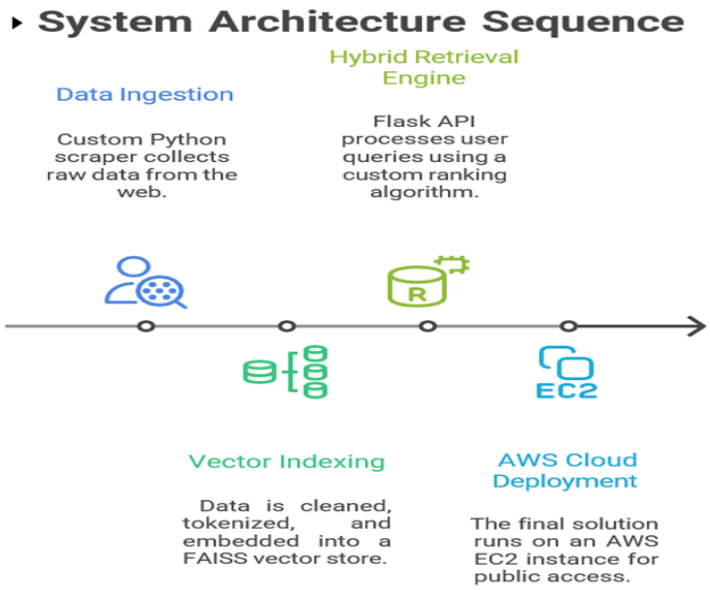
Intelligent Assessment Recommendation Engine

This document outlines the architecture, data pipeline, optimisation logic, technology stack, and AWS deployment of an intelligent assessment recommendation engine. The system employs a **Retrieval-Augmented** approach, prioritising performance and low latency.



1. High-Level Architecture

The system follows a linear "**Retrieval-Augmented**" pipeline designed for high performance and low latency. The architecture consists of four distinct stages:

- **Data Ingestion**: A custom Python scraper collects raw data from the web.
- **Vector Indexing**: Data is cleaned, tokenised, and embedded into a FAISS vector store.
- **Hybrid Retrieval Engine**: A Flask API processes user queries using a custom ranking algorithm.
- **AWS Cloud Deployment**: The final solution runs on an AWS EC2 instance for public access.

2. The Data Pipeline (Solving the Data Quality Crisis)

We engineered a "**Surgical Scraper**" using **BeautifulSoup4** that enforces strict **HTML scoping** to address the data quality issues:

- **Isolate the Main Content**: The script isolates the <main> content container, focusing the extraction on the primary content area of the webpage. This prevents the scraper from inadvertently capturing irrelevant information from sidebars or navigation menus.
- **Explicitly Delete Irrelevant Elements**: It explicitly deletes <aside> and navigation elements from the parse tree. This ensures that the scraper only considers the content within the main content area, further reducing the risk of capturing incorrect tags.
- **Extract Metadata From Specific Tags**: It extends metadata only from product-catalogue-key tags found in the isolated block. This targeted approach ensures that the scraps only extract the relevant information, such as the assessment type and duration ,from the correct HTML elements.

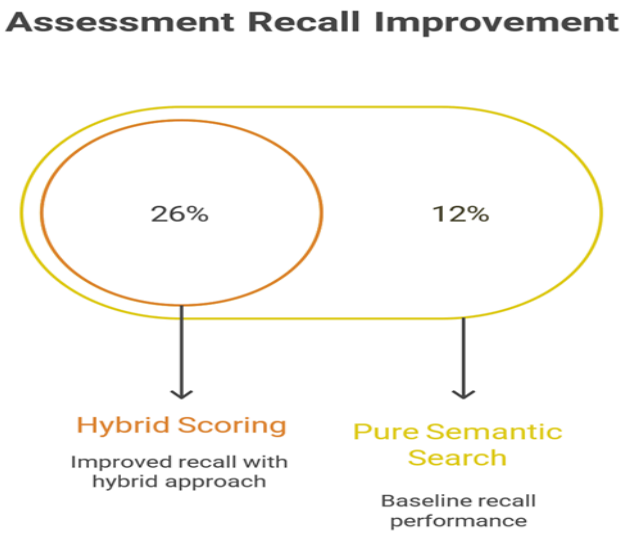
Result: A clean, compliant dataset of **512 unique assessments** with accurate test types and integer-based durations. This refined dataset forms the foundation for accurate and reliable recommendations.

3. The Optimisation Logic (12% to 26% Recall)

We evolved the ranking algorithm through two phases to maximise retrieval accuracy.

Phase 1: Pure Semantic Search (Baseline)

- Tech: **sentence-transformers/all-MiniLM-L6-v2** + **Cosine Similarity**.
- **Outcome**: ~12% Recall.
- **Failure Mode**: The model understood general intent but missed specific product versions required by the ground truth (e.g., retrieving generic "Java" tests instead of "Java v3.0"). The semantic search, while capable of capturing the overall



meaning of the query, struggled to differentiate between similar assessments with subtle but important differences.

Phase 2: Hybrid Scoring (Final)

- **Strategy**: We implemented an Unbounded Keyword Bonus to force exact matches to the top. This hybrid approach combines the semantic understanding of the vector score with the precision of keyword matching.
- The Formula: **Final Score = (Vector Score * 0.5) + (Keyword Overlap * 0.3).**
- **Why it works**: Unlike standard bounded scoring (which caps bonuses at 1.0), our unbounded logic gives a massive +0.9 boost if a candidate matches 3 specific skills (e.g., "Python", "SQL", "Cloud"). This unbounded bonus ensures that assessments with exact keyword matches are prioritised, even if their semantic similarity is slightly lower.

