Horizontal Line 5, Shape

**MapSmart Technical Documentation**

Mapping Legacy Data to New Systems in Seconds

Created by: Hitansh Bhatt, Rex Parmar, Muhammad Ali, Tim Rozer, and Angelo Rivera

Created on: Monday, June 14, 2025

Horizontal Line 5, Shape

**Table of Contents**

* 1. Project Overview
  + 1.1 Project Description
  + 1.2 Goals & Objectives
  + 1.3 Scope
  + 1.4 MapSmart Team
* 2. Application Features
  + 2.1 Deployment
  + 2.2 Main Application Flow
  + 2.3 RAG Application Flow
  + 2.4 Demo
* 3. Architecture & Design
  + 3.1 Technology Stack
  + 3.2 System Architecture Diagram
  + 3.3 RAG Architecture Diagram
  + 3.4 AWS Services
* 4. Development Process
  + 4.1 Sprint Overview
  + 4.2 Code Structure
  + 4.3 Modules & Components
* 5. Testing & Quality Assurance
  + 5.1 Testing Process
  + 5.2 Key Challenges
  + 5.3 Known Issues
  + 5.4 Future Improvements
  + 5.5 Team Notes
* 6. Appendix
  + 6.1 Access Keys
  + 6.2 Sprint Documentation

**1. Project Overview**

**1.1 Project Description**

Data migration is a common challenge for organizations transitioning from legacy systems. This process often involves reformatting or restructuring data to fit new schemas, which is typically manual, time-consuming, error-prone, and expensive.

MapSmart is an AWS-hosted data processing pipeline designed to automate the mapping of semi-structured data (Files, Databases) by creating embeddings for text using Generative AI. It takes as input either files or database credentials if the information needs to be extracted from a database. The **source** file represents the desired target schema, while the **target** file represents the legacy format to be aligned. Our team has used GenAI tools offered by AWS Bedrock to help create these mappings. The MapSmart UI will output an Excel sheet that contains all mappings between the input files/databases. Additionally, a built-in **chatbot assistant** helps users understand and interact with the mapping process more intuitively.

**1.2 Goals & Objectives**

Below are the main goals the MapSmart team planned and were able to achieve in our 3-month co-op term.

* **Automated Schema Mapping:** Developed a core service that infers column-to-column mappings between legacy and target data structures
* **Diverse File/Database Support:** Support for .csv, .xlsx, .txt file formats as well as connections to Oracle, Sybase, MySQL, SQL Server, PostgreSQL, and MongoDB databases
* **Chatbot Assistant:** Implemented a conversational interface that guides users through the mapping process and explains AI-generated mappings

**1.3 Scope**

This project is useful for **any industry** or company looking to migrate their data. Our current testing process has had a focus on the **Pharmaceutical field**, where it demonstrated strong performance in aligning complex, compliance-heavy datasets. The system can be further customized or fine-tuned to specific domains/industries using:

* Domain-specific prompt engineering
* Fine-tuned foundational models
* Knowledge Base for RAG application
* Custom rule sets for regulatory alignment

**1.4 MapSmart Team**

Our team of five Software Engineer Interns collaborated over a 3-month period to develop MapSmart.

(Frontend Development) Angelo Rivera - <https://www.linkedin.com/in/angelo-riv/>

(Frontend Development) Rex Parmar - <https://www.linkedin.com/in/rex-parmar/>

(DevOps Engineer) Muhammad Ali - <https://www.linkedin.com/in/muhammad-ali-052908152/>

(Backend Development) Hitansh Bhatt- <https://www.linkedin.com/in/hitanshbhatt/>

(Backend Development) Tim Rozer - <https://www.linkedin.com/in/tim-r-042783220/>

**2. Application Features**

**2.1 Deployment**

**Frontend**

1. Clone the repo: git clone <https://github.com/Canada-Co-Op-May-25/MapSmartUI.git>
2. npm install
3. npm run start
4. Host Git Repository on Amplify

**Backend**

All the backend code is deployed in Lambda functions (see *Modules & Components* section) and are connected together using SQS (Simple Queue Service).

