ChatGPT

Prompt Ingestion and Memory System

- **User Story:** *As a government analyst, I want to upload project materials (text, files, links) and have the assistant remember key details for use in drafting policy documents.*
- **Design & Implementation:** GovDoc Copilot provides an **IngestInputChain** to handle user inputs. This chain dispatches to upload tools (`uploadTextInput`, `uploadFileInput`, or `uploadLinkInput`) that extract raw text and metadata [1]. For example, uploading a PDF or link triggers content scraping, then parsing via an LLM into a structured **ProjectProfile** with fields like title, scope, stakeholders, etc. [2] [3]. The raw text is logged in a **PromptLog** entry (in a database) with a summary and metadata for later retrieval [4]. The ProjectProfile is saved to the database (e.g. via `ProjectProfileEngine`) to persist the project's context. In memory, prompt logs act as a running "memory" of all user-provided content. (Per design, the system was intended to use **Redis or a vector DB** for chunked storage, but currently it relies on the PromptLog database.) The **memory_retrieve** tool later queries PromptLog for relevant entries (by project/session) to gather prior inputs [5] [6]. Duplicates are filtered and results returned as a list of snippets (e.g. workshop notes) for use in drafting. This provides a working memory of user inputs across the session.
- **Testing:** In a test scenario, a user uploading a sample document should result in a new PromptLog entry and an updated ProjectProfile. For example, running `IngestInputChain` with `input_method: "link"` on a URL returns a `project_profile` JSON and logs the content. Automated tests (see `test_runner.py`) cover end-to-end ingestion: asserting that after ingestion, calling `memory_retrieve` returns the expected text snippet [7]. One would also verify that invalid inputs (missing `project_id` in metadata, etc.) are handled gracefully (the chain raises errors if required metadata is missing [8]).
- **Gaps & Improvements:** *Implementation gap:* The current memory retrieval pulls all entries by tool type but lacks semantic filtering – e.g. it does not yet embed or rank content by relevance to a section (the code for vector search is not present). Incorporating a vector database (e.g. Chroma or RedisSearch) for semantic memory lookup would improve relevance, as originally planned. *Modernization opportunity:* Use streaming or chunked upload for very large files, and integrate real-time feedback on what was extracted. Also, the PromptLog could be extended to store conversation turns beyond just uploads, enabling a richer long-term memory. The design already anticipated augmenting PromptLog for traceability [9], so ensuring the schema (e.g. adding fields for section tags or embeddings) would enhance memory context usage.

Reference Document Alignment (Embedding Flow)

- **User Story:** *As a policy writer, I want the assistant to align my draft with official strategies, mandate letters, and past reports by pulling in relevant excerpts.*
- **Design & Implementation:** The system supports a **reference document corpus** that can be indexed and queried. Users can list available references or upload new ones (e.g. "2025 mandate letter") via `uploadReferenceDocument` (similar to other upload tools). Uploaded references are cleaned, chunked, and embedded into a vector store for later semantic search. The key tool is `alignWithReferenceDocuments`, which takes a user query or draft context (e.g. "What does the government say about AI?") and retrieves top-matching excerpts from the indexed docs. Internally

this likely uses OpenAI embeddings or similar to find semantically related passages (though the current branch's code for this is minimal). In practice, the implemented placeholder is the `reference_doc_context` step, which produces a summary of relevant reference document content for a given section. For example, when drafting the "Strategic Alignment" section, the tool might fetch snippets from the **GC AI Strategy 2025–2027** or **Digital Ops Strategic Plan** that match the section's themes. These excerpts plus their citations are packaged into a summary string ("summarized reference documents"). The **SectionSynthesizer** then injects this into the draft prompt – in code, it accepts a `reference_doc_context` field and includes it in the prompt template under an "[alignment]" or "[Corpus]" section [10] [11] (the prompt template expects placeholders like `{{alignment}}` or `{{corpus}}` for reference material). Additionally, the system uses a structured map of artifact requirements (**gate_reference_v2.yaml**) to guide alignment. For example, `section_synthesizer.get_section_intents()` looks up the current gate and section to retrieve any predefined *"intents"* or checklist items that the section must cover [12] [13]. These intents (e.g. "ensure alignment with Digital Standards for IT") serve as embedded reference cues to the LLM. The output of align/reference tools includes source attributions so the assistant can later cite them.

