AMERICAN UNIVERSITY OF BEIRUT

TA Agent Builder Project

Submitted by

Omar Ramadan, Rasha Malaeb, Zaynab Al Haj

EECE 798S: Agentic Systems

Final Project Report

Term: Fall 2025 - 2026

Date of Submission: Saturday, November 29th, 2025

Table of Contents

1. Introduction

1.1 Problem Statement

University courses generate substantial volumes of educational material: lecture slides, PDFs, scanned handouts, recorded lectures, assignments, and supplementary readings. Students routinely seek clarification on deadlines, theoretical concepts, project requirements, and exam preparation. Traditional support mechanisms such as office hours, email, and course forums present several limitations:

1. **Scalability**: Instructors cannot respond to all student questions in real time.
2. **Availability**: Office hours are constrained and may not align with students' schedules.
3. **Consistency**: Responses vary depending on the responder and the context in which the question is asked.
   Accessibility: Information is often fragmented across multiple documents and platforms.

While the rapid adoption of Large Language Models (LLMs) in educational settings has empowered professors with conversational learning assistants, current solutions, primarily ChatGPT and other general-purpose interfaces, remain insufficient. Professors must manually upload course materials into an agent, and every modification (adding, updating, or deleting files) requires repetitive re-uploads. These assistants remain tied to static, individual model instances that do not evolve with course content, resulting in inefficiency, outdated responses, and an additional layer of technical overhead.

1.2 Motivation

Three primary factors drive the need for an automated and adaptive course assistant solution:

Challenge	Description
Heavy Course Material	Each semester, professors manage large volumes of course files (slides, PDFs, assignments, scanned notes), which frequently change.
Manual Updates Are Time-Consuming	Current platforms require re-uploading documents whenever content changes or new material is added, leading to workflow friction.

Limited Technical Familiarity	Most instructors do not wish to deal with LLM configuration, prompt engineering, vector databases, or custom agent integrations.

The motivation is to bridge this gap by introducing an automated Teaching Assistant (TA) Agent Builder. Instead of relying on static uploads, instructors simply connect their Google Drive folder, allowing the assistant to autonomously sync course materials. This removes manual intervention, keeps the agent continuously up-to-date, and provides students with a reliable channel for accurate, context-grounded responses.

1.3 Objectives

The project aims to:

1. ***Simplify Agent Creation***
   Enable professors to create course-specific teaching assistant agents via an intuitive web interface, without requiring technical expertise.

2. ***Automate Document Synchronization***
   Maintain a continuously updated knowledge base through dynamic synchronization with instructors' Google Drive folders.

3. ***Provide Accurate Responses***
   Use Retrieval-Augmented Generation (RAG) to ensure that student queries are grounded in real course materials.

4. ***Support Conversation Memory***
   Enable multi-turn sessions that preserve context across interactions, improving response relevance and user experience.

5. ***Enable Monitoring and Insights***
   Provide instructors with a dashboard summarizing student queries, trending topics, and recurring areas of confusion.

6. ***Support Multiple Model Providers***
   Allow flexibility in selecting LLM providers to balance accuracy, latency, and cost based on instructors' needs.

7. ***Deliver a Public Chat Interface***
   Generate a shareable student public assistant link so learners can access support anytime, without relying on general-purpose tools.

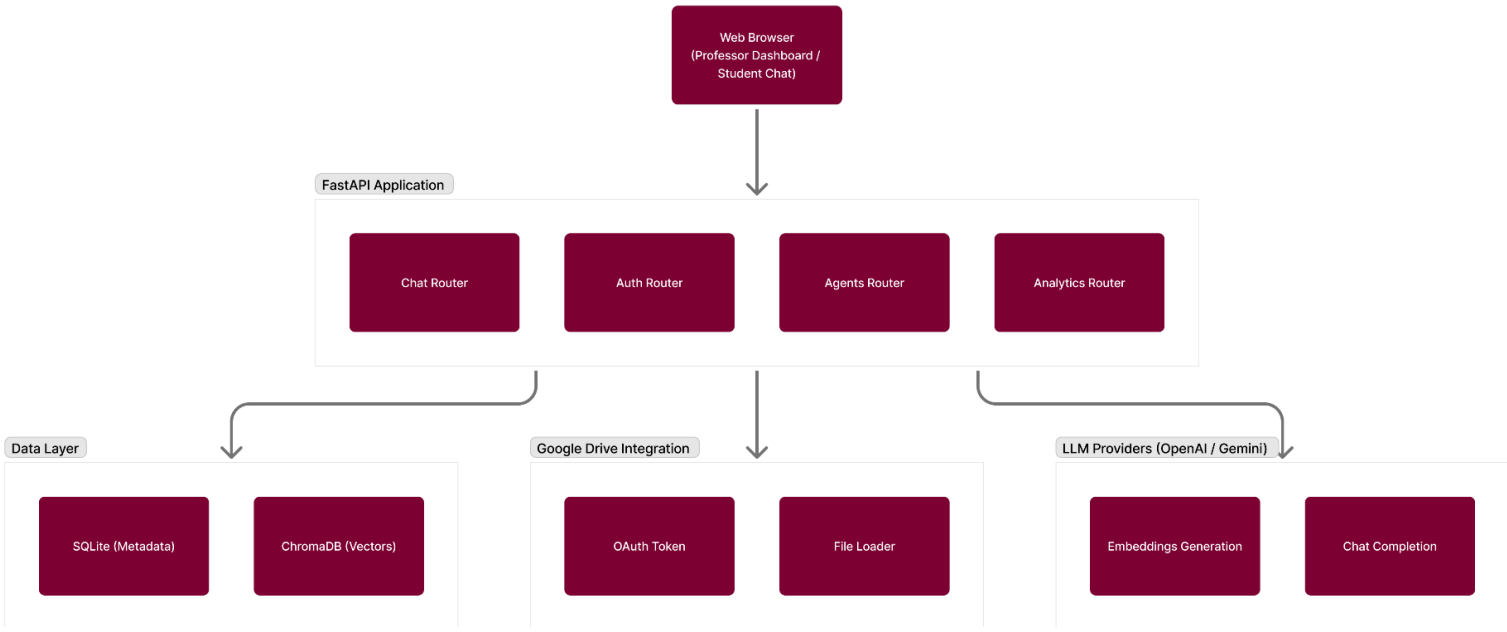
2. System Architecture

2.1 High-Level Overview

The system adopts a modular, layered architecture that separates user interaction, application logic, and data persistence. This structure improves maintainability, scalability, and extensibility.

At the top layer, the **presentation layer** consists of the web interfaces:

- A **professor dashboard** used to configure and manage teaching assistants.
- A **student chat interface** through which learners interact with the AI assistant.



The **application layer**, implemented using **FastAPI**, provides core system functionality. It handles authentication, agent creation, chat processing, analytics, and background tasks such as document synchronization. Each major feature is encapsulated in a dedicated routing module to maintain clear separation of concerns and reduce cross-module dependencies.

The **data layer** combines a relational database (SQLite) for metadata and ChromaDB for vector storage. SQLite maintains users, agents, configuration parameters, logs, and synchronization metadata. ChromaDB serves as the vector retrieval engine for the document embeddings, enabling scalable Retrieval-Augmented Generation (RAG) for student queries.

External services form an integral part of the architecture. Course materials are synchronized from **Google Drive**, while the conversational model and embedding generation are obtained from integrated LLM providers such as **OpenAI** or **Gemini**. These integrations are abstracted behind provider interfaces to allow switching or extending LLM backends without modifying core business logic.

