ChatGPT

1. User Stories @ Definition of Ready

- **As a policy analyst, I want to provide basic project information and have an AI** GovDoc Copilot **generate a** draft Business Case **so that I can quickly get an initial structured document to refine.**
  Acceptance Criteria: Given a minimal project profile (title, scope, alignment, stakeholders, etc.), the system generates a multi-section Business Case draft (e.g. Problem Statement, Options Analysis, Recommendation) with relevant content and in a standard format. The draft should include evidence or citations for key facts (if any were retrieved) and preserve the logical section structure. The output is persisted (e.g. stored in a database or document store) and made available for download or further editing.
  Assumptions: A structured **Project Profile** exists or can be created from user input before drafting begins. External knowledge (policies, stats) can be fetched via web search to enrich the content. The AI has predefined templates or guidelines for Business Case sections.
  Dependencies: Availability of OpenAI API for content generation, web search API for evidence retrieval, and a database for storing profiles and document sections. Prompt templates for each section must be defined (e.g. YAML configs) and accessible by the system [1] [2].

- **As a user, I want to upload or input a project description and have it automatically parsed into a** Project Profile **so that the AI can use structured data for document drafting.**
  Acceptance Criteria: The system accepts text, file, or link input describing a project. It then extracts key fields (project title, sponsor, scope, budget, timelines, etc.) into a structured profile JSON. The profile is saved in the system (database) with a unique `project_id`. Any missing optional fields are set to null. On successful ingestion, the system returns a status confirmation and the structured profile data. If required fields (like `project_id`) are missing or the input format is unsupported, the system returns an error.
  Assumptions: The user provides sufficient project details in the input for the AI to extract meaningful fields. An initial `project_id` (or name) is supplied or determinable.
  Dependencies: OpenAI's GPT-4 for extracting the profile from text [3] [4], the Project Profile data model and database (for storing the profile), and file parsing utilities for non-text inputs (if uploading PDFs, etc.).

- **As a policy analyst, I want to assemble a final Business Case document from previously generated sections so that I can review or share a complete proposal.**
  Acceptance Criteria: The system can compile all saved draft sections for a given project's Business Case (by artifact type and version) into one document. Sections should be in the correct order as per the template (e.g. Introduction, Problem Statement, Analysis, Recommendation, etc.), consistently formatted (headings, numbering), and combined into a single output (e.g. a markdown or PDF document). The assembled document is saved (e.g. in an `artifact` record or uploaded to a cloud drive) and a link or identifier is returned. The assembly should only proceed if all major sections are available in the database; if any section is missing or incomplete, the system should flag it.
  Assumptions: Draft sections have been generated and stored for the given project and document type. The user has rights to save or export the final document (e.g. to Google Drive if integrated).
  Dependencies: Database of section drafts (each tagged by project and section type), document

formatting logic or template (to ensure consistent style), and an optional cloud integration (e.g. Google Drive API) for storing the final output [5].

2. Design Decomposition

Technical Architecture and Modules

Entry Points & API Layer: The system is built as a FastAPI application (see `main.py` / `api_router.py`) exposing endpoints that correspond to the main stages of Business Case drafting. For example, there are likely endpoints such as `POST /profile` to ingest project info, `POST /section` to generate a section draft, and `POST /artifact` to assemble the document. These endpoints accept JSON inputs and invoke the backend toolchains. The OpenAPI schema (`openapi.json`) would define models like `ProjectProfile`, `ArtifactSection`, etc., and parameters like `project_id`, `section_type`, etc., for each route. *(OpenAPI spec not shown due to the private branch, but WP12 design outlines analogous "phase service" endpoints [6] [7].)*

Ingestion & Project Profile: The **IngestInputChain** (`app/engines/toolchains/IngestInputChain.py`) orchestrates the first phase. It accepts raw input in various forms (text, file, or link) along with metadata (including `project_id`). Internally, it uses a Tool Registry to select the appropriate upload parser tool: e.g. `"text"` inputs use `uploadTextInput`, files use `uploadFileInput`, etc. [8]. The chosen *upload tool* reads the content and returns raw text and some metadata (like the provided `project_id`). The chain then calls `generate_project_profile(text, metadata, existing)` which uses GPT-4 to extract structured fields from the raw text according to a predefined schema [3] [4]. The prompt for this step instructs the AI to output a JSON with the required project fields (title, budget, dates, alignment, etc.), using `null` for missing values [3] [9]. If an existing profile exists for that project (the chain checks via `ProjectProfileEngine().load_profile`), it merges the new data with old to avoid overwriting non-null fields [10]. The finalized profile (a Python dict) is then saved to the database via `ProjectProfileEngine().save_profile` [11]. The output of the chain includes the status, the new `project_profile` data, and echoes the raw text and metadata [12]. The ProjectProfileEngine (in `app/engines/project_profile_engine.py`) encapsulates DB access – likely using an ORM or direct SQL – for the `project_profile` table. The table schema supports all profile fields (mostly strings, dates, numbers) with only `project_id` (primary key) and `last_updated` being required [13]. This design ensures all necessary project context is stored once and reused for any document generation tasks.

