ChatGPT

User Stories @ Definition of Ready

- **As a project team member, I want to ingest a new project input (text or file) to create a structured Project Profile so that the system can persist key project details for all future document workflows.**
  Acceptance Criteria: Upon providing a valid input method (`text`, `file`, or `link`) with required metadata (`project_id`, etc.), the system extracts project fields via GPT and saves a `project_profile` record in the database [1] [2]. The saved profile must include title, dates, budget, stakeholders, etc., with `project_id` and `last_updated` always populated [3] [4]. If required fields are missing or the input_method is invalid, the service returns a clear error (e.g., "Missing project_id" or invalid method) without partial saves [5] [6].
  Assumptions: OpenAI GPT-4 API is available for text-to-profile extraction [7], and the SQL database is accessible and initialized with the `project_profile` schema [8]. The input file (if any) is reachable by the service.
  Dependencies: OpenAI API key configured; SQLAlchemy models (ProjectProfile) synced with the database; the `uploadFileInput` tool for file parsing is registered in the ToolRegistry [9].

- **As an AI Product Pod (GPT agent), I want to automatically generate a document section draft based on an existing project profile and prior notes so that I can produce high-quality content with context and citations.**
  Acceptance Criteria: When the planner receives an intent to `"generate_section"` with inputs like `artifact_id`, `section name`, and `project_id`, it should orchestrate the **GenerateSectionChain** [10]. This chain must: (1) retrieve relevant memory entries (e.g. recent prompts or notes) for context [11], (2) call an LLM to **synthesize** a draft section, and (3) call the LLM again to **refine** the draft for clarity and completeness [11]. All three sub-tools – e.g. `memory_retrieve`, `section_synthesizer`, `section_refiner` – are executed in order [12]. On completion, the system saves the output as an `ArtifactSection` record linked to the correct `artifact_id` and `gate` (stage) in the database [12]. The section content should include any citations or source links if applicable. The planner returns a JSON containing the final section text and a trace of steps taken. If any step fails (e.g., no profile found, LLM error), the chain aborts and returns an error status with no DB commit.
  Assumptions: A Project Profile for the given `project_id` exists in the DB (the chain will load it at start [13]). Historical prompt logs (PromptLog entries) or embedded memory are available for `memory_retrieve` to pull context [11] – if not, that step may return an empty context gracefully. OpenAI API is accessible for content generation. The `ReasoningTrace` mechanism is in place to log the chain-of-thought.
  Dependencies: `planner_orchestrator.py` has an entry mapping `"generate_section"` intent to the `GenerateSectionChain` [14]. The ToolRegistry is populated with the required tool classes for memory lookup and LLM calls. Database models for `ArtifactSection` and `ReasoningTrace` are defined and migrated. (Future integration with a vector store for embeddings is noted but currently mocked or minimal [15].)

- **As an AI Product Pod or user, I want to assemble a complete artifact (document) from approved section drafts so that I can produce a final deliverable (e.g. a PDF or markdown) with all sections in order.**
  Acceptance Criteria: Given an intent `"assemble_artifact"` with inputs like `artifact_id`, `project_id` and target `version`, the system (via **PlannerOrchestrator**) invokes an **AssembleArtifactChain** that composes the final document [16]. This chain will: (1) **load** all relevant section texts and metadata for the specified artifact (e.g., all sections from the `ArtifactSection` table for that project's gate) in the correct order, (2) **format** or adjust each section (e.g., ensure consistent headings, merge content), (3) **merge** sections into one cohesive document, and (4) **finalize** the document (perform any cleanup like removing placeholders) [16]. Finally, it uses `storeToDrive` to upload the compiled document to an external Google Drive folder, returning a shareable link [16]. The acceptance is met if the final artifact is saved (e.g., a PDF/Markdown in Drive) with all sections included, and a corresponding entry is logged (e.g., in a `DocumentVersionLog` with the Drive URL) [17]. The system should also update any relevant status (like marking sections as "compiled" or updating the project's current gate). Errors (like missing sections or Drive API failure) should be caught and reported without leaving partial data (e.g., if upload fails, the final assembly is marked failed).
  Assumptions: All required sections have been generated and stored beforehand. Google Drive API credentials (OAuth service account) are configured and valid [18]. The folder structure (as defined in `drive_structure.yaml`) exists or will be created by the tool. The user has appropriate permissions for the target Drive.
  Dependencies: The `assemble_artifact_chain.py` (or equivalent logic in PlannerOrchestrator) is implemented to orchestrate `loadSectionMetadata`, `formatSection`, `mergeSections`, `finalizeDocument`, and then call `storeToDrive` [16]. The `storeToDrive.py` tool uses the Google Drive Python SDK and is configured with the service account JSON or token [19] [18]. A `DocumentVersionLog` model/table is available to record the output version and link [20].

