# CSC 898 Report Fall 2025

Shreyas Raghuraman

## Problem Statement

The California higher education system places strong emphasis on the community college transfer pathway, enabling students to begin their academic journey at a California Community College and later transfer to a California State University or University of California campus. While this pathway is designed to be affordable and accessible, successfully navigating it is often complex and error-prone. Students must interpret articulation agreements, major-specific prerequisites, general education patterns, and campus-level transfer policies that are distributed across multiple disconnected platforms. As a result, students frequently struggle to determine which courses are transferable, which satisfy degree requirements, and how individual academic decisions impact their long-term transfer goals.

High advisor-to-student ratios significantly restrict the availability of personalized and continuous support, particularly for transfer-intending students who require guidance across multiple semesters.

Addressing this problem requires more than a static planning tool or a simple question–answer chatbot. A system designed to support transfer students must maintain awareness of a student's academic history, declared goals, and prior advising interactions while responding dynamically to evolving questions and constraints. Furthermore, because such a system operates on sensitive educational data, it must incorporate secure authentication, controlled access, and reliable data persistence to meet real-world deployment standards.

This project addresses these challenges by exploring the design and implementation of a cloud-native, agentic advising system built on Amazon Web Services. The core technical problem is to architect an end-to-end system that supports authenticated users, persistent multi-turn conversations, and flexible reasoning workflows while remaining scalable, modular, and secure. The system must reliably store and retrieve conversational state, route user requests through backend services, and evolve from rigid workflow-driven logic toward more adaptive agent-driven execution.

A central motivation of this work is to evaluate how modern AWS services such as Amazon Cognito, S3, CloudFront, API Gateway, Lambda, DynamoDB, and emerging agent frameworks can be combined to support a production-grade conversational advising application. In particular, the project examines the transition from predefined orchestration mechanisms to agent-centric reasoning models and analyzes how this shift impacts system flexibility, maintainability, and development velocity.

The goal of this semester's work is not to deliver a complete advising solution, but to establish a robust architectural foundation and a working prototype that demonstrates secure access, persistent conversation management, backend integration, and the early stages of agentic reasoning. This foundation serves as the basis for continued development toward richer advising logic, advanced tool integration, and more sophisticated conversational intelligence in future phases.

## System Overview and Architectural Design

To address the challenges outlined in the problem statement, we designed a cloud-native, serverless system architecture that supports secure access, persistent conversations, and flexible agentic reasoning. The architecture emphasizes modularity and separation of concerns, allowing individual components to evolve independently while maintaining a coherent end-to-end flow. Amazon Web Services was selected as the deployment platform to leverage managed services for scalability, reliability, and security without introducing operational overhead.

The system follows a layered architectural model consisting of a frontend access layer, a centralized API layer, an agent execution core, and a data persistence layer. Each layer is designed to handle a specific responsibility within the overall advising workflow, ensuring that user interaction, reasoning logic, and data storage remain decoupled.

