ChatGPT

1. User Stories @ Definition of Ready

- *As a project stakeholder, I want to provide project details (text, file, or link) and have the system create a Project Profile, so that all key project information is structured and stored for reuse.*
  Acceptance Criteria:
- Given a valid input (e.g. a text description or uploaded file), when the input is submitted, then the system generates a structured `project_profile` (with fields like **project_id**, **title**, **scope_summary**, etc.) and saves it in the database [1] [2] .
- If a profile for the given project_id already exists, the system merges new details into the existing profile (preserving any non-empty existing fields).
- The profile entry includes a timestamp (last_updated) and is persisted via the ProjectProfileEngine (e.g. in a SQL table) [3] .
- Assumption/Gaps: The *OpenAI API* is available to parse raw text into JSON. The prompt schema for the profile is defined, and the system expects certain fields; if any required field (like project_id) is missing, the system will raise an error.

- **Dependencies:** Requires OpenAI GPT-4 API access and a database connection (SQLAlchemy models) for storing profile data [4] [5] .

- *As a policy author, I want to generate a document section (e.g. "Problem Statement") using AI, so that I can quickly get a first draft with relevant context and citations.*
  Acceptance Criteria:

- Given an existing project profile and a target artifact/section identifier, when the "generate section" intent is invoked, then the system loads the latest project profile from the DB [6] and retrieves relevant prior context (e.g. recent Q&A or memory logs).
- The system formulates a search query based on the project context and memory [7] [8] , uses it to fetch external info (e.g. government docs), and **summarizes** the web results for the LLM (if applicable) [6] .
- The AI then drafts the section content using all available inputs – project profile, internal memory, external research – following a prompt template (ensuring the output is well-structured and aligned with project goals) [9] [10] . A refinement pass is applied to polish tone, clarity, and completeness [11] .
- The final section draft is saved as an **ArtifactSection** record, including metadata like section_id, artifact_id, source citations, status, and timestamp [12] [13] . The system returns a success status with the generated text.
- Assumption/Gaps: The memory retrieval currently uses past prompt logs; a more robust embeddings-based retrieval ("queryCorpus") is not yet implemented [14] . Web search integration is limited to query generation – actual web data injection may require manual steps or future extension.

- **Dependencies:** Requires the **PlannerOrchestrator** to map the intent "generate_section" to the appropriate tool chain [15] . Depends on OpenAI API for content generation (via `section_synthesizer` and `section_refiner` tools) and possibly an external search API for

knowledge retrieval. Database access is needed to store the generated section and to fetch any stored memory or profile data.

- *As a user, I want to assemble a full artifact document from generated sections, so that I can review or share a complete draft policy.*
  Acceptance Criteria:

- Given an artifact ID (and assuming some or all sections have been generated), when the assemble command is invoked, then the system loads the project profile and all relevant ArtifactSection entries for that artifact from the database [16].
- The system composes the final document by formatting each section (e.g. adding headings) and merging sections in the correct order. It then performs a final refinement on the combined document for consistency and flow (e.g. ensuring the document reads as one cohesive piece) [17].
- The assembled document (final artifact text) is saved back to the database (e.g. as updated ArtifactSection records or a new entry denoting the complete artifact) and/or prepared for output. If configured, the document is also uploaded via the Drive integration (saving a Google Docs link) [17] [18].
- The user receives confirmation (e.g. a status or a link to the compiled document). The process should propagate the project_id through, ensuring the output is tied to the correct project context (for storage in the right place) [17].
- Assumption/Gaps: Google Drive upload (`storeToDrive`) was planned but may not yet be fully implemented [19] [18]. For now, assembly results might be delivered as text or a file download. It's assumed all required sections are available; missing sections might result in placeholders or an error.

- **Dependencies:** Depends on the PlannerOrchestrator to handle the "assemble_artifact" intent and invoke tools like loadSectionMetadata, mergeSections, finalizeDocument, etc. [17]. Relies on the database for retrieving section content and possibly on external APIs for export (Google Drive API for `storeToDrive`, if enabled).

- *As a user, I want to provide feedback or edits on a section and have the system revise it, so that the content can be improved iteratively.*
  Acceptance Criteria:

- Given an existing section draft and user feedback (e.g. "clarify this point" or edited text), when the revise function is invoked, then the system uses the feedback to prompt the AI to update the section content accordingly (maintaining style and context) [20].
- The revised section is saved to the ArtifactSection store (either as a new version or overwriting the draft) along with a log of changes. The status of the section is updated (e.g. marked as "revised" or version incremented) and the revision action is recorded in the reasoning trace or change log [20].
- If the feedback is an explicit edit, the system may perform a diff to highlight changes or ensure no loss of critical info. On success, the user can retrieve the updated section for review.
- Assumption/Gaps: It's assumed the revise functionality (`revise_section_chain`) is available and uses an LLM prompt to incorporate feedback. The exact method of providing feedback (direct text vs. tracked changes) might be undefined, and complex revisions may require manual verification.

- **Dependencies:** Requires the section to have been generated previously (existing ArtifactSection entry). Depends on LLM to intelligently apply the feedback. Also relies on logging mechanisms to track what changed for audit (e.g. in ReasoningTrace or a version log).

- *As a compliance officer, I want the system to log the reasoning and sources behind AI-generated content, so that outputs are traceable and trustworthy.*
  Acceptance Criteria:

- When any toolchain (ingest, generation, assembly, revise) is executed, then the system records a **ReasoningTrace** detailing each step taken (tool calls, decisions, timestamps) and any source URLs or references used [21] [22].
- Each AI call (e.g. draft generation or refinement) logs its prompt and a summary of the result to a PromptLog or trace file. The final output of a chain includes not only the content but also the collected trace (e.g. as a YAML or JSON structure) [23] [24].
- All logs are linked to the session/user and project for later retrieval. For example, after generating a section, one can retrieve the trace to see which tools were invoked (memory retrieval, web search, etc.) and what sources were cited [23].
- Assumption: ReasoningTrace is stored in a persistent form (database or file) and is accessible for debugging. Privacy is respected (no sensitive data in logs unless permitted). This logging does not significantly slow down the user experience.
- **Dependencies:** Relies on a logging subsystem (`log_tool_usage`, etc. [25]) invoked by each tool. Depends on persistent storage (YAML files or a database table for traces). No external API, but uses internal schemas for PromptLog and trace (ensuring they are structured for easy parsing).

