Design

Steps:

1. **Choose your image source**:
    - **Local File** in your repo: path/to/image.png
    - **External URL**: https://example.com/image.png
2. **Place the image in your repo (if local)**:
    - For example, if you have an images folder in your project, place my_diagram.png there.
3. **Reference the image in Markdown**:
    - Example with a local file:

md

CopyEdit

```
![System Architecture Diagram](images/my_diagram.png "System Architecture Diagram")
```

    - Example with a URL:

md

CopyEdit

```
![External Image](https://example.com/my_diagram.png "Diagram Title")
```

4. **Alt Text and Title**:
    - **Alt text** (the first part in brackets) improves accessibility by describing the image for screen readers.
    - **Title** (in quotes) appears as a tooltip on hover in some Markdown renderers (optional).

Resizing or Aligning (Advanced)

By default, Markdown doesn't support resizing or alignment. If you need more control, you can embed HTML directly:

md

CopyEdit

```
<p align="center">
  <img src="images/my_diagram.png" alt="System Architecture Diagram" width="400" />
</p>
```

- **HTML** syntax allows specifying exact width/height or alignment.
- Not all Markdown renderers allow raw HTML, but GitLab and GitHub do.

With this approach, you can easily insert diagrams, screenshots, or any other images into your .md files.

You said:

cv-analyzer-ai-powered-cv-summary image CV_Analyzer_Flowchart_v1.png

Reasoned for a couple of seconds

Below is a quick example of how to embed your **CV_Analyzer_Flowchart_v1.png** image in a Markdown file named, for example, README.md within your cv-analyzer-ai-powered-cv-summary project.

---

md

CopyEdit

CV Analyzer – AI-Powered CV Summary

![CV Analyzer Flowchart](CV_Analyzer_Flowchart_v1.png "Flowchart showing the overall CV Analyzer process")

Here's what each piece does:

1. **![CV Analyzer Flowchart]** – The **Alt Text** describes the image (important for accessibility).
2. **(CV_Analyzer_Flowchart_v1.png "Flowchart showing the overall CV Analyzer process")** – The file path to the image and the optional **title** in quotes.

Make sure that:

- The image file (**CV_Analyzer_Flowchart_v1.png**) is in the **same directory** as your Markdown file (or update the path accordingly if not).
- You've committed/pushed the image to your repository so it's available in your cv-analyzer-ai-powered-cv-summary project.

If you want to place the image in a subfolder like images/, just update the path:

md

CopyEdit

![CV Analyzer Flowchart](images/CV_Analyzer_Flowchart_v1.png "Flowchart showing the overall CV Analyzer process")

That's it! Once committed, anyone viewing your README.md (or other .md file) on GitLab or GitHub will see the diagram rendered inline.

You said:

Sequence Diagram ### 3. Sequence Diagram (Simplified) A rough outline of how the system components interact: 1. **User** runs python main.py --pdf resume.pdf --max-words 100. 2.

CLIHandler: 1. Parses args (pdf path, max_words). 2. Calls **PDFParser** to extract text. 3. **PDFParser**: 1. Reads and cleans PDF text. 2. Passes raw text to **SectionDetector**. 4. **SectionDetector**: 1. Identifies sections (Education, Experience, etc.). 2. Returns structured text segments back to CLIHandler (or directly to Summarizer). 5. **CLIHandler** calls **Summarizer**, passing: - Extracted text or structured sections. - Max words, any focus keyword. 6. **Summarizer**: 1. Sends prompt to Gemini LLM API. 2. Receives summary text. 3. Returns summary to CLIHandler. 7. **CLIHandler** calls **OutputManager** to display the summary. 8. **OutputManager**: 1. Prints summary to console. 2. (Optional) Writes summary to a file if desired. --- draw the Sequence Diagram

Reasoned for a few seconds

Below is a **Mermaid**-based sequence diagram representing the described flow. When placed in a Markdown file and viewed in GitLab or GitHub (with Mermaid support), it will render as a visual UML sequence diagram.

mermaid

CopyEdit

sequenceDiagram

participant U as User

participant CLI as CLIHandler

participant PDF as PDFParser

participant SD as SectionDetector

participant SUM as Summarizer

participant OM as OutputManager

U->>CLI: python main.py --pdf resume.pdf --max-words 100

CLI->>CLI: Parse arguments (pdf path, max_words, etc.)

CLI->>PDF: Extract text from PDF

PDF->>PDF: Read & clean PDF text

PDF->>SD: Pass raw text for section detection

SD->>SD: Identify Education, Experience, Skills...

