Marketing Content Generator Report

Course Name: Generative AI

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1	Aditi Gaikwad	EN22CS301040
2	Akhilesh Tiwari	EN22CS301085
3	Alfez Khan	EN22CS301102
4	Akshat Sakalye	EN22CS301093
5	Aditya Patidar	EN22CS301060
6	Aanchal Chaturvedi	EN22CS301014

Group Name: Group 06D1

Project Number: GAI-06

Industry Mentor Name: Aashruti Shah (Program Director, Datagami)

University Mentor Name: Prof. Ajaj Khan

Academic Year: 2025-2026

1 - Problem Statement & Objectives

1.1 – Scope of the document

This document outlines the architecture, design, and operational workflows of the "Marketing Content Generator" project. The scope of this application encompasses a locally hosted, full-stack web environment designed to automate the drafting of professional marketing copy. It details the integration of a React.js frontend, a highly performant FastAPI backend, a personalized Vector Database, and the gemini-flash Large Language Model (LLM). It covers the entire user journey, from authentication to the dynamic generation and regeneration of content based on three specific user inputs: topic, platform, and tone.

1.2 – Intended Audience

The primary users of this system are marketing professionals, digital ad agencies, and content creators who require rapid, high-quality, and platform-specific content iteration. Secondary audiences for this document include system evaluators, academic mentors, and software developers reviewing the system's technical design and GenAI integration.

1.3 – System Overview

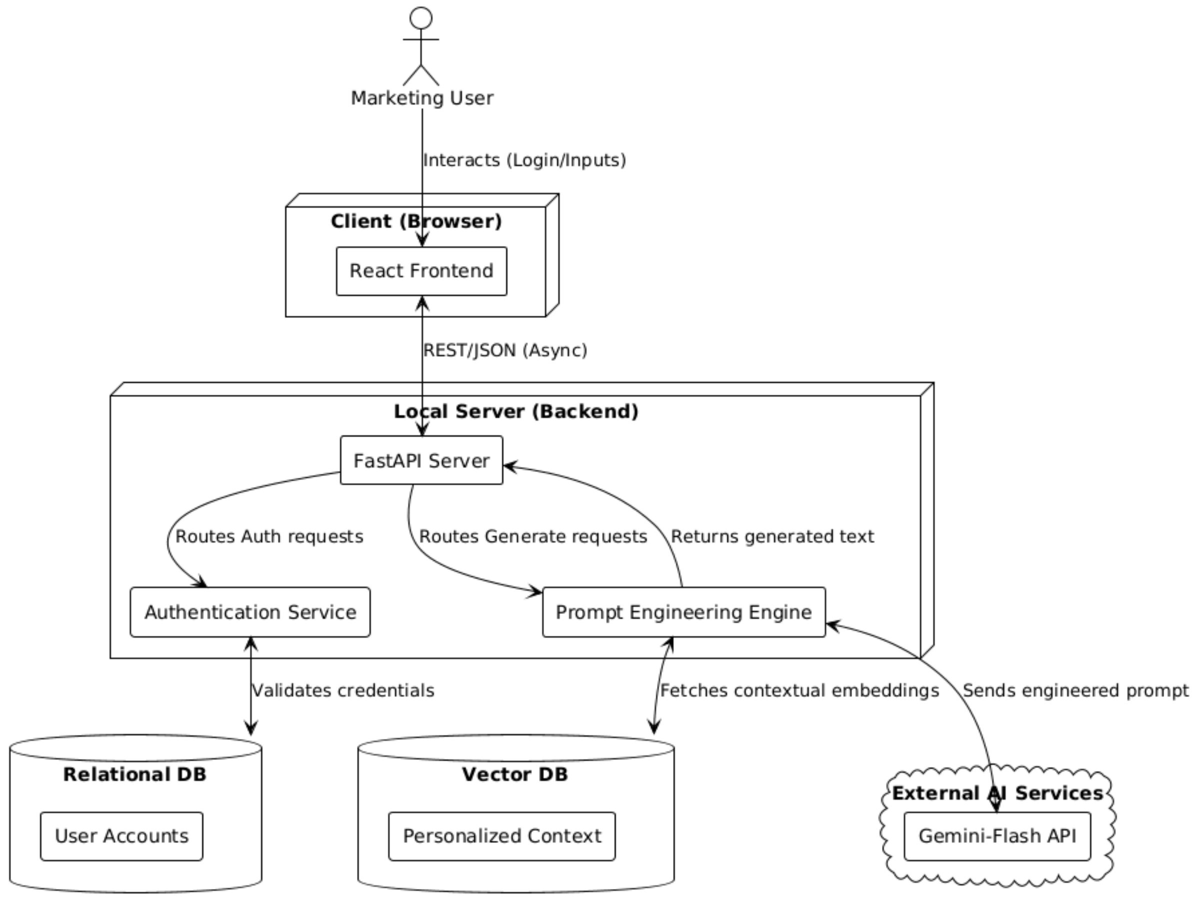
The Marketing Content Generator is a decoupled web application operating on a local server. Users navigate to a browser-based home page where they must sign up or log in. This authentication creates a personalized session linked to a Vector Database, which stores user-specific contexts and preferences. Once authenticated, users access a dashboard to input a topic, select a target platform, and define a tone. The FastAPI backend processes these inputs, constructs an engineered prompt enriched with Vector DB context, and requests generation via the gemini-flash API. The generated content is returned and displayed instantly on the dashboard, with functionality to repeatedly regenerate unique variations on demand.