2.2 Component Architecture

The application is decomposed into well-defined modules, each responsible for a specific domain:

Module	Responsibility
main.py	Application bootstrapping, router registration, and middleware configuration
auth.py	User account creation, login, session validation, and token management
agents.py	Creation, retrieval, and update of TA configurations; validation of agent parameters
chat.py	Multi-turn conversation management, streaming responses, and memory summarization
rag.py	Document ingestion, text extraction, embedding generation, and retrieval pipelines
providers.py	Abstract interfaces for LLMs and embeddings (OpenAI, Gemini, etc.)
analytics.py	Clustering of student queries, FAQ suggestions, and instructor insight tools
scheduler.py	Automated background jobs for document synchronization and re-indexing
security.py	Password hashing, access control, cookie/session policies
models.py	SQLAlchemy ORM database models
progress.py	Snapshotting of embedding progress to provide real-time UI feedback

Each component is designed to enable independent iteration. This modular design also supports future extensions, such as additional LLM providers or alternative storage systems, with minimal architectural disruption.

2.3 Data Flow

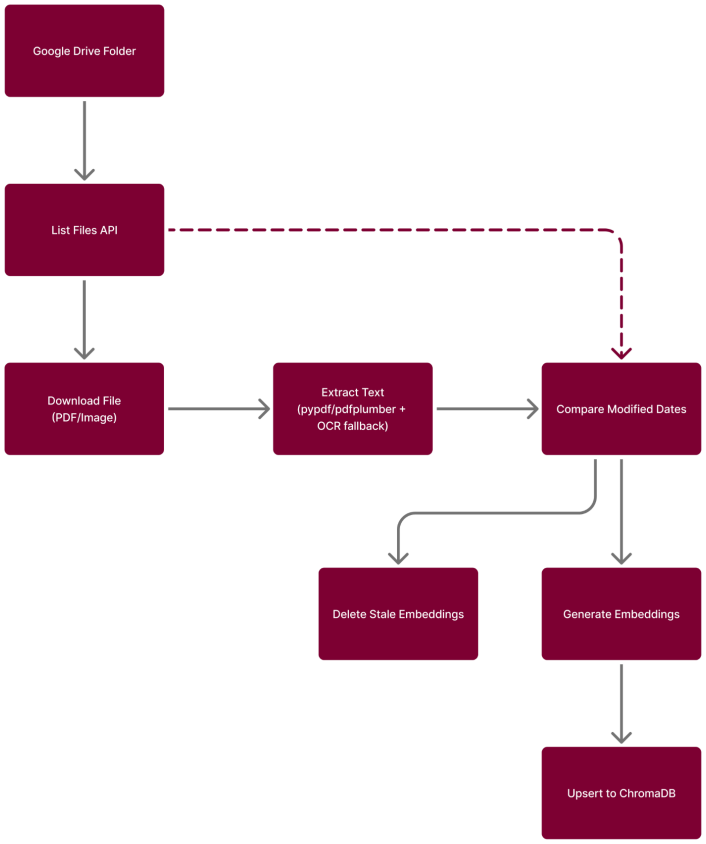
The system manages two core processes: document ingestion and query processing.

2.3.1 Document Ingestion Flow

When a professor links a Google Drive folder, the system periodically synchronizes its contents. Files are listed via the Google Drive API, downloaded as needed, and processed using PDF parsers (pypdf, pdfplumber) with OCR fallback for scanned documents. Each file is compared against its previously ingested version using timestamps and hash metadata.

- ***New files*** are embedded and inserted into ChromaDB.
- ***Updated files*** trigger replacement of existing embeddings.
- ***Deleted files*** result in removal of stale embeddings.

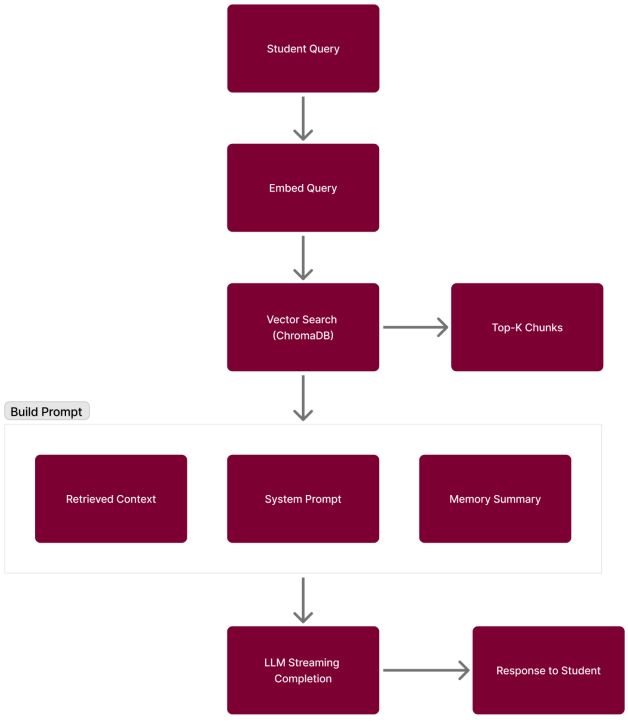
The synchronization is continuous and autonomous, eliminating the need for manual re-uploads or agent retraining.



2.3.2 Query Processing Flow

When a student submits a question, the system embeds the query and performs a **vector similarity search** against ChromaDB to retrieve the most relevant contextual passages. These passages, along with system prompts and conversation memory, are assembled into a structured prompt that ensures grounded, context-aware responses.

The selected LLM produces a streaming response that is delivered back to the student chat interface. Summarization logic maintains conversation history while preventing context overflow and maintaining model efficiency.



2.4 Professor Workflow

The platform is designed to make the creation and operation of teaching assistants accessible to instructors with minimal technical effort. The workflow proceeds as follows:

1. **Authentication and Access**
   Professors log into their account and are directed to a personalized dashboard where agents associated with their courses can be managed.

2. **Agent Configuration**
   Using a guided form, the instructor specifies the *Agent Name, Persona & Instruction Set, Model Provider & Version* and *Google Drive Authorization*.

3. **Automated Synchronization**
   Once configured, the agent automatically ingests course materials from the selected Drive folder. Subsequent revisions to slides, handouts, or documents are detected and synchronized in the background without requiring manual uploads.

4. **Agent Deployment**
   The system generates a **public student chat link**, accessible on any device. It also creates a **management interface** through which professors can review student questions, refine agent behavior, or adjust provider settings.

5. **Student Interaction**
   Students access the assistant at their convenience. Since materials remain synchronized, responses consistently reflect the current state of the course, independent of lecture pace or administrative updates.

This workflow enables instructors to deploy persistent, up-to-date, and scalable AI assistants without technical intervention, while ensuring that students receive accurate support grounded in real course content.

3. Key Features

3.1 Agent Builder UI

Before instructors begin creating or configuring their course-specific TA agents, they first access the platform through a unified authentication interface. The login and registration page provides a simple and secure entry point into the system, ensuring that each professor's agents, settings,

and Drive connections remain isolated and protected. **Figure 1** shows the main authentication screen through which users can log in or create an account.