2. 🛠 Design Decomposition

Technical Design: The system is organized into modular **toolchains** and **tools**, orchestrated by a central Planner/Orchestrator component. Key modules include:

- **Planner Orchestrator (`planner_orchestrator.py`):** The entry-point that maps high-level *intents* (like `"generate_section"`, `"assemble_artifact"`) to the appropriate toolchain class. It acts as a controller, ensuring the right sequence of tools run for a given request [15]. The orchestrator maintains a registry or mapping of available chains and uses a method (e.g. `select_tool_chain`) to instantiate and invoke them [26]. This design allows GPT or external callers to simply specify an intent and input, and the Planner will execute the corresponding workflow (e.g. `Planner().run({"intent": "generate_section", ...})` as illustrated [27]).

- **Toolchain Classes (`app/engines/toolchains/*.py`):** Each major workflow is encapsulated in a class with a uniform interface (typically a `run(inputs)` method) [15]. For example:

- *IngestInputChain*: Ingests raw input into a structured profile. Internally it delegates to an upload tool (text/file/link) to get raw text, then calls an OpenAI API to parse that text into a profile JSON (based on a schema) [2]. It then uses the **ProjectProfileEngine** to save the profile to the DB [28] [29].
- *GenerateSectionChain*: Implements the "compose_and_cite" pipeline for drafting a single section [30]. It sequentially calls tools: **memory_retrieve** (to fetch relevant context from logs or knowledge base), **section_synthesizer** (to compose an initial draft with citations using GPT), and **section_refiner** (to polish the draft) [11]. It uses **memory_sync** utilities to log each tool's output and to persist the final section via the DB (ArtifactSection) [11] [13].
- *AssembleArtifactChain*: Gathers all sections of an artifact and combines them. It likely uses tools like **loadSectionMetadata** (to retrieve section texts and titles from the DB), **formatSection** (to apply any formatting templates to each section), **mergeSections** (to concatenate sections into one document),

and possibly **finalizeDocument** (to run a final AI pass for coherence) [17] . Finally, it may call **storeToDrive** to upload the document if configured [17] . The chain returns a composite result (e.g. a confirmation or the merged content).

- *GenerateFullArtifactChain*: An end-to-end chain that automates an entire document creation. It likely orchestrates multiple GenerateSectionChain calls (for each required section defined in the artifact schema) and then invokes AssembleArtifactChain. This chain coordinates the workflow so that a single command produces a fully drafted and assembled artifact [31] .
- *ReviseSectionChain*: A chain (mentioned in tests) that takes user feedback for a section and applies a refinement. It might reuse the LLM with a "revise" prompt and update the ArtifactSection record, logging the changes [20] .

All toolchain classes follow similar patterns: they prepare input data, invoke a series of tool calls (often via a **ToolRegistry** or direct instantiation), log the outcomes, and aggregate a result. They typically return a dict or object containing the final output (and sometimes the save status and a reasoning trace) [11] .

- **Tool Components:** Individual tools perform atomic tasks within chains. They are often implemented as classes with a `run_tool(input_dict)` method (and sometimes a `validate` method) [32] . Tools are loosely analogous to functions or API calls that the chain uses. Examples include:
- **Upload Tools** (`uploadTextInput`, `uploadFileInput`, `uploadLinkInput`): Responsible for taking raw user input (direct text, file path, or URL) and returning unified text plus metadata. For instance, `uploadFileInput` likely reads the file content into text, and `uploadLinkInput` might fetch content from a URL. The Ingest chain picks the tool based on input_method and calls `tool.run_tool`, which returns `{"text": "...", "metadata": {...}}` [33] . These tools ensure the raw input is ingested in a consistent format.
- **Memory Retrieval Tool** (`memory_retrieve`): Fetches relevant historical data (prompt logs, Q&A pairs, knowledge base entries) related to the project or session. In the current design, this likely queries a **PromptLog** table or in-memory log to find entries matching the current artifact/section context [11] . This tool provides contextual paragraphs or summaries which the section synthesizer can use. (A planned improvement was to use semantic search on a vector store – a `queryCorpus` tool – but that wasn't implemented yet [14] .)
- **Query Prompt Generator** (`query_prompt_generator`): Takes the project profile and memory context and crafts a search query in natural language [34] [7] . It uses a Jinja-templated prompt loaded from YAML to instruct GPT to output an optimized query string [8] . The result (e.g. a query about policy precedents) can then be used by a web search step. This tool encapsulates the logic of turning context into a search query via LLM.
- **Web Search & Summarization** (`webSearch` & summarizer): Although not a fully standalone module in code, the design includes using the query from above to perform a web search and then summarize the findings. We see evidence of this in the prompt templates (placeholders for `{{web}}` content in the section synthesis prompt) and design docs [6] . Likely, an external call (via an API or a manual step in testing) fetches top documents, and then an LLM tool summarizes those results into a concise context paragraph. This summarized web context is then fed into the section synthesizer prompt. (In a future implementation, this could be a dedicated tool that automates the search and summary in one step.)
- **Section Synthesizer** (`section_synthesizer`): The core drafting tool. It assembles all inputs – project profile, memory snippets, any corpus or alignment data, and web info – into a single prompt (using a template from YAML) and calls the OpenAI API to produce a draft section [9] [10] . The output is expected to be a well-structured section of text, possibly with inline citations or

placeholders for sources. This tool likely uses the `llm_helpers.chat_completion_request` utility under the hood to call GPT with appropriate parameters (model, temperature, etc.).