2 – System Design

2.1 – Overall Architecture

The application employs a modern Client-Server architecture, heavily relying on API-driven communication.

- **Presentation Layer (Client):** A Single Page Application (SPA) built with React.
- **Application Layer (Server):** A Python-based FastAPI server acting as the central orchestrator. FastAPI was selected for its native asynchronous capabilities and high performance, making it ideal for managing LLM API timeouts and concurrent requests.
- **Data & AI Layer:** Comprises a local relational database for user management, a local Vector Database (e.g., ChromaDB/Pinecone) for semantic context storage, and the external Google gemini-flash API for generative capabilities.



2.2 – Application Design

The application is logically divided into self-contained modules to ensure maintainability:

1. **Authentication Module:** Handles the creation of user profiles, password hashing, and session token generation using FastAPI's dependency injection system.
2. **Input Collection Module (React):** A dedicated UI component featuring a form restricted to three specific fields (Topic, Platform, Tone) to ensure strict parameter control before payload transmission.
3. **Context Management Module:** Interfaces with the Vector DB to retrieve embeddings associated with the logged-in user's session, establishing the "personalized environment."
4. **Generative Module:** The core FastAPI function that concatenates user inputs with vector context, enforces prompt engineering rules, and handles the asynchronous HTTP request to the gemini-flash endpoint.

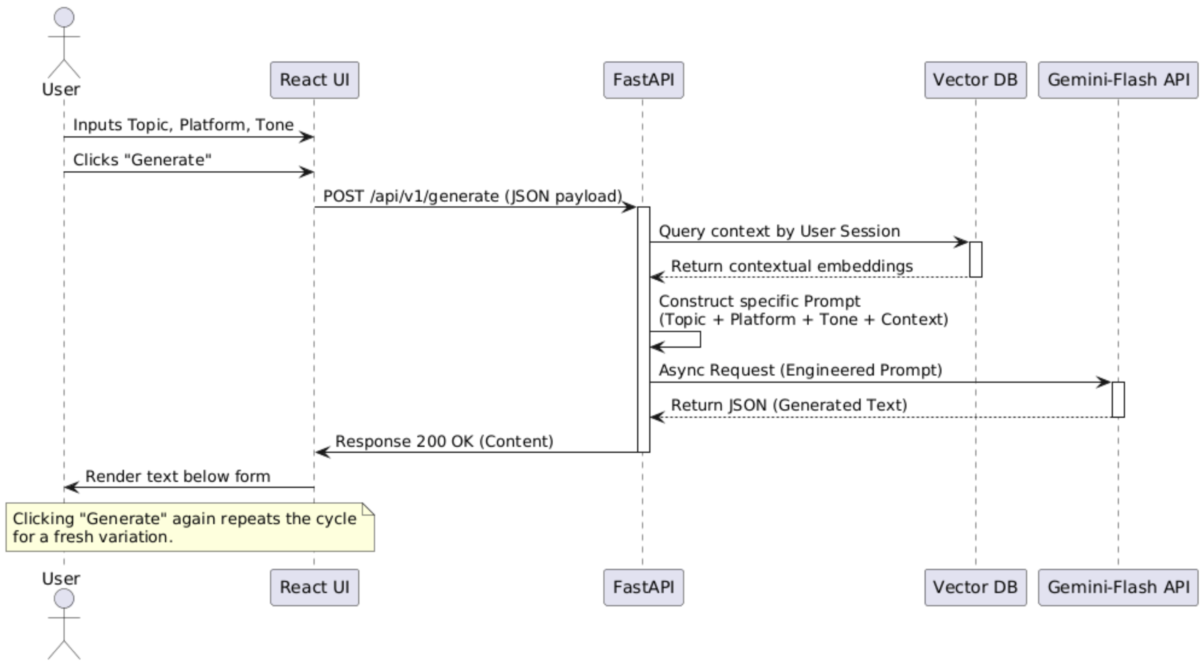
2.3 – Process Flow

The operational sequence from the user's perspective is linear, fast, and repeatable:

Initialization: The user launches the local server and navigates to the application URL in their browser.

1. **Authentication Gateway:** The user encounters the login/signup screen. Valid credentials grant access to the main dashboard and establish a personalized session.
2. **Parameter Definition:** The user fills the three required fields:
    1. *Topic:* (e.g., "Launch of new organic coffee blend")
    2. *Platform:* (e.g., "Instagram Reels")
    3. *Tone:* (e.g., "Energetic and Trendy")
3. **Execution & Processing:** The user clicks "Generate". The React frontend sends an async POST request to the FastAPI endpoint. FastAPI pulls the user's background context from the Vector DB, compiles the master prompt, and pings gemini-flash.