1. Clone the repo: git clone <https://github.com/Canada-Co-Op-May-25/MapSmart-Backend.git>

**2.2 Main Application Flow**

* **Admin Page** The Admin page allows users to securely configure access credentials for various data sources. Here, users can add and manage login credentials required to connect to Amazon S3 buckets and supported databases, ensuring seamless integration with the platform. Additionally, they can view error logs for all jobs.
* **Onboarding Page** The Onboarding page guides users through the job creation process. Users can define a new job by specifying essential details such as the job name, input files, and database sources. This structured setup ensures that all required parameters are captured for successful execution.
* **Execution Page** The Execution page provides access to job outputs. Upon completion, users can download the resulting files directly to their local machines for further review or processing. This page serves as a centralized hub for managing and accessing job results efficiently.

**2.3 RAG Application Flow**

The RAG Application flow, Fredrick, can also be utilized to create mapping sheets in a conversational manner.

A screenshot of a computer

AI-generated content may be incorrect.

**2.4 Demo**

Below is a link to our demo video demonstrating the features and the use of the MapSmart tool. Please access this [link](https://cognizantonline-my.sharepoint.com/:v:/g/personal/2409087_cognizant_com/EdgAi5jEyzVCkLmzy2N1nhcBPBpKu4QcLkziABwrazul-A?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D&e=GRwgwa) with your Cognizant account.

**3. Architecture & Design**

**3.1 Technology Stack**

**Frontend**

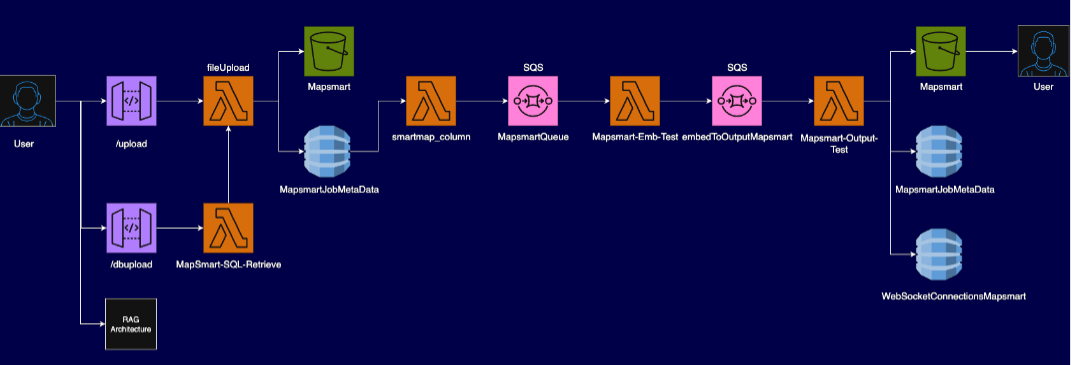
Our MapSmart frontend is built with modern technology stacks used in millions of other applications and businesses. It provides a responsive, intuitive user interface for file uploads, chatbot interactions, and viewing the generated mapping sheets.

| **TECHNOLOGY** | **PURPOSE** |
| --- | --- |
| React.js | Core library for building dynamic and component-based user interfaces |
| Node.js (npm) | Used as the JavaScript runtime for managing frontend dependencies and running local development tools |
| axios | Promise-based HTTP client used to communicate with the backend APIs |
| react-router-dom | Handles client-side routing for seamless page transitions |
| jsPDF | Generates downloadable PDF files (ex. user summaries or logs) |
| ChartJS | Used for rendering analytics and visual summaries in the dashboard |
| react-icons | Provides scalable icons for a consistent and modern UI |
| CountUp | Animates number counters for dynamic UX |