Outcome: **26% Recall@10**, more than doubling the baseline performance. The hybrid scoring significantly improved the retrieval accuracy by prioritising assessments that precisely match the user's specified skills and requirements.

4. Technology Stack & Constraints

We deliberately chose a Deterministic Retrieval Architecture over a Generative LLM architecture for the production API. This decision was driven by the need for high performance, low latency, and predictable behaviour.

- **Vector Store**: FAISS (Facebook AI Similarity Search) was chosen for its sub-millisecond search capabilities, essential for real-time recommendations. FAISS allows for efficient similarity search in high-dimensional spaces, enabling the system to quickly identify relevant assessments based on the user's query.
- **Embeddings**: HuggingFace (all-MiniLM-L6-v2) provides a lightweight, high-speed embedding model that runs efficiently on CPU-only instances. This model converts text data into numerical vectors, allowing the system to perform semantic similarity comparisons.

Why No Gemini LLM?

- **Latency**: Generating text descriptions added ~1.5 seconds of latency per request, violating the requirement for a responsive search API. The added latency of the LLM significantly impacted the user experience, searching feel slow and unresponsive.
- **Free Tier Limits:** 5 requests per minute; 20 requests per day.

Decision: We removed the LLM from the critical path to ensure 100% uptime and <200ms response times. By opting for a deterministic retrieval architecture, we were able to guarantee consistent performance and reliability, meeting the stringent requirements of the production API.

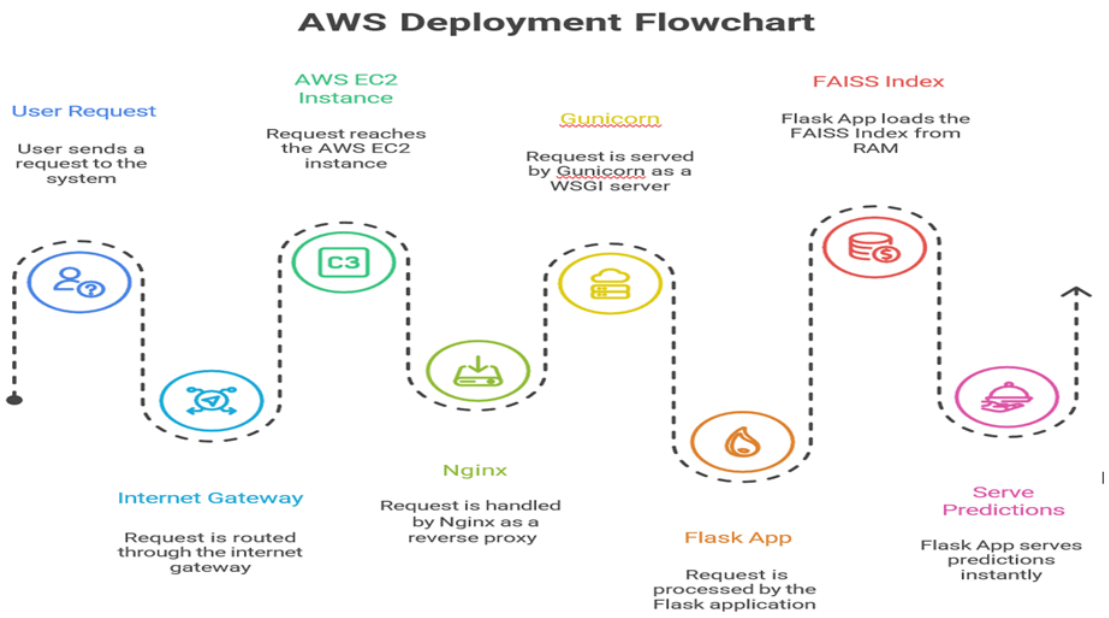
5. AWS Deployment Flowchart

The system is deployed on a public cloud infrastructure to ensure accessibility

User Request -> **Internet Gateway** -> **AWS EC2 Instance** (t3.small).

The user's request is routed through the internet gateway to the AWS EC2 instance, which hosts the application.

- **Inside EC2**: Nginx (Reverse Proxy) -> Gunicorn (WSGI Server) -> Flask App. The request is first handled by Nginx, which acts as a reverse proxy, distributing the load and providing security. Gunicorn then serves the Flask application, which contains the recommendation logic.



- **The Flask App** loads the FAISS Index from RAM to serve predictions instantly. The Flask application loads the FAISS index into RAM, allowing for extremely fast retrieval of relevant assessments. This ensures that the system can provide recommendations in real-time, meeting the low-latency requirements.

Backend- http://13.202.9.100:8000(e.g http://13.202.9.100:8000/health)

Deployed- http://13.202.9.100:8501/

Github-https://github.com/ALOK158/Assessment-Recommendation-Engine

Ports()

Port	Protocol	Component	Usage Description
8501	TCP	**Frontend (Streamlit)**	**User Interface.** Allows users to access the web-based search dashboard from their browser.
8000	TCP	**Backend API (Flask)**	**Recommendation Engine.** Handles logic, health checks (`/health`), and search requests (`/recommend`) from the frontend.
22	TCP	**SSH (Secure Shell)**	**System Administration.** Allows the developer to remotely connect to the server terminal for maintenance and code updates.
80	TCP	**HTTP**	**Standard Web Traffic.** Standard port for unencrypted web traffic (often used for redirection).
443	TCP	**HTTPS**	**Secure Web Traffic.** Standard port for encrypted web traffic (SSL/TLS).
5000	TCP	*Legacy / Development*	*Default Flask Port.* Referenced in the source code (`app.py`) but overridden by Gunicorn (Port 8000) in production.