4. **Output & Iteration:** The generated text is parsed by FastAPI and pushed back to the React UI, displaying beneath the input form. If the user clicks "Generate" again without changing inputs, the process repeats, leveraging the LLM's temperature settings to produce a completely new, unique variation of the copy.

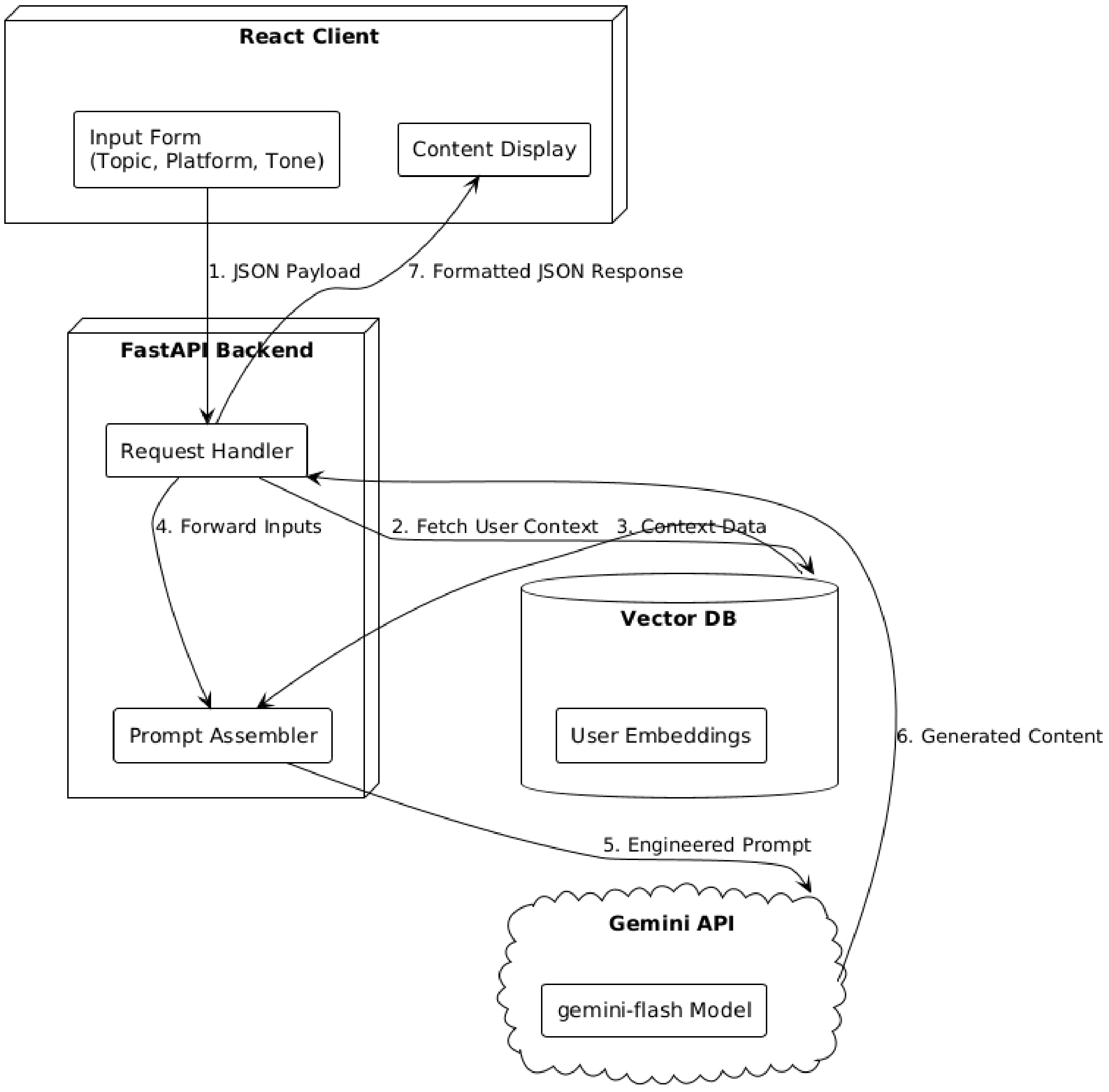

User
React UI
FastAPI
Vector DB
Gemini-Flash API
Inputs Topic, Platform, Tone
Clicks "Generate"
POST /api/v1/generate (JSON payload)
Query context by User Session
Return contextual embeddings
Construct specific Prompt (Topic + Platform + Tone + Context)
Async Request (Engineered Prompt)
Return JSON (Generated Text)
Response 200 OK (Content)
Render text below form
Clicking "Generate" again repeats the cycle for a fresh variation.

2.4 – Information Flow

The information flow dictates how user inputs and background data traverse the system to produce the final marketing copy.