- **As a system integrator or QA engineer, I want all AI tools and actions to be exposed via a clear API (OpenAPI spec) so that GPT agents or external services can invoke these tools consistently and we can validate their inputs/outputs easily.**
  Acceptance Criteria: The project provides an **OpenAPI JSON** (`openapi.json`) that enumerates every available endpoint/tool with schemas. For example, there are REST endpoints under `/tasks/` for task management (start, complete, reopen tasks) [21], under `/memory/` for memory indexing and search [22], and others like `/audit/validate_changelog`, `/git/rollback_commit`, `/actions/list` etc. Each tool route is documented with its purpose and parameters [23]. The API must be served by the FastAPI app (e.g., via an `/openapi.json` route) and be in sync with the actual implementation. **Definition of Ready** is achieved when calling `GET /actions/list` returns a categorized list of tools [24] and matches the OpenAPI spec, and when GPT's planner can reference this spec to choose tools. All input models (e.g., task schemas, memory query) should have validation (likely via Pydantic) such that invalid requests receive 4xx errors.
  Assumptions: The FastAPI application (`main.py` and `api_router.py`) includes all routers for tasks, memory, audit, etc., and FastAPI's automatic documentation is enabled. Tools are registered either via code or via a manifest (`tool_catalog.yaml`) to populate the OpenAPI. (It's assumed that `tool_catalog.yaml` is kept up-to-date to reflect new tools/endpoints, though a WP plan noted it was pending [25].) The user (or GPT agent) will use the documented endpoints exactly as specified.
  Dependencies: FastAPI and Pydantic libraries for API and schema definitions; the

`tool_registry.py` to map tool calls within the backend (for internal invocation when GPT uses a tool); and the existence of `openapi.json` in the repo (possibly auto-generated or manually exported) to serve as a contract for available operations [26]. Any UI or external system can reference this spec to integrate with the AI Delivery system.

Design Decomposition

Technical Architecture (Modules, Chains, Tools, Libraries)

Modular Engines and Toolchains: The system is organized into modular *chains* and *tools* that correspond to phases of document processing. Each **toolchain** is a Python class encapsulating a multi-step workflow for a specific intent. For example, `IngestInputChain` handles input ingestion (text/file to profile) and `GenerateSectionChain` handles drafting a section [10]. Chains implement a standard `run(inputs: dict)` method that orchestrates sequential calls to tools and other engines [14]. Each step in a chain is performed by a **Tool**, typically a small class with a common interface (`validate()` and `run_tool()`) [27]. Tools are often single-responsibility actions, such as retrieving memory, calling an LLM to synthesize text, or saving output to storage [11]. They are implemented in the `app/tools/tool_wrappers` directory – for example, `memory_retrieve.py`, `section_synthesizer.py`, `section_refiner.py` – each defining a `Tool` subclass that can be invoked by name [11]. The system uses a **Tool Registry** (`ToolRegistry`) to map tool names to their classes, enabling chains to dynamically fetch and use tools by string key [28] [29]. This registry may be populated by scanning the tool modules or via a config file (the presence of `tool_catalog.yaml` suggests an intent to externalize this mapping) – ensuring new tools can be added with minimal code changes [30].

Planner Orchestrator: At the heart is the `PlannerOrchestrator` (sometimes referred to simply as Planner) which serves as the **controller** deciding which chain to run for a given high-level intent [14]. It contains logic to select the appropriate toolchain based on an input "intent" string (e.g., `"generate_section"`, `"assemble_artifact"`) and then invokes that chain's `run()` method with the provided context [14] [31]. The orchestrator acts as the interface for GPT agents to trigger complex workflows – rather than calling individual tools, a GPT Pod can call the Planner with an intent and let it handle the sequence [31]. Internally, `PlannerOrchestrator.run(intent, inputs)` likely uses a mapping (dictionary or if/elif) to route intents to chain classes, as indicated by design docs [14]. The orchestrator also ensures that common preparatory steps are done, such as loading the Project Profile from the DB at the start of each run (so that every chain has access to up-to-date project context) [32]. It may also handle top-level exception catching – if any tool in the chain fails, the orchestrator can log the failure and return an error response to the caller.

Tool Wrappers and External Services: Each tool wrapper is responsible for a specific atomic action. Many tools wrap calls to external services or complex logic behind a simple interface: - *LLM Integration:* Tools like `section_synthesizer` and `section_refiner` call OpenAI's GPT-4 via the OpenAI Python SDK [7] [33]. The code centralizes this in helper functions (e.g., `chat_completion_request` in `llm_helpers.py`) which constructs the prompt messages and temperature, and returns the model's reply [34]. This abstraction makes it easier to call GPT and swap models if needed. Prompt templates are defined externally in YAML files and loaded at runtime; for example, the `query_prompt_generator` tool fetches a prompt template from the repository's `app/prompts/generate_section_prompts.yaml` on GitHub [35], allowing prompt text to be updated without code changes. - *Memory and Context:* The `memory_retrieve` tool likely interacts with stored conversation logs or embeddings. Initially, it searches the **PromptLog** (a log of previous user inputs and AI outputs) to pull any entries relevant to the current section or artifact [11]. The design anticipated vector embedding search ("**queryCorpus**") for more advanced memory recall [15], but currently the tool might use simple keyword or ID-based lookup (as a stub). In future, a vector database (Qdrant or similar) could be integrated, with an embedder syncing documents to that store (a reference to a Qdrant-based `memory_sync.py` exists in a different context, indicating planning for embeddings). - *Persistence Tools:* Some tools handle saving outputs. For example, `memory_sync.py` is used to log tool outputs to the database [36] or to update indices. It likely provides functions like `log_tool_usage(tool_name, input_summary, output_summary, ...)` to record each step's result for traceability [37]. Similarly, a tool like `loadSectionMetadata` fetches section info from DB for assembly, and `storeToDrive.py` wraps the Google Drive API for uploading files [38] [17]. - *Utility Tools:* e.g., `webSearch` is mentioned as a tool in the section generation chain [12]. This suggests an ability to do a web search (perhaps via an API or custom integration) to bring external information. It's likely implemented as a Tool that calls out to an external search API and returns results. Another is `formatSection` or `mergeSections` – these may simply manipulate text (concatenate sections, apply markdown formatting) and do not require external services.