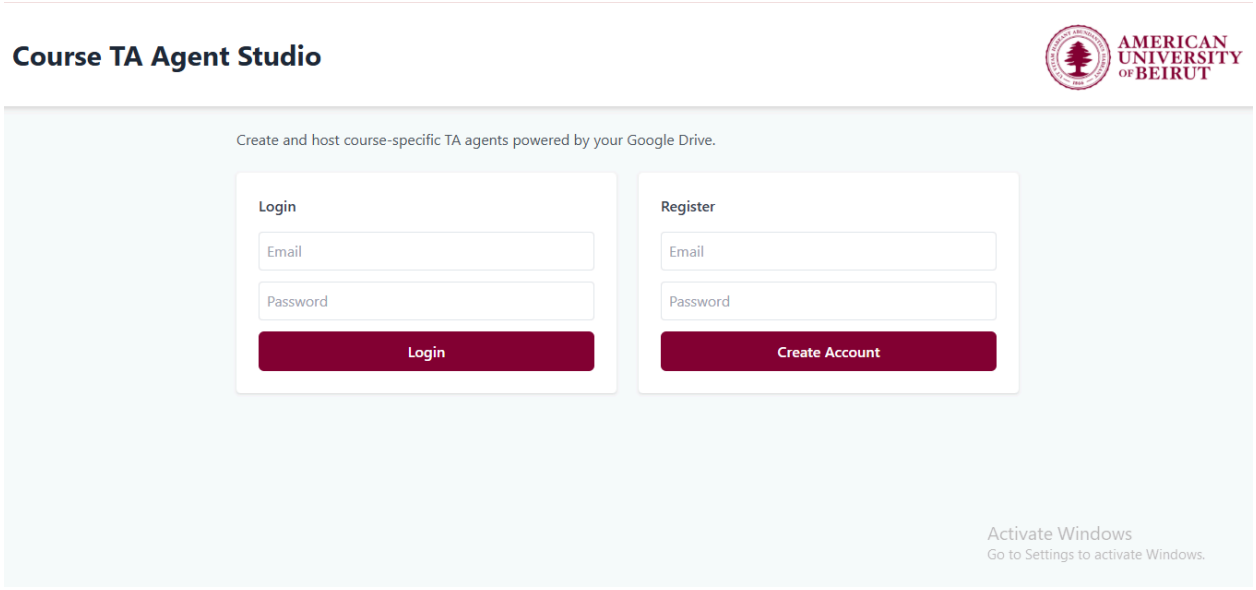


Figure 1. TA Agent Studio login interface

Once authenticated, instructors are directed to the main dashboard, which serves as the control center for creating and managing their teaching assistant agents. The dashboard includes an onboarding panel that outlines the steps required to set up an agent beginning with connecting a Google Drive folder that contains the course materials.

Figure 2 illustrates the initial dashboard state, where the system prompts the user to authorize Google Drive before agent creation can begin.

After Drive authorization, the Agent Builder interface becomes fully enabled. Instructors can now fill out the complete configuration form, specifying the agent's identity, persona, model provider, and embedding strategy. The dashboard also displays the list of existing agents, allowing easy management and updates.

Figure 3 shows the fully active Agent Builder form once Drive access has been successfully granted.

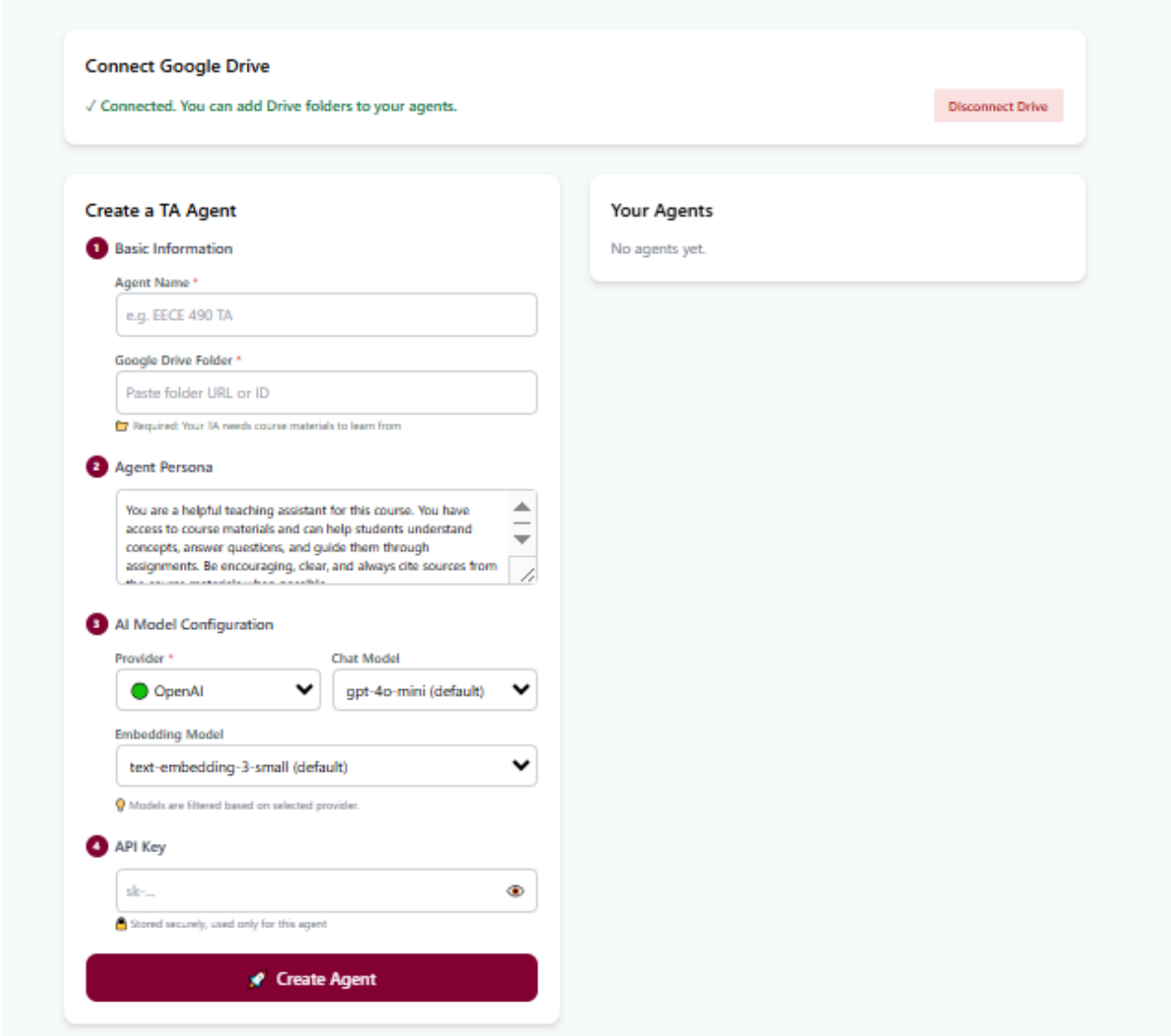


Figure 3. Fully enabled Agent Builder interface after Google Drive connection

3.1.1 Configuration Inputs

Within the Agent Builder interface, instructors define each agent using a structured set of configuration fields, including:

- ***Agent Name***: A recognizable identifier for the teaching assistant (e.g., *EECE 798S TA*).
- ***Google Drive Folder URL***: A link to the drive folder that contains the course materials.
- ***Persona and Description***: A natural language prompt describing the assistant's tone, expertise, and behavioral guidelines.

- ***Provider Selection***: Selection of the underlying model ecosystem such as OpenAI or Gemini.
  Model Configuration: Choice of both the conversational model and the embedding model to be used.
- ***API Key***: A per-agent API key that isolates usage and billing at the agent level.

3.1.2 Validation Before Deployment

To prevent misconfigured or unusable agents, the system conducts a validation pipeline prior to creation:

1. The provided API key is verified through a test embedding request to ensure credentials are valid.
   The system checks that the Google Drive folder exists, is accessible to the user, and contains files.
2. All required configuration parameters are examined for completeness.

Only when all validation checks pass is the agent successfully deployed.

3.2 Google Drive Synchronization

Course materials are maintained through a robust and automated synchronization mechanism. Once an instructor connects a Google Drive folder, the system continuously monitors its contents and reflects changes within the agent's knowledge base.

Key Capabilities

- ***OAuth Integration:*** The system employs Google OAuth 2.0 with offline access, allowing secure token refresh without repeated user authentication.
- ***Change Detection***: Documents are compared based on metadata such as modification timestamps so that only altered files are processed.
- ***Incremental Embedding****:* Files that have not changed are skipped, reducing unnecessary computation.
- ***Deletion Awareness****:* When materials are removed from the source folder, their associated embeddings are also removed.
- ***Scheduled Indexing****:* A background scheduler periodically re-evaluates the folder to ensure the assistant remains aligned with the latest course content.