**Backend**

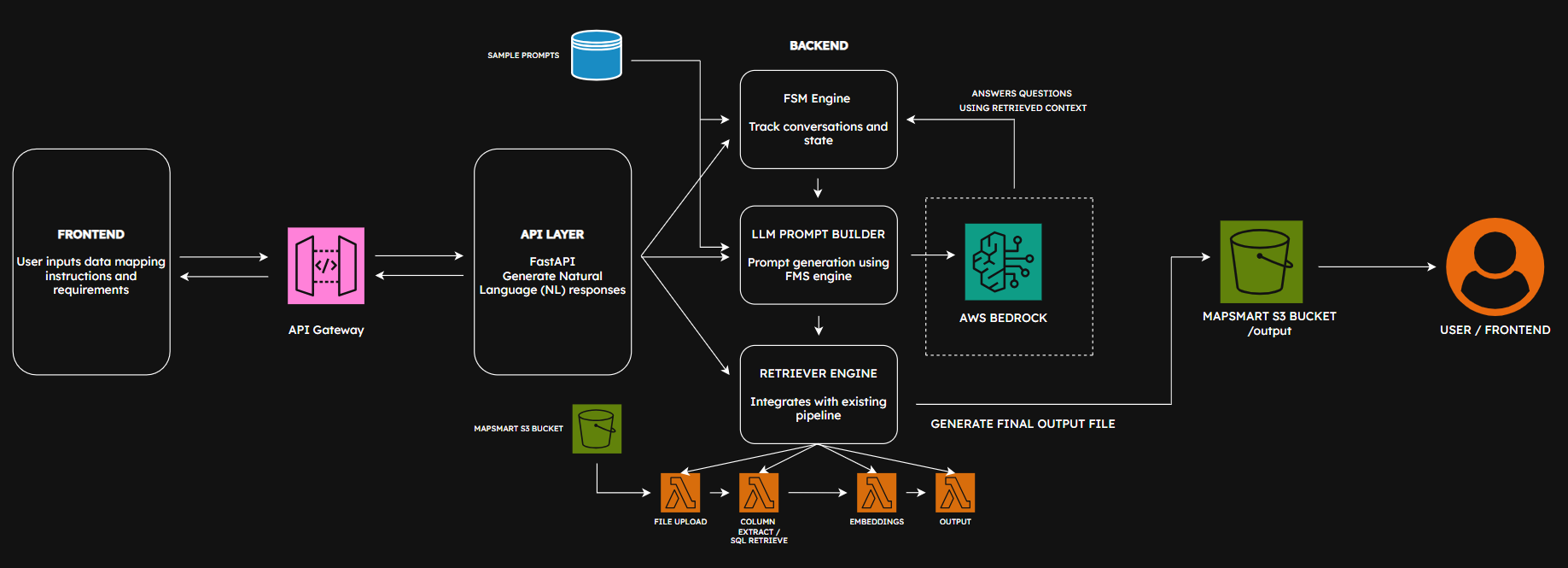
The MapSmart backend is built using a modern, cloud-native technology stack designed for scalability, modularity, and ease of deployment. It powers intelligent column-level data mapping between structured sources using GenAI and serverless computing principles.

| **TECHNOLOGY** | **PURPOSE** |
| --- | --- |
| Python 3.13 | Core programming language for all backend logic, including embedding generation and mapping |
| Amazon Bedrock (Titan v2) | Foundation model used to generate embeddings for semantic similarity between column metadata |
| Pandas & NumPy | For data transformation and manipulation |
| AWS Lambda | Serverless compute environment to host backend functions as microservices |
| AWS API Gateway | Exposes Lambda functions through a RESTful API for external access |
| AWS S3 | Secure storage for user-uploaded files and generated outputs |
| Boto3 (AWS SDK for Python) | Python SDK to interact with AWS services like S3, Bedrock, and Lambda |
| OpenPyXL / XlsxWriter (via Pandas) | Libraries for exporting structured Excel outputs with formatting and metadata annotations |

**3.2 System Architecture Diagram**

The system follows a modular, serverless architecture built entirely on AWS to support scalable and event-driven column-level data mapping. User inputs are received via RESTful endpoints (/upload for files and /dbupload for databases), triggering Lambda functions that store data in Amazon S3 or retrieve metadata from databases. Job metadata is tracked in a centralized DynamoDB table (MapsmartJobMetaData). A core mapping pipeline is initiated via the smartmap\_column Lambda function, which publishes tasks to an SQS queue for parallel processing. Subsequent Lambdas (Mapsmart-Emb-Test and Mapsmart-Output-Test) consume messages from the queue to generate column embeddings, compute mappings, and write the final Excel outputs to S3. Real-time updates and results are pushed to users, completing an end-to-end, low-latency GenAI mapping workflow.

**3.3 RAG Architecture Diagram**



The Retrieval-Augmented Generation (RAG) architecture enhances the column mapping workflow with a natural language chatbot interface. Users input data mapping instructions through a frontend UI, which communicates via API Gateway to the backend. The FSM (Finite State Machine) engine tracks conversational state and feeds relevant user inputs and sample prompts into an LLM Prompt Builder. This builder interacts with AWS Bedrock to generate intelligent responses based on retrieved context. The Retriever Engine pulls metadata or file inputs from S3 and integrates with the existing mapping pipeline powered by Lambda functions—handling file uploads, column extraction, embedding generation, and output generation. The final Excel mapping file is stored in an S3 bucket and delivered back to the user, closing the loop in a seamless, end-to-end GenAI-powered mapping experience.