Planner Orchestrator & Toolchain Selection: The **PlannerOrchestrator** (`planner_orchestrator.py`) acts as an intelligent controller that, given an **intent** and some input, routes to the appropriate toolchain or sequence of tools. For Business Case drafting, two primary intents are used: - `generate_section` – draft a specific section of an artifact (e.g. "problem_statement" of an "investment_proposal"). - `assemble_artifact` – compile all sections of an artifact into a final document.

When invoked (e.g. via `PlannerOrchestrator().run("generate_section", inputs)`), the orchestrator will load the relevant context (notably the project profile from the DB) and then execute the corresponding chain of tools [2]. In the current implementation (as of *sandbox-curious-falcon* branch), the orchestrator may directly instantiate and run a `GenerateSectionChain` class, or perform the steps internally – the design evolved such that by WP17/WP24, a dedicated class `GenerateSectionChain`

exists [14] [15]. The **ToolRegistry** (`app/tools/tool_registry.py`) provides a mapping of tool names to their implementations (tool classes in `tool_wrappers/` or chain classes in `toolchains/`). The orchestrator uses this registry to fetch and invoke tools dynamically. For example, the `generate_section` pipeline uses tool names like `"memory_retrieve"`, `"webSearch"`, `"section_synthesizer"`, etc., which are looked up in the registry and executed in sequence.

Section Drafting Toolchain: The **GenerateSectionChain** (`app/engines/toolchains/generate_section_chain.py`) encapsulates the multi-step process of drafting a single section of the Business Case. Given inputs such as `project_id`, `artifact` (document type), and `section` (section name), it proceeds as follows: 1. **Load Context:** It ensures the latest `ProjectProfile` is loaded (via ProjectProfileEngine) so that the LLM has structured project info. It may also retrieve recent user inputs or prior reasoning traces (the "memory") relevant to this section. For example, a tool `memory_retrieve` is used to fetch any stored reasoning or evidence associated with the project and section [16]. If vector embeddings are used, this step would perform a similarity search on past content (e.g. relevant policy text, prior user feedback) to ground the draft – the design docs mention using "memory embeddings" for this purpose [17], implying an embedding store could be queried to augment context. 2. **Generate Search Query:** Before writing the section, the chain may decide to gather external information. The **QueryPromptGenerator** tool (`app/tools/tool_wrappers/query_prompt_generator.py`) is invoked to craft an optimal web search query [18]. It uses a Jinja2 template from the `generate_section_prompts.yaml` config – specifically the `search_query_generation` prompts [19]. The template is filled with the project profile summary and up to 10 lines of "memory" context (recent inputs or partial drafts) [20] [21]. The result is an AI-generated search query (returned as `{"query": "..."}` [22]). 3. **Web Search & Evidence Gathering:** Using the query, the **webSearch** tool is called. This tool likely calls an external search API or knowledge base to retrieve top results. (The implementation details were not explicitly shown, but presumably it uses a Bing or Google search API and returns snippets or links.) The results may be summarized by another helper tool (according to the plan, web results summarization was to be integrated via prompt templates [23]). The outcome is a set of relevant facts or excerpts with source URLs. 4. **Draft Synthesis:** Next, the **SectionSynthesizer** tool generates the section draft using GPT-4. This tool uses another block from the `generate_section_prompts.yaml` prompt config – likely a system prompt defining the section's purpose and a user prompt template that includes the project profile and any retrieved evidence or memory. For example, if drafting the **Problem Statement** section, the prompt might instruct: "Write a Problem Statement for project X, clearly stating the business problem, impacts, and urgency, using the following context…". The SectionSynthesizer passes such messages to the LLM via `llm_helpers.chat_completion_request()` [24] and obtains a draft text, potentially with placeholders like "[1]" for citations. 5. **Refinement:** The raw draft is then fed into **SectionRefiner** tool. This tool's job is to post-process the draft for tone, clarity, and completeness [16]. For example, it might enforce a professional tone, fill any missing pieces, and ensure the content meets length or style guidelines. It may also resolve citation placeholders by appending reference details from the evidence. The refiner likely uses another LLM prompt (e.g. "As a senior editor, refine the following section draft for style and ensure it's self-contained. Improve clarity and maintain a formal tone."). The output is a polished section text. 6. **Persistence:** The final section content is then saved as an **ArtifactSection** record in the database [25]. The system would create a new entry linking the `project_id`, `artifact_id` (document type), `section_id` (section name), and perhaps a version or timestamp. It also stores the text and any source citations. This is done via a database model or through the ProjectProfileEngine/another engine. A log of this generation step (a **ReasoningTrace**) is also recorded for traceability – capturing the chain of tools used and their outputs for audit/debugging [26]. The chain returns a result indicating success and including the saved section content or an identifier.