This approach eliminates manual re-uploading and ensures that the student-facing assistant is always synchronized with updated slides, assignments, and lecture notes.

3.3 Retrieval-Augmented Generation (RAG) Pipeline

The system relies on a Retrieval-Augmented Generation pipeline to deliver grounded responses. Rather than relying on a model's generic knowledge, the assistant generates answers based on the instructor's actual materials.

3.3.1 Document Processing

Multiple input formats are supported, including structured text and PDFs. The pipeline first attempts native text extraction from documents. If content is not directly readable (for example, scanned documents or images embedded in PDFs), an OCR-based fallback mechanism is used. This ensures that even imperfect documents can be indexed and leveraged for retrieval.

3.3.2 Embedding Strategy

Embedding models are selected at the agent level. Instructors may opt for OpenAI's embedding models (*text-embedding-3-small*) or Gemini's embedding models (e.g., *text-embedding-004*). Embeddings are stored in ChromaDB and are associated with document chunks to facilitate fine-grained contextual retrieval.

3.3.3 Retrieval Configuration

Queries are resolved using a top-K vector search to surface the most relevant document fragments. The system applies configurable context limits to prevent excessively long responses and to maintain clarity in the generated answers. These constraints ensure that the agent remains focused, efficient, and grounded in course materials.

3.4 Memory Management

The conversational engine uses a hybrid memory strategy that balances short-term continuity with long-term contextual retention.

3.4.1 Short-Term Memory

A sliding window of recent messages is preserved in full detail and is injected directly into the prompt. This allows the assistant to understand ongoing conversation flow and respond naturally to immediate user requests.

3.4.2 Long-Term Memory

Older sections of the conversation are progressively compressed into concise summaries. These summaries capture essential facts, key decisions, and unresolved questions while avoiding raw message accumulation. When the number of active messages exceeds a defined threshold, the system generates an updated summary and marks older entries as archived. This architecture enables extended dialogue without overwhelming the model or degrading response quality.

3.5 Agent Management

Once an assistant is deployed, instructors can manage its configuration and behavior through a dedicated management page.

The system provides a **public chat link** that can be shared with students for direct access to the agent. Instructors can copy the link or open the public chat in a new tab.

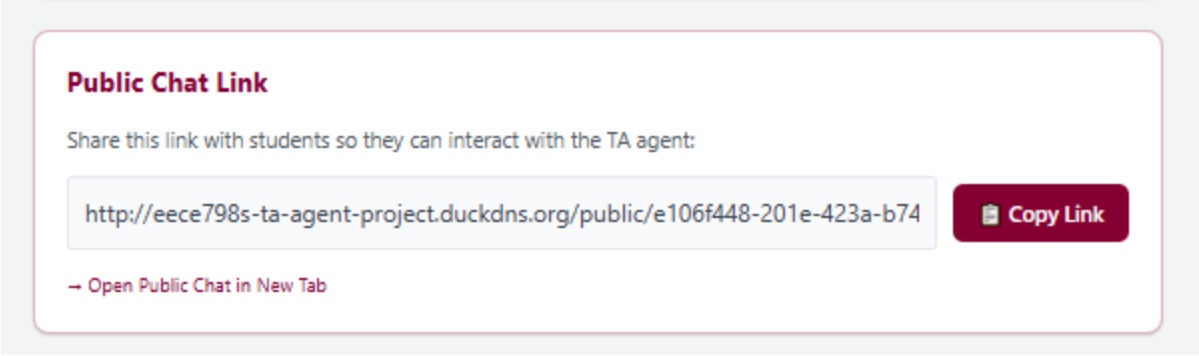


Figure 4. Public chat link and agent actions interface.

The public chat interface allows students to interact with the assistant and view any suggested questions generated from the indexed course materials.

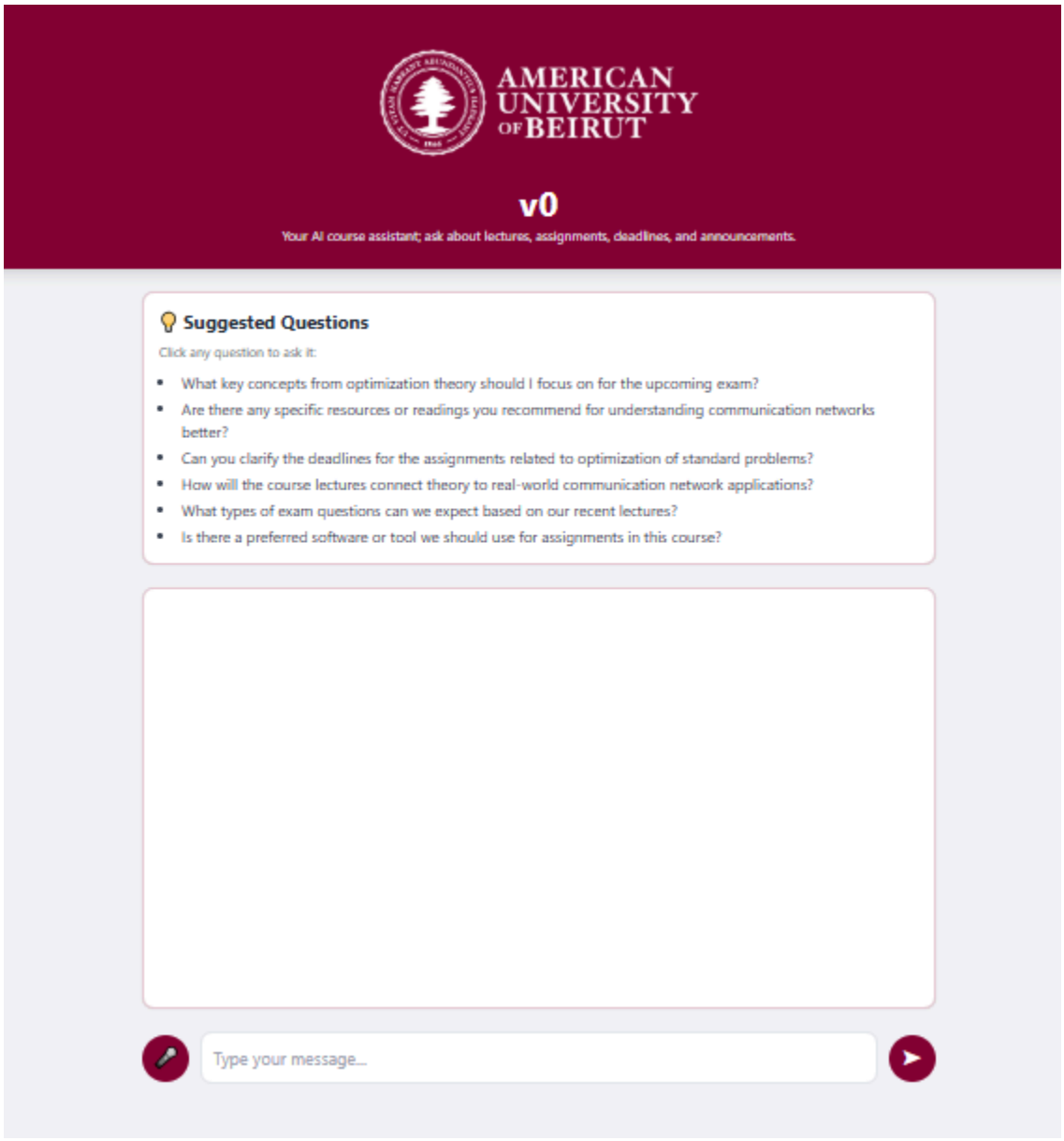


Figure 5. Student-facing public chat interface with suggested questions.

Instructors may also update the agent's settings at any time, including its name, persona, linked Drive folder, model provider, embedding model, and API key. The same page allows posting or modifying announcements that appear directly in the student chat.