1. **Input Capture:** The user submits a JSON payload containing exactly three parameters (topic, platform, tone) via the React frontend.
2. **Context Augmentation:** The FastAPI backend intercepts this payload and extracts the user's session ID. It uses this ID to query the local Vector Database, fetching historical preferences, previously successful tone markers, or saved industry jargon specific to that user.
3. **Prompt Synthesis:** A dedicated internal service merges the explicit inputs (Topic, Platform, Tone) with the implicit Vector data to create a highly structured, engineered prompt.
4. **LLM Execution:** This final prompt is transmitted securely over HTTPS to the gemini-flash API.

5. **Delivery:** The generated string is returned to FastAPI, stripped of any markdown formatting artifacts if necessary, and sent back to the React UI to be displayed directly beneath the generation form.



2.5 – Components Design:

The system is highly modularized:

- **React SPA Component:** Manages browser routing. Contains the AuthView (Login/Signup) and the DashboardView. The dashboard maintains a local state for the three input fields and a state array for the generated outputs to handle rapid regeneration.

- **FastAPI Auth Router (/auth):** Handles JWT (JSON Web Token) issuance and validation. It ensures that routes accessing the Vector DB are protected and mapped to the correct user.
- **FastAPI Generation Router (/generate):** An asynchronous endpoint (async def generate_content()) that manages the concurrent I/O operations of querying the local Vector DB and the external Gemini API without blocking the server.
- **Prompt Engineering Service:** A Python utility class containing the template logic. It ensures that instructions like "adhere to character limits for [Platform]" or "utilize a [Tone] voice" are consistently applied before hitting the LLM.

2.6 – Key Design Considerations:

- **Asynchronous I/O:** Using FastAPI's native async/await capabilities is critical. When multiple users (or one user rapidly clicking "Generate") hit the API, the local server will not freeze while waiting for the gemini-flash model to respond.
- **Data Isolation (Tenancy):** The Vector DB must strictly partition data by user ID. A user creating medical marketing content should never receive personalized context embeddings belonging to a user creating automotive marketing content.
- **Idempotency & Regeneration:** The generate endpoint is designed to be stateless regarding the LLM but stateful regarding the user. Submitting the exact same Topic, Platform, and Tone multiple times will deliberately yield different results (leveraging the LLM's natural variance) to provide the user with multiple drafting options.

2.7 – API Catalogue:

The backend exposes the following core RESTful endpoints:

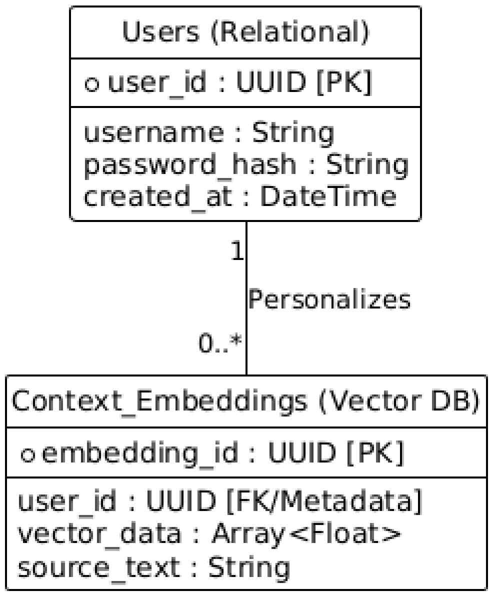
Endpoint	Method	Payload/Parameters	Description
/api/v1/auth/signup	POST	username, password, email	Creates a new user in the local relational DB
/api/v1/auth/login	POST	username, password	Authenticates user and returns a Bearer Token

/api/v1/generate	POST	topic (str), platform (str), tone (str)	Require Auth Token. Fetches vector context, calls gemini-flash, and returns generated text.

3 – Data Design

The data architecture utilizes a hybrid approach, combining a traditional relational database for structured transactional data and a specialized vector database for unstructured semantic data.

3.1 – Data Model:



1. **Relational Model (SQLite):**
    a. *Users Table:* Stores user_id (Primary Key), username, and hashed_password.
    b. *History Table (Optional/Audit):* Stores generation_id, user_id (Foreign Key), topic, platform, tone, and generated_output.
2. **Vector Model (ChromaDB/Pinecone):**
    a. *Embeddings Collection:* Stores the multi-dimensional vector representation of a user's successful past content or custom instructions.

Indexed by a unique embedding_id and tagged with metadata including the user_id to ensure strict personalization boundaries.

3.3 – Data Access Mechanism:

- **Relational Data:** Accessed using an Object-Relational Mapper (ORM) like SQLAlchemy within the FastAPI environment. This provides SQL injection protection and simplifies database queries into Python objects.
- **Vector Data:** Accessed via the official Python client library for the chosen Vector DB (e.g., chromadb package). Queries are executed using similarity search algorithms (like Cosine Similarity) filtered by the authenticated user's ID.

3.3 – Data Retention Policies

Because the system is deployed on a local server, data retention is managed directly by the host machine.

- User account credentials persist indefinitely until manually deleted.
- Vector embeddings persist to continually refine the personalized environment. If the user requests a "reset" of their environment, the associated vector records tagged with their user_id are dropped.