(In the WP7 test, `GenerateSectionChain().run(input)` *is used directly, demonstrating this flow: it takes* `artifact`*,* `section`*, and* `project_id` *and produces a saved draft* [27]*. PlannerOrchestrator can also trigger this chain internally when handling a "generate_section" intent.)*

Document Assembly Toolchain: Once all key sections have been drafted (or when the user requests it), the **AssembleArtifactChain** (`app/engines/toolchains/assemble_artifact_chain.py`) compiles the full Business Case. The inputs are the `project_id` and `artifact_id` (and possibly a target version or gate). The steps include: 1. **Retrieve Sections:** The chain loads all relevant sections for the project's artifact from the DB. A tool `loadSectionMetadata` (or similar) fetches the titles and content of each section [28]. The section order and inclusion might be determined by a reference template (for example, `project/reference/gate_reference_v2.yaml` defines which sections apply to a given artifact type and project phase [29]). 2. **Format Sections:** Each section draft is passed through a formatting step to ensure consistency. The **FormatSection** tool might add markdown headings, normalize bullet points, or ensure each section starts on a new page if exporting to PDF. It could also insert section numbers or combine short sections if needed. 3. **Merge Sections:** The content is then concatenated in the correct order by a **MergeSections** tool [28]. This produces a single combined document (likely in markdown or a similar markup). 4. **Final Touches:** A **FinalizeDocument** step may perform any cross-section adjustments – for instance, ensuring the tone is consistent across sections, adding a table of contents, or writing an executive summary if required. (In WP12 designs, a "validate_and_commit" phase was considered to verify structure and content before finalizing [30]. Validation might include checking that all required sections are present and properly labeled.) 5. **Storage/Output:** Finally, the assembled document is saved and optionally exported. The **storeToDrive** tool uploads the document to a Google Drive folder (as per WP20 integration) and returns a shareable URL [31]. In addition, an `Artifact` record might be stored in the database with metadata: which project, artifact type, version, timestamp, and storage link. The orchestrator or chain returns a success response with the document link or the merged text.

(The test flow confirms this: calling `PlannerOrchestrator().run("assemble_artifact", input)` *loads the profile and assembles the artifact* [32]*. In later refactoring,* `AssembleArtifactChain()` *exists and is used similarly* [33]*. The assembled output is printed or returned as JSON, likely containing the drive URL or combined content.)*

Full Artifact Generation (Orchestration): There is also a higher-level workflow to generate an entire Business Case in one go. The **GenerateFullArtifactChain** (`generate_full_artifact_chain.py`) was introduced (per WP24) to orchestrate running all section generations followed by assembly [34] [35]. This chain likely uses the section template (from `gate_reference_v2.yaml`) to determine which sections are needed, calls `GenerateSectionChain` for each (possibly in sequence or in parallel threads), waits for all to complete, and then triggers `AssembleArtifactChain`. The result is a one-call solution to go from a project profile to a complete first draft of the Business Case. This was built to facilitate end-to-end testing and perhaps to allow the AI agent to do everything with one command. (In the test script, we see `GenerateFullArtifactChain.run()` being called with just `project_id`, `artifact`, etc. [35].)

Prompts and LLM Integration: Prompt templates are defined in YAML under `app/prompts/`. The `generate_section_prompts.yaml` file contains structured prompt blocks for various sub-tasks of drafting [1]. For instance, it likely has sections like `search_query_generation` (used by QueryPromptGenerator) and others for `section_draft` or `section_refine`. Each block provides a **system prompt** and a **user prompt template**. The code uses a helper

`llm_helpers.get_prompt(file, block)` to fetch these from the GitHub repository at runtime [36]. The use of Jinja2 templating allows injecting runtime data (project profile fields, memory context, web info) into the prompt string before sending to OpenAI. All calls to the OpenAI ChatCompletion API are funneled through `llm_helpers.chat_completion_request()`, which standardizes model parameters (e.g. always using `gpt-4` and a certain temperature) [24]. This design centralizes LLM usage for easier updates (e.g. switching models or keys in one place).

Database (Data Schema): Besides the `project_profile` table, the system uses tables or data structures for: - **ArtifactSection**: stores each section draft. Likely fields: `project_id`, `artifact_id` (document type, e.g. "investment_proposal"), `section_id` (e.g. "problem_statement"), the textual content, `sources` (list of citation URLs or references), `created_at`, `status`. This allows sections to be versioned or updated independently. The WP17b plan explicitly mentions an `ArtifactSection` schema with fields for text, sources, status, etc. [37]. - **PromptLog / ReasoningTrace**: logs of AI reasoning or tool usage per session. Each time the AI (or planner) runs a toolchain, it could append a record of what was done, which tools were used, and a summary of inputs/outputs. The `log_tool_usage` function in `memory_sync.py` (called by IngestInputChain and presumably others) records usage of each tool, including a summary of input and output [38]. This log can be used for debugging or feeding back into memory retrieval. There might be a `prompt_log` table or a YAML/JSON trace stored for each run.

Integration & Flow Orchestration: The overall system orchestrates these components as follows: 1. **Profile Creation** – via API or internal call, IngestInputChain produces a ProjectProfile and saves it [39]. 2. **Section Generation** – triggered by either an API call for a specific section or by an AI agent deciding a section is needed. The PlannerOrchestrator prepares inputs (loads profile, etc.) and calls GenerateSectionChain, which uses multiple tools (memory retrieval, search, LLM prompts) to create and save the section [2]. This can be repeated for each section of the Business Case. 3. **Document Assembly** – once sections are ready (or on demand), the orchestrator (or directly AssembleArtifactChain) collects the sections and builds the final document [5]. It then commits the document to persistent storage (DB and/or Drive). 4. **Iteration** (Optional) – The design allows the AI to be in the loop. For example, the AI could review the assembled draft and prompt the user for feedback or additional info, then incorporate that by regenerating a section or refining the document (as described in the Planner Phase Services proposal [40] [41]). While this interactive loop may be a future enhancement, the current system is structured to allow re-entering the generation phase with updated inputs.