- **Section Refiner** (`section_refiner`): A follow-up GPT tool that takes the raw draft from the synthesizer and improves it. The refiner might use a simpler prompt like "Polish the following text for clarity and consistency" (the exact prompt might be defined in code or a template). It corrects grammar, ensures the tone is professional, and that the section is complete. The output is a final version of that section's text [35] [11]. In the chain, this refined text replaces the draft.
- **Formatting/Merging Tools** (`formatSection`, `mergeSections`): Utility tools used during assembly. `formatSection` could apply standard formatting to a section (like adding section titles, numbering, or converting markdown if needed). `mergeSections` concatenates multiple section texts into one document in the correct order, inserting any necessary transitions or page breaks. These might be simple functions rather than AI calls.
- **Finalize Document (Refine Document)**: Similar to section_refiner but at full document scope. This would take the merged whole artifact text and possibly run an LLM to ensure the sections flow well together, the language is consistent, and no sections conflict or repeat. The "refine_document_chain" mentioned in development likely encapsulates this step. If implemented, it uses the project context (and possibly a summary of each section) to prompt GPT to refine the entire artifact.

- **Output/Storage Tools**:

    - **ProjectProfileEngine** (`project_profile_engine.py`): an interface to the database for project profiles. It provides methods like `save_profile` and `load_profile` to store or retrieve a project profile (likely using SQLAlchemy under the hood) [36] [37].
    - **ArtifactSection storage**: There may not be a singular "ArtifactSectionEngine" class; instead, the toolchains or memory_sync directly use SQLAlchemy models to commit sections. For instance, after generating a section, the chain might create an `ArtifactSection` ORM object and add it to the session (the model includes fields like section_id, artifact_id, text, etc. [38]). The design emphasizes this in WP17b deliverables ("Add ArtifactSection write logic" [39]).
    - **ReasoningTrace Logger**: Implemented in `memory_sync.py` (and possibly writing to a YAML or DB). The function `log_tool_usage` records each tool invocation with input and output summaries [40]. At the end of a chain, a comprehensive reasoning trace (sequence of steps and results) is stored, either as a YAML file or in a DB table, to allow traceability [22].
    - **StoreToDrive** (`storeToDrive.py`): A planned integration for Google Drive. Its purpose is to take the final artifact (perhaps as a markdown or PDF) and upload it to a structured Drive folder, then return a shareable link [41] [42]. In the current branch, this appears to be designed but not fully implemented (the WP20 plan outlines how it *should* work [19]). For now, it's a placeholder tool that might do nothing or a stub returning a dummy URL.
    - **PDF Renderer** (`pdf_renderer.py`): Likely a utility to convert content (like a summary or report) into a PDF file. In the context of the Concussion app (see below), it's used to produce a clinical PDF export [43]. It could also be repurposed for exporting the policy artifact if needed.

- **Ancillary Modules and Services:**

- **LLM Utilities** (`app/tools/utils/llm_helpers.py`): This module centralizes interactions with the OpenAI API. For example, it provides a `chat_completion_request(messages, temperature)` that wraps `openai.ChatCompletion.create` calls [44]. It also includes helper methods like `get_prompt` to fetch and parse prompt templates from the repository (via raw

GitHub URLs) [45]. By centralizing this, all tools call the API in a consistent way, and things like retries, API keys, or system-level prompt prefixes can be managed in one place.

- **Data Models** (`app/db/models/*.py`): Defines the ORM classes for persistence. We saw `ProjectProfile` model (fields for title, sponsor, dates, etc.) [5]. There is likely an `ArtifactSection` model (with fields per WP17b design: section_id, artifact_id, text, sources, etc. [13]) and possibly `ReasoningTrace` and `PromptLog` models. For instance, a `PromptLog` might log each tool run or user query (fields like input_summary, output_summary, user_id, timestamp). These models correspond to tables created in the SQLite or PostgreSQL database. A `database.py` probably initializes the engine and session (the code references SessionLocal in some tools [46]).
- **FastAPI App (for Concussion use-case)**: In addition to the PolicyGPT-focused components, the branch includes a FastAPI app exposing some routes (e.g. `/export_summary`, `/log_symptoms`). This is part of a **Concussion Recovery assistant** module (as evidenced by files like `export_summary.py`, `get_triage_flow.py`, etc., and the `tool_catalog_v2.md` describing those routes [47] [48]). This is somewhat separate from the PolicyGPT chains – it's a different context where user data (symptoms, incidents) are logged and PDFs are generated for clinicians. The technical design here involves FastAPI routers, Pydantic models for request/response [49], and engines like `epic_writer` and `pdf_renderer` to produce FHIR bundles or PDFs [50] [51]. While not the primary focus of the PolicyGPT flow, it shows the repo is a sandbox hosting multiple AI-driven tools. In a migration, these might be split out or modularized, but currently they share the same codebase for convenience.

Interface Design: The system's interface can be considered in a few ways:

- **Internal API (Orchestrator Interface):** Other components (like a GPT agent or a script) interact with the PlannerOrchestrator by calling a method (e.g. `planner.run(intent, data)`). Under the hood, this is how the GPT integration would work: the GPT could decide on an action like `assemble_artifact` and the system would execute it. For example, in the design it's noted that GPT would call: `result = planner.run({"intent": "generate_section", ...})` [27]. The input is a dictionary containing the intent and necessary parameters (project_id, artifact_id, etc.), and the output is a structured result (containing the content and metadata). This is not a user-facing UI, but a programmatic interface which could be invoked via CLI or a chat platform plugin.