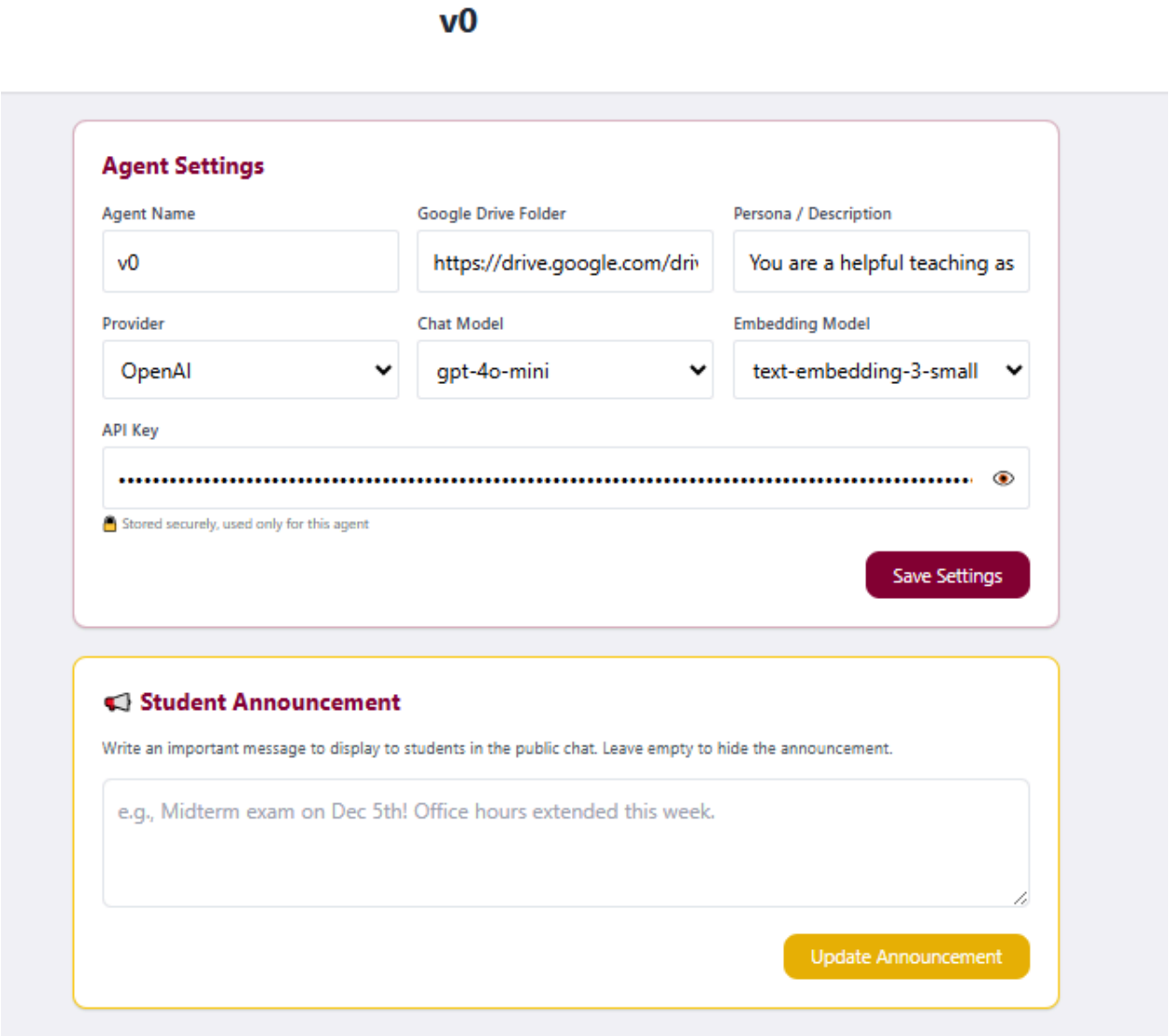


Figure 6.Agent settings and announcement management page.

Supported actions include updating agent parameters, posting announcements for students, viewing query logs, and permanently deleting the agent along with its embeddings and logs.

The announcement feature is particularly useful for time-sensitive updates such as assignment deadlines, exam reminders, or schedule changes, ensuring students receive information without relying solely on email or LMS postings.

3.6 Agent Monitoring and Analytics

The system includes analytics capabilities that provide instructors with insight into student usage trends and knowledge gaps.

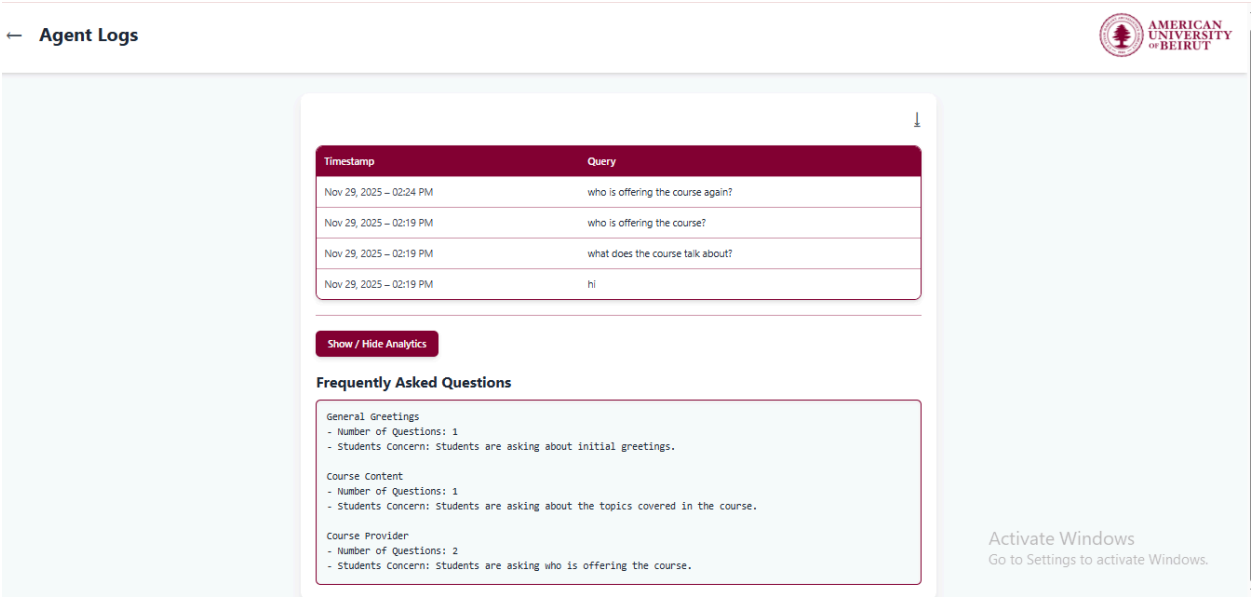


Figure 7. Query logging and FAQ generation interface.

3.6.1 Query Logging

Each student request is captured along with metadata such as the timestamp and agent identifier. This information supports historical review, auditing, and pedagogical decisions.

3.6.2 FAQ Generation

Student queries are periodically transformed into synthesized insights. Embeddings of past questions are clustered to identify recurring themes, such as exam preparation or assignment guidelines. Representative questions from each cluster are passed through a summarization model to produce instructor-friendly FAQs that highlight what students are commonly struggling with.

4. Implementation Details

This section outlines the system architecture and technical components used to develop the Teaching Assistant (TA) Agent Builder platform. The solution emphasizes modularity, reliability, and extensibility to ensure seamless integration with future enterprise-level deployments.

4.1 Technology Stack

The system is implemented using a modern, highly scalable backend architecture with strong support for asynchronous operations, secure authentication, document parsing, and LLM interaction.