Example Trace: *Suppose the user provides a text describing a project.* The **GovDoc Copilot** backend calls `IngestInputChain`, which uses GPT-4 to extract a profile (e.g. `{ project_id: "pegasus", title: "Green Energy Initiative", ... }`) and saves it [11]. The user then requests a draft Business Case. The system's planner invokes `GenerateSectionChain` for each required section: - For "Problem Statement", the chain fetches the profile and recent logs, generates a search query (maybe "green energy initiative problem context"), performs a web search, and gets results (e.g. stats about carbon emissions). GPT-4 then writes a Problem Statement using the profile and facts (citing sources). The draft is refined and saved with sources. *(These steps correspond to memory_retrieve + webSearch + section_synthesizer + section_refiner)* [2]. - Similar steps happen for "Options Analysis", "Recommendation", etc., each time using the project profile (and any new info) as input. After sections are drafted, the system calls `AssembleArtifactChain` to merge them. It orders the sections as per a template, maybe inserts an automatically generated Introduction or Summary if needed, and produces the full document. It then saves this artifact and perhaps uploads a PDF to Drive [5]. The final output is a link or document ID the user can open to see the assembled Business Case.

Interface Design (FastAPI & Schema):

- **FastAPI Routers:** The application likely has routers for different functional areas, e.g., `project_profile` routes and `document_generation` routes. For instance:
- `POST /projects/{project_id}/profile`: to call the ingestion chain with a file or text payload. Returns the saved profile JSON.
- `POST /projects/{project_id}/sections`: to generate a new section. The request body could include `artifact_type` and `section_name` (and possibly any custom instructions). This would call PlannerOrchestrator with `generate_section`.
- `GET /projects/{project_id}/sections/{section_name}`: to retrieve a generated section (from DB) – useful for front-end to display draft content.
- `POST /projects/{project_id}/artifact`: to assemble the artifact (Business Case) from all sections, calling PlannerOrchestrator `assemble_artifact`. Could allow a query param for `version`.
- `GET /projects/{project_id}/artifact`: to fetch the assembled doc or its metadata (e.g., Drive link).

Each endpoint request/response adheres to Pydantic models (e.g., `ProjectProfileModel`, `SectionDraftModel`, `ArtifactModel`). For example, the profile model contains fields as in the schema (title, sponsor, etc.) and the section model contains at least `section_name` and `content`. The Artifact model might contain a list of sections or a URL if exported.

- **OpenAPI Documentation:** The OpenAPI spec (`openapi.json`) would list these endpoints and models. It would describe that the **Business Case drafting** endpoints require an existing project profile. It likely tags them under "GovDoc Copilot" or "Document Drafting" for clarity. Authentication might be handled via API key or token (not described in the code we saw, but possibly assumed).

- **Example** – *Draft Section Endpoint:*
  Request: `POST /projects/pegasus/sections` with JSON body: `{"artifact": "investment_proposal_concept", "section": "problem_statement"}`.
  Processing: The API handler constructs an input dict and calls `PlannerOrchestrator().run("generate_section", input)`. Internally, this loads profile for "pegasus", runs the GenerateSectionChain as detailed, and obtains an output dict containing (among other things) `save_result` which includes the section text and metadata [27].
  Response: HTTP 200 with body: `{"section": "problem_statement", "content": "<draft text>", "sources": ["[1] ... URL ..."], "project_id": "pegasus", "status": "draft_saved"}`. If the profile was missing or an error occurred, an HTTP 400 or 500 with error message is returned.

Data Flow & Embedding Logic:

Data flows through the system in structured JSON-like objects: - **Project Profile Data**: flows from user input → LLM extraction → DB. It is then pulled from DB → used in prompts for sections → possibly updated with new info (though usually static per document). - **Memory and Embeddings**: The term *memory* refers to stored conversational or contextual data. Each time a tool is run, `memory_sync.log_tool_usage` records a summary of input and output [38]. This log can be persisted (perhaps in a `prompt_log` table or in-memory). Additionally, important text (like the project profile and finalized sections) might be embedded

into a vector store so that `memory_retrieve` can later do semantic searches. For example, after ingesting the profile, an embedding for the profile content could be stored (using OpenAI Embeddings API) with the project_id as metadata. Then when drafting a section, the system could retrieve the top-N similar pieces of text from the profile or prior interactions as additional context. While implementation details aren't explicit, WP17b's objective explicitly mentions using memory embeddings in the section drafting pipeline [42], so the system likely uses an embedding-based similarity search for relevant context. Any retrieved text is then inserted into the prompts (the QueryPromptGenerator already does something similar, concatenating recent memory lines into the query prompt [43]). - **External Research Data**: When web search is performed, relevant snippets and URLs flow into the LLM prompt for drafting. The SectionSynthesizer might output the draft with citation placeholders, and the SectionRefiner could then append footnotes or endnotes with the actual URLs. These source URLs are then stored alongside the section content (e.g., in the `sources` field of ArtifactSection). Thus, the final assembled document can include a reference list or the system can display sources to the user (fulfilling the "compose and cite" requirement).