- **Command-Line Interface (CLI) & Testing Scripts:** The repository includes **test scripts** (under `project/test/`) that act as a simple CLI to run these flows. For instance, `WP7/test_runner.py` demonstrates an end-to-end run: it manually sets up an input dict for IngestInputChain, calls `IngestInputChain().run()`, then calls PlannerOrchestrator for generating a section and assembling an artifact [52] [53]. Similarly, `WP24/test_script.py` runs each toolchain in isolation to verify outputs [54] [55]. These scripts indicate that a developer or tester can execute the chain classes directly from a Python environment. There aren't traditional CLI commands (like no ArgParse interface), but one could easily wrap these calls into a CLI tool or interactive notebook. The **Definition of Ready** for user stories implies that before actual UI integration, these scripts serve as the primary interface to confirm behavior.

- **Web/API Interface for Concussion App:** For the concussion recovery portion, a **FastAPI** web service is defined. Routes like `/log_incident_detail` or `/export_summary` accept HTTP requests with JSON payloads and trigger underlying logic [56] [57]. For example, a POST to `/export_summary` with a user_id will gather that user's data and return a PDF URL and a FHIR

medical data bundle [58] [59]. This shows a more traditional REST interface, but it's specific to that use-case. There is no evidence of a similar REST API for the PolicyGPT document generation (e.g. no `/generate_section` route in the catalog). The PolicyGPT flows were likely intended to be invoked by an AI agent (like a custom GPT integration) or via internal scripts rather than directly through an HTTP endpoint. A future implementation could expose such endpoints (e.g. a POST `/generate_section` that takes a project_id and section and returns the content), but currently this appears to be absent.

- **User Interface (UI) Considerations:** No front-end UI (web page or GUI) for the policy generation is present in this repo. The "frontend" mentioned in tool_catalog_v2 is for the concussion app's front-end that might call those FastAPI endpoints. For PolicyGPT, the UI likely was the chat interface of a "Custom GPT" where the AI could call the Planner's functions behind the scenes. For example, the user might prompt the GPT, "Draft the problem statement for project Alpha," and the GPT (with the system's help) maps that to a `generate_section` call. This kind of interface is implied by references to GPTPods and ProductPods (which seem to be AI agents or automated committers in the project). So, in summary, the interface for PolicyGPT is **indirect** – it's meant to be part of an AI-driven workflow rather than a direct form the user fills out.

- **Dependencies and Integration Points:** The design interacts with external systems at a few points:

- **OpenAI API:** All content generation and refinement relies on OpenAI's GPT models (e.g. GPT-4) via API calls [4] [60]. Thus, an API key and internet access are required in a deployed setting. The `OPENAI_API_KEY` is expected as an environment variable [61].
- **Database:** A relational database is used via SQLAlchemy for persistence. Whether it's SQLite (likely for local dev) or a cloud DB, the code uses models and session (SessionLocal) to store and fetch data [46]. Any migration must account for migrating this data or at least the schema (project_profile, artifact_section, etc.).
- **Google Drive API:** Intended for final document storage. OAuth credentials and the Drive SDK would be needed to enable `storeToDrive`. The plan includes using a service account or user OAuth flow and organizing files in specific folder structures [62] [18].
- **Requests to external URLs:** The system fetches its own prompt templates from GitHub raw URLs at runtime [63]. Also, any web search would involve HTTP requests to search engines or specific sites. These outbound calls mean the runtime environment needs internet connectivity and proper error handling for failures.
- **FastAPI (for certain tools):** If the concussion app components are run, the FastAPI server becomes an interface. That part depends on Uvicorn (or similar) to host, and clients (like a React app or other services) to consume the endpoints. It's a separate interface pathway, which could be run in parallel with the PolicyGPT modules.

Data Design:

- **Data Flow & State:** The system maintains state primarily in the **database** and transient memory during a chain execution. A typical flow begins with user input (text file, etc.), which after Ingestion yields a `ProjectProfile` record in the DB [29] [64]. That profile (and possibly initial memory entries) forms the context for subsequent steps. When generating a section, the chain will *read* from the DB (load project profile, fetch relevant logs) [6], then produce new content. The new section content is then *written* to the DB as an `ArtifactSection` record [12]. Multiple sections can be

written independently. When assembly is triggered, it will *read* all `ArtifactSection` entries for the given artifact (and possibly the profile again) and then write back a composed result (which could be updating an existing record or creating a compiled document record). Throughout, each tool's action may also write to a **PromptLog** or **ReasoningTrace** (likely in YAML or a separate table) to document what was done [22]. The final output (e.g. a Drive URL or PDF file path) might be stored in a `DocumentVersionLog` table along with metadata [65] if the Google Drive integration is completed. This ensures that all critical outputs are captured for later retrieval or auditing.