- **Testing:** A typical test would involve uploading a known reference (say, a policy PDF containing a distinctive phrase) and then asking the assistant an alignment question containing that phrase. The expected result is that `alignWithReferenceDocuments` returns an excerpt containing the phrase with a citation. In integration tests, after uploading reference docs, calling the drafting chain for a section triggers `reference_doc_context`; one would assert that the `output_summary` of that tool includes expected keywords from the reference and is not empty. (Currently, because the embedding retrieval logic is rudimentary, tests might stub out the vector search with a known response). The design also envisions a `listReferenceDocuments` tool to verify which references are indexed – one could test that after uploading, the new document appears in the list with correct metadata.
- **Gaps & Improvements:** *Implementation gap:* The actual embedding-based search is not fully realized in this branch – there is no explicit call to an embedding API or vector DB in the code. The `reference_doc_context` appears as a logged step but the logic behind retrieving specific excerpts is likely incomplete or stubbed. In the future, integrating a true vector search (e.g. using Pinecone or RedisBloom for semantic queries) is needed to deliver on this capability. *Modernization opportunity:* The system could proactively suggest relevant references by comparing section drafts to the corpus (e.g. using similarity thresholds). Also, handling for multi-lingual reference docs or large corpora (perhaps via batching queries or using LLM-based retrieval QA) would improve robustness. Finally, better integration of reference checks – e.g. an automatic "policy alignment score" or checklist validation – could be built on top of the reference alignment once the foundation is solid.

Web Research and Citation Logging

- **User Story:** *As a user, I want the assistant to fetch up-to-date facts and examples from the web, and log those findings with citations so I can incorporate evidence into my document.*
- **Design & Implementation:** GovDoc Copilot includes a research assistant mode within the chat. When the user asks an open-ended question ("What's been done on this issue elsewhere?"), the system goes into a **web research** workflow. First, the LLM formulates a search query via the `query_prompt_generator` tool, which uses a Jinja2 prompt template (see *search_query_generation* in prompts) to condense the project context and memory into an optimal query [14] [15]. The assistant (likely via an API like SerpAPI or a custom search integration) executes this query and retrieves top results. Summaries of those results are compiled; the transient

`web_search_context` logs show "summarized web results" were prepared. Next, the user is often asked if the findings should be saved for drafting. If they agree, the `record_research` tool is invoked to formally log the research notes. This tool takes raw notes plus metadata and structures them. If metadata (title, source, date, URL for each source) is provided by the search step, `record_research` formats each into a citation string [16]. If not, it can even call an LLM to parse unstructured notes into a structured list (with title/source) [17]. The final output is saved into PromptLog with an `input_summary` like "global_context | record_research" and a rich JSON in `output_summary` containing: an **answer** (the synthesized research summary), **documents** (a list of snippet texts), **citations** (formatted reference strings), and **metadatas** (structured source info). This design ensures that any external info gathered (e.g. statistics, precedents from other jurisdictions) is preserved along with its source. These research notes are then available as **global context** for drafting – e.g. the assistant can pull them in when generating relevant sections. (Indeed, the `record_research` implementation calls `log_tool_usage` internally to index this in the system memory [18].)

- **Testing:** To test this flow, one would simulate a user query that requires web info (e.g. "Find recent AI training initiatives in other governments"). The expected outcome is that the system produces a search query (which could be logged or returned by `query_prompt_generator`), fetches some example results (this part might be mocked in tests due to external API), and then successfully calls `record_research`. A unit test for `record_research` can supply a dummy `notes` string plus a list of `metadatas` and ensure the output JSON contains the concatenated answer and citations [16] [18]. We would also verify that `record_research` handles cases like missing metadata by falling back to structuring via LLM. Integration-wise, after saving research, calling `memory_retrieve` should ideally retrieve the `record_research` entry as well (though currently `memory_retrieve` filters by upload tools only [5], an improvement would be to include these global notes). The logs from a real run confirm the research note was saved and included citations (e.g. references to IPAC, CSPS, GSA in the output), which is a good end-to-end validation.
- **Gaps & Improvements:** *Implementation gap:* The actual web search step is partially outside the code – there is no explicit `webSearch` API call in this branch, suggesting it might rely on an external service or be manually driven. Formalizing this (e.g. integrating a search API or a browser tool) would complete the loop. Additionally, the `memory_retrieve` not capturing `record_research` logs means these research notes might not be automatically pulled into section generation (unless the planner explicitly passes them). Extending memory queries or adding a dedicated `global_context` retrieval (to fetch all saved research notes) would fix this. *Modernization opportunity:* The research step could leverage newer LLM abilities to directly browse and cite (e.g. using a Bing or Browser plugin approach) rather than a two-step query+record process. Moreover, implementing rate-limited web searches with caching of results in a vector index could speed up repeated queries. Finally, enhancing `record_research` to tag the notes with relevant section IDs or topics could allow finer-grained inclusion of research only where appropriate in the document.