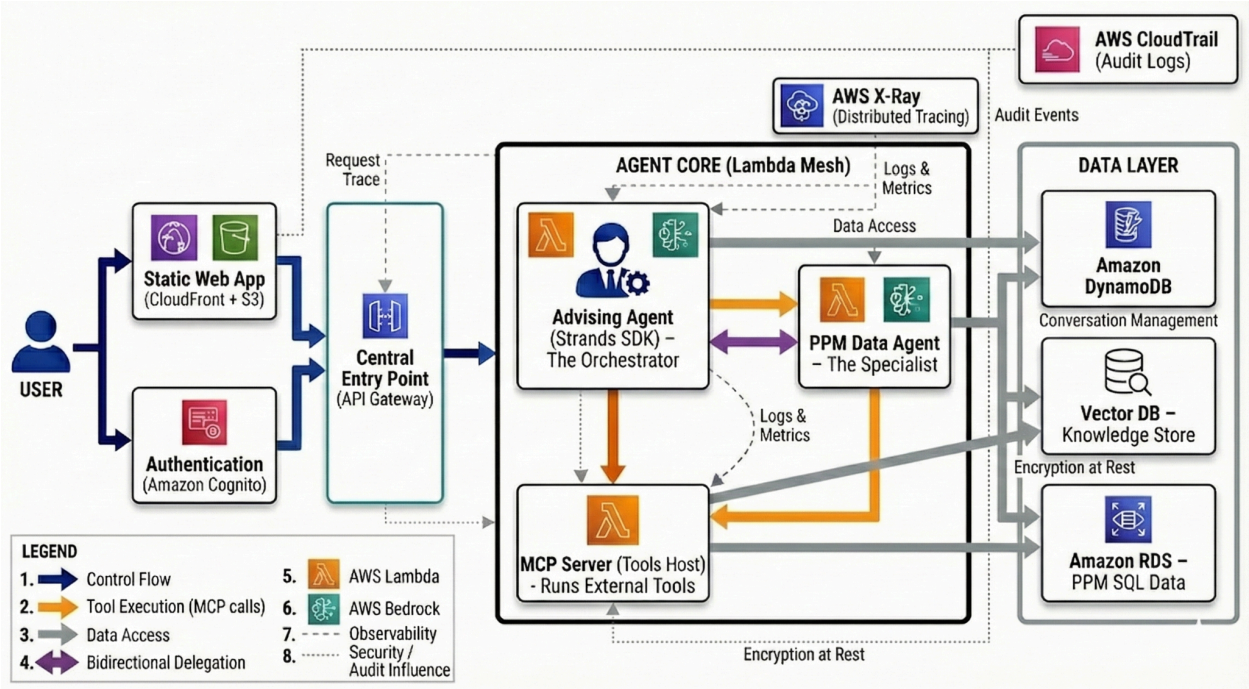


**Figure 1: High-level system architecture of the cloud-native agentic advising platform**

At the entry point, users interact with a static web application delivered through Amazon CloudFront and Amazon S3. This frontend layer provides the user interface for authentication and conversational interaction while remaining agnostic to backend implementation details. User identity and access control are handled through Amazon Cognito, ensuring that all subsequent requests are securely authenticated before entering the system.

All authenticated requests are routed through a centralized entry point implemented using Amazon API Gateway. This layer serves as the controlled interface between the frontend and backend services, enforcing authorization policies and providing a consistent request boundary.

By centralizing ingress through API Gateway, the system maintains clear observability and simplifies backend evolution.

The core reasoning functionality resides within an agent execution layer implemented as a mesh of AWS Lambda functions. At the center of this layer is an orchestrating advising agent built using the Strands SDK. This agent is responsible for interpreting user intent, maintaining conversational flow, and delegating specialized tasks to other agents or tools when required. Rather than relying on rigid, predefined workflows, the system supports dynamic delegation and bidirectional communication between agents, enabling more flexible and context-aware reasoning.

External tools are hosted through a dedicated MCP server, which allows the advising agent to invoke structured operations such as querying academic program data or performing semantic searches over knowledge stores. This design enables clean separation between reasoning logic and tool execution, improving extensibility and maintainability.

Persistent state and knowledge are managed through a dedicated data layer. Amazon DynamoDB is used to store conversation history and session metadata, allowing the system to resume advising interactions across sessions. Structured academic data is stored in relational databases, while vector-based knowledge retrieval supports semantic querying for unstructured information. All data stores are configured with encryption at rest, and access is tightly controlled through IAM policies.

Observability and auditability are treated as first-class concerns within the architecture. Distributed tracing, logging, and metrics collection are integrated across all layers to support debugging and performance analysis, while audit logs capture system-level events to ensure compliance and traceability.

This architectural design establishes a robust foundation for a production-grade advising system. By decoupling frontend delivery, agentic reasoning, tool execution, and data persistence, the system can scale to support real users while remaining adaptable to future enhancements in agent behavior and reasoning complexity.

## Frontend and Authentication Layer

The frontend and authentication layer serves as the primary interaction point between students and the advising system. This layer was designed with three core objectives in mind: global accessibility, strong security guarantees, and minimal operational overhead. To achieve these goals, the frontend is implemented as a static web application using Next.js and is deployed using managed AWS services that provide scalability and reliability without requiring server-side infrastructure.

The user interface is built using Next.js with a static export configuration, allowing the application to be generated as a collection of static HTML, CSS, and JavaScript assets. This design choice eliminates the need for server-side rendering while enabling fast page loads and

predictable behavior across environments. The application uses the App Router architecture and is written in TypeScript to ensure type safety and maintainability. Tailwind CSS is used to provide a consistent and responsive user interface across devices. Client-side state management is employed to handle authentication status and user interactions, ensuring that the frontend remains lightweight and decoupled from backend services.