- **Key Data Schemas:**

- **Project Profile:** Represents a project's core information. Each profile is identified by a `project_id` (string key). The schema includes fields like `title`, `sponsor`, `project_type`, `total_budget`, `start_date`, `end_date`, `strategic_alignment` (how the project aligns with strategy), `current_gate` (stage of the project), `scope_summary`, `key_stakeholders`, `major_risks`, `resource_summary`, and `last_updated` timestamp [5] [66]. Most fields are text or nullable, with only `project_id` and perhaps `title` being required. This corresponds to a SQL table `project_profile` [67]. The profile acts as the single source of truth for project context, and is referenced by project_id in other tables.
- **Artifact Section:** Represents a section of a policy artifact (document). Each record would include a composite key or fields for `artifact_id` (which document or template it belongs to, e.g. "investment_proposal"), `section_id` (the specific section name or code, e.g. "problem_statement"), and `project_id` (to tie back to the project) – together these identify a unique section draft. Additional fields store the generated `text` content of the section, a list of `sources` or citations (possibly as a JSON or comma-separated string of URLs), a `status` (draft, refined, revised, etc.), `generated_by` (which could track which AI model or tool version produced it), and a `timestamp` for when it was last updated [38]. This likely corresponds to an `artifact_section` table. The content here is what gets assembled into full documents. If versioning is needed, version numbers or separate records would be used for new drafts of the same section.
- **ReasoningTrace / PromptLog:** The system likely stores a log of tool usage. While not explicitly shown in code, WP17b design mentions a `ReasoningTrace` stored as YAML [22] and logging to `PromptLog`. We can infer a **PromptLog** table might have fields like `id`, `tool_name`, `input_summary`, `output_summary`, `user_id`, `session_id`, `timestamp` [68]. In fact, the `log_tool_usage` call in IngestInputChain logs the tool name (e.g. "uploadFileInput"), an input summary (like the file name or a snippet of text), an output summary (first 200 chars of extracted text), plus session and user IDs and metadata [40]. These logs serve as the "memory" for subsequent memory_retrieve – by looking up recent logs for a given project or session, the system can recall what was done or what content was provided earlier. The **ReasoningTrace** might be more high-level, possibly capturing the entire chain's process (maybe linking to the PromptLog entries or summarizing decisions). It might be stored as a text blob (YAML/JSON) associated with a run ID, project_id, and intent. This trace would include the sequence of tool invocations and could be stored in a `reasoning_trace` table or simply written to an output file for offline analysis.
- **Document Version Log:** (Planned) For Drive integration, a `document_version_log` table is proposed [65]. Fields include an auto `doc_version_id`, `artifact_name` (or ID), `gate` (if the project management process gate is relevant), `file_path` (local or cloud path), `google_doc_url` (the link on Drive), `submitted_by` (user who triggered it), and

`submitted_at` timestamp. Each time a full artifact is uploaded, a new record would note where it lives on Drive. This allows retrieving the latest or past versions and auditing when they were generated.

- **Other domain-specific tables:** The concussion app part has its own data models (e.g. `ConcussionAssessment`, `SymptomLog`, `IncidentReport`, etc., not detailed in the PolicyGPT context). For example, `ConcussionAssessment` is queried in the export_summary flow [51]. These tables hold user medical data and are used in that context. They don't overlap with the policy generation data, except sharing the database. In a redesign, they would be separate modules or even services.

- **Data Transformations:** Data is often transformed between steps. For instance, the **ProjectProfile** is created by parsing unstructured text via LLM into structured JSON, then stored. That structured profile is then transformed into prompt-friendly text (profile_summary) for query generation [69]. Memory logs (which might be a list of JSON log entries) are transformed into a `memory_context` string by taking recent entries and truncating content [7]. When synthesizing a section, multiple data sources (profile, memory, web, etc.) are merged into one big prompt via Jinja template [10] – essentially transforming structured data back into unstructured prompt text for the LLM. After generation, the AI's unstructured output (just a block of text, possibly with JSON if it were told to format output) is parsed or handled. For the profile, the AI output is JSON parsed directly [70]; for section generation, the output is likely plain text which is directly saved. If citations are expected, the AI might output them in a structured way (e.g. "[1]" referring to sources), and the code would need to post-process to extract actual source links if provided. The **final assembly** step transforms multiple text fields into one text document, and then possibly into a PDF or Google Doc (binary or external format). Each of these transformations needs to preserve data integrity (e.g., ensure no data is lost when merging, and formatting doesn't accidentally alter content meaning).

- **Schema Evolution & Gaps:** The documents indicate a few places where the schema might evolve. For example, adding **project profile versioning** was considered (meaning the ProjectProfile could branch or maintain history) [71]. Also, handling alternate section schemas or templates is mentioned, implying the data model might need to accommodate different artifact structures (perhaps a table that defines which sections belong to which artifact type, e.g., gate_reference YAML) [72]. These are not fully implemented, but in migrating the system, one should design the data model to be flexible for such future needs (perhaps a separate metadata table for artifact templates and a version field in profiles and sections).

3. Test Suite Outline

To ensure the system works as expected, we propose a multi-layered test approach covering end-to-end scenarios, individual components, and data integrity. Tests can be outlined as follows:

System-Level Scenarios (Black Box):

1. **End-to-End Artifact Generation (Happy Path):**
   Scenario: A user provides a project brief (text file with project details) and requests a full artifact generation for a specific template (e.g. an "Investment Proposal").
   Steps: Simulate file upload input to the ingestion chain, then invoke the generate_full_artifact_chain

for that project and artifact.
Expected Outcome: The test asserts that a ProjectProfile is created (with correct fields from the text), all expected sections for the artifact are generated and saved (e.g. Problem Statement, Solution Approach, etc.), and the final assembled document text is produced. Verify that the final output includes content from each section in logical order and that a corresponding ArtifactSection or document record exists in the DB for the compiled artifact [31] . Also verify that a reasoning trace was logged for the run.
Validation: Check that no required fields in the profile or sections are null, the text is non-empty and reasonably formatted (e.g. contains multiple paragraphs), and that sources (if any were required) are present in the section texts.

2. **Section Generation with Context & Memory:**
Scenario: The system is asked to generate a single section (e.g. "Risks and Mitigations") for an existing project which already has some PromptLog history (simulated).
Steps: Pre-load the database with a ProjectProfile for the project and insert a few PromptLog entries (as if the user had prior conversations or uploaded related info). Invoke GenerateSectionChain for the "Risks" section.
Expected Outcome: The output should reflect use of the provided context – e.g. if a prior log entry mentioned a specific risk, the generated section should include or address that [35] . The test asserts that the section text is returned and saved, and that it's not just generic text but influenced by the memory (this might be checked via presence of keywords from the prompt log in the output). Also verify that the sources or any external info are integrated if the chain attempted a web search (this could be simulated by injecting a dummy web summary).
Validation: Ensure that if memory entries exist, the memory_retrieve tool did incorporate them (perhaps by checking logs or the trace). The section content should have a minimum length and contain the section heading. The function should handle gracefully if memory is empty (still returns a section).