SD-->>CLI: Return structured text sections

CLI->>SUM: Summarize CV (pass text, max_words, focus)

SUM->>SUM: Send prompt to Gemini LLM API

SUM->>SUM: Receive summary text

SUM-->>CLI: Return summarized content

```
  CLI->>OM: Provide summary for output

  OM->>OM: Print summary to console

  OM->>OM: (Optional) Write summary to file
```

How to Use It

1. **Copy-Paste** this code block into your Markdown file (e.g., README.md or design.md).
2. **Commit and push** to GitLab/GitHub.
3. **View** the rendered diagram in the repository's file viewer or wiki (assuming Mermaid is supported/enabled).

You can further customize node labels, arrow styles, or add notes using Mermaid syntax.

You said:

Mermaid not suupurt

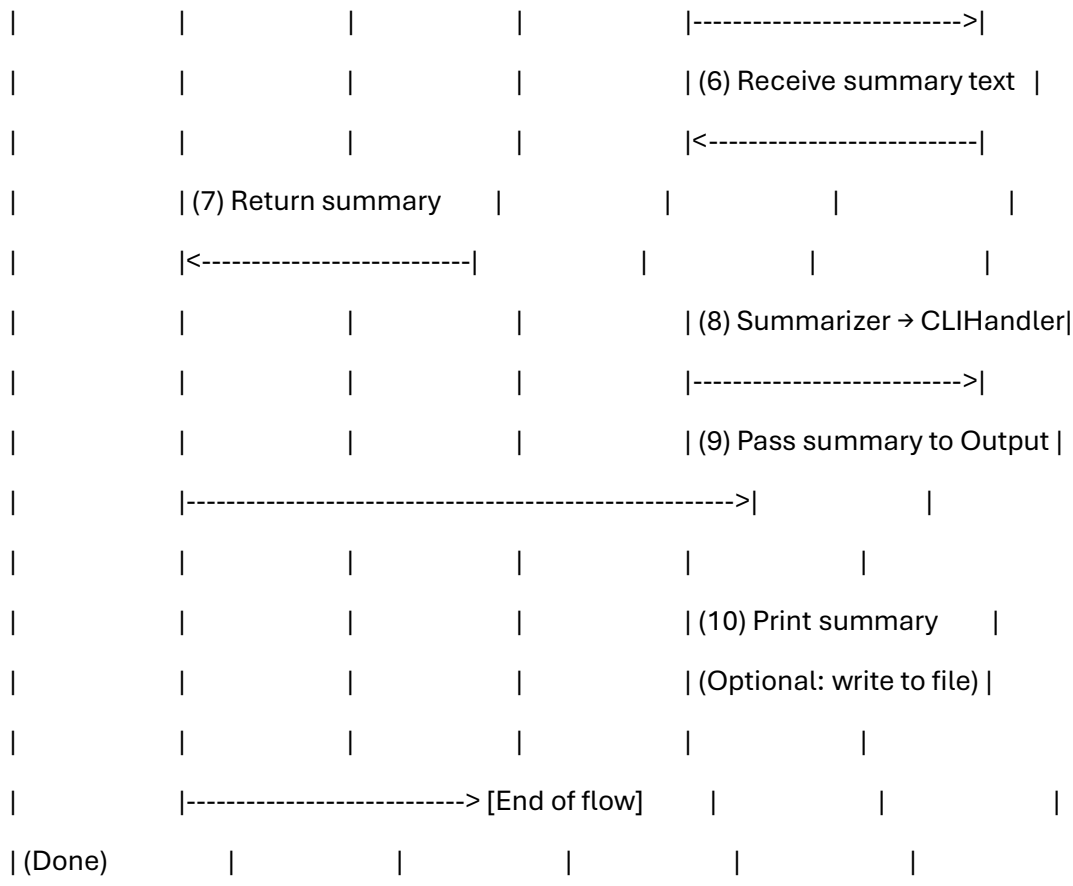
Reasoned for 14 seconds

Below is an **ASCII-based sequence diagram** that outlines how the system components interact without relying on Mermaid. Simply copy and paste this into your .md file, and it will display as text:

text

CopyEdit

```
Sequence Diagram (ASCII Representation)


 User              CLIHandler          PDFParser          SectionDetector        Summarizer
OutputManager

 | (1) python main.py      |                |                |                |                |

 | --pdf resume.pdf        |                |                |                |                |

 | --max-words 100         |                |                |                |                |

 |------------------------>|                |                |                |                |

 |                | (2) Parse CLI arguments   |                |                |                |

 |                |------------------------->|                |                |                |

 |                |                | (3) Read & clean PDF     |                |                |

 |                |                |------------------------->|                |                |

 |                |                |                | (4) Identify sections    |                |

 |                |                |                |------------------------->|                |

 |                |                |                |                | (5) Send prompt to Gemini  |
```