For deployment, the static frontend assets are hosted in an Amazon S3 bucket configured with restricted public access. Rather than exposing the bucket directly, Amazon CloudFront is used as a global content delivery network, providing HTTPS enforcement, low-latency access, and geographic distribution. Origin Access Control is enabled to ensure that CloudFront is the only entity permitted to retrieve objects from the S3 bucket, preventing direct public access. Client-side routing is supported through custom error handling rules, allowing the single-page application to function correctly even when deep links are accessed directly through the browser.

Authentication is handled using Amazon Cognito, which provides a managed identity solution suitable for applications handling sensitive educational data. The system uses an OAuth 2.0 authorization code flow with the Cognito Hosted UI, allowing users to authenticate securely without exposing credentials to the frontend application. Cognito manages user registration, login, password policies, and account recovery, while issuing time-bound access, ID, and refresh tokens upon successful authentication. This approach ensures compliance with modern security practices while offloading identity management complexity from the codebase.

The frontend integrates with Cognito using AWS Amplify, which simplifies token handling and authentication state management. Upon login, users are redirected to the Cognito Hosted UI, and after successful authentication, they are returned to the application with an authorization code. This code is exchanged for tokens, which are stored locally and synchronized with Amplify's internal storage mechanisms. Authentication state is managed through a centralized context provider, allowing protected routes such as the dashboard and chat interface to enforce access control consistently across the application.

Once authenticated, all subsequent requests from the frontend are routed through Amazon API Gateway. Authentication tokens issued by Cognito are automatically attached to API requests and validated using a Cognito user pool authorizer. This ensures that only authenticated users can access backend services, and that all requests entering the system can be traced back to a verified identity. By enforcing authentication at the API Gateway level, backend services remain isolated from direct exposure while benefiting from centralized access control and request validation.

Overall, this frontend and authentication layer provides a secure, scalable, and cost-effective foundation for student interaction. By combining static hosting, global content delivery, and managed identity services, the system delivers a responsive user experience while ensuring that sensitive user data and conversational context are protected. This design allows the frontend to evolve independently of backend logic and supports seamless integration with the agentic reasoning components described in subsequent sections.