Key **external libraries** include: **FastAPI** (for serving the API and defining endpoints), **SQLAlchemy** (for ORM models and DB interactions [39]), **OpenAI SDK** (for GPT calls [40]), **Requests** and **PyYAML** (for fetching and loading prompt templates [35]), **Jinja2** (for templating dynamic parts of prompts [41] [42]), and possibly **redis-py** (if Redis caching is implemented for quick data access – see "Redis" section below).

Database Layer: Under `app/db`, the system defines SQLAlchemy models for persistent objects. For example, `ProjectProfile` is a model for the project profile table with columns for title, sponsor, dates, etc. [4]; similarly, we expect models like `ArtifactSection` (storing section drafts with fields like section_id, artifact_id, text, status) and `ReasoningTrace` (storing the reasoning or chain-of-thought logs). These models inherit from a Base (declarative base) defined likely in `database.py`, and the system probably uses SQLite or Postgres via SQLAlchemy. There is also mention of a `PromptLog` and `WebSearchLog` – these could be models capturing each user query or web search made for traceability [43] [44]. The **ProjectProfileEngine** (`app/engines/project_profile_engine.py`) acts as a DB interface for project profiles – providing methods to `load_profile(id)` and `save_profile(data)` so that higher-level code (like IngestInputChain) doesn't need to write SQL itself [8]. This engine likely uses the SQLAlchemy session to fetch or upsert records in the `project_profile` table.

Redis (Caching and Pub/Sub): The presence of a `redis` folder suggests an intended use of Redis, possibly for caching or background task queueing. While the code references to Redis were minimal in our review, a likely design is to use Redis as an in-memory store for ephemeral data like session state or interim results. For example, the system might cache recent memory retrieval results or store ongoing task states in Redis to reduce database load. It could also serve as a message broker if the architecture later incorporates asynchronous task execution (e.g., offloading heavy LLM calls to worker processes). In the current sandbox, this integration appears to be either stubbed or lightly used (no explicit calls found in the chain logic); however, ensuring the design can accommodate Redis helps with scaling and decoupling components. We note it here as part of the design for potential migration: for instance, the PlannerOrchestrator could push a task to a Redis-backed queue for asynchronous execution by a worker, rather than blocking an API call.

Interface Design (APIs, Routes, Integration Points)

FastAPI Application & Routing: The application is implemented as a FastAPI service (e.g., instantiated in `main.py` with `app = FastAPI()`), exposing a variety of REST endpoints that correspond to "tools" or actions the system can perform. These routes are organized via APIRouters, likely separated by domain: - **Task Management Endpoints:** under a `/tasks` router – handles operations on tasks in the project's task.yaml. For example: `POST /tasks/start` to start a task, `POST /tasks/complete` to complete it, `POST /tasks/append_chain_of_thought` to log intermediate reasoning, `POST /tasks/reopen` to reopen a closed task [21]. These allow external input (from a human lead or a Pod agent) to manage the task lifecycle. - **Memory Endpoints:** under `/memory` – for example, `POST /memory/index` to index new files into the memory (and memory.yaml) [45], `GET /memory/search` to query stored memory by keyword [45], `POST /memory/diff` to compare listed vs actual files, `POST /memory/validate-files` to verify all memory references are valid [22]. These tie into the Memory management features of the system. - **Artifact and Document Endpoints:** though much of the artifact creation is handled by internal chains, the system likely exposes some routes for retrieving outputs or initiating sequences. For instance, there might be a `POST /artifact/generate_section` that under the hood calls PlannerOrchestrator (this could be how GPT triggers it via a tool invocation). The OpenAPI spec may not expose raw Planner calls to end-users, but rather specific tool routes (since GPT uses the spec to decide). Another example is `/actions/list` – which returns a list of all tools and routes, effectively a self-discovery endpoint for the AI [24]. - **Audit/QA Endpoints:** e.g., `/audit/validate_changelog` to verify that all recent file changes have corresponding changelog entries [46]. This would scan commits and the changelog.yaml. - **Utility Endpoints:** like `/git/rollback_commit` to revert a commit by ID [46], or `/getFile` and `/batch-files` to fetch file content from the repo (the guide mentions these for allowing the AI to read code or docs on demand [24]). - **Profile/Project Endpoints:** Possibly endpoints exist to get or update the project profile (e.g., `/project/profile/{id}` GET or PUT) for external interfaces, though GPT likely uses internal chains instead. The openapi.json would clarify this if available.