3. **Profile Ingestion Error Handling:**
Scenario: A user inputs data with missing critical metadata.
Steps: Call IngestInputChain with an input dict lacking `project_id` in metadata or with an unsupported method type.
Expected Outcome: The chain should raise a clear error (ValueError) indicating the missing/invalid input [73] [74] .
Validation: Confirm that no partial profile is saved to the DB when input is invalid. For a file path that doesn't exist, verify that the error is handled (the uploadFileInput tool should throw an exception which propagates or is caught and logged). This test ensures robust input validation paths are in place.

4. **Assembly with Partial Sections:**
Scenario: Some sections of an artifact are already in the database (perhaps manually written or generated earlier), and the user requests assembly.
Steps: Insert 2 out of 3 required ArtifactSection records for an artifact, then call AssembleArtifactChain for that artifact.
Expected Outcome: The chain should gather the existing sections, detect that one section is missing. Depending on intended behavior, it might either (a) call the generation for the missing section on the fly, or (b) assemble what's available and mark placeholders for missing parts, or (c) return an error.

Validation: Determine expected design – if the intended design is that all sections must be present, then test asserts that assembly fails gracefully with an informative message. If dynamic generation is intended, then after assembly, all sections including the previously missing one should now be present (implying the chain invoked generation internally). The final document should include content for all sections (no gaps). Also verify that the combined text flows correctly (for instance, check that section boundaries in the output correspond to the ordering defined by the artifact template).

5. **Revision Workflow:**
   Scenario: A user requests a revision on a section with specific feedback (e.g. "elaborate on X" or provides a modified sentence).
   Steps: Ensure an ArtifactSection exists for the section. Then invoke ReviseSectionChain with the section ID and the feedback input.
   Expected Outcome: The test should observe that a new version of the section text is produced which addresses the feedback (for example, it includes the elaboration requested). The ArtifactSection entry should be updated or a new entry created for the revised version, and a log of the revision should be recorded (perhaps in a ChangeLog or by differences in text).
   Validation: Compare the revised text to the original to confirm the specific change was made. Ensure that the revision did not unintentionally drop other content. Check that the ReasoningTrace for the revision includes the feedback details (to confirm the tool correctly received and used the input).

6. **Concurrent Generation Stress Test:** (If applicable)
   Scenario: Simulate multiple requests coming in (for example, two different projects being processed at once).
   Steps: Run two chain processes in parallel threads: e.g., generate_full_artifact_chain for Project A and Project B simultaneously.
   Expected Outcome: Both processes complete successfully without interfering with each other's data (Project A's sections all have project_id A, and B's have B, no mix-up).
   Validation: Ensure that database entries for profiles and sections are correctly attributed to the right project. Check for any race conditions – e.g. the sequence numbers or timestamps should make sense (no collisions), and no deadlocks occur in the database. This ensures the system can handle multi-user or multi-session usage, which is likely in a real deployment with concurrent chats.

Component/Unit-Level Tests:

- **Tool Outputs:** For each tool, create unit tests with controlled inputs:
- *uploadFileInput:* Provide a small text file and ensure the tool returns the exact text and proper metadata (filename, size, etc.). Use a temporary file for this test.
- *uploadLinkInput:* Mock an HTTP request (or use a sample HTML file) to verify it can fetch and strip content from a URL. Check handling of inaccessible URLs (should throw an error or return a message).
- *memory_retrieve:* Seed a fake PromptLog (could be an in-memory list of dicts or a temporary SQLite with entries) with known entries, then query for a specific artifact or session. Assert that the tool returns the most relevant entries (e.g. most recent 5) and in the correct format (likely a list of snippets). If the log has nothing, confirm it returns an empty list or suitable default.
- *query_prompt_generator:* Construct a dummy project_profile dict and a list of memory entries, call `run_tool`. Verify that the returned `query` string contains phrases from the profile (e.g. project

title or keywords from scope) [69] [75] . This test might use monkeypatching to stub `chat_completion_request` to return a predictable string (so we're not reliant on the actual OpenAI API for unit testing).
- *section_synthesizer:* Here we'd definitely stub the OpenAI call. Supply representative strings for profile, memory, etc., ensure it builds the prompt correctly. We can spy on `llm_helpers.get_prompt` usage if applicable. The test asserts that `run_tool` produces an output containing the key parts (e.g. if we stub GPT to echo back inputs, we can check that the output includes those inputs, meaning they were passed through correctly). Also test behavior when one of the context inputs is empty or very large (to see if it truncates or handles it).
- *section_refiner:* Stub the LLM to return a known transformation of input text (like input text in uppercase as a dummy "refinement"). Check that `run_tool(raw_text)` returns the refined text (uppercase in this dummy scenario). This ensures that the refiner is actually applying the LLM result and not, say, returning the original text unchanged.

- *log_tool_usage (memory_sync):* Unit test this logging function by calling it with sample data. Instead of a real DB, point it to an in-memory log or a dummy that captures the input. Verify that the formatted log entry contains tool_name, input_summary, output_summary truncated to expected lengths, and that it handles special cases (very long text gets truncated, None values are handled, etc.).