3.4 – Data Migration

- For the relational database, tools like Alembic are used alongside SQLAlchemy to track schema changes (e.g., adding a new field to the user table) without losing existing data.
- For the Vector DB, if the embedding model is upgraded in the future, a Python migration script will be required to read the original source_text, re-embed it using the new model, and overwrite the existing vector_data arrays.

4 – Interfaces

The Marketing Content Generator utilizes three primary categories of interfaces to ensure seamless communication between the user, the local server, and external AI services.

4.1 – User Interface (UI)

Built with React, the UI is a Single Page Application (SPA) designed for minimal friction.

- **Authentication View:** Clean forms for user Sign-Up and Login, managing the gateway to the personalized environment.

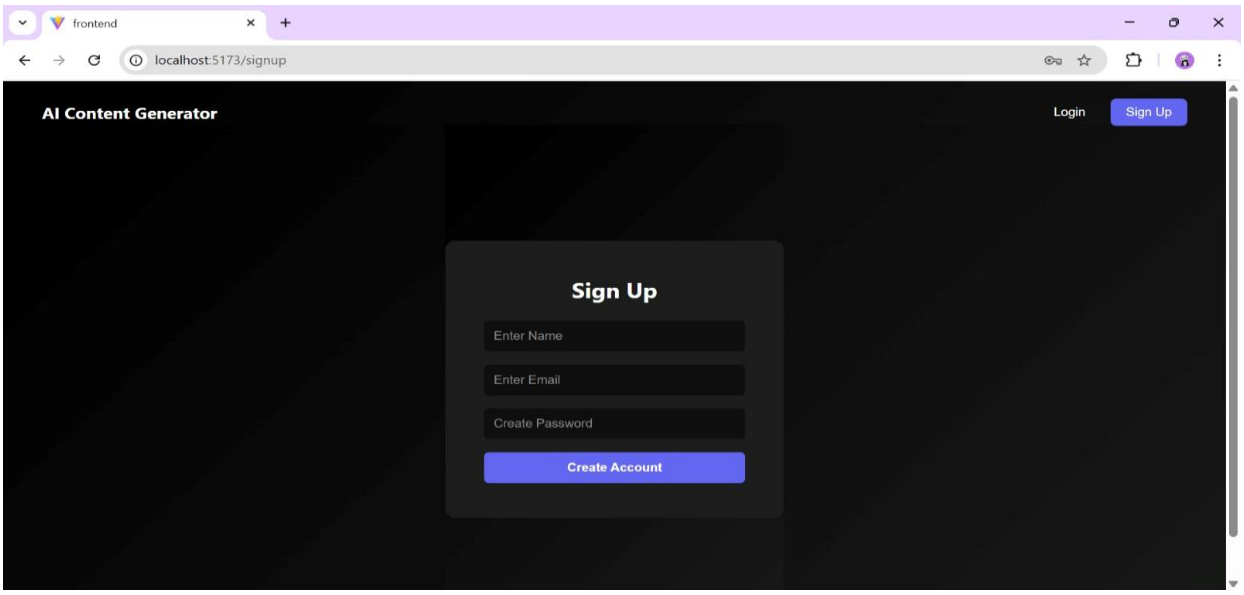


Fig. Screenshot of Sign Up Page

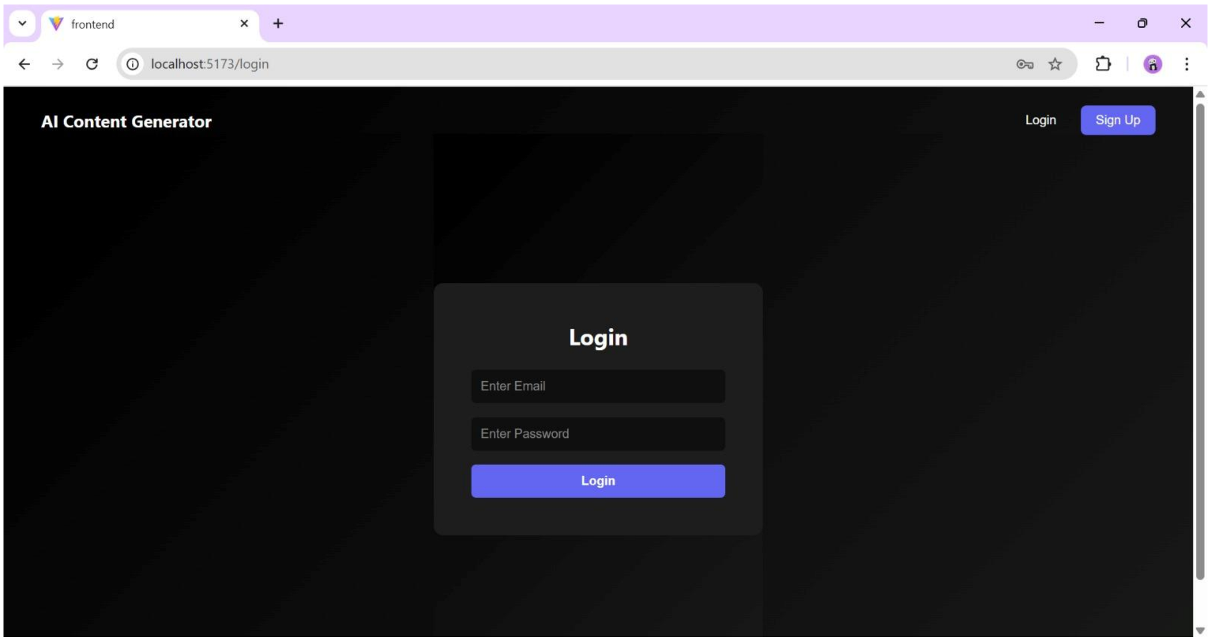


Fig. Screenshot of Login Page

- **Generator Dashboard:** The primary workspace. It features exactly three input fields—Topic (text input), Platform (dropdown or text), and Tone (dropdown or text). Directly beneath this form is a dynamic display area where the generated text renders. A prominent "Generate" button triggers both the initial creation and subsequent regenerations.

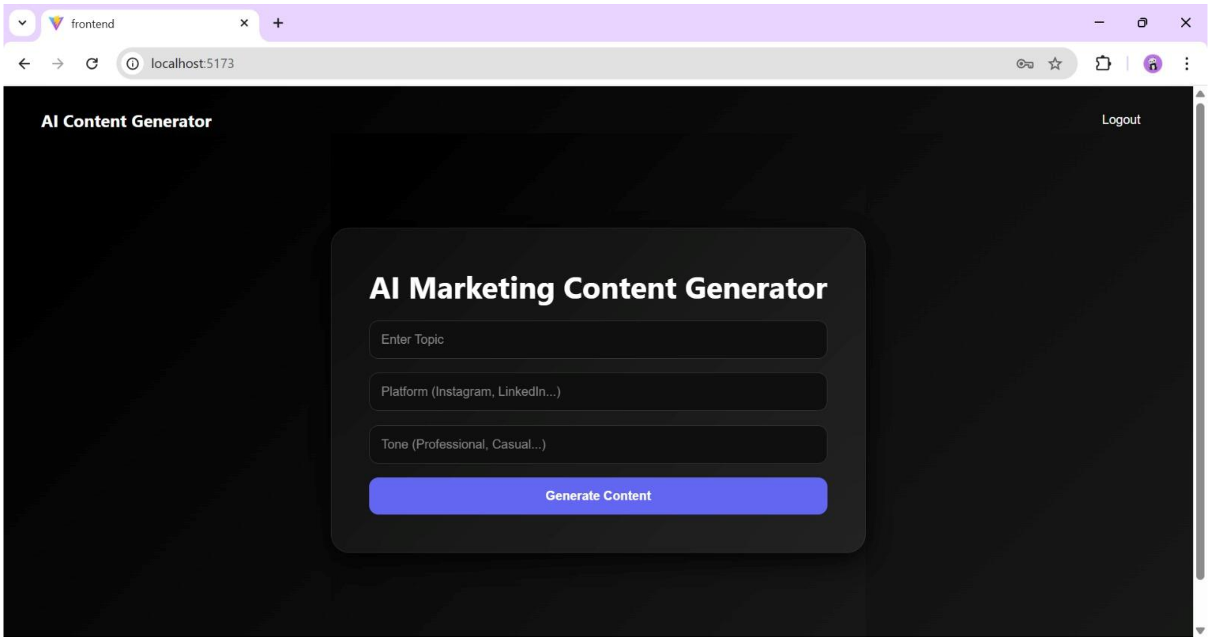
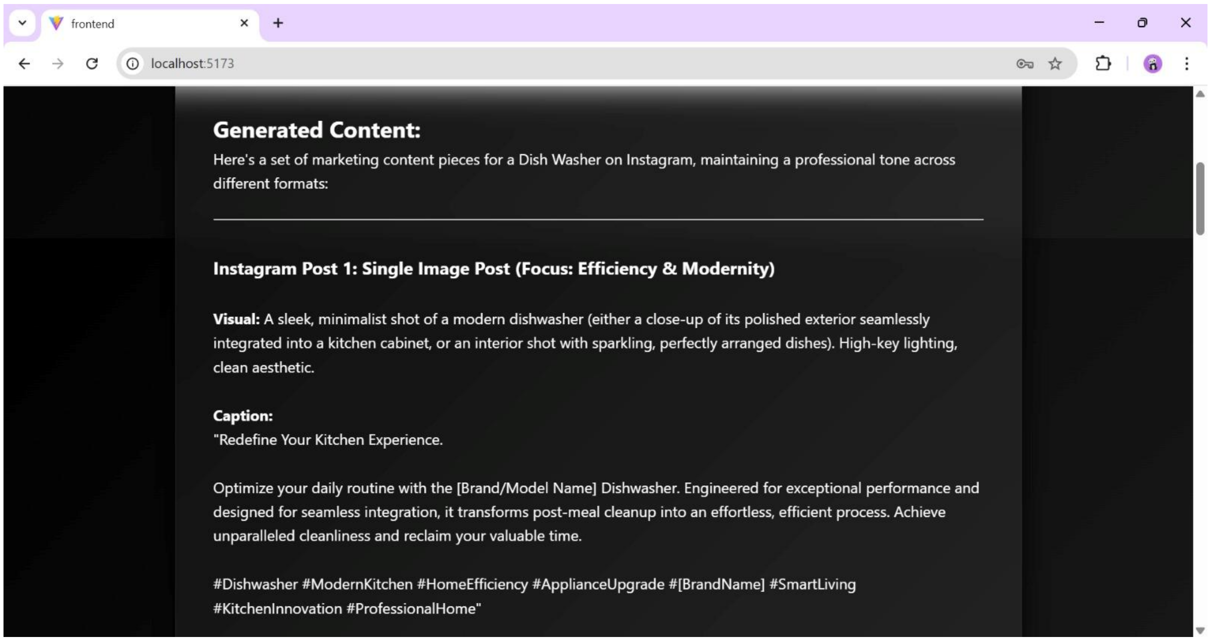