Each endpoint is defined likely in `api_router.py` or similar, grouping the routers. For instance, `api_router.py` might include something like:

```python
from fastapi import APIRouter
router = APIRouter()
router.include_router(tasks_router, prefix="/tasks")
router.include_router(memory_router, prefix="/memory")
...
```

In turn, those sub-routers map to functions or class-based views that invoke the appropriate backend logic (possibly calling PlannerOrchestrator or directly a Tool function). The OpenAPI specification (openapi.json) is either generated by FastAPI (if the app runs) or exported – it serves as a contract showing all these routes, their inputs, and outputs.

Integration with GPT (OpenAPI as Tools): A critical interface is how GPT agents use these endpoints. The system is **OpenAPI-driven for GPT tooling** [47] [23]. That means the GPT has access to the OpenAPI spec and can call the API endpoints as tools (similar to how OpenAI Plugins work). For example, when GPT decides to use a tool, it essentially crafts an HTTP request to one of these documented endpoints (the orchestration likely handles this behind the scenes). The PlannerOrchestrator may be invoked directly by GPT via an endpoint or function call when complex multi-step operations are needed, but simpler atomic actions (like logging a thought, or getting the next task) are direct API calls. This design makes the AI's capabilities transparent and externally callable in a controlled manner.

User Interface (CLI/Docs): While no traditional UI is described, there is a CLI test harness (`project/test/...` scripts like `test_runner.py`) for developers to simulate E2E flows [48] [49]. This indicates developers can run the pipeline locally, providing test inputs and observing JSON outputs. Moreover, documentation in `project/docs` (like the Onboarding Guide) serves as a UI in terms of guiding human operators to interact with the system via prompts and the available tool set [50] [51]. If a front-end were built, it would likely call the same FastAPI endpoints (for example, a web dashboard could hit `/tasks/next` to show the next recommended task). Additionally, the system outputs artifacts like PDFs to an external system (Google Drive) which serves as an interface for end-users to retrieve final documents.

Inter-service Communication: Within the codebase, different components interact via function calls (e.g., Planner calling a chain, chain calling ToolRegistry to get a tool, tool calling DB engine). There isn't an internal message queue or event bus observed (no active use of Redis pub/sub yet), so the design is presently synchronous and in-process. The **Redis** component could be introduced as an interface between processes or services if the architecture is expanded (for example, using Celery with Redis broker for background tasks). For now, all interfaces are either in-memory function calls or REST API calls at the boundary.

Data Design (Schemas, Memory Flow, Transformations)

Database Schema & Models: The core data objects and their schemas: - **Project Profile:** As noted, the `project_profile` table stores project-level info. Key fields include `project_id` (primary key), `title`, `sponsor`, `project_type`, `total_budget` (numeric), `start_date`, `end_date` (dates), `strategic_alignment`, `current_gate` (int for gate/stage), `scope_summary`, `key_stakeholders`, `major_risks`, `resource_summary`, and `last_updated` timestamp [4]. Most fields are optional (nullable) except a few like title may be marked not null per design (initially some were non-null causing issues, which were fixed by allowing nulls where appropriate [52]). The system ensures at least `project_id` and `last_updated` are always present [3]. This profile acts as a single source of truth for project context [53]. - **ArtifactSection:** This table holds individual section drafts that belong to an artifact (document) and a gate. Expected fields (from design) are: `section_id` (unique identifier for the section, possibly a slug like "problem_statement"), `artifact_id` (which document this section belongs to, e.g., "investment_proposal_concept"), `gate_id` (project stage number, e.g., 0 for concept gate) [54], `project_id` (to link back to the project), `text` (the content of the section, likely Text type), `sources` (any source citations/URLs included), `status` (draft/approved), `generated_by` (which pod or tool created it, e.g., "ProductPod" or an AI identifier), and a `timestamp` or last_updated. The chain populates this after refining the section [55] [11]. In the WP7 test, after generation, the section was indeed stored in `ArtifactSection` table with correct data [12]. - **ReasoningTrace / PromptLog:** The **ReasoningTrace** likely stores the chain-of-thought or decision trace for each major operation. It might not be a SQL table – the WP17b plan mentioned "ReasoningTrace YAML + summary string" [56]. Possibly the final reasoning is dumped to a YAML or JSON file for transparency (and attached in test evidence [57]). However, there could also be a `prompt_log` or `web_search_log` table in the DB for logging each prompt and search. For example, if `memory_retrieve` queries recent prompts, those prompts must have been logged somewhere (PromptLog table with columns like `id`, `project_id`, `content`, `speaker`, `timestamp`). WP7 test results mention a `web_search_log` missing a `project_id` field causing a schema error [43], implying such a table/model exists and needed that foreign key set. The system logs usage of each tool via `log_tool_usage()` which likely writes to one or multiple of these logs (including which user/session invoked it, what input summary and output summary were) [37] [58]. - **Supporting Tables:** There is mention of a `DocumentVersionLog` in the context of Drive integration [20]. This would store records of each document assembly (fields: version id, artifact_id, gate, file path, Google Drive URL, who submitted, when). A `StageLog` might track stage transitions (from the concussion app context, or maybe general usage of get_stage_guidance). The schema includes a `current_gate` in profile to know the stage, but logs of stage changes might be kept if needed. - **Redis Data:** If Redis is used, data likely stored there would be ephemeral and keyed by project or session. For example, caching the last memory search results under a key like `memory_cache:{project_id}` or storing an ongoing assemble task status under `assemble_status:{project_id}`. Without specifics in code, one can assume any data kept in Redis is duplicate of DB state for quick access or transitional state that doesn't warrant a DB entry.