**3.4 AWS Services**

| **Category** | **Service** | **Purpose** |
| --- | --- | --- |
| **Data Processing** | Amazon Bedrock (Titan) | Generates semantic embeddings and powers GenAI-based mapping logic. |
| AWS Lambda | Executes serverless functions across each pipeline stage. |
| Amazon Simple Queue Service (SQS) | Handles decoupled microservice communication using asynchronous queues. |
| **Storage & Databases** | Amazon S3 | Stores input/output mapping files and artifacts. |
| Amazon DynamoDB | Tracks job metadata and WebSocket connections in real time. |
| Amazon RDS | Extracts structured metadata from relational DBs (PostgreSQL, Oracle, etc). |
| **Deployment & API** | AWS Amplify | Deploys the frontend application with CI/CD support. |
| Amazon API Gateway | Manages secure RESTful API endpoints for frontend-backend communication. |
| **Security** | AWS Web Application Firewall (WAF) | Protects against common web exploits and malicious traffic. |
| **Cost Management** | AWS Budgets | Sets budget alerts and monitors usage thresholds |
| AWS Cost Explorer | Analyzes detailed cost breakdowns across services. |
| **Logging & Monitoring** | Amazon CloudWatch | Tracks logs, metrics, and alarms in real time. |
| AWS CloudTrail | Provides audit logs for all API interactions across services. |
| Amazon SNS (Simple Notification Service) | Sends alerts and status notifications (e.g., job complete/fail). |
| Amazon S3 Glacier Flexible Retrieval | Archives long-term mapping data cost-effectively. |
| AWS Backup | Manages backups of S3, RDS, and other critical services. |

**4. Development Process**

**4.1 Sprint Overview**

**Sprint 1:​**

**Frontend**: Created the static version of the website. Designed the initial layout of buttons. ​

**Backend**: Designed the file-to-file pipeline and developed the initial version of the core embeddings logic. ​

​

**Sprint 2:​**

**Frontend**: Implemented the core functionality of the website, such as the onboarding and executing process. ​

**Backend**: Developed the DB-to-DB comparison pipeline and improved the structure of the output files. ​

​

**Sprint 3: ​**

**Frontend**: Make the website compatible with the DB-to-DB functionality.​

**Backend**: Implemented various other DBs such as Postgres, Sybase, DB2, MySql etc. to the catalog of databases our projects works on. ​

**Sprint 4:​**

**Frontend**: Created the email notifications and greatly the user experience on the website. ​

**Backend**: Conducted E2E testing on the various DBs added.​

**Sprint 5:​**

**Frontend**: Added the RAG AI Assistant (Fredrick)​

**Backend**: Deployed a RAG application and improved the Amazon Titan embeddings to cover numerous edge cases. ​

**4.2 Code Structure**

**Backend** - Stored in the MapSmart-Backend repository

* All the code for the lambda functions is stored in the prod folder within the repository

**Frontend** - Stored in the MapsmartUI repository

* The code for the react app is stored in the repository, this is the code used to deploy the frontend in AWS Amplify

**4.3 Modules & Components**

**4.3.1 Lambda Functions**

**adminCredUpload/** - Handles secure storage and retrieval of administrator database credentials in DynamoDB, masking sensitive fields like passwords and access keys when returning data to users.

**fileUpload/** - Processes multipart file uploads from both API requests and SQS messages, stores files in S3, enriches column metadata with AI-generated descriptions using AWS Bedrock, and creates DynamoDB records for tracking upload jobs.

**getLogin**/ - Authenticates users by validating username and password credentials against hashed passwords stored in DynamoDB using PBKDF2 encryption.

**infoUpload**/ - Captures and stores S3 bucket configuration information (bucket name, endpoint, access keys) in DynamoDB for administrative purposes.

**mapsmart-GetAllWorkspaces**/ - Retrieves and returns all workspace records from the MapsmartWorkspaceData DynamoDB table with proper JSON serialization handling for Decimal types.

**mapsmart-createWorkspace**/ - Creates new workspaces by accepting name and description parameters, generating unique workspace IDs, and storing them in DynamoDB with timestamps.