Layer	Technology	Purpose
Backend Framework	FastAPI ≥ 0.104	High-performance asynchronous API service with OpenAPI auto-generation
Relational Database	SQLite + SQLAlchemy ORM	Persistent storage of users, agents, credentials, and conversational metadata
Vector Database	ChromaDB ≥ 0.4.16	Semantic indexing and embedding retrieval of course documents
LLM Providers	OpenAI, Google Gemini	Conversational inference and embedding generation
Document Processing	pypdf, pdfplumber, pytesseract, Pillow	Parsing of native PDFs and OCR for scanned documents
Google Integration	google-api-python-client , google-auth	Drive synchronization and authentication via OAuth
Background Scheduling	APScheduler	Automated document syncing and indexing tasks
Session Management	itsdangerous, starlette-session	Secure session tokens and signed cookies
Template Rendering	Jinja2	Server-side view layer for public chat interface
Testing Framework	pytest, pytest-cov	Unit and integration test coverage

The selected stack allows for both rapid prototyping and robust production deployment, while maintaining flexibility to integrate alternative LLM providers and cloud storage services in future iterations.

4.2 Database Architecture

The system follows a hybrid relational-vectorized storage strategy. Relational tables maintain user identity, agents, and interaction metadata, while course documents are embedded and indexed using ChromaDB to enable semantic retrieval.

This architecture ensures:

- ***Clean multi-tenant isolation*** (every agent belongs to a single professor).
- ***Efficient retrieval and querying*** using semantic embeddings.
- ***Auditability and observability*** via query logs and conversations.

4.3 Provider Abstraction Layer

To guarantee flexibility and remove vendor lock-in, the platform implements a unified abstraction over LLM providers. Each model provider adheres to a common interface responsible for text generation, streaming responses, and embedding computation.

4.3.1 Provider Implementations Overview

```
class ProviderBase:
  def __init__(self, model: str, api_key: str | None = None):
    self.model = model
    self.api_key = api_key

  def stream_chat(self, messages: List[Dict[str, str]]) -> Iterable[str]:
    raise NotImplementedError

  def embed(self, texts: List[str]) -> List[List[float]]:
    raise NotImplementedError

  def complete(self, messages: List[Dict[str, str]]) -> str:
    raise NotImplementedError
```

4.3.2 API Key Validation

Provider authentication is validated by performing a dry-run operation against the embeddings API:

```
def validate_api_key_with_provider(provider: str, api_key: str, embed_model: str) -> Tuple[bool, Optional[str]]:
    """
    Validate an API key by making a test embedding call.
    Returns: (is_valid, error_message)
    """
    if provider == "openai":
        return validate_openai_api_key(api_key, embed_model)
    elif provider == "gemini":
        return validate_gemini_api_key(api_key, embed_model)
    else:
        return False, f"Unknown provider: {provider}"
```

This approach prevents users from configuring non-functional agents due to invalid credentials.

4.4 Document Processing Pipeline

Course materials often include mixed-format PDFs: native text, scanned documents, and low-quality exports. The platform performs hierarchical text extraction using a two-stage pipeline:

```
def _extract_text_from_pdf(file_bytes: bytes, title: str = "") -> str:
    """
    Two-stage PDF text extraction:
    1. Fast path: pypdf.PdfReader (for selectable text)
    2. Fallback: pdfplumber + OCR (pytesseract) for scanned pages
    """
    # Attempt direct extraction
    reader = PdfReader(tmp.name)
    fast_text = ""
    for page in reader.pages:
        fast_text += (page.extract_text() or "") + "\n"

    if fast_text.strip():
        return fast_text

    # OCR fallback
    with pdfplumber.open(tmp.name) as pdf:
        for page in pdf.pages:
            extracted = page.extract_text() or ""
            if extracted.strip():
                text += extracted + "\n"
            else:
                img = page.to_image(resolution=300).original
                text += pytesseract.image_to_string(img) + "\n"

    return text
```

This ensures consistent knowledge representation even when professors upload scanned slides or handwritten documents.

4.4.1 Embedding Integration

ChromaDB expects a callable embedding function compatible with its ingestion pipeline:

```
class ProviderEmbeddingFunction:
    """Chroma-compatible embedding wrapper."""
    def __init__(self, provider: ProviderBase):
        self._p = provider

    def __call__(self, input: Documents) -> Embeddings:
        texts = list(input) if not isinstance(input, list) else input
        return self._p.embed(texts)
```

4.5 Security Architecture

Security is critical for educational platforms handling student queries, instructor credentials, and external API keys.

4.5.1 Password Hashing

User credentials are stored using bcrypt, leveraging automatic salting and irreversible hashing:

```
def hash_password(password: str) -> str:
    salt = bcrypt.gensalt()
    return bcrypt.hashpw(password.encode(), salt).decode()

def verify_password(password: str, hashed: str) -> bool:
    return bcrypt.checkpw(password.encode(), hashed.encode())
```

4.5.2 Session Control

User login sessions are managed through signed cookies:

```
signer = URLSafeTimedSerializer(SECRET_KEY)
def set_session_cookie(response: Response, user_id: int):
    token = signer.dumps({"user_id": user_id})
    response.set_cookie(COOKIE_NAME, token, max_age=86400, httponly=True)
def get_current_user_id(request: Request) -> Optional[int]:
    token = request.cookies.get(COOKIE_NAME)
    if not token:
        return None
    try:
        data = signer.loads(token, max_age=86400)
```

```
        return data.get("user_id")
    except:
        return None
```

Key security principles applied:

- No storage of plaintext credentials
- Cookie integrity through cryptographic signing
- Limiting session scope to 24-hour intervals
- Provider-level credential isolation

5. Evaluation

5.1 Methodology

The evaluation was conducted using a dedicated testing harness (eval/eval_rag_models.py) designed to measure end-to-end agent performance. The process consists of four steps:

1. ***Index Construction***
   All evaluation documents are embedded and stored separately from the production index. This ensures isolation and reproducibility.

2. ***Query Execution***
   For each evaluation question, the system retrieves the top-K relevant document chunks using semantic similarity and then generates an answer through the selected LLM.

3. ***Metric Computation***
   Generated responses are compared against ground-truth answers using accuracy, semantic similarity, and token-level scoring.

4. ***Aggregation and Reporting***
   Results are consolidated across all queries and summarized to highlight relative performance across models.

This approach closely mirrors real usage patterns, ensuring that evaluation reflects operational behavior rather than idealized benchmarks.

5.2 Golden Evaluation Set

A curated evaluation set of 50 question-answer pairs was created using actual course materials from EECE 798S. The questions span lectures on Generative AI, Large Language Models, prompting strategies, Retrieval-Augmented Generation, and supplemental instructional content.

Each entry contains the source document, the question posed, and an authoritative answer. For example, a question from the "Large Language Models" lecture asks what qualifies a model as "large." The reference answer states that models are considered large when they consist of millions to billions of parameters distributed across deep transformer layers.

This dataset represents realistic student inquiries and provides a reliable basis for measuring retrieval quality and answer correctness.

5.3 Metrics

Performance was assessed using a combination of retrieval, linguistic, and operational metrics:

- ***Top-1 Retrieval Accuracy****:* Whether the highest-ranked document corresponds to the correct source.
- ***Top-K Retrieval Recall****:* Whether the correct source appears within the K most relevant results.
- ***F1 Score****:* Token-level overlap between predicted and reference answers.
- ***Semantic Similarity***: Cosine similarity between embeddings of predicted and reference responses.
- ***Average Latency***: Mean time to respond to a user query.
- ***Token Consumption***: Total tokens generated and processed.
- ***Cost per Query****:* Estimated provider cost per evaluation question.

Collectively, these metrics capture both answer quality and operational efficiency, which are essential for real-world deployments.

5.4 Results Overview

The evaluation compared nine models: four Gemini variants and five OpenAI GPT variants across four key metrics: semantic similarity, cost per question, total token consumption, and latency. The results for each provider family are shown in Figures 9–16.