All these flows are orchestrated such that the **PlannerOrchestrator** knows what data to inject at each step. The orchestrator likely builds a context object containing: `{ project_profile: {…}, memory: [...], inputs: (artifact, section, etc.), user_id: … }` which it passes to the toolchain. Each tool may augment this context (e.g., add the search results) and pass along to the next. In the end, final outputs and logs are written out to the DB.

Illustration – Section Generation Sequence:

1. **API call** – passes `project_id="pegasus", artifact="investment_proposal_concept", section="problem_statement"` to Planner.
2. **PlannerOrchestrator** – loads profile for "pegasus" from DB [44]; collects recent logs (embedding search) for context; calls GenerateSectionChain.
3. **GenerateSectionChain.run** – receives a dict with profile, section, etc. It calls `memory_retrieve` → gets relevant text (e.g., "Project Pegasus has goal X...") and adds it to context.
4. **QueryPromptGenerator** – called with `project_profile` and `memory` → returns `query="renewable energy adoption rate statistics Canada"`.
5. **webSearch** – called with that query → returns snippets (e.g., "Canada aims for 50% renewable by 2025 [Source1]", URL1).
6. **SectionSynthesizer** – called with profile + snippets → returns draft text: *"Problem Statement: Canada's energy consumption still relies heavily on non-renewables... (as per federal target of 50% renewable by 2025[1]) ..."*.
7. **SectionRefiner** – called with draft → returns refined text: *"Canada's current energy portfolio remains 60% non-renewable, undermining climate goals. Problem Statement: The project addresses this gap by ... (Source: Department of Energy report [1])"*.
8. **Save Section** – the refined content and citation [1]: URL1 are saved as a record in ArtifactSection table.
9. **Return** – the chain returns success and perhaps the saved content. Orchestrator may log the reasoning trace (steps 3–8).
10. *(Later)* **AssembleArtifactChain** – fetches all sections (Problem Statement, etc.) from ArtifactSection where `project_id=pegasus, artifact=investment_proposal_concept`, merges them, and saves the full doc (possibly uploading to Drive and getting a link).

The above breakdown reveals the interplay of modules: the **toolchains** define the sequences, **tool_wrappers** implement individual LLM-powered tasks, **prompts** provide the content for those tasks, and the **db** layer persists outputs and state.

Notable Classes & Libraries:

- **OpenAI API Client** – The code uses `openai` Python package to call GPT-4 [45] . All LLM calls go through this.
- **Jinja2** – for prompt templating (filling YAML prompt templates with runtime data) [46] .
- **SQL/ORM** – likely SQLAlchemy or Pydantic+SQLModel for DB models (ProjectProfile, ArtifactSection). For example, `ProjectProfile.py` defines fields as Python types for the ORM [13] .
- **requests** – to fetch prompt YAML from GitHub raw URLs [47] (and possibly for web search if using REST API).
- **FastAPI & Pydantic** – to define API endpoints and data models (not explicitly shown, but implied by structure).

(In summary, the design is a modular pipeline: from ingestion to section drafting to document assembly, with a Planner orchestrating tool usage. It emphasizes traceability (logging each step) and reusability of context (project profiles, memory logs) to incrementally build a comprehensive Business Case.)

3. Test Suite Outline

To ensure the system works as expected, a combination of unit tests for individual tools and integration tests for the end-to-end flow is used. Key components and their test scenarios include:

- **Project Profile Ingestion Tests:**
- *Test valid text input:* Provide a sample project description text and metadata (`project_id`, etc.) to `IngestInputChain`. Assert that the output contains a `project_profile` dict with keys like `title, scope_summary, strategic_alignment` filled, and that `project_profile["project_id"]` matches the input metadata or extracted value. Verify that `save_profile` was called and the profile can be retrieved from the DB (e.g. via `ProjectProfileEngine.load_profile`).
  Acceptance: The chain returns status `"profile_saved"` and all required fields are present (or null) [12] .
- *Test missing required field:* Call `IngestInputChain.run` with an input missing `project_id` in metadata. Expect a `ValueError` or error response indicating the missing field (the code explicitly checks for `project_id` in metadata and raises an error if not present [48]).

- *Test file and link ingestion:* Simulate uploading a file (or use a small text file) and a URL. Ensure `uploadFileInput` and `uploadLinkInput` tools are invoked via ToolRegistry and their outputs (extracted text) feed into profile generation. Verify the profile is created similarly for these methods.

- **Tool Wrapper Unit Tests:**
  Each tool in `app/tools/tool_wrappers/` should be tested in isolation with controlled inputs:

- *QueryPromptGenerator:* Provide a dummy `project_profile` (with known fields) and a fabricated `memory` list. Mock the GitHub prompt YAML fetch if needed (or use a sample YAML string) to ensure

the template is loaded. Assert that `run_tool` returns a dict with a `"query"` key that is a non-empty string [22]. Also verify that the query content reflects the profile (e.g. contains the project title) – basically, check the Jinja template render works.

- *SectionSynthesizer and SectionRefiner:* These likely call OpenAI, so in unit tests, mock the `llm_helpers.chat_completion_request` to return a fixed string (to avoid actual API calls). Then ensure that `run_tool` processes inputs properly. For example, for SectionSynthesizer, input might include a profile and some evidence; the test would check that the combined prompt (maybe accessible via logs or by patching `chat_completion_request`) follows the template. For SectionRefiner, give it a rough draft text and see that it produces a refinement (in practice, since it's mostly an LLM call, the test might just ensure no exceptions and output is returned).