Memory Embedding and Retrieval Flow: The system's approach to memory is twofold: 1. **Symbolic Memory (PromptLog):** All interactions (prompts and outputs) are logged in a structured way so that they can be later retrieved by simple queries. The `memory_retrieve` tool likely filters these logs by artifact or topic. For example, if generating a section "Problem Statement," it might pull all PromptLog entries tagged with "problem_statement" or relevant keywords. The result is a collection of past Q&A or notes that the synthesizer will use as additional context [11]. This forms a chain of evidence the AI can use to not repeat itself and to maintain consistency. 2. **Vector/Semantic Memory:** While the initial implementation may not fully incorporate it, the design clearly anticipates it. The mention of *embeddings* and a future `queryCorpus` tool [15] suggests that documents or knowledge base could be embedded (using e.g., OpenAI embeddings or similar) and stored in a vector index. Indeed, a separate memory sync script (possibly named the same) was referenced for Qdrant integration. In a migrated or modernized version, after each document or relevant text is uploaded (via `/memory/index`), an embedding would be created and stored in a vector store, and `/memory/search` would perform a similarity search. In the sandbox code, if `memory_retrieve` is implemented simply, it might just return recent PromptLog entries until the vector search is in place.

Data Transformations: At each stage, data is transformed and validated: - **Text Input → JSON Profile:** IngestInputChain takes raw text (and file content) and, using GPT with a schema prompt, transforms it into a JSON that matches the ProjectProfile schema [59] [60]. The chain then merges this JSON with existing profile data (to not overwrite fields that were already present unless new info is provided) [61]. It also cleans and types the fields (converting strings to dates, numbers where needed) [62] [63] – this was critical to avoid serialization issues (e.g., empty strings to None, numeric types correct) [52]. Finally, it serializes and saves the profile. - **Profile/Memories → Draft Section Text:** GenerateSectionChain takes the project profile (possibly injecting key fields like title, scope, stakeholders into the prompt) and a summary of memory (like last 10 memory lines) to feed the LLM prompt [64] [65]. The LLM output, which is a raw section text, is then fed (with context) into a second LLM call for refinement [11]. This two-step transformation ensures initial content followed by polishing. The output is then wrapped in a JSON structure with metadata (the chain likely returns `{"final_output": ..., "trace": ..., "save_result": ...}` where `save_result` might contain the DB save confirmation) [11]. - **Multiple Sections → Final Document:** The assemble process takes multiple text sections and combines them. This could involve sorting by an outline order (perhaps defined in `gate_reference_v2.yaml` – which likely lists which sections belong in which artifact for each gate) [66]. Indeed, `gate_reference_v2.yaml` is referred to as a section schema [66], likely mapping gates to required section IDs and their titles. The assemble logic uses that reference to ensure all required sections are present and properly ordered, hence the earlier issue where a section ID didn't match the expected schema and had to be fixed [67]. Once concatenated, formatting might convert markdown to PDF or ensure the content flows. The `finalizeDocument` tool could convert the compiled markdown to PDF (perhaps using a library or API) before upload. Finally, the Google Drive upload returns a URL which is then written to the `DocumentVersionLog` along with metadata [17].

Data Validation & Integrity: Throughout these transformations, validation is key: - **Schema Validation:** The system relies on predefined schemas (YAML specs or Pydantic models). For instance, the YAML prompt for profile extraction explicitly defines the fields and types expected [68]. Similarly, `gate_reference_v2.yaml` serves as a contract for what sections and IDs are valid for each artifact/gate, and the code cross-checks outputs against it (ensuring, for example, that a "Problem Statement" section gets a matching `section_id` that the final assembly expects) [67]. - **Error Handling:** The code uses try/except blocks around DB operations and LLM calls. IngestInputChain catches exceptions when trying to load an existing profile (logging and proceeding with an empty profile if none) [69]. It also raises clear ValueErrors for missing required input fields up front [70]. These errors propagate to the API responses so that a bad request doesn't result in partial processing. - **Logs and Traceability:** Every tool usage is logged (tool name, a brief input summary, output summary, user/session IDs) via `log_tool_usage` [58]. This provides an audit trail in the data: one can trace, for any given artifact generation, which tools were invoked in what order and what they produced. The ReasoningTrace (likely stored as JSON/YAML in an outputs folder) provides a step-by-step account including intermediate GPT prompts and decisions, aiding debugging and verification [71] [72].

In summary, the data design ensures that the state of the project and artifacts is persisted in a relational DB for reliability, while transient working data (like intermediate prompts or external info) is fetched on the fly or cached. The structures are in place to support advanced memory via embeddings and quick caching via Redis as the system scales, even if the initial sandbox implementation uses simpler approaches.