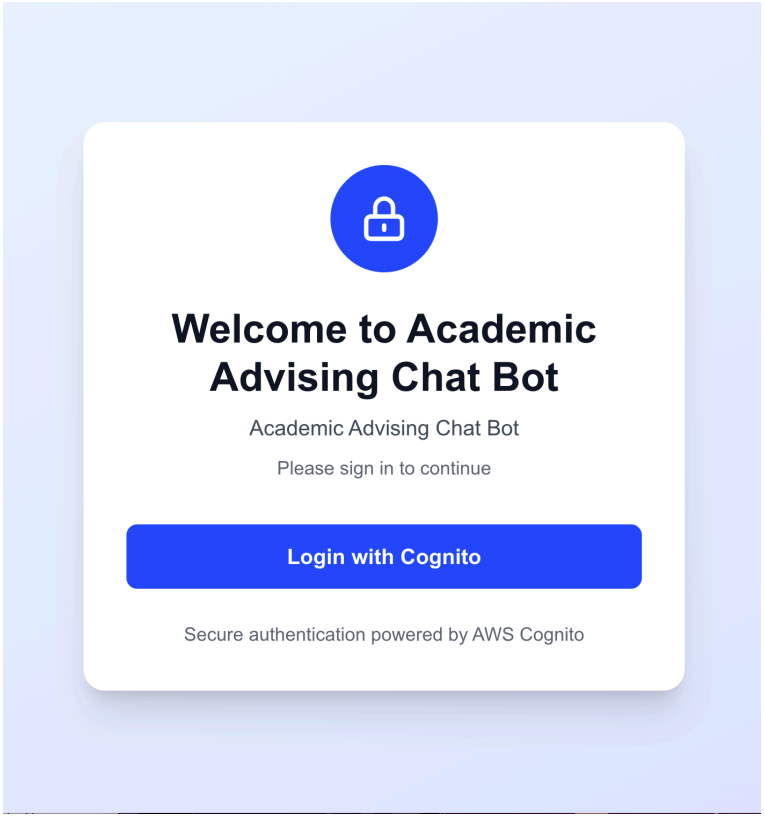


**Figure 2: Login Interface of the frontend hosted in S3 and Cloudfront**

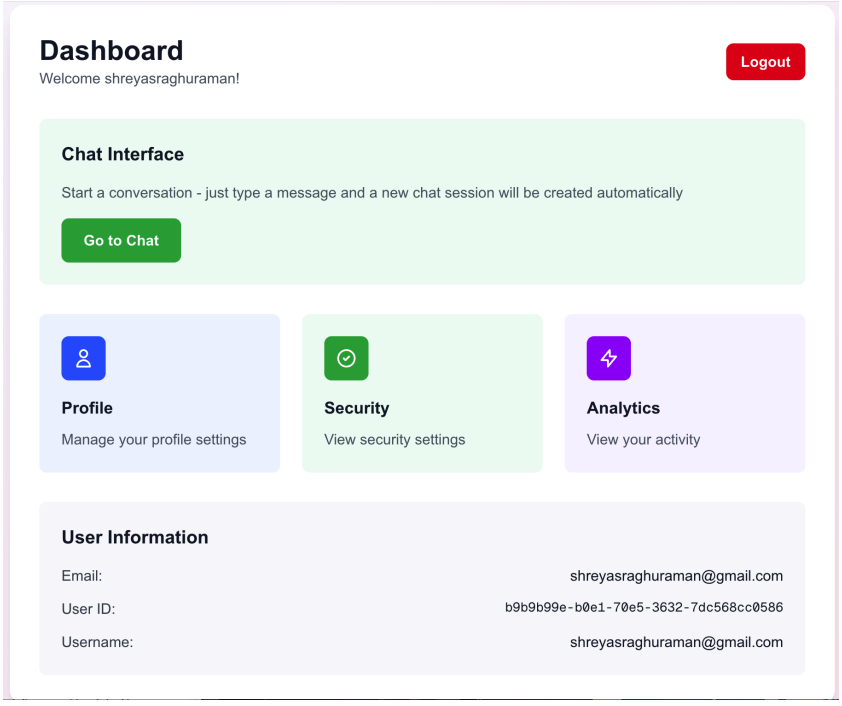


**Figure 2: Dashboard Interface of the ChatBot**

## API Gateway and Request Flow

Amazon API Gateway serves as the centralized ingress point between the authenticated frontend application and the backend serverless services that manage conversation state and agent execution. This layer plays a critical role in enforcing security boundaries, routing requests to appropriate compute resources, and maintaining a consistent interface between the user-facing application and the underlying agentic infrastructure.

The system is deployed using a regional REST API configured in the `us-west-1` region and exposed through a single production stage. All backend functionality related to conversational sessions and message handling is accessed through this API, ensuring that no Lambda functions or data stores are directly reachable from the public internet. By centralizing access through API Gateway, the system achieves a clear separation between presentation logic and backend execution while enabling observability and controlled evolution of backend services.

Authentication and authorization are enforced at the API Gateway level using a Cognito User Pool authorizer. Every request originating from the frontend includes a JSON Web Token issued during user authentication. API Gateway validates the token's signature, issuer, and expiration before allowing the request to proceed. This mechanism ensures that only authenticated users can access backend resources and that each request can be reliably associated with a verified user identity. As a result, backend services remain free from authentication logic and can instead rely on identity claims propagated through the request context.

The API is organized around a session-centric resource model that mirrors the conversational structure of the advising system. At the top level, session resources represent individual advising conversations associated with a user. Nested message resources allow retrieval and storage of conversational exchanges within a given session. This hierarchical structure provides a natural mapping between REST endpoints and the underlying data model while maintaining clarity and extensibility for future endpoints.

All API methods use Lambda proxy integrations, allowing the full HTTP request context—including headers, path parameters, and authorization claims—to be forwarded directly to the target Lambda function. This design minimizes transformation logic at the API Gateway layer and enables backend functions to make routing and processing decisions based on complete request information. Each Lambda function is responsible for a single, well-defined operation such as listing sessions, creating a new session, retrieving messages, or storing a new message.

Within each Lambda invocation, user identity is extracted from the authorization claims injected by API Gateway. This identity is used as the partition key for all data access operations, ensuring strict user-level isolation. Conversation sessions and messages are persisted in Amazon DynamoDB using a composite key design that supports efficient querying while maintaining logical separation between session metadata and message content. Time-to-live attributes are applied to records to support automatic cleanup of stale data without manual intervention.

Cross-origin access is supported through explicit CORS configuration at the API Gateway level. Preflight requests are handled using mock integrations that return the required headers, allowing modern browsers to safely issue authenticated requests from the CloudFront-hosted frontend. While the current configuration permits all origins for development flexibility, the design anticipates tighter restrictions in a production environment.