- **Chain Logic:** We should test the chains in isolation with mocked tools where possible:

- *IngestInputChain:* Replace actual OpenAI call with a stub that returns a predetermined JSON (for example, given a prompt, return `{"project_profile": { ... }}` with some fields). Then run IngestInputChain with a text input. Verify it calls ProjectProfileEngine.save (which we can monkeypatch to a dummy) with the parsed profile. Check that the returned dict from `run()` contains `status: "profile_saved"` and the `project_id` matches the input metadata [64] . Also test the branching where an existing profile is found: pre-load ProjectProfileEngine.load to return a dict and ensure merging logic works (fields from new profile override empties in old) [28] .
- *GenerateSectionChain:* We can monkeypatch ToolRegistry to return mock tool instances for memory_retrieve, section_synthesizer, section_refiner. Configure these mocks: memory_retrieve returns some dummy context list, section_synthesizer returns a draft text "Draft X", section_refiner returns "Refined X". Then run the chain's `run(input)` and expect the output dict's `final_output` or equivalent contains "Refined X". Also verify that it attempted to save via memory_sync or DB (perhaps monkeypatch ArtifactSection model's save). Essentially, this ensures the chain orchestrates calls in the right order and passes data along. Edge case: test when memory_retrieve returns nothing (the chain should still proceed with just profile data).
- *AssembleArtifactChain:* Mock the database calls to return sample sections (e.g. three sections of text). Also mock finalizeDocument (if it calls GPT) to just concatenate with a separator. Then run assemble chain. Verify the result combined text is in the correct order and format. If the chain is supposed to call storeToDrive, mock that call so it doesn't actually attempt external upload but returns a dummy link; then check that the dummy link is perhaps included in output. If the chain uses PlannerOrchestrator or ToolRegistry internally, ensure those are accounted for in the test setup.

- *PlannerOrchestrator:* This can be tested by registering a dummy intent and chain. For example, add a fake chain in ToolRegistry or planner's registry (like intent "dummy_test" mapped to a class that just sets an internal flag when run). Then call PlannerOrchestrator().run("dummy_test", {...}). Assert that the dummy chain's run was invoked and the result was returned. Also test an unknown intent: it should throw an error or handle it gracefully (no toolchain found scenario).

- **Data Validation & Integrity Checks:**

- After running an ingest + generate flow in a test, directly query the database (or the SQLAlchemy session) to ensure the ProjectProfile and ArtifactSection tables have the expected entries. For example, confirm that `project_profile.project_id` matches in both tables (foreign key relation, if enforced) and that fields like title, etc., are not null where they shouldn't be. If possible, test that `last_updated` is auto-updated on subsequent profile edits.
- Test that numeric and date fields in the profile are correctly typed: e.g. input "total_budget: 1.5" becomes a Numeric in the DB, or an invalid date in text results in that field being None (the code does attempt date parsing and sets None on failure [76] [77]). This is important for data integrity — the test can feed a profile text with an incorrectly formatted date and then check that in the saved profile, the date field is null and a warning was logged.
- Verify unique constraints and error on duplicates: attempt to save a ProjectProfile with an existing primary key via IngestInputChain and ensure it merges rather than duplicates (no integrity error). Similarly, ensure that two ArtifactSections with same composite keys (project, artifact, section, gate) aren't created unintentionally – subsequent generation of the same section could update instead of insert, or use a versioning scheme. This might require simulating the scenario and checking either that a duplicate was avoided or intentionally allowed with a different key (depending on design).
- Check that deletion or update cascades are not problematic: e.g. if a ProjectProfile were deleted (not sure if UI allows it), ArtifactSections might become orphans. While this might be out of scope for the main flows (no deletion functionality exposed), a migration should consider referential integrity. A test could manually delete a profile and see what happens when generating a section (should error that profile not found).
- **Security**: Though not explicitly mentioned, one can include tests for injection or misuse. For example, test that if a user inputs a malicious string in the project details (like SQL injection attempt in text), it is handled as plain text and doesn't break anything (since the system uses parameterized queries via ORM, it should be fine). Or test that extremely long inputs are handled (maybe truncated in prompt or split) without crashing.

By covering these tests, we ensure the system's core functionalities meet the expected behavior and that the data produced is consistent and valid. The mix of scenario tests and granular unit tests will catch both high-level integration issues and low-level logic bugs.

4. Modernization Opportunities

As we prepare for migration or reimplementation, several improvements and refactoring opportunities have been identified to make the system more robust, maintainable, and forward-compatible:

- **Decouple and Modularize Components:** The current repository serves multiple purposes (policy document generation and a concussion app) within one codebase. This could be separated into distinct modules or services. For example, the PolicyGPT toolchains can be one service, and the Concussion FastAPI app another. This separation would reduce confusion and make each system easier to deploy and maintain. If they remain in one repo, a clearer modular structure (with namespaces or folders distinguishing the contexts) would help. In a microservice architecture, they would communicate via APIs rather than share a database. This also allows scaling each piece independently based on load (e.g. heavy GPT usage vs. web API usage).

- **Introduce a Standard Orchestration Framework:** The custom planner and toolchain pattern could be reimplemented or augmented using established libraries (like **LangChain** or similar workflow orchestration frameworks) for LLMs. These frameworks provide abstractions for memory, tools, and chains of calls, which might reduce the need for custom ToolRegistry and manual prompt handling. Adopting such a framework could bring in features like easier parallel calls, better error propagation, and integration with vector stores for memory out-of-the-box. It would also be more familiar to new developers. However, care should be taken to preserve the logic and data flows already in place (e.g. the specific prompt templates and multi-step refinement approach) when translating to a new framework.

- **Enhance Prompt and Template Management:** Currently, prompt templates are stored in YAML and fetched from GitHub at runtime [63]. In a production setting, it would be better to package these prompts with the application or load them from a config service, rather than hitting a raw GitHub URL (which could introduce latency or break if no internet). A modernization could involve a **Prompt Management System** – perhaps storing prompts in a database or using feature flags to swap prompt versions. Additionally, more prompts (for refinement, revision, etc.) could be templatized to allow easier tuning without code changes. Using Jinja2 for template rendering is good; we can extend that to all AI prompts (the refiner prompt, for instance, could be externalized rather than hard-coded). This makes the system more adaptable as requirements change or new models with different prompting styles are used.

- **Improve Logging and Monitoring:** The system logs usage of tools and stores reasoning traces, but these could be leveraged more. We should integrate with a logging framework or APM (Application Performance Monitoring) for better observability. For instance, instrument each chain run with timing and success/failure metrics. Ensure that logs (especially from `log_tool_usage` and errors) are output to a centralized log or dashboard for real-time monitoring. Moreover, currently reasoning traces are stored but not actively analyzed; a modern approach might automatically flag anomalies in traces (like if the AI had to retry many times or gave an empty response) and alert developers. This would make the system more enterprise-ready by facilitating debugging and continuous improvement based on actual usage patterns.