Test Suite Outline

To ensure the system works as expected and is robust to changes, we outline a comprehensive test suite with multiple levels:

1. End-to-End System Tests (Black Box): These tests simulate real usage patterns, treating the service as a whole (via its API or via orchestrator calls): - *Project Ingestion Flow:* Provide a sample input (text file or text blob) and associated metadata to the ingestion endpoint or directly call `PlannerOrchestrator.run("ingest_input", data)`. Assert that the response indicates success and that a new ProjectProfile entry is created in the database with correct values. Verify that `last_updated` is set and non-null fields are as expected (e.g., `title` from the input appears in DB) [73] [1]. Also verify that a log entry for the upload tool was created (e.g., in PromptLog or usage log). - *Section Generation Flow:* Using a known project profile (from above or a fixture), call the generate section API (or Planner with intent "generate_section") with a specific `artifact_id` and `section` name. Assert the response contains a well-formed section draft and trace. Then confirm the side effects: a new row in `artifact_section` table for that section is present, with the expected `section_id`, content text, and links to the correct project and artifact [12]. Check that the reasoning trace was logged (either by retrieving a `ReasoningTrace` entry or checking an output file) and that each tool (`memory_retrieve`, `section_synthesizer`, etc.) reported a usage log entry. This test should be run with variations: one where relevant memory exists (to see it influence output) and one where no prior memory exists (ensuring it still succeeds, perhaps with an empty context). - *Artifact Assembly Flow:* After generating multiple sections for an artifact, call the assemble artifact endpoint or function. Assert that the final output status is success and contains (or provides access to) a compiled document link [16]. Validate that the document content is a concatenation of the sections in the correct order. If possible (in a test environment), intercept the call to `storeToDrive` with a mock that simulates upload to avoid external dependency, and verify it was called with a file containing all sections. Check the database/log: a new DocumentVersionLog entry should be created with the expected artifact_id, version, and a dummy URL (from the mock) [20]. Also ensure that each tool in the chain (`loadSectionMetadata`, `mergeSections`, etc.) logged its step. An edge case scenario to test: assembling when a section is missing or when Drive is unreachable, expecting a graceful failure message. - *Task Lifecycle:* (If relevant endpoints are available) simulate a user starting a task via `/tasks/start`, then appending some reasoning via `/tasks/append_chain_of_thought`, and completing it via `/tasks/complete`. Verify that each action returns the appropriate response (e.g., starting a task returns a prompt for the Pod with task details [21]). After completion, check that `task.yaml` (or the in-memory representation) was updated with the task's new status and that `changelog.yaml` or `reasoning_trace.yaml` entries were made for any outputs committed during the task [74]. This ensures the integration between user commands and internal state is sound.

2. Component/Unit Tests: - *Tool Unit Tests:* For each Tool in `tool_wrappers`, write tests for its `validate` and `run_tool` methods in isolation. For example, for `query_prompt_generator.Tool`, provide a minimal `input_dict` with and without required fields (`project_profile`, `memory`) to confirm it raises `ValueError` on missing data [70]. For the run logic, since it calls external URLs and OpenAI, substitute the network calls with mocks: e.g., patch `requests.get` to return a known YAML string, and patch `chat_completion_request` to return a preset query. Then assert the output of `run_tool` is as expected (e.g., it should return a dict with a `"query"` key) and that the prompt composed internally matches the template. Similarly test `memory_retrieve.Tool`: you can seed a fake PromptLog (perhaps monkeypatching a method it uses to fetch logs) and see that it returns a list of relevant entries or an empty list when none found. - *Chain Unit Tests:* Test the logic of each chain class by injecting doubles for its tools. For instance, for `GenerateSectionChain.run`, instead of calling real tools, monkeypatch the ToolRegistry to return dummy tool instances with a controlled behavior (one returns a fixed memory list, one returns a fixed draft text, one returns a fixed refined text). Then call the chain's `run` with sample inputs and verify that it assembles the outputs correctly and calls the expected sequence. This isolates the chain's orchestration (e.g., ensuring it passes the right data between tools and handles outputs properly). One should check that if a tool returns an error or empty result, the chain responds as designed (propagating error or continuing with defaults). - *Database Integration Tests:* Using a test database (e.g., SQLite in-memory), test the `ProjectProfileEngine` and other DB interfaces. For example, create a ProjectProfile object and save via `save_profile`, then load it and confirm data persists. Test merging logic: if `save_profile` is supposed to merge with an existing record (as IngestInputChain does by loading old, merging fields, then saving) [61], simulate an existing DB entry and see that after saving new data, fields are correctly updated or preserved. Similarly, verify that `ArtifactSection` insertion works and that all NOT NULL constraints can be satisfied with the data provided (this would catch issues like the initial non-nullable fields problem [52]). - *OpenAPI Schema Test:* As a quality check, load the `openapi.json` and ensure it is valid (conforms to OpenAPI 3.0) and that for each expected endpoint, the schema matches what the backend expects. This can be done by programmatically comparing the OpenAPI spec to Pydantic models or by using FastAPI's `TestClient` to hit each endpoint with a sample request and ensuring a 200 or appropriate error for invalid input. For instance, call the `/tasks/start` with missing fields and expect a validation error, then with all fields and expect success, confirming that Pydantic model enforcement is correct. This cross-checks documentation with implementation [26] .