From an end-to-end perspective, a typical request follows a well-defined lifecycle. A user action in the browser triggers a JavaScript request from the static frontend. This request is routed to API Gateway, where the authentication token is validated and the request is authorized. API Gateway then invokes the appropriate Lambda function, passing along identity claims and request parameters. The Lambda function performs the required read or write operation against DynamoDB and returns a structured response. API Gateway attaches the necessary headers and returns the response to the frontend, where the user interface is updated accordingly.

This API Gateway–centric design provides a secure and scalable backbone for the advising system. By combining managed authentication, serverless execution, and a clear resource model, the system supports persistent multi-turn conversations while remaining flexible enough to accommodate future agentic capabilities. As additional agent workflows and tools are introduced, new endpoints and integrations can be added incrementally without disrupting existing functionality, preserving both system stability and development velocity.

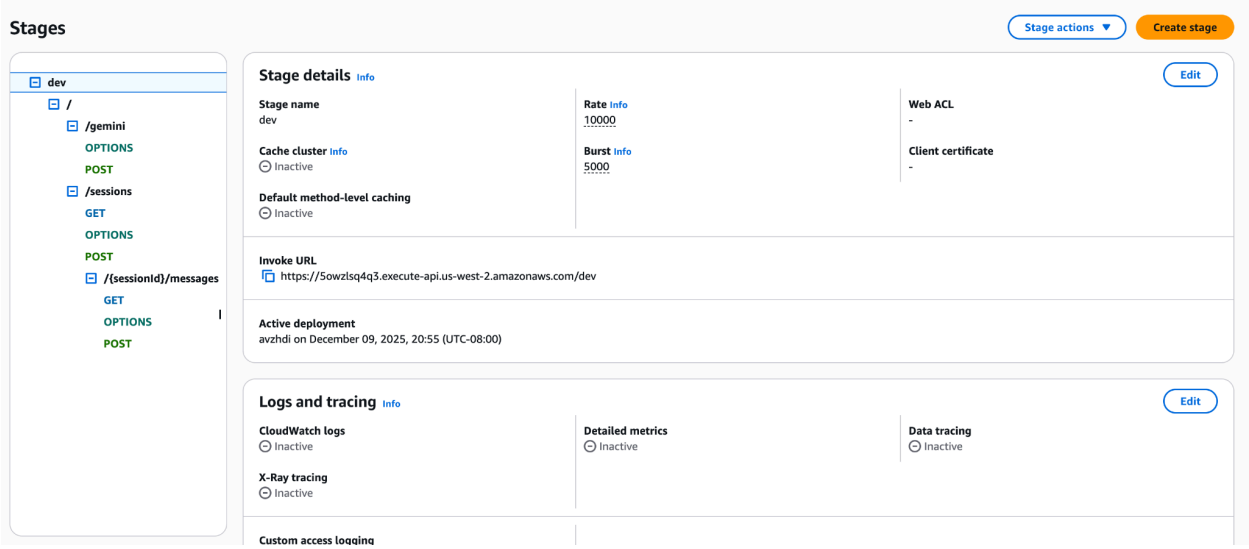


**Figure 4: All the current endpoints created in API Gateway and active in stage dev**

# Conversation Management with Amazon DynamoDB

Persistent conversation state is a foundational requirement for an academic advising system that operates across multiple sessions and semesters. Unlike stateless chatbots that discard context after each interaction, this system must reliably store and retrieve conversational history while enforcing strict user isolation and maintaining scalability. To meet these requirements, conversation state management is implemented using Amazon DynamoDB as the primary persistence layer.

DynamoDB was selected due to its fully managed, serverless nature, predictable performance at scale, and native support for flexible schema design. The system employs a single-table design pattern, allowing all conversation-related entities to be stored within a single table while supporting efficient query access patterns. This approach minimizes operational complexity and avoids the need for joins or cross-table transactions, which are ill-suited for high-throughput serverless workloads.

The DynamoDB table is configured in the same region as the API Gateway and Lambda functions to minimize latency. It operates in on-demand billing mode, allowing the system to scale automatically based on traffic without capacity planning. Server-side encryption is enabled by default using AWS-managed keys, and resource-level tagging is applied for cost tracking and management.

At the core of the design is user-based partitioning. Each authenticated user is assigned a unique partition key derived from their Cognito identity. All data associated with a user's conversations and messages is stored under this partition, ensuring that queries are naturally scoped to a single user and preventing cross-user data access. This design enforces isolation at the data model level rather than relying on application logic alone.