**mapsmart-emb-test**/ - Generates high-performance embeddings using AWS Bedrock Titan models with optimized batch processing, vectorized similarity calculations, and strategic domain-specific boosts for column mapping.

**mapsmart-output-test**/ - Processes similarity matrices from SQS messages to generate Excel mapping files with advanced features like primary key prioritization, confidence scoring, and multi-role ID handling, then uploads results to S3.

**mapsmart-rag-test/** - Implements a conversational RAG (Retrieval-Augmented Generation) chatbot that processes multipart form data with file uploads, manages session state for data mapping workflows, and triggers mapping pipelines when requirements are met.

**mapsmartBookmarkJobs/** - Updates the bookmark status of mapping jobs in DynamoDB by accepting job ID and bookmark boolean values from client requests.

**mapsmartConnection**/ - Serves as a WebSocket connection handler that simply returns a 200 status code for maintaining persistent connections.

**mapsmartDeleteJob**/ - Removes mapping job records from DynamoDB based on provided job IDs with comprehensive error handling and CORS support.

**mapsmartDisconnect**/ - Handles WebSocket disconnections by scanning and removing all connection entries associated with a specific connection ID from the WebSocketConnectionsMapsmart table.

**mapsmartEmailSender**/ - Monitors DynamoDB streams for job status changes and automatically sends email notifications to users when their mapping files are ready for download.

**mapsmartErrorHandling**/ - Provides comprehensive error processing by enriching error context, generating user-friendly messages, determining retry strategies, logging structured data, and sending responses back to SQS queues.

**mapsmartErrorLogs**/ - Processes CloudWatch log streams to extract and filter error messages, then stores structured error data as JSON files in S3 for analysis and monitoring.

**mapsmartRegisterJob**/ - Registers WebSocket connections with specific job IDs in DynamoDB to enable real-time notifications for mapping job progress.

**mapsmartVersioning**/ - Handles file versioning by processing multipart form uploads, creating backups of existing files, updating metadata in DynamoDB, and triggering downstream processing via SQS messages.

**mapsmart\_sql\_retrieve**/ - Connects to various database types (Oracle, PostgreSQL, SQL Server) using unified extractors to retrieve table metadata and column information, then formats and sends the results to SQS for further processing.

**smartmap\_column**/ - Extracts column metadata from uploaded Excel/CSV files, normalizes column names, processes both existing and new source files, and sends structured extraction results to SQS for embedding generation.

**4.3.2 Databases**

**MapsmartAdminCredentials** - Stores the data regarding DB and bucket credentials. So for example it will store the Oracle login credentials for the DB that's connected to RDS.

**MapsmartJobMetaData** - This table stores the metadata for each mapping job ran, this includes the job id, email, name, file paths and names.

**MapsmartLoginCredentials** - This table stores the data of the different users who use the application, this is required for login purposes.

**MapsmartWorkspaceData** - This table holds information regarding the different workspaces, a workspace is a section where multiple jobs can be run. Different users might have different workspaces which they use to run their jobs.

**WebSocketConnectionsMapsmart** - This holds the information regarding websocket connections, it makes it easier to automatically get the output files without needing to refresh.