frontend
localhost:5173
AI Content Generator
Logout
AI Marketing Content Generator
Enter Topic
Platform (Instagram, LinkedIn...)
Tone (Professional, Casual...)
Generate Content

Fig. Screenshot of Home Page



Generated Content:
Here's a set of marketing content pieces for a Dish Washer on Instagram, maintaining a professional tone across different formats:

Instagram Post 1: Single Image Post (Focus: Efficiency & Modernity)

Visual: A sleek, minimalist shot of a modern dishwasher (either a close-up of its polished exterior seamlessly integrated into a kitchen cabinet, or an interior shot with sparkling, perfectly arranged dishes). High-key lighting, clean aesthetic.

Caption:
"Redefine Your Kitchen Experience.

Optimize your daily routine with the [Brand/Model Name] Dishwasher. Engineered for exceptional performance and designed for seamless integration, it transforms post-meal cleanup into an effortless, efficient process. Achieve unparalleled cleanliness and reclaim your valuable time.

#Dishwasher #ModernKitchen #HomeEfficiency #ApplianceUpgrade #[BrandName] #SmartLiving #KitchenInnovation #ProfessionalHome"

Fig. Screenshot of Generated Output

4.2 – Application Programming Interfaces (API):

The FastAPI backend acts as the central API gateway for the local system.

- **Internal REST API:** The React frontend communicates with FastAPI exclusively via HTTP methods (POST) using standard JSON payloads. This decoupled interface ensures the frontend and backend can be modified independently as long as the JSON contract remains intact.

4.3 – External Interfaces:

- **LLM API Integration:** The backend communicates with Google's gemini-flash model over secure HTTPS. The interface is managed via the official Google GenAI Python SDK or direct REST calls, authenticating securely using an environment-stored API key.

5 – State and Session Management

Maintaining the correct state is critical for a personalized and responsive user experience.

5.1 – Frontend State Management (React):

The React application manages short-term, UI-level state:

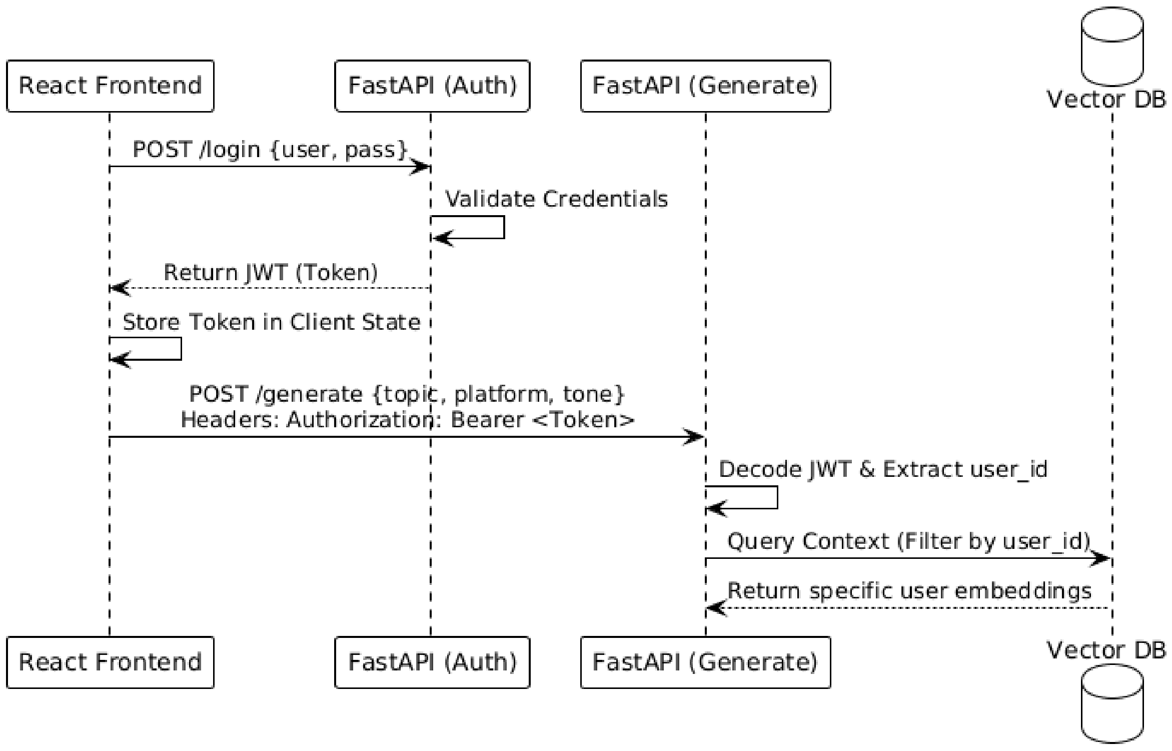
- **Form State:** Captures the current string values for Topic, Platform, and Tone.
- **Output State:** Holds the string of the generated content returned by the API.
- **Loading State:** A boolean toggle that triggers visual feedback (like a spinning loader) while the system waits for the gemini-flash API to return the response.