3. Data Validation Tests: - *Schema Conformance:* Write tests to ensure outputs conform to defined schemas. E.g., after running GenerateSectionChain, take the resulting `section_id` and verify it appears in the `gate_reference_v2.yaml` for that artifact's gate (using the YAML file as oracle) – this ensures section naming is correct [67] . Also verify that all required sections per gate in `gate_reference_v2.yaml` can round-trip through the system (this could be a parameterized test that for each section in the schema, attempts to generate a dummy section and then assemble them). - *Null/Type Handling:* Insert edge values into the system to ensure they are handled: e.g., a Project Profile with an empty string for sponsor -> after `IngestInputChain`, confirm it stored as NULL in DB (and not an empty string) [75] . Try extremely large numbers for budget or future dates to see if the system can parse and not overflow types. Test serialization of complex types: ensure that datetime fields (like `last_updated`) can be JSON serialized in all API responses (the test runner's `default_serializer` indicates this was a known issue [76]). Essentially, any data leaving the system via API should be JSON serializable – a test can iterate over a sample response dict and ensure no Python objects (datetime, Decimal, etc.) remain. - *Logging and Trace Consistency:* After performing a chain run, verify that for each tool that should have been called, a log entry exists with matching names and IDs. For example, generate a section and then query the PromptLog/ReasoningTrace for entries – you should find entries for `memory_retrieve`, `section_synthesizer`, `section_refiner` with the corresponding session or user id [37] [77] . This catches any mismatches between the log schema and usage (like the earlier bug where `log_tool_usage()` had a wrong parameter name causing a missing field [77]).

By covering end-to-end scenarios, individual components, and data integrity, this test suite will give confidence that the system is behaving as intended. It will also catch regressions: for instance, if someone changes the ProjectProfile schema or prompt format, the tests on generation and DB save will flag if something no longer aligns (like a non-nullable field introduced without default would cause a test failure when saving profile). Additionally, having explicit schema tests prevents silent mis-matches between what the AI expects and what the system provides.

Modernization Opportunities

Through reverse engineering the current implementation, several opportunities emerge to improve modularity, maintainability, and scalability of the system:

- **Adopt Database Migrations:** The project currently creates and alters tables in an ad-hoc manner (e.g., manual dropping/recreating to fix schema issues [52]). Implementing a migration framework (such as Alembic for SQLAlchemy) would provide version-controlled DB schema changes [78] . This ensures that as models like ProjectProfile or ArtifactSection evolve, migrations can be applied systematically across environments. It also helps during deployment or reimplementation – one can bring up the database to the latest schema with a single command, reducing the risk of "works on my machine" issues.

- **Refine Planner Orchestrator Structure:** The PlannerOrchestrator is central to intent mapping, but as more toolchains are added, the mapping logic could become unwieldy (a long if/elif or dictionary of string to class). A modernization could involve a more **declarative registration** of toolchains. For example, each Toolchain class could be annotated or registered in a central `tool_catalog.yaml` or in the code via a decorator. The orchestrator can then dynamically discover available toolchains and their intents. This was hinted by the plan to use `tool_catalog.yaml` (which was missing as of a certain work package) [25] . Completing that implementation would allow adding new capabilities without modifying the orchestrator code – one would update the YAML or add a new class, and the system would pick it up. Additionally, improving the Planner's test harness will harden it against future changes – e.g., simulate various sequences and edge cases [79] .

- **Enhance Separation of Concerns:** Currently, some tools perform multiple roles (for example, IngestInputChain not only calls GPT but also merges profiles and writes to DB). Introducing clearer separation can help. One idea is to factor out **pure logic** vs **side effects**:

- Keep GPT prompt creation and parsing logic in one layer (which could be easily unit tested with sample prompts and outputs).
- Keep database writes in a repository or service layer (e.g., a ProfileService that knows how to merge and save profiles). This way, the chain can call ProfileService for saving without concerning itself with SQLAlchemy details.

- External API integrations (Drive, web search) could be abstracted behind interfaces so they can be stubbed or replaced (e.g., a `StorageService` with methods `upload_file(project_id, gate, artifact, file)` that `storeToDrive` implements). This would ease migrating to a different storage backend in future or adding caching.

- **Improve Prompt Management:** The use of external raw GitHub links to fetch prompts at runtime is brittle [35] – it introduces a dependency on external availability and complicates offline or self-hosted deployments. A modern approach is to package these prompt templates with the application (for instance, include the YAML files in the repository and load them from the local filesystem or a config directory). If dynamic updates are desired, consider a small versioning mechanism or an admin interface to update prompts, rather than pulling from GitHub on each call. This change would improve reliability and performance (no network call needed for prompts) and simplify migration (the system is more self-contained).

- **Consistency in Tool Interface:** All tool wrappers currently use a similar pattern but this could be reinforced by using an abstract Base class or interface for tools. For example, define a base `Tool` class in a common module that all tools inherit, enforcing the presence of `validate()` and `run_tool()` methods. This was implicitly done (many tools have a class named Tool), but making it explicit can help developers follow the pattern and possibly enable polymorphism (e.g., a list of tools could be managed uniformly). If not already using Pydantic for tool inputs, consider defining Pydantic models for each tool's input and output schema. This would automatically provide validation and documentation (and could be tied into the OpenAPI spec generation for each tool route).