- **Strengthen Error Handling and Validation:** In the migration, pay attention to making the system resilient. Some parts of the code assume ideal input (for example, assembling an artifact assumes all sections present). We should harden these by gracefully handling missing data or external API failures. For instance, if OpenAI API fails or times out, implement retries or fallbacks (maybe a simpler model or a cached response). If Google Drive upload fails, the system could still return the document content with a warning instead of failing completely. Input validation can be more thorough – e.g. if numeric fields in profile come in scientific notation or with currency symbols, strip/convert them safely, etc. In testing, these failure modes should be covered so the system can handle edge cases without crashes.

- **Integrate Vector Database for Memory:** A clear opportunity is to upgrade the **memory_retrieve** mechanism. Currently it likely fetches recent logs, which might not scale or capture semantic relations. Introducing a vector store (like Pinecone, FAISS, or Weaviate) to index project knowledge (profile, prior sections, relevant documents) would allow semantic search. The `queryCorpus` tool noted as a backlog item [14] can be implemented to query this store. This would improve the relevance of the context the AI gets, leading to better section drafts. It also enables long-term

memory beyond just the latest prompts – earlier conversations or external docs can be recalled even if not explicitly keyword-matched. Along with this, an embedding generation step (using OpenAI embeddings or similar) would be added whenever new content is saved (profile or sections) to keep the vector index updated.

- **Concurrency and Scaling:** To modernize for a production environment, consider how to scale the system. The current design is mostly synchronous and single-threaded (since it was a sandbox). We can introduce asynchronous processing for I/O-bound steps: for instance, fetching web content or calling external APIs could be `async` to improve throughput. We could also allow the generation of multiple sections in parallel when using `generate_full_artifact_chain` (e.g. spawn tasks for each section, since GPT calls for different sections are independent). This will cut down total generation time significantly for large artifacts. If using FastAPI or another web framework for an API, ensure the endpoints are non-blocking and can handle concurrent requests. Additionally, using a task queue (Celery or similar) for long-running jobs (like full artifact generation) could free up request threads and provide status updates to users.

- **User Experience & Collaboration:** On the front-end side, once the core is migrated, a UI can be built or improved. For example, integrating with Google Docs for real-time collaboration (as hinted by Journey C in WP20 design [78]) would be a modern touch – after the AI drafts a document, a user could edit it on Google Docs, and the system could later ingest the edited version to learn from the changes. Implementing that diff and update loop would make the system interactive and continuously learning. Also, providing a dashboard for users to see all generated artifacts, sections status (draft, revised, approved), and to trigger actions (generate or assemble) via buttons would greatly enhance usability. In modernization, one could create a web app or chatbot interface that uses the orchestrator API under the hood.

- **Security and Permissions:** A modern reimplementation should also incorporate security best practices. For instance, if multiple users or teams use the system, ensure that project data is segregated (one user shouldn't access another's project profile or documents). If exposing an API, implement authentication (API keys or OAuth). Logging should avoid sensitive data exposure (perhaps redact or omit certain fields from debug logs). The Google Drive integration must handle credentials securely (no plain tokens in code – use env vars or vaults). While these aspects were not central in the sandbox, they become crucial in a production migration.

- **Completing Unfinished Features:** There are clear partially implemented features (marked as TODO or in WPs). Modernization should address these to unlock full value:

- Finish the **Google Drive integration** so that on artifact assembly, the document is automatically uploaded and the link stored [19]. This involves finalizing OAuth setup and testing with real Drive API calls.
- Implement the **context_summary** in prompts (WP24 tasks T17/T18) – e.g., when generating a section, include a summary of previous sections if available to improve coherence. This would require tracking the content of already generated sections and possibly having a tool to summarize them. It can significantly improve the final document quality by preventing contradictions or repetition across sections.
- Develop a **tool catalog/manifest** (mentioned as missing [79]) that declaratively lists all tools and their inputs/outputs. This could allow dynamic loading of tools and easier maintenance (the Planner

could read this catalog rather than a hardcoded registry). It also helps when exposing capabilities (imagine an API endpoint that lists available actions and their descriptions, generated from this catalog).

By addressing these opportunities, the system will be easier to maintain and extend, and better prepared for real-world deployment. The migration is not just a code port – it's a chance to realign the design with best practices, ensuring longevity and adaptability for the "AI delivery" platform moving forward. Each improvement should be weighed for effort vs. benefit, but collectively they set a strong foundation for the next generation of the system.

---

1 3 6 12 16 17 25 71 72 WP7_exit_report.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP7/WP7_exit_report.md

2 4 28 29 33 36 37 40 60 64 68 70 73 74 76 77 IngestInputChain.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/engines/toolchains/IngestInputChain.py

5 66 67 ProjectProfile.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/db/models/ProjectProfile.py

7 8 32 34 44 63 69 75 query_prompt_generator.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/tool_wrappers/query_prompt_generator.py

9 10 generate_section_prompts.yaml
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/prompts/generate_section_prompts.yaml

11 14 15 21 22 23 24 26 27 30 35 WP17b_exit_report.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_exit_report.md

13 38 39 WP17b_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_design_plan.md

18 19 41 42 62 65 78 79 WP20_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP20/WP20_design_plan.md

20 31 61 test_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP24/test_plan.md

43 45 changelog.yaml
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/outputs/changelog.yaml

46 49 50 51 57 58 59 export_summary.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/export_summary.py

47 48 56 tool_catalog_v2.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/delivery/tool_catalog_v2.md

52 53 test_runner.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP7/test_runner.py

54 55 test_script.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP24/test_script.py