**4.3.3 APIs**

**Mapsmart - API Gateway (REST API)**

| **Methods** | **Input Fields** | **Response** |
| --- | --- | --- |
| /bookmark | **Method**: POST  **Content-Type**: application/json  **Body**: {  job\_id: string,  isBookmarked: boolean  } | Typical Status 200 response, no needed information provided. |
| /chat | **Method**: POST  **Content-Type**: multipart/form-data  **Body (FormData)**:{  message: string, // context string of conversation  source\_file?: File, // optional file upload  target\_file?: File // optional file upload  } | {  response: string,  debug\_info: {  pipeline\_triggered: boolean  },  pipeline\_result?: {  success: boolean,  record\_id: string  }  } |
| /dbupload | **Method**: POST  **Content-Type**: multipart/form-data  **Body (FormData)**: {  content?: File, // file upload when available  db\_inputs: string, // JSON stringified array of database connection objects  file\_inputs?: {  field\_name: "file\_input", // Fixed field name  filename: string, // Original filename (formData.sourceFile.name or formData.targetFile.name)  role: "source" // Fixed role as "source"  },  email: string,  jobName: string,  workspace\_id: string,  sourceFilePath: string,  targetFilePath: string  }  **Possible db\_inputs json strings:**  [  {  id: string, // Source credential ID (formData.sourceDBCreds)  db\_type: string, // Database type in lowercase  database: string, // Source database name (formData.sourceDBName)  table\_name: string, // Source table name (formData.sourceTableName)  driver: "ODBC Driver 18 for SQL Server", // Fixed driver string  role: "source" // Role as "source"  },  {  id: string, // Target credential ID (formData.targetDBCreds)  db\_type: string, // Database type in lowercase (same as source)  database: string, // Target database name (formData.targetDBName)  table\_name: string, // Target table name (formData.targetTableName)  driver: "ODBC Driver 18 for SQL Server", // Fixed driver string  role: "target" // Role as "target"  }  ]  **Db\_type examples:**  "MySQL" → "mysql"  "PostgreSQL" → "postgresql"  "Oracle" → "oracle" | {  id: string, // credential ID  db\_type: string, // "mysql", "postgresql", etc.  database: string,  table\_name: string,  driver: string, // e.g., "ODBC Driver 18 for SQL Server"  role: "source" | "target"  } |
| /deleteJob | **Method**: DELETE  **Content-Type**: application/json  **Body**:{  job\_id: string  } | Typical Status 200 response, no needed information provided. |
| /getAllJobs | **Method**: GET | {  count: number,  jobs: Array<{  id: string,  job\_name: string,  email: string,  status: string, // "uploaded", "source\_updated", etc.  timestamp: string,  uploaded\_at?: string,  is\_bookmarked?: boolean,  source\_filename?: string,  target\_filename?: string  }>  } |
| /getErrorLogs | **Method**: GET  **Query Parameters**: ?date={formattedDate} (YYYY-MM-DD format) | {  files: Array<{  key: string, // error log file location  url: string // presigned download URL  }>  } |
| /getJobs | **Method**: GET  **Query Parameters**: ?workspace\_id={groupId} | {  jobs: Array<{  id: string,  job\_name: string,  email: string,  timestamp: string,  source\_filename: string,  target\_filename: string,  is\_bookmarked?: boolean  }>  } |
| /getLogin | **Method**: POST  **Content-Type**: application/json  **Body**:{  username: string,  password: string  } | {  exists: boolean  } |
| /getOutput | **Method**: GET  **Query Parameters**: ?job\_id={job.id} | {  jobs: Array<{  files: Array<{  presigned\_url: string  }>  }>  } |
| /replaceFile | **Method**: POST  **Content-Type**: multipart/form-data  **Body (FormData)**:{  job\_id: string,  file\_type: string, // "existing\_source" or "new\_source"  file\_data: File,  filename: string  } | Typical Status 200 response, no needed information provided. |
| /upload | **Method**: POST  **Content-Type**: multipart/form-data  **Body (FormData)**:{  targetFile: File,  sourceFile: File,  sourceFilePath: string,  targetFilePath: string,  email: string,  jobName: string,  workspace\_id: string  } | Typical Status 200 response, no needed information provided. |
| /uploadAdminCred | **Multiple Modes:** Fetch, Save, Update, Delete  **Request (Fetch Mode):**  **Method:** POST  **Content-Type:** application/json  **Body:**{  username: string,  mode: "fetch"  }  **Request (Save/Update Mode)**:  **Method**: POST  **Content-Type**: application/json  **Body for S3**:{  id: string,  type: "s3",  username: string,  bucketName: string,  endpoint: string,  secretKey: string,  accessKeyId: string  }  **Body for Database**:{  id: string,  type: string, // "mysql", "postgresql", etc.  username: string,  host: string,  credentialUsername: string,  password: string  }  **Request (Delete Mode)**:  **Method**: POST  **Content-Type**: application/json  **Body**:{  id: string,  username: string,  mode: "delete"  } | {  records: Array<{  id: string,  type: string,  // S3 fields or database fields based on type  bucketName?: string,  endpoint?: string,  secretKey?: string,  accessKeyId?: string,  host?: string,  username?: string,  password?: string  }>,  record?: Object // single record for Admin.js usage  } |
| /uploadSingle | **Method**: POST  **Content-Type**: multipart/form-data  **Body (FormData)**:{  job\_id: string, // The ID of the job being edited  file\_type: string, // Either "existing\_source" or "new\_source"  file\_data: File, // The new file to upload  filename: string // The name of the uploaded file  } | Typical Status 200 response, no needed information provided. |
| /uploadWorkspace | **Request (Create):**  **Method**: POST  **Content-Type**: application/json  **Body**:{  name: string,  description: string  }  **Request (Fetch):**  **Method**: GET | **Response (Fetch):**{  workspaces: Array<{  id: string,  workspaceName: string,  description: string,  createdAt: string  }>  } |