- *WebSearch tool:* If implemented, mock the external API call (e.g., monkeypatch `requests.get` or the Bing API SDK call to return a preset response). Feed in a query and ensure the tool returns results in expected format (maybe a list of snippets or a summarized text). Also test edge case: no results or API failure (should the tool return an empty result or error? The system should handle that gracefully, perhaps by proceeding without external content).

- **GenerateSectionChain Integration Tests:**
  This is a critical path, so a full integration test can be done in a controlled setting:

- *Test end-to-end section generation:* Using a small known project profile (could even use the one from WP24 test: title "Test Project", etc. [49]) as input, run `GenerateSectionChain().run`. Since this will call out to multiple tools, one approach is to monkeypatch tools to deterministic stubs for test. For example, patch `QueryPromptGenerator.run_tool` to return a fixed query, patch `webSearch` to return a fixed snippet, and patch the LLM calls to return a canned draft text. Then run the chain and verify:
  - The returned output structure has `save_result` containing the section text and maybe a reference [50].
  - The section text contains content derived from the fake snippet (proving it attempted to use evidence).
  - The ArtifactSection was saved in DB: use the DB engine to fetch it and confirm the text matches.
  - The ReasoningTrace/logs contain entries for each tool (if exposed or stored).
- *Test missing profile handling:* Call GenerateSectionChain with a non-existent `project_id` (or omit `project_profile` if the chain is supposed to fetch itself). It should load via ProjectProfileEngine; if none found, this is an error scenario. The expected behavior might be an exception or a result indicating failure (the system might require profile to exist). Ensure the chain does not proceed to LLM calls without a profile (or it yields a meaningful error like "Project profile not found").

- *Multiple sections sequence:* Although each section is generated independently, an integration test can generate two different sections and ensure they both get saved and do not overwrite each other. For instance, generate "problem_statement" and "solution_options" sequentially for the same project and then query the DB or use assemble to confirm both exist.

- **AssembleArtifactChain Tests:**

- *Test assembly with all sections present:* Insert or use an existing set of ArtifactSection entries for a project (perhaps reuse the outputs from the previous test). Run `AssembleArtifactChain().run({"project_id": X, "artifact": Y, ...})`. Verify that the result includes a merged document. If the chain returns the content directly, check that it concatenated sections in the correct order and format (maybe sections separated by headers). If it returns a reference (like a Drive link or path), verify that link is non-empty and presumably well-formed. Also check in the DB if a new Artifact or Document record is created (if applicable).
- *Test assembly with missing section:* Remove one required section and run assemble. The expected result might be that the chain still produces output (skipping missing sections) or it raises an error/warning. The test should assert the system's chosen behavior – e.g., perhaps it still merges what's available and flags the missing sections in the output. Ensure it doesn't just fail silently. (If design follows WP12 validate step, it might list `missing_inputs` – but current implementation likely simpler.)

- *Test drive upload (if applicable):* This is more of an integration test with external service. It can be tricky to test without actual Drive credentials; instead, one could mock the `storeToDrive` tool to simulate success. The test would ensure that after `AssembleArtifactChain`, the returned structure contains a `Drive_url` or similar when storeToDrive runs. If storeToDrive fails (e.g., network error), ensure the chain returns a proper error status.

- **Full Flow End-to-End Test:**
  A scenario test where we simulate a user request from start to finish:

- Run IngestInputChain with a sample input text.
- For each section in a small set (maybe 1–2 sections), run GenerateSectionChain.
- Run AssembleArtifactChain.
  This sequence should execute without exceptions and produce a final document. The final output can be checked for consistency (e.g., the content of the final doc contains content from the generated sections, and those in turn reflect the original input context).

The repository includes a test runner for exactly this purpose in WP7 [51] and an updated script in WP24 [52] [33]. For example, the WP7 test runner does: ingest → generate section → assemble, printing outputs at each step [53] [27] [32]. These scripts serve as integration tests to validate that all pieces work together.

- **Edge Cases & Validation Tests:**
- Test extremely short input (or empty fields) for profile generation – ensure no crash and profile fields become null.
- Test very long input (to see if chunking is needed or if it hits token limits) – possibly ensure the system can handle by truncation or summarization before feeding to LLM.
- Test section generation when web search yields no results – the draft should still be produced using just the profile (verify the content is not empty or the system doesn't hang waiting for input).
- Test concurrency if applicable: two section generation requests for the same project at the same time – ensure no DB conflicts (profiles being updated concurrently, etc.). This might involve threading or async tests.
- Input validation tests: ensure that if required keys are missing in JSON payload (e.g. no `section` specified), the API returns a 422 validation error (FastAPI would do this via Pydantic models automatically).

Test Organization: The test suite would be organized with unit tests (possibly in `test_tool_wrappers.py`, `test_toolchains.py`) and integration tests (maybe high-level scenario tests in `test_end_to_end.py`). The presence of `project/test/WP7/test_runner.py` and similar indicates that in this project, tests may be written more as scenario scripts rather than classic unittest framework tests. Nonetheless, they achieve coverage of major workflows and are manually inspected for correctness [51]. Going forward, these could be formalized into assertions rather than just printouts, to automatically verify success criteria.