Semantic Similarity

Figure 8 shows the semantic similarity scores for all models. Gemini achieved the highest alignment with ground-truth answers, with Gemini-2.5-Flash reaching 0.71, followed by Gemini-2.0-Flash at 0.69. OpenAI models trailed slightly, ranging from 0.56 (GPT-4.1-Nano) to 0.65 (GPT-5).

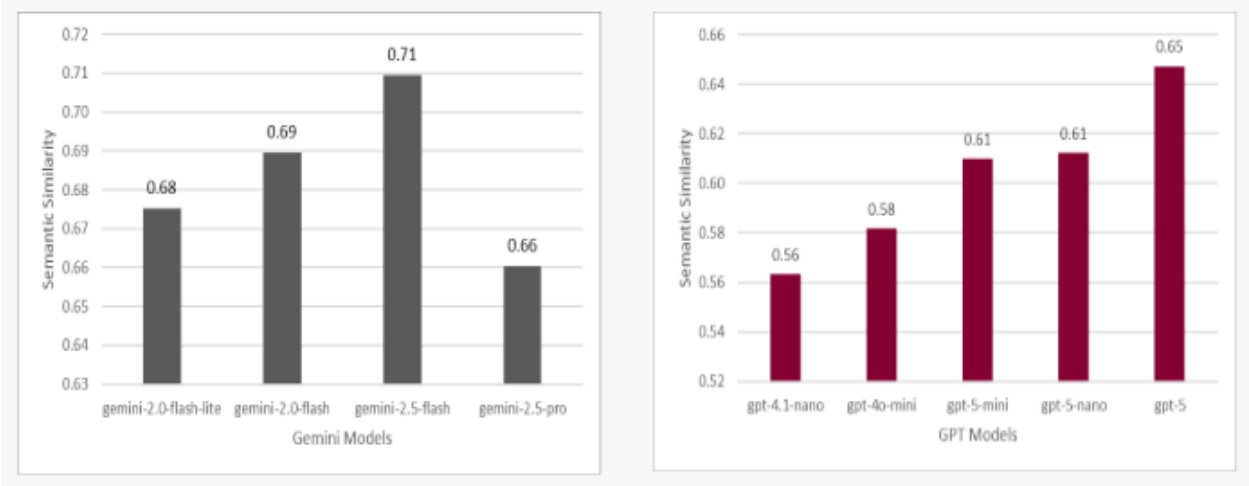


Figure 8. Semantic Similarity Across Gemini and GPT Models

Cost per Question

Cost results are summarized in Figure 9. Lightweight models exhibited extremely low costs: Gemini-2.0-Flash-Lite and GPT-4.1-Nano cost only a fraction of a cent per query. Premium models such as Gemini-2.5-Pro and GPT-5 were significantly more expensive, up to 20× higher, making them less practical for large-scale classroom deployments.

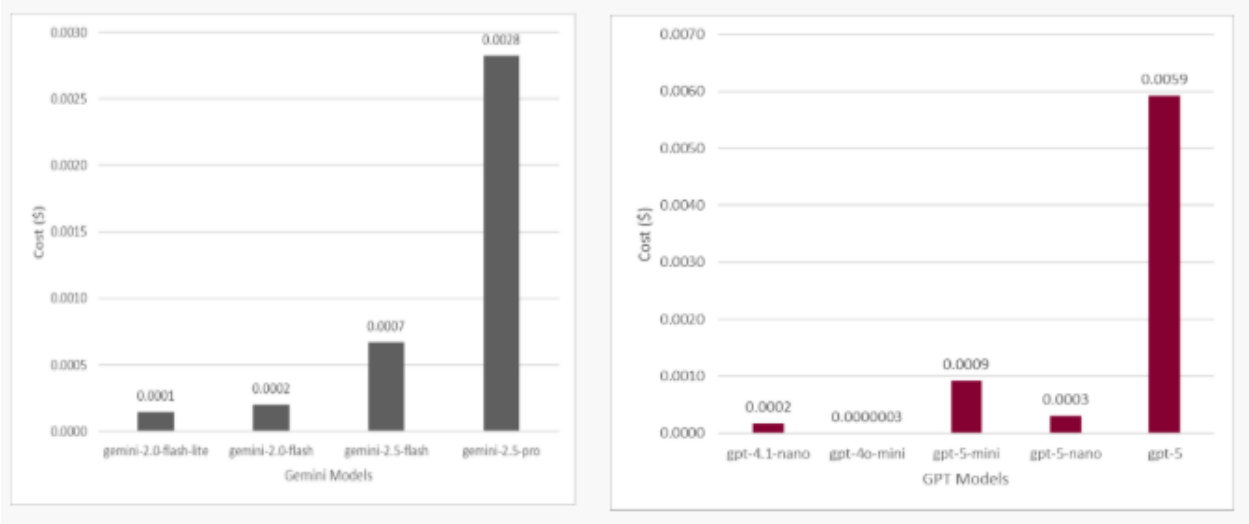


Figure 9. Cost Across Gemini and GPT Models

Total Token Consumption.

Figure 10 reports total tokens generated and processed across all evaluation queries. Smaller Gemini models stayed near **82k–83k tokens**, while OpenAI models showed greater variation, peaking at **~96k tokens** for GPT-5-Nano and GPT-5. Higher token consumption directly contributed to increased cost and latency.

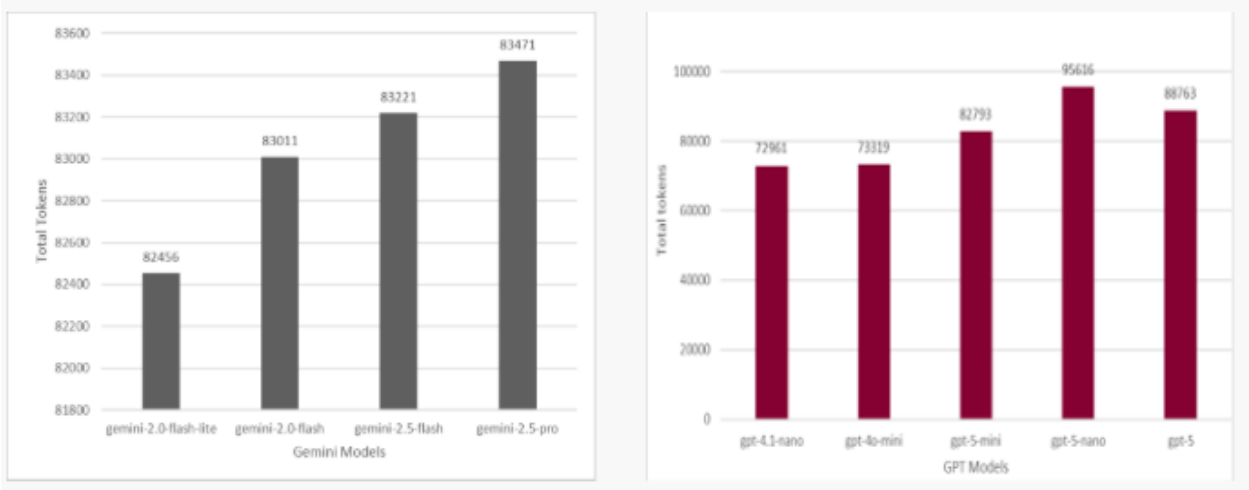


Figure 10. Total Tokens Across Gemini and GPT Models

Latency

Latency results appear in Figure 11. Gemini-2.0-Flash produced the fastest responses at 0.89 seconds, while Gemini-2.5-Pro was the slowest at 7.24 seconds. Similarly, OpenAI's smallest models responded in under 2 seconds, whereas GPT-5 exhibited the highest latency at 8.75 seconds.