**FileNotificationSocket - API Gateway (Websocket)**

**$connect** - refers to the **mapsmartConnection** lambda function, used to connect to the socket

**$disconnect** - refers to the **mapsmartDisconnect** lambda function, used to disconnect from the socket

**registerJob** - refers to the **mapsmartRegisterJob** lambda function, used to register websocket connections

**4.3.4 Storage Buckets**

Mapsmart - Stores all uploaded files in the respective provided filepaths

Mapsmart-error-logs - Stores all error logs (errors/date/file.json)

**4.3.5 Triggers and Queues**

**Dynamo DB Trigger** - The file upload lambda function saves the job metadata and this triggers a DB trigger for new inputs, this triggers the smartmap\_column lambda function.

**SQS Message Flow Pipeline:**

mapsmart-rag-test/ → sends to RagToUpload queue

mapsmart\_sql\_retrieve/ → sends to DBExtractToFile queue

mapsmartVersioning/ → sends to MapSmartVersioningToColumns queue

smartmap\_column/ → sends to MapsmartQueue queue

mapsmart-emb-test/ → sends to EmbedToOutputMapsmart queue

mapsmartErrorHandling/ → sends to MapsmartQueue and MapsmartRetryQueue queues

Message Receivers:

fileUpload/ ← receives from SQS (processes file upload events)

mapsmart-emb-test/ ← receives from SQS (processes embedding generation requests)

mapsmart-output-test/ ← receives from SQS (processes similarity matrices for Excel generation)

**5. Testing & Quality Assurance**

**5.1 Testing Process**

We followed a path of testing each component while building it:

* Every API was tested on Postman first and then from the frontend to test if data is being transmitted from backend to frontend and vice versa.
* Every lambda function was tested immediately after development. For example, fileUpload lambda function was tested by uploading a file from frontend immediately after building the function.
* Every backend actions were being monitored by CloudWatch to catch any backend errors and to fix it as soon as possible.
* We ran around 250+ testing jobs to match outputs and mapping sheets.
* While testing the mappings we considered so many test cases like null mappings, duplicate mappings, concatenation of columns and also brief descriptions etc. This helped us to make tool more flexible to all cases and all types of inputs.
* We also set up various types of Databases to test DB to DB mappings. Like MySQL, MSSQL, PostgreSQL, Oracle DB etc.
* At the end we gathered feedback from mentors and knowledge of what quality improvements can be done to project and finally implemented those.

**5.2 Key Challenges**

* Access to AmazonQ to make a chatbot assistance for the tool took too much time and in the end we did not get access to it
* Gaining access to some services and tools for the project took more time than expected.
* Not being able to use more than one Embedding models to test mappings and enhance quality of mapping sheet
* Change in requirements in middle sprint and unable to understand requirements clearly in starting phase of project (Sprint 1,2)

**5.3 Known Issues**

* While mapping two files and columns, descriptions of the columns plays a crucial part of it. So if descriptions are incorrect or not much descriptive it can lead to incorrect mapping sheet
* In the UI, error messages for job failure is not in human readable format
* Chatbot (Fredrick) can not perform Database to Database or Database to File mapping job
* Editing jobs is possible, but it does not run the entire job again with the new files.

**5.4 Future Improvements**

* Access to AmazonQ would be very beneficial for MapSmart.
* Providing knowledge based and business logics and rules to RAG applications can make MapSmart a company specific tool.
* More AWS Bedrock embeddings models can be leveraged to enhance mapping quality even more.
* More Databases can be integrated with the tool as a requirement of a specific company or client.