4. Modernization Opportunities

During the reverse-engineering, several areas emerged where the design could be improved for maintainability, extensibility, and clarity:

- **Reduce Hard-Coding and Remote Dependencies:** The current system fetches prompt templates from a specific GitHub branch at runtime [47]. This introduces external dependency and potential latency or failure if offline. A modernization would embed these prompts into the application or a config service. For instance, the YAML prompt files could reside in the app package (or be baked into a container image) so that prompt loading doesn't rely on an HTTP GET each time. This also allows using version control tags or feature flags to switch prompts without changing code. Additionally, tool names and sequences are somewhat hard-coded (e.g., the planner uses literal strings like `"memory_retrieve"`). Introducing a configuration or plugin system for toolchains would decouple the logic – e.g., define the `generate_section` chain steps in a config file, so modifying the chain (adding a new tool) doesn't require code changes.

- **Streamline Orchestration vs. Chains:** There is some **duplication in how tasks are invoked**. For example, originally `PlannerOrchestrator` could run the `generate_section` sequence internally, but later a dedicated `GenerateSectionChain` class was created [14]. In tests, sometimes the chain is invoked directly, other times via the orchestrator [32]. Similarly, an AssembleArtifactChain was introduced while PlannerOrchestrator also handled assembly. This overlap can confuse the flow and maintenance – developers must update logic in two places. A modern refactor would clearly separate concerns: either the PlannerOrchestrator purely routes to chain classes (one unified way), or deprecate the orchestrator in favor of calling chain classes directly. The trend in WP24 was to break monolithic planner logic into discrete services [54] [55], which is good – following through on that, the orchestrator can be slimmed down to just a router that picks the right chain class for a given intent. All the detailed step logic should reside in the chain implementations (which seems to be the direction). This will eliminate duplicate implementations of similar functionality and ensure consistency.

- **Improve Modularity of Tools and Chains:** The toolchain code currently uses a lot of sequential, imperative logic inside the `run` methods. For example, IngestInputChain's run method not only calls the upload tool, but also merges profiles, cleans fields, and writes to DB all in one method [56] [11]. This makes it harder to reuse or test parts of the process. By adopting a more functional or pipeline approach, each step could be a separate function or method. For instance, splitting "parse input to text", "LLM extract profile", "merge with existing", "save to DB" into distinct methods would allow unit testing each in isolation and potentially reusing them (e.g., if a new ingestion method is added, it could reuse the same LLM extraction step). Similarly, the section generation chain could break out steps for "fetch context", "generate draft", "refine draft" into separate functions or even use

a workflow library. This modularity also helps in injecting alternate implementations (for example, a different refinement model or a different search provider) without altering the whole chain logic.

- **Explicit Tool Interface and Registry:** The system uses a ToolRegistry, but how tools are registered isn't clearly shown (perhaps each tool class is added to a dict in `tool_registry.py`). A more modern approach would use entry points or decorators to auto-register tools, making it easier to add new ones. Also, standardizing the interface (all tools have `validate(input)` and `run_tool(input)` as seen in QueryPromptGenerator [57]) is good; this could be enhanced by an abstract base class `BaseTool` that enforces this contract. It could also handle common concerns like logging inputs/outputs automatically (currently, `log_tool_usage` is called in chains manually [38]). A base class could call `log_tool_usage` for every tool run by default, reducing the chance of a developer forgetting to log a new tool's usage. Overall, a plug-in architecture (maybe using Python entry points) could let the system discover new tool classes dynamically, making the system more extensible for future features (e.g., adding a "QualityCheckTool" without modifying the orchestrator).

- **Decouple Business Logic from LLM Prompts:** Right now, a lot of the system's behavior is baked into prompt engineering (e.g., the entire extraction schema is defined in a prompt string [3], or the way citations are included is dictated by the prompt template). This is expected in LLM-based systems, but as a modernization, one could introduce more post-processing logic outside the LLM to reduce reliance on prompt correctness. For example, instead of asking GPT to output JSON and then parsing it [58], one could use OpenAI's *function calling* feature to get structured data more reliably (not sure if GPT-4 function calling was available in mid-2025, but if so, that would be a robust approach). Similarly, citation handling could be made more deterministic by having the LLM output markers and then programmatically attaching the actual URLs from the search results (ensuring no hallucinated sources). This would make the system more trustworthy and easier to validate. Essentially, critical data transformations (like assembling JSON, merging text) could be done with code, using the LLM primarily for generating natural language content.

- **Address Tight Coupling to Specific Document Schema:** The code and prompts are somewhat tailored to a particular document type (e.g., an "investment proposal" with certain sections and a notion of stage gates). If in the future the GovDoc Copilot should handle other document types (say briefing notes, policy reports, etc.), it would be beneficial to make the section template and logic data-driven. There is already a `gate_reference_v2.yaml` that likely defines section outlines [29]. However, the code may still explicitly reference "artifact_id" and "gate_id" in multiple places. A modernization could involve a **Document Template Registry**: a config where each artifact type lists its required sections, and any special instructions. The chains then iterate over that config rather than hardcoding section names in code or prompts. This reduces the need to write new code for new document formats—just update config and prompt templates.

