of text data, capable of generating, summarising, and reasoning about natural language.

Localhost: Refers to the local machine a program is running on, typically used during development before an application is deployed to a remote environment.

Lovable: An AI-powered platform for building apps and websites by chatting with an AI.

Low-Code: A development approach that allows applications to be built with minimal hand-written code, typically through visual interfaces or configuration.

Markdown: A lightweight markup language that uses plain text formatting syntax to produce structured documents.

Material UI: A React component library implementing Google's Material Design system.

Model Retraining: The process of updating an AI model with new data to maintain or improve its performance.

Next.js: A React framework designed for server-side rendering and static site generation.

Node.js: A JavaScript runtime environment that allows JavaScript to be executed on the server.

Partition Key: The primary component of a DynamoDB table's key, used to determine which partition a record is stored in.

Polling: A technique where a client repeatedly checks a server or resource at intervals to determine whether a condition has been met.

Presigned URL: A time-limited URL granting temporary access to a private S3 object without requiring AWS credentials.

Prompts: Instructions provided to an LLM alongside a user query to guide the style, format, and scope of its response.

Proxy Integration: An API Gateway configuration that forwards the entire HTTP request to a backend Lambda function, which is then responsible for constructing the full HTTP response.

RAG (Retrieval Augmented Generation): An AI technique that enhances LLM responses by first retrieving relevant documents from a knowledge base and providing them as context to an AI.

Reranking: A post-retrieval step that reorders search results based on their relevance to the query, improving the quality of content passed to the LLM.

Response Streaming: A technique that sends generated text to the client as it is produced, reducing the time to first visible output. Often used in AI apps such as ChatGPT, Gemini, and Claude.

S3: Amazon Simple Storage Service. A popular and inexpensive object storage service provided by AWS.

S3 Vectors: An AWS vector storage service backed by S3, offering a simpler and potentially more cost-effective alternative to Aurora PostgreSQL for vector storage.

Schema: The defined structure of a database, describing how data is organised, what fields exist, and how records relate to one another.

SDK (Software Development Kit): A library provided by AWS to allow easy interaction with AWS services through JavaScript and other languages.

Secure Development: A set of practices aimed at building software that is resistant to security vulnerabilities.

Security Questionnaire: A structured assessment used by organisations to evaluate the security and compliance posture of a vendor or business partner.

Semantic Search: A search technique which uses vectors to retrieve results based on the meaning of a query rather than exact keyword matches.

Serverless: A cloud computing model in which the cloud provider manages the underlying infrastructure, automatically scaling resources as needed. Developers deploy code without provisioning or maintaining servers.

Single Page Application (SPA): A web application that loads a single HTML page and dynamically updates content in the browser without full page reloads.

Sort Key: The secondary component of a DynamoDB composite key, used to sort and filter records within a partition. Combined with the partition key, it uniquely identifies a record.

SQS: Amazon Simple Queue Service. A fully managed message queuing service used for decoupling and scaling microservices.

State: The data representing the current condition of an application or component at a given point in time. State is a frequent concern in React applications.

Swagger: A specification and tooling framework for documenting RESTful APIs.

Tailwind: A CSS framework for styling web applications.

TanStack Query: A data-fetching and cache management library for React.

Token: The basic unit of text processed by an LLM, roughly corresponding to three-quarters of an English word.

Type Hinting: Annotations in code that indicate the expected data types of variables, parameters, and return values, helping IDEs surface errors earlier in the development process.

TypeScript: A statically typed superset of JavaScript that adds type annotations and compile-time type checking.

Vectors: Numerical representations of data, typically long arrays of floating-point numbers, that encode semantic meaning.

Vector Stores: Databases optimised for storing and querying vector representations of data.

VPC: Virtual Private Cloud. An isolated network environment within AWS in which resources such as databases can be deployed.

9. Bibliography

Amazon Web Services, Inc. (2024). What is RAG? - Retrieval-Augmented Generation Explained - AWS. [online] Available at: <https://aws.amazon.com/what-is/retrieval-augmented-generation/>

Chipeta, C. (2025). What is a Security Questionnaire? | UpGuard. [online] www.upguard.com. Available at: <https://www.upguard.com/blog/security-questionnaire>

Tokarev Artem (2025). AWS Bedrock Knowledge Bases: Comparing S3 Vector Store vs. OpenSearch, PostgreSQL & Neptune. [online] Hackernoon.com. Available at: <https://hackernoon.com/aws-bedrock-knowledge-bases-comparing-s3-vector-store-vs-opensearch-postgresql-and-neptune>