**5.5 Team Notes**

* **Accomplishments** 
  + Leveraged GenAI like Titan, Several AWS Services and RAG Model
  + Created a company specific RAG Application
* **Our Stats**
  + 380+ Jobs created and tested
  + 20+ Lambda function created
  + Budget managed to be less than $150 in 3 Months

**6. Appendix**

**6.1 Access Keys**

**MapSmart Account**

* Username: admin
* Password: adminpass

**6.2 Sprint Documentation**

**Sprint 1 Tasks - Frontend**

| **Section** | **Task** |
| --- | --- |
| Dashboard | * Dashboard Page with Basic Stats |
| Execution | * Execution Page where you can view + download metadata​ * Download Output File |
| Onboarding | * Onboarding Page to create jobs​ * Support for .xlsx, .txt and .csv​ * Choose file save path, clone, view, or delete jobs |
| Admin | * Passes Admin Credentials to DynamoDB table​ |
| Other Minor Features | * 404 Page​ * Custom Error UI​ * Navbar |

**Sprint 1 Tasks - Backend**

| **Section** | **Task** |
| --- | --- |
| File Handling | * Architectured the S3 folder structure to track the input and output files * Established a working connection with the frontend * Developed the file-to-file comparison pipeline |
| Embeddings | * Created the initial version of the embedding logic for the mapping sheet |
| Final Mapping Output | * Able to output the final file-to-file mapping sheet |

**Sprint 2 Tasks - Frontend**

| **Section** | **Task** |
| --- | --- |
| Dashboard | * Grab Job Data from DynamoDB (API)​ * Add any other additional components (ex. Latest job) |
| Execution | * Downloading a file for all jobs ​ * Downloading metadata of performed jobs for the entire day ​ * Ensure View Button works for any job (API) |
| Onboarding | * Implement email notification to users for job status​ * Integrating database file uploading​ * More UI Improvements​ * Group Jobs by Workspaces |
| Admin | * Create a Lambda Function for Onboarding to validate credentials from DynamoDB (API)​ * Form Validation​ * Ensure Encryption |
| Other Minor Features | * Error Handling​ * Enhance Styling​ * Web Security​ * Job Filter/Search​ * Duplicate File Handling (API) |
| Extra | * Admin Console for multi-credentials for Databases + S3​ * Sign-on Page​ * Bookmarked Jobs |

**Sprint 2 Tasks - Backend**

| **Section** | **Task** |
| --- | --- |
| File Handling | * Handling different types of Databases (SQL, Oracle, Sybase etc.)​   + If the mapping sheet is not provided, our system will generate one itself.​ * File version handling inside the 'bkp/' folder.​ * Generate Embeddings for Database​ * Implemented Filename/Table, and format mapping |
| Embeddings | * Dynamic "threshold" and concat\_range. (on hold for now)​ * Incorporated new columns (format, etc.) |
| Final Mapping Output | * Adding the metadata columns. * Structuring the output files as per the requirements |

**Sprint 3 Tasks**

| **Section** | **Task** |
| --- | --- |
| MapSmart Webpage | * Database File Handling​ * Basic AWS Security Features​ * Job submission form changes (Making different forms for databases and files) * Credential Management on the onboarding page. * Continue working on the database form submission​ * Error Logs​ * Email Notification permissions allow being sent to all emails​ * Review Job Status |
| Amazon Q | * Implement an Interactive UI​ * Chatbot structure |
| Backend | * Handle the remaining databases (Postgres, MySQL, Sybase)​ * Database final mapping sheet output​ * Improved the mapping and embedding logic |
| Testing | * E2E testing​ * Validating SQL outputs |

**Sprint 4 Tasks**

| **Section** | **Task** |
| --- | --- |
| MapSmart Webpage | * Email Notification permissions allow being sent to all emails​ * Live Job Status​ * Enhance UI/UX for Error Logging​ * "Create a User" Page for Admin |
| RAG Model | * Implement an Interactive UI​ * RAG Chatbot |
| Backend | * Implement code to handle Sybase​ * Make further improvements (if required) to mapping logic based on SQL E2E testing |
| Testing | * Validating SQL outputs |

**Sprint 5 Tasks**

| **Section** | **Task** |
| --- | --- |
| Frontend | * Improve AI Assistant UI |
| RAG Application | * Deploy local code to the Lambda functions​ * Finish implementing the RAG application backend (intent extractor and LLM response)​ * Implement chat session state (already have FSM)​ * Integrate RAG backend with Fredrick UI​ |
| Backend | * Improvised the Amazon Titan embeddings to handle variations in output​ * Minor output formatting bugs |