The sort key encodes entity type and hierarchy through structured prefixes. Conversation sessions are stored using a sort key prefixed with a conversation identifier, while individual messages within a session use a composite sort key that includes both the session identifier and a message identifier. This encoding allows multiple entity types to coexist within the same table while enabling efficient range queries based on conversation or message scope.

Conversation session records store high-level metadata such as title, optional description, creation timestamp, and an expiration timestamp. Messages store the message content, sender type, creation timestamp, optional threading reference, and a longer expiration timestamp. By separating session metadata from message content while maintaining a shared partition key, the system supports efficient retrieval of either conversations or messages without redundant data duplication.

Time-to-live attributes are applied to both conversation and message records to support automatic data lifecycle management. Conversation records are configured with a shorter retention period, while messages persist for a longer duration. DynamoDB's built-in TTL mechanism automatically removes expired items without requiring background cleanup jobs or

scheduled tasks. This approach reduces long-term storage costs and ensures that stale conversation data does not accumulate indefinitely.

Access to DynamoDB is mediated exclusively through AWS Lambda functions invoked via API Gateway. These functions extract the authenticated user identifier from the request context and use it as the partition key for all read and write operations. Importantly, the user identifier is never accepted from client input, preventing tampering or unauthorized data access. All queries and writes are therefore implicitly scoped to the authenticated user.

The primary access patterns supported by the system include listing all conversations for a user, retrieving all messages within a specific conversation, creating a new conversation session, and storing individual messages. Each of these operations is implemented using DynamoDB Query or PutItem operations that rely on exact partition key matches and prefix-based sort key conditions. This ensures constant-time partition access and avoids inefficient table scans.

Message retrieval queries are configured to return results in chronological order, enabling the frontend to reconstruct conversation history exactly as it occurred. DynamoDB's consistency guarantees are sufficient for this use case, as write operations are immediately durable and read-after-write consistency is acceptable within the conversational context.

From a concurrency perspective, the design supports multiple simultaneous users and concurrent message writes within the same conversation. DynamoDB's internal concurrency handling eliminates the need for explicit locking or synchronization logic in the application layer, simplifying the implementation while preserving correctness.

Overall, this DynamoDB-based conversation management layer provides a scalable, secure, and cost-effective foundation for persistent multi-turn interactions. By combining user-based partitioning, structured sort key encoding, and automatic lifecycle management, the system ensures that conversational state remains reliable and accessible across sessions while maintaining strong isolation and operational efficiency. This persistence

**Example DynamoDB Schema for Conversation State Management**

This section illustrates the logical schema used to store conversational state in Amazon DynamoDB. The system follows a single-table design pattern, where multiple entity types are co-located within a single table and differentiated through structured key prefixes.

**Table Name:** ChatApp
**Partition Key (PK):** User identifier derived from Amazon Cognito (cognito:username)
**Sort Key (SK):** Composite key encoding entity type and hierarchy

**Entity Type 1: Conversation (Session)**

Each conversation represents a distinct advising session associated with a single authenticated user. Conversation records store session-level metadata and act as logical containers for messages.

**Key Pattern**
PK = user_id
SK = CONV#{session_id}

**Example Item**

PK: "user@example.com"

SK: "CONV#550e8400-e29b-41d4-a716-446655440000"

title: "Fall 2025 Transfer Planning"

description: "Initial advising session for CSU transfer"

created_at: 1704067200

expires_at: 1704672000

**Field Descriptions**

- **PK:** Unique user identifier from Cognito, enforcing user-level isolation
- **SK:** Conversation identifier prefixed with `CONV#`
- **title:** Human-readable session name
- **description:** Optional session context
- **created_at:** Unix timestamp indicating session creation time
- **expires_at:** Time-to-live attribute used for automatic cleanup

Conversation records are configured with a shorter retention period to prevent accumulation of inactive sessions.

**Entity Type 2: Message**

Message records store individual conversational turns exchanged between the user and the advising agent within a session.