5.2 Backend Session Management (FastAPI):

Because the system requires a personalized Vector Database environment for each user, the application is strictly authenticated.

- **JWT Authentication:** Upon successful login, FastAPI generates a JSON Web Token (JWT). The React frontend stores this token (e.g., in localStorage or memory) and includes it in the Authorization header of every subsequent request.
- **Stateless Personalization:** While the REST API remains stateless (meaning the server doesn't hold the user's connection in memory), the JWT contains the

user_id. FastAPI uses this ID on every /generate request to isolate and query only the vector embeddings belonging to that specific user.


React Frontend
FastAPI (Auth)
FastAPI (Generate)
Vector DB
POST /login {user, pass}
Validate Credentials
Return JWT (Token)
Store Token in Client State
POST /generate {topic, platform, tone}
Headers: Authorization: Bearer <Token>
Decode JWT & Extract user_id
Query Context (Filter by user_id)
Return specific user embeddings

6 – Caching

In many web applications, caching is used to store the output of expensive API calls. However, because a core requirement of the Marketing Content Generator is the ability to hit "Generate" multiple times to yield *new* content variations for the exact same inputs, caching LLM responses is intentionally restricted.

- **Prompt & Context Caching:** Instead of caching the final output, the FastAPI server implements an in-memory LRU (Least Recently Used) cache for the local Vector DB queries. If a user rapidly regenerates content, the system caches their specific vector embeddings for a short duration. This skips the database read operation on subsequent clicks, speeding up the prompt engineering phase before the gemini-flash call.

- **No Output Caching:** To preserve the dynamic variability of the LLM, the final engineered prompt is always sent to the gemini-flash endpoint, guaranteeing a fresh piece of marketing copy on every generation cycle.

7 – Non-functional Requirements

7.1 – Security Aspects:

Operating a local server handling AI request requires strict security measures:

- **Environment Variable Protection:** The gemini-flash API key and the FastAPI secret keys for JWT signing are never hardcoded. They are stored securely in a local .env file, which is excluded from version control (e.g., added to .gitignore).
- **Password Cryptography:** User passwords are not stored in plain text. FastAPI utilizes libraries like passlib and bcrypt to securely hash and salt all passwords within the relational database.
- **Prompt Injection Mitigation:** The backend incorporates sanitization logic within the Prompt Engineering Engine to ensure user inputs (Topic, Platform, Tone) do not contain malicious instructions intended to bypass the system's baseline instructions.

7.2 – Performance Aspects

- **Asynchronous Execution:** By leveraging FastAPI's async def routing and an asynchronous HTTP client (like httpx), the server can handle multiple concurrent generation requests. It does not block the main execution thread while waiting for the Gemini API to respond.
- **Low-Latency LLM:** The selection of the gemini-flash model specifically addresses performance. It is optimized for rapid text generation, minimizing the wait time on the React frontend and providing a snappy, interactive experience.
- **Vector DB Efficiency:** By utilizing a local vector database tailored for fast nearest-neighbor searches (like ChromaDB), the retrieval of the user's personalized context adds negligible overhead (often under 50ms) to the total generation time.

8 – References

- **FastAPI Documentation:** Technical guidelines for asynchronous Python API development and dependency injection. (https://fastapi.tiangolo.com/)
- **React.js Documentation:** Framework principles for component-based UI and state management. (https://react.dev/)
- **Google Gemini API Documentation:** Implementation details for the gemini-flash model, REST endpoints, and best practices for prompt engineering.
- **Vector Database Documentation:** (Reference the specific Vector DB you end up installing, e.g., ChromaDB or Pinecone docs) for managing embeddings and semantic search.
- **Course Materials:** Concepts and architectures discussed in the Generative AI course at Medicaps University – Datagami.