Gatsby QA System Report

Introduction:

The Gatsby QA System transforms The Great Gatsby PDF into an interactive reference tool. Readers may type any question about characters, symbols, themes, or events and receive an answer presented as a set of flashcards with direct links to the relevant pages in the novel. All processing required for a single query—retrieval, answer generation, and UI render—completes in around 5-8 seconds on an ordinary laptop without relying on a GPU.

```
PDF → preprocessing → FAISS index (offline)
                                  ▲
                                  |
Front-end → FastAPI → retrieve → re-rank → LLM API → JSON   (/ask)
```

Data Preparation and Retrieval Flow:

At startup in in_p.py, it extracted the full text of The Great Gatsby using the pdfminer.six library. The text is segmented into overlapping chunks of approximately 500 tokens with 100 tokens of overlap to retain context. Each chunk is encoded into a vector embedding using a sentence-transformer model (paraphrase-MiniLM-L6-v2) and indexed with FAISS (Facebook AI Similarity Search) for efficient similarity retrieval. The resulting index (gatsby_index.faiss) and the serialized list of chunks (chunks.pkl) enable sub-second nearest-neighbor lookups.

A FastAPI backend (main.py) exposes a single POST endpoint, /ask, which accepts a JSON payload containing the user's question. When a request arrives, the backend retrieves the top-K most relevant chunks from FAISS, re-ranks them with a lightweight CrossEncoder (ms-marco-MiniLM-L-6-v2) to improve precision.

Answer Generation:

Answer generation is handled through the OpenAI chat-completion endpoint. Gpt-4.1-nano is used for cost-optimized options. A concise system prompt plus a small set of few-shot examples instruct the remote model to return a strict JSON object. The object contains a short title and subtitle, followed by an array of items. Each item supplies a description, an analytic paragraph, an unabridged quotation, its page number, and an expandable list of further page references. Because the OpenAI service runs externally, no specialised hardware is necessary on the host machine; Runtime combined typically amounts to 5-8 seconds.

API Layer and Types:

The back-end (see backend/main.py) exposes a single POST /ask route that accepts a JSON body containing one field — question. The response conforms to a schema with fields

title, subtitle, and an items array where every entry holds name, description, analysis, key_quote, quote_page, and page_references.

Frontend Design and UI:

All front-end assets reside in the /frontend folder and are served as static files by FastAPI. The interface is implemented in plain HTML, CSS, and vanilla JavaScript. On first load the page shows a welcome banner and three example questions. Submitting a question or clicking on each of the sample questions will hide the banner, trigger an animated skeleton loader, and send a fetch request to /ask. Once the JSON answer arrives, the script populates the header with the title and subtitle, creates a slideable horizontal tab strip—one tab per item—and renders an information panel for the active tab. The panel comprises description, analysis, a highlighted key quote, and a grid of page-reference cards whose buttons open the public domain PDF at the cited pages.

Responsive design rules ensure that on small screens the two-column grid collapses gracefully, reference cards become swipeable, and the tab bar scrolls horizontally. Subtle CSS transitions and gradient backgrounds provide a modern look without external frameworks.

Performance Evaluation and Iteration:

On a 2018-era Intel Core i7 laptop answer retrieval and re-ranking finish within half a second. The OpenAI request-and-response cycle averages roughly 4-7 seconds under a typical broadband connection, leaving a room within the 1-3 seconds budget for front-end rendering. Memory overhead is modest: the FAISS index and cached embeddings occupy under thirty megabytes, and the back-end process idles below five hundred megabytes including Python, the embedding models, and the HTTP server.

Deployment Steps:

The project is self-contained. Running pip install -r requirements.txt sets up all dependencies—no CUDA libraries are needed. An environment variable OPENAI_API_KEY must be present before starting the service. After the initial preprocessing pass completes, launching the application is as simple as

uvicorn backend.main:app --host 0.0.0.0 --port 8000

Navigating to http://localhost:8000 opens the single-page interface. The preprocessing step is automatically skipped on subsequent runs provided the index and chunks files remain in the data directory.