**Key Pattern**
PK = user_id
SK = CHAT#{session_id}#MSG#{message_id}

**Example Item**

PK: "user@example.com"

SK:
"CHAT#550e8400-e29b-41d4-a716-446655440000#MSG#123e4567-e89b-12d3-a456-426614174000"

message: "Which courses transfer to CSU for Computer Science?"

sender: "User"

created_at: 1704067210

expires_at: 1735603200

reply_to: null

**Field Descriptions**

- **PK:** Same user identifier used for all session data
- **SK:** Composite key encoding session and message identity
- **message:** Text content of the conversational turn
- **sender:** Message originator (User or Assistant)
- **created_at:** Unix timestamp for ordering messages chronologically
- **expires_at:** Time-to-live attribute with longer retention than sessions
- **reply_to:** Optional reference for message threading

**Access Pattern Labels**

The schema supports the following access patterns without table scans:

- **List all conversations for a user**
  Query by PK with SK beginning with `CONV#`
- **Retrieve all messages for a conversation**
  Query by PK with SK beginning with `CHAT#{session_id}#MSG`
- **Create a new conversation**
  Insert item with `CONV#` sort key
- **Store a new message**
  Insert item with `CHAT#{session_id}#MSG#` sort key

**Time-to-Live (TTL) Strategy**

Two independent TTL policies are applied:

- Conversation records expire after a shorter duration to remove inactive sessions

- Message records persist longer to preserve conversational context

DynamoDB automatically deletes expired items without requiring background cleanup processes.

**Schema Design Rationale**

This schema ensures:

- Strong user isolation through partition-key scoping
- Efficient query performance using prefix-based sort keys
- Automatic lifecycle management via TTL
- Support for multi-session, multi-message conversations at scale

The schema serves as the persistence backbone for stateful, multi-turn advising interactions and enables agentic reasoning components to operate with full historical context.

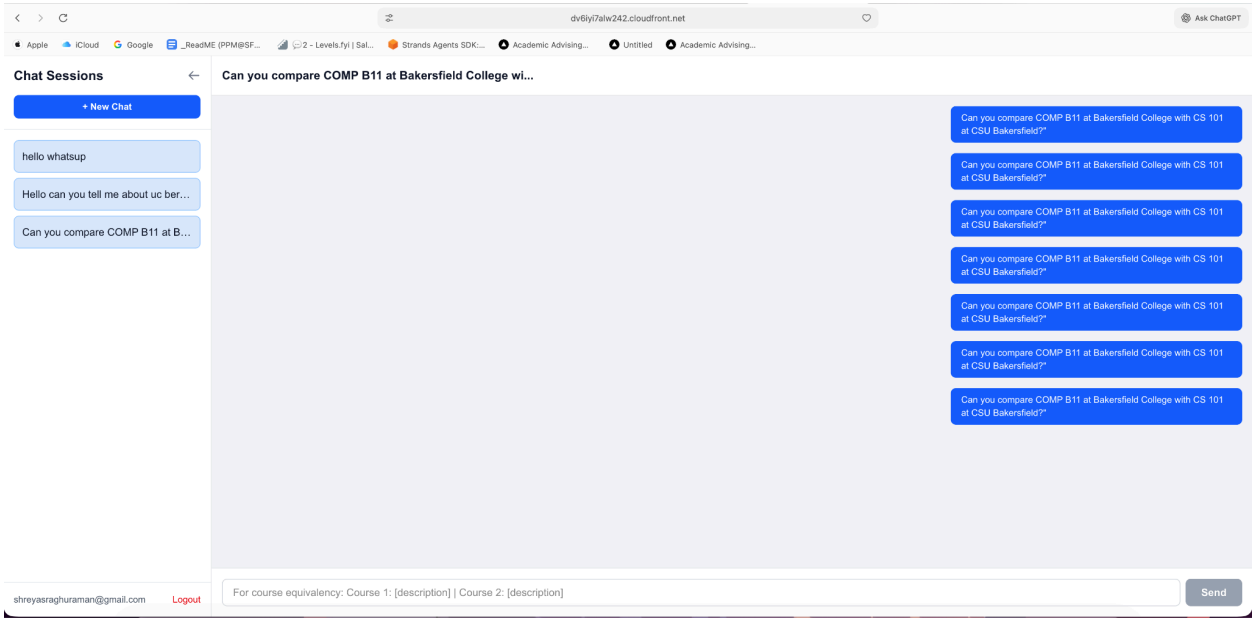


**Figure 5: Conversation Management in the ChatBot with sessions**

# Agent Structure and Current Implementation

The advising system adopts an agent-centric architecture built using the Strands SDK, with the goal of transitioning from rigid, workflow-driven logic to flexible, model-driven reasoning. Rather than encoding decision paths directly in application code, the system delegates control flow to intelligent agents capable of interpreting user intent, maintaining conversational context, and dynamically invoking tools or other agents as needed.