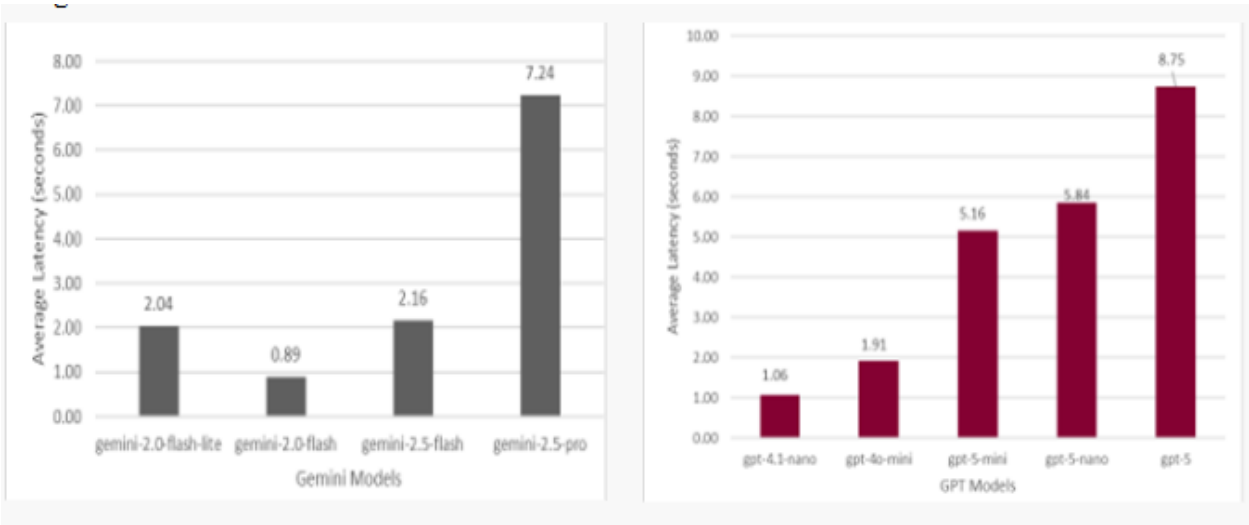


Figure 11. Average Latency Across Gemini and GPT Models

5.5 Analysis and Recommendations

The findings confirm that the RAG pipeline performs reliably across providers, with both Gemini and OpenAI achieving high retrieval accuracy. The core architecture; indexing, chunking, and semantic search; consistently guides models toward correct source material.

Gemini models exhibit stronger semantic consistency in our academic context, often producing well-structured explanations that mirror instructor-style responses. However, this advantage must be weighed against latency and cost considerations.

Smaller or mid-range models deliver the best cost-performance balance. gpt-4o-mini and gemini-2.0-flash-lite provide extremely low operational cost while maintaining acceptable accuracy. Gemini-2.0-flash is the optimal choice when response speed is critical. For quality-driven deployments, gemini-2.5-flash provides the best semantic alignment without excessive cost.

Premium models offer diminishing returns. While gpt-5 and gemini-2.5-pro are powerful, their higher latency and cost do not translate into proportional improvements in accuracy or semantic quality, making them unsuitable for scalable teaching assistant deployments.

6. Testing

6.1 Test Architecture

The project employs **pytest** as the testing framework, leveraging fixtures for dependency injection, resource cleanup, and mock management. The structure of the test suite is organized by functional domains to maintain clarity and isolation:

```
tests/
├── __init__.py           # Marks the test directory as a package
├── conftest.py           # Global fixtures (database, mocks, sample objects)
├── test_agents.py        # Tests for agent logic and configuration
├── test_chat.py          # Tests for conversation flow and memory handling
├── test_models.py        # Tests for SQLAlchemy models and relationships
├── test_progress.py      # Tests for embedding progress metrics
├── test_providers.py     # Tests for LLM provider integration
├── test_rag.py           # Tests for RAG pipeline and document retrieval
└── test_security.py      # Authentication, hashing, and session tests
```

Representative Fixtures:

```
@pytest.fixture
def test_db(test_engine):
    """Create an isolated in-memory SQLite database instance."""
    TestingSessionLocal = sessionmaker(bind=test_engine)
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

@pytest.fixture
def sample_agent(test_db, sample_user):
    """Provision a sample agent instance used across tests."""
    agent = Agent(
        owner_id=sample_user.id,
        name="Test TA Agent",
        provider="gemini",
        model="gemini-2.5-flash",
    )
    test_db.add(agent)
    test_db.commit()
    return agent
```

Fixtures are scoped to ensure that each test operates on a clean state, enabling deterministic and reproducible results.

6.2 Test Coverage

The table below summarizes test distribution across modules and the corresponding functionality covered:

Module	Tests	Focus Areas
test_providers.py	18	Provider API handling, key validation, embedding & chat endpoints
test_agents.py	24	Agent configuration, defaults, validation logic
test_security.py	14	Password hashing, authentication, session handling
test_models.py	16	Document loading, URL parsing, retrieval workflows
test_rag.py	18	Document loading, URL parsing, retrieval workflows
test_progress.py	13	Progress computation, embedding tracking
test_chat.py	18	Interaction flow, summarization, memory persistence

7. Deployment

7.1 Local Development

Prerequisites: Python 3.11+, Tesseract OCR, and a Google Cloud project with OAuth credentials.

7.2 Docker Deployment

docker-compose.yml:

```
version: '3.8'
services:
```

```
  app:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - GEMINI_API_KEY=${GEMINI_API_KEY}
      - SECRET_KEY=${SECRET_KEY}
```

Deployment Commands:

```
docker-compose up --build -d      # Start container
docker-compose logs -f           # View logs
docker-compose down              # Stop container
```

7.3 Hosting using a VPS provider:

We have deployed the docker build that we have created on DigitalOcean, an online Virtual Private Server provider that does not need extensive identity verification (unlike Hetzner which took forever to validate our identity and thus was not feasible as a deployment option due to time constraints).

We purchased a 1 GB Memory, 25 GB Disk, FRA1 - Ubuntu 24.04 (LTS) x64 virtual machine, and inside it, we were constantly cloning and fetching our GitHub repository so that we can get the newest code changes, then rebuild the docker build from the virtual machine's side.

Complemented with DuckDNS, a free dynamic DNS hosted on AWS that allows the redirection of IP addresses (i.e. 209.38.201.148 → eece798s-ta-agent-project), as well as Caddy, a lightweight web server and reverse proxy that allowed the handling HTTPS certificates (which is required for authenticating using Google Drive), we were able to host our application fully on the internet. Thus, our application becomes accessible to any admin consumer that registers and authenticates their credentials, and also used by any student with access to a public link for any agent hosted on that same virtual machine.

If you are interested to check the deployed application, please consult the following link: https://eece798s-ta-agent-project.duckdns.org/

8. Limitations and Future Work

8.1 Current Limitations

1. Agents index a single Google Drive folder; nested folders are not yet supported.
2. Public chat relies on session cookies, which can be lost if cleared.
3. Large document batches block the UI during initial embedding.
4. All agents share a single ChromaDB instance, limiting multi-tenant isolation.

8.2 Future Enhancements:

1. Recursive folder indexing for multi-level document structures.
2. Asynchronous document processing to avoid UI blocking.
3. Multi-modal RAG support for images and diagrams.
4. Role-based access control for TAs and staff.
5. Student feedback integration to improve retrieval quality.
6. Multi-language support for OCR and embeddings.

9. Conclusion

Course TA Agent Studio provides a scalable, reliable, and easy-to-use platform for deploying AI teaching assistants. Its intuitive agent creation interface, RAG-powered retrieval, and hybrid memory management enable professors to deliver persistent, accurate course support without technical expertise.

Evaluation demonstrates high retrieval performance across multiple LLM providers, with Gemini-2.5-flash achieving the highest semantic similarity (0.71) at a reasonable cost. The modular design ensures extensibility, while comprehensive testing guarantees reliability.

Overall, this project validates the effectiveness of RAG-augmented conversational agents in educational settings and establishes a foundation for future advancements in AI-assisted learning.