- **OpenAPI and Documentation Synchronization:** Currently, there's an **openapi.json** in the codebase and hints at maintaining it manually [26] . It's easy for such specs to drift from the code implementation. A modernization step is to **auto-generate OpenAPI docs** from the FastAPI routes (which FastAPI does out of the box) or integrate the manual spec as tests (e.g., test that the manual openapi.json matches FastAPI's generated schema). Alternatively, if manual curation is needed for clarity, create a script to regenerate it whenever the code changes, or include it as part of CI. This ensures GPT always has an up-to-date tool spec. The WP20 notes about syncing OpenAPI with the tool catalog [26] underscore this need – it will reduce confusion during migration if documentation and reality are aligned.

- **Logging and Monitoring:** While extensive logging is present (usage logs, reasoning traces), centralizing logs can be a next step. For example, instead of writing reasoning traces to disparate YAML files, the system could aggregate them in a timeline or database for easier querying (perhaps using an ELK stack or a simple admin UI). Additionally, adding unique identifiers to each orchestrator run (trace IDs) would help trace through logs. As the system scales or moves to production, consider instrumenting it with monitoring – track how long each tool call takes (to identify slow steps like external API latency) and how often certain errors occur. This data can guide further refactors (e.g., if webSearch is slow, maybe cache results in Redis; if a particular validation error is frequent, maybe enforce it earlier).

- **Redis Utilization or Removal:** If the Redis integration is currently a stub or minimally used, clarify its role and either leverage it fully or remove it to simplify architecture. For instance, if Redis is intended for caching recent PromptLog queries, implement that: after `memory_retrieve`, store the results in Redis with a short TTL so that if the same context is needed again it's faster. Or if it's for session state (storing current tasks per user session), implement that such that stateless API calls become stateful sequences when needed. On the other hand, if analysis shows that Redis isn't providing clear benefits in the current scope, it might be omitted in a refactor to reduce complexity (one less service to maintain). The key is to avoid having unused components that could become outdated or introduce deployment overhead.

- **Refactor External Integrations:** The Google Drive integration is a valuable feature but can be abstracted. For modernization, one could create a generic `FileStorage` interface with methods `upload(file, path)` and `download(file_id)` and have two implementations: one for Google Drive, another for local or S3, etc. This would make testing easier (one can plug in a local filesystem implementation for tests) and future migrations (maybe moving from Drive to another storage) smoother. Similar treatment can be given to any other external services (e.g., if web search currently calls an API, encapsulate the API key and URL in a single module).

- **Code Style and Organization:** As the project grows, organizing the code into clear packages is helpful. The current structure under `app/` is fairly good (tools, engines, db, prompts). We might suggest moving the FastAPI routers into their own package (e.g., `app/api/`) rather than top-level `api_router.py`, to keep all API-related code together. Also, adopting a naming convention (some files use CamelCase like `IngestInputChain.py`, others snake_case) uniformly to snake_case Python files would be more PEP8 compliant and make the project look more consistent. While cosmetic, these changes ease new developer onboarding.

- **Testing and CI:** Lastly, improving the test suite and possibly setting up continuous integration is a modernization must. Given the complexity of orchestrated AI behavior, having automated tests run on each change will catch regressions like the `tool_name` parameter mismatch or serialization issues early [77]. Invest in more **edge case tests** – e.g., ensure that if OpenAI API returns a malformed JSON for project profile, the system handles it (maybe by retrying or cleaning the string) rather than crashing. The retrospective notes already pointed out the need to simulate more edge cases in PlannerOrchestrator tests [79] – implementing those will make the system more resilient to unexpected inputs or API changes.

Each of these improvements will support a smoother migration or reimplementation. By cleaning the separation of concerns and externalizing configurations, the core logic (the "what" it does) is more easily transferable to new frameworks or environments. By having robust migrations and tests, the system's behavior is well documented and reproducible, which is crucial when porting to a new stack or scaling out. Overall, these steps will make the AI Delivery system more **maintainable**, **extensible**, and **production-ready** for future developments.

Sources: The analysis above is based on the repository's documentation and code excerpts, including Work Package reports and test outcomes that highlighted current design and issues (e.g., profile integration [1] [66], section generation flow [10] [11], assembly process [12], and noted improvements [78]). These guided the identification of areas where the system can be improved while preserving its successful patterns.

---

[1] [3] [8] [9] [13] [32] [37] [53] [66] WP7_exit_report.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP7/WP7_exit_report.md

[2] [12] [16] [43] [44] [52] [57] [67] [73] [77] WP7_test_results.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP7/WP7_test_results.md

[4] [39] ProjectProfile.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/db/models/ProjectProfile.py

[5] [6] [7] [28] [29] [33] [58] [59] [60] [61] [62] [63] [68] [69] [75] IngestInputChain.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/engines/toolchains/IngestInputChain.py

[10] [11] [14] [15] [30] [31] [36] [55] [71] [72] WP17b_exit_report.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_exit_report.md

[17] [18] [19] [20] [25] [38] WP20_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP20/WP20_design_plan.md

[21] [22] [23] [24] [45] [46] [47] [50] [51] [74] onboarding_guide.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/docs/onboarding_guide.md

[26] tool_catalog_v2.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/delivery/tool_catalog_v2.md

27 35 41 42 64 65 70 query_prompt_generator.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/tool_wrappers/query_prompt_generator.py

34 40 llm_helpers.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/utils/llm_helpers.py

48 49 76 test_runner.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/test/WP7/test_runner.py

54 56 WP17b_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_design_plan.md

78 79 WP7.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/retrospectives/WP7.md