## Current Agent Structure

The current implementation follows a **multi-agent orchestration model** consisting of specialized agents with clearly defined responsibilities. At the center of the system is a primary advising agent, which acts as the main interface between the user and the backend reasoning infrastructure. This agent is responsible for interpreting user input, determining intent, and coordinating downstream actions.

Supporting this primary agent are additional specialized agents that encapsulate distinct reasoning or data-access capabilities. Each agent is implemented using the Strands SDK and exposed as an independent execution unit, enabling modular development and clear separation of concerns.

In the present prototype, three logical agent roles are implemented:

- **Orchestrator Agent**: Serves as the primary reasoning and coordination agent. It receives user input, identifies high-level intent (e.g., course similarity, pathway generation, general queries), and decides whether to invoke tools directly or delegate tasks to other agents.
- **PPM Data Agent**: Acts as a domain-specific knowledge agent responsible for answering questions related to academic programs, courses, and pathways. In the current implementation, this agent returns placeholder responses but is architected to support future integration with real databases and retrieval systems.

Agent-to-agent communication is implemented using Strands' native A2A (Agent-to-Agent) protocol. Each agent exposes an HTTP interface, allowing other agents to invoke it as a service. Communication follows a standardized message format, enabling structured delegation and bidirectional information flow.

## Delegation and Tool Usage Model

A key design principle in the current system is **asymmetric delegation**. The primary advising agent is permitted to invoke the secondary agent either as a tool (for scoped, single-turn assistance) or to fully delegate control for a portion of the conversation. In contrast, secondary agents do not invoke the primary agent, ensuring a clear hierarchy and preventing cyclical dependencies.

In this system, control does not always return to a single primary agent after every step. Instead, the agent that handled the previous turn determines what happens next. If the last response was produced by the PPM data agent, that agent may continue the conversation and handle follow-up actions directly, as long as the task remains within its responsibility. This allows the system to maintain continuity and avoid unnecessary handoffs between agents.

The delegation flow is therefore dynamic and context-driven. The primary agent initiates the process and decides which agent should handle a request, but once an agent is active, it can continue operating until the task is complete or requires capabilities outside its scope. Only then is control passed back or redirected. This approach keeps interactions efficient, reduces orchestration overhead, and allows each agent to operate naturally within its area of responsibility.

## Implementation Status

The current implementation establishes a fully functional agentic backbone with the following characteristics:

- Agents are implemented using the Strands SDK and run locally via FastAPI during development.
- Each agent includes a Lambda-compatible handler, enabling serverless deployment without architectural changes.
- Agent communication is operational using HTTP-based A2A messaging.
- Tool interfaces for course similarity and pathway generation are implemented with placeholder logic to validate orchestration flow.
- Session awareness is partially implemented, with conversational context preserved within agent execution but not yet fully externalized.

While the reasoning and delegation mechanisms are functional, the system currently operates on mock data and simplified tool logic. This is an intentional design choice, allowing architectural validation before integrating production data sources.

## Design Rationale

This agent structure was chosen to maximize extensibility and maintainability. By encapsulating reasoning, parameter handling, and domain knowledge into separate agents, the system avoids monolithic logic and supports independent evolution of each component. The asymmetric delegation model further simplifies control flow while enabling sophisticated reasoning patterns.

Most importantly, this structure provides a clear migration path toward more advanced agentic behavior. As additional agents, tools, or reasoning strategies are introduced, they can be integrated without disrupting existing functionality.

# Planned Evolution and Next Steps

The next phase of development will transition the current prototype into a production-ready, plug-and-play agentic platform.

Key planned enhancements include:

- **MCP-Based Tool Hosting**: Externalizing tools into dedicated MCP servers to decouple tool execution from agent logic.
- **Serverless Deployment**: Deploying each agent as an independent AWS Lambda function, enabling horizontal scaling and fault isolation.
- **Persistent Session Management**: Externalizing conversation state to durable storage, allowing agents to resume context across invocations.
- **Infrastructure as Code**: Using Terraform to provision API Gateway, Lambda functions, IAM roles, and data stores in a reproducible manner.
- **Real Data Integration**: Connecting the PPM agent to production databases and vector-based knowledge stores.
- **Adaptive Agent Routing**: Enhancing session-level logic so that agent selection and delegation are informed by prior conversational context.

Together, these steps will transform the current Strands-based prototype into a modular, cloud-native agentic advising system capable of supporting real users at scale.