Explanation of the Steps

1. **User**: Runs python main.py --pdf resume.pdf --max-words 100.
2. **CLIHandler**: Parses arguments (pdf path, max_words) and validates them.
3. **PDFParser**: Reads and cleans PDF text.
4. **SectionDetector**: Identifies major sections (Education, Experience, Skills).
5. **Summarizer**: Sends prompt to the Gemini LLM API with the structured text and focus/word constraints, then receives the summary.
6. **CLIHandler**: Receives the finalized summary from the Summarizer.
7. **OutputManager**: Prints the summary to the terminal (and optionally writes to a file).

This ASCII sequence diagram provides a readable, platform-agnostic alternative to Mermaid, preserving the essential system flow.

You said:

Phase 3: Design

Reasoned for a second

md

CopyEdit

Phase 3: Design

In this phase, we refine our architecture through a **Class-Responsibility-Collaborator (CRC)** outline and simple design diagrams. The goal is to ensure each component is well-defined and easy to maintain before coding.

---

1. High-Level Design Approach

We follow **XP's principle of simple design**:

- **Keep classes small** and focused on a single responsibility.

- **Minimize dependencies** between classes.

- **Favor clarity** over premature optimization.

---

2. CRC Cards

Below is a CRC (Class-Responsibility-Collaborator) description for our key classes. These classes are tentatively located in the `src/` folder (or as makes sense in your project structure).

Class	**Responsibilities**	**Collaborators**
`CLIHandler`	- Parse command-line arguments (PDF path, max words, focus) - Validate user input (e.g., check if file exists) - Initiate main application flow	`PDFParser` `Summarizer`
`PDFParser`	- Extract text from the given PDF - Preprocess text (basic cleaning, splitting)	`CLIHandler` `Summarizer`

| `Summarizer` | - Interface with **Gemini** (google-generativeai library)
- Construct prompt based on user focus and max-words
- Return generated summary text | - `CLIHandler`
- `PDFParser` (for raw CV text) |

| `SectionDetector` | - Identify major sections in the raw PDF text (Education, Experience, Skills, etc.)
- Provide structured data (e.g., dictionary of section_name → text_content) | - `PDFParser`
- `Summarizer` (can refine prompts or pass structured data) |

| `OutputManager` | - Print final summary to terminal
- (Optional) Write summary to a text file if the user desires | - `Summarizer`
- `CLIHandler` |

---

3. Sequence Diagram (Simplified)

A rough outline of how the system components interact:

1. **User** runs `python main.py --pdf resume.pdf --max-words 100`.
2. **CLIHandler**:
   1. Parses args (pdf path, max_words).
   2. Calls **PDFParser** to extract text.
3. **PDFParser**:
   1. Reads and cleans PDF text.
   2. Passes raw text to **SectionDetector**.
4. **SectionDetector**:
   1. Identifies sections (Education, Experience, etc.).
   2. Returns structured text segments back to CLIHandler (or directly to Summarizer).
5. **CLIHandler** calls **Summarizer**, passing:
   - Extracted text or structured sections.
   - Max words, any focus keyword.
6. **Summarizer**:
   1. Sends prompt to Gemini LLM API.
   2. Receives summary text.
   3. Returns summary to CLIHandler.

7. **CLIHandler** calls **OutputManager** to display the summary.

8. **OutputManager**:

   1. Prints summary to console.

   2. (Optional) Writes summary to a file if desired.

---

4. Design Considerations

1. **Error Handling**

   - The `CLIHandler` must catch invalid arguments (e.g., missing PDF) and provide clear error messages.

   - The `PDFParser` should handle unreadable or corrupt PDFs gracefully, possibly returning partial text or a structured error.

2. **Extensibility**

   - `SectionDetector` could be enhanced with more complex NLP or pattern recognition.

   - `Summarizer` logic can add advanced prompt engineering or fallback strategies if Gemini is unreachable.

3. **Testing**

   - Each class should have unit tests (e.g., `test_pdf_parser.py`, `test_summarizer.py`).

   - System-level tests ensure the end-to-end flow works as expected.

---

5. LLM Interactions

All design-related conversations with the LLM are recorded in:

- [Phase 3 LLM Chat Transcript](./chats/phase3_llm_chat.txt)

Next Steps

We'll move on to **Phase 4: Coding & Testing**, where we implement the classes, adhere to PEP 8, and create both unit and system-level tests to validate our design.

You said:

back to step2