- **Logging and Monitoring:** Enhancements could be made to better track the AI's performance and the workflow. Currently, ReasoningTrace is stored per run (likely as a YAML text log) [26], but there's an opportunity to integrate with observability tools. For instance, each step could emit structured logs or events. This would help in debugging issues (like if the webSearch returned irrelevant info, one can see the query and results in the trace). Modernizing here could mean integrating with an analytics DB or dashboard to monitor how often users regenerate sections, which prompts fail, etc. This ties in with the earlier mention of using a formal test to ensure each part is working; in production, similar monitoring would catch anomalies.

- **Error Handling and User Feedback:** The current design assumes a mostly smooth flow (the AI is expected to always produce something). But in practice, errors can happen (API errors, no content found, etc.). The system could be improved by handling these more gracefully. For example, if the OpenAI API fails mid-way, the orchestrator could catch the exception and mark the section status as "error" instead of just propagating a 500. Dependencies like the internet for web search should be optional – if web search fails, the system might continue drafting with just the profile, possibly warning the user that external info couldn't be retrieved. Also, validating final outputs (e.g., check that each section has a minimum length, or that the assembled doc isn't missing placeholders like "[REF]") would be a good addition to ensure quality. These validations and fallback strategies would make the copilot more robust for real-world use.

- **Eliminating Untracked Assumptions:** Some assumptions in the code are not explicitly enforced. For instance, the orchestrator assumes a project profile exists for the given project_id when generating a section (WP7 notes that profile is loaded early and passed throughout [59]). If that assumption fails (no profile), the system might break. A modern approach would explicitly check and either prompt the user to create a profile or even auto-trigger profile generation from minimal info if possible. Another assumption: the sequence of tools yields a good draft in one go. In reality, an analyst might want to iterate (the WP12 plan addresses this with phase checkpoints [40]). While full implementation of interactive loops might be future work, designing the system to allow re-entering a chain with adjustments is useful. For example, the user might say "Regenerate the Recommendation section with a more optimistic tone." The architecture should allow calling the section synthesizer with a tone parameter or an alternate prompt. Currently, to do that, one might have to hack the prompt template or code. Making the system more *parametric* (allow certain user instructions or style preferences to flow through) would enhance flexibility.

- **Performance and Scaling:** As more documents and sections are generated, the memory retrieval might slow down if logs aren't pruned or indexed. Modernization can include introducing a vector database (like Pinecone or FAISS) for memory, and implementing caching of LLM outputs (for example, if the same section is requested twice for the same profile without changes, reuse the prior output). Also, currently each tool call is sequential and synchronous. Using asynchronous execution (supported by FastAPI and Python async) could improve throughput, especially for I/O-bound steps like web API calls. For instance, web search and OpenAI calls could be run concurrently for multiple sections. The `GenerateFullArtifactChain` could potentially launch multiple section generations in parallel if the LLM API and system resources allow, reducing total time to draft the whole doc.

- **Code Quality and Documentation:** There are artifacts of quick development – e.g., inconsistent naming (some functions use `camelCase`, others `snake_case` like Python standards), and the use of magic strings. A cleanup pass to adhere to Python conventions and type hints would reduce bugs. Some files (especially in `framework/` or `legacy/`) can be removed or clearly separated if they are not relevant, to avoid confusion. Ensuring each public method or complex function has a docstring explaining its purpose would help future developers. The repository does have design docs and retrospectives (e.g., WP7_retrospective.md, WP17b_retrospective.md) – continuing this practice but also adding inline documentation in code is valuable.

In summary, **modularizing the pipeline**, **centralizing configuration**, and **hardening the system against failures** are key areas. By addressing these, GovDoc Copilot's Business Case drafting feature can become more robust and easier to evolve. The goal of modernization would be to transform the current prototype-

level orchestration into a production-grade service: one that is configurable, testable, and resilient, while maintaining the powerful LLM-driven capabilities that allow users to go from a blank page to a complete Business Case with just a prompt and a click.

Sources:

- WP7 Exit Report – *Project Profile & Document Generation E2E flow* 60 61
- WP17b Design – *Section drafting chain (memory retrieval, synthesize, refine)* 16
- Query Prompt Generator – *Generating search query from profile and memory* 18 22
- LLM Helpers – *Unified chat completion and prompt loading* 36 24
- Test Runner – *Usage of chains and orchestrator in sequence* 27 32

---

1 memory.yaml
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/memory.yaml

2 5 13 28 29 39 44 51 59 60 61 WP7_exit_report.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP7/WP7_exit_report.md

3 4 8 9 10 11 12 38 48 56 58 IngestInputChain.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/engines/toolchains/IngestInputChain.py

6 7 30 31 40 41 54 55 planner_phase_services_spec.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP12/planner_phase_services_spec.md

14 15 27 32 50 53 test_runner.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP7/test_runner.py

16 17 25 26 37 42 WP17b_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_design_plan.md

18 19 20 21 22 43 46 47 57 query_prompt_generator.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/tool_wrappers/query_prompt_generator.py

23 34 WP24_task_list.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP24/WP24_task_list.md

24 36 45 llm_helpers.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/utils/llm_helpers.py

33 35 49 52 test_script.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP24/test_script.py