Section Generation and Revision Workflows

- **User Story:** *As a user, I want the AI to generate well-structured draft sections of my document for me, and later revise those sections based on feedback.*
- **Design & Implementation:** This is handled by two chain workflows: **generate_section_chain** (for initial drafting) and **revise_section_chain** (for iterative edits). When the user requests a new draft section (e.g. "Draft the Problem Statement"), the Planner invokes `generate_section_chain`. Under the hood, this orchestrates a series of tool calls to gather context, then calls the LLM to

compose the section. According to design, the steps include: 1) **Memory retrieval** – fetch prior user inputs and notes via `memory_retrieve` [19] ; 2) **Reference alignment** – gather relevant reference excerpts via `reference_doc_context` (alignment search); 3) **Web context** – include any saved research or do a quick web query (`web_search_context`); 4) **Prior artifacts** – summarize any prior related documents (e.g. previous gate versions) via `prior_artifacts_summary` ; and 5) **Compose section** – call `section_synthesizer` to draft the section content [20] . In practice, the implementation prepares a consolidated prompt for the LLM: The SectionSynthesizer collects the project profile info and all the above context pieces (it expects fields like `memory_summary` , `prior_artifacts_summary` , `reference_doc_context` , `web_search_context` , plus the section's metadata) [10] [11] . It also looks up any section-specific intents from the gate reference (ensuring required points are addressed) [12] [13] . Then it fills a prompt template ("Draft a well-structured section using all inputs…") with these values and sends it to OpenAI (e.g. GPT-4) [21] [22] . The raw LLM output (which may be multi-paragraph Markdown) is captured as `raw_draft` . The chain then calls **SectionRefiner** to polish this draft. The `section_refiner` uses another prompt template focusing on tone, clarity, and completeness, applying it to the `raw_draft` [23] . The result is a refined section text (`refined` draft) [24] . Each step is logged via `log_tool_usage` for traceability (the design sets that each tool in the chain writes to the PromptLog/ReasoningTrace [25]). The final refined section is saved as an **ArtifactSection** record in the database with metadata like section_id, artifact_id, etc., along with source citations collected during generation [26] . (In the current code, sections are stored in memory/PromptLog and returned to the user, and a database model for ArtifactSection exists in the design if not in code). The **revise_section_chain** is triggered when the user provides feedback on a section. For example, "Add stakeholder evidence to this section" would invoke a series of tools: a `feedback_structurer` to interpret the free-text feedback and identify which section(s) and what kind of edit is needed, followed by a `section_rewriter` tool that modifies the section accordingly (likely by prompt instructing the LLM to apply the specific change). The chain might use a `diff_summarizer` to log what changed and then update the stored ArtifactSection and an audit trail (ReasoningTrace). In this branch, the revision tools are in planning – e.g. placeholders for feedback parsing and rewriter exist, but their implementations may be minimal. However, we do see evidence of a review flow: after generation, the system can fetch each section for review (`fetchReviewSection`) and format it for display (`formatSection`) before finalizing. This indicates sections are being temporarily stored (possibly in Redis or in memory) during the editing cycle.

- **Testing:** To test section generation, one would simulate a full chain call. In isolation, unit tests can verify `section_synthesizer` : provide it with a dummy profile and some context strings, then assert the returned draft contains all major parts. For example, if `memory_summary` contains a specific bullet point, the output should reflect that content (ensuring the prompt template succeeded). The logs from an integrated test (as in `WP7_test_runner.py`) show the sequence of tool calls and we would expect to see: memory, context summaries, then a section draft generated and refined, and an entry in the database or log for the new ArtifactSection. For revision, a test could feed a sample section text and a piece of feedback into the (hypothetical) feedback_structurer and check that it identifies the correct section and an action ("add evidence"). Then passing the section and instruction to `section_rewriter` should yield a modified text (e.g. now containing a "Stakeholders:" paragraph). Since revision chain is not fully implemented in code, this might be tested conceptually or with stubbed tools. The user journey was manually tested in runtime: after drafting, the user asked for changes and the system responded by logging revisions (in our log, we see multiple `fetchReviewSection` calls which would correspond to retrieving updated sections

for review). Ensuring that each revision is idempotent and stored (perhaps via versioning in ArtifactSection) would be part of acceptance tests.

- **Gaps & Improvements:** *Implementation gap:* The current branch doesn't show the full `revise_section_chain` tool implementations – likely placeholders exist but need development. There's mention of tools like `feedback_structurer` and `section_rewriter` in the design, but these aren't present or mapped in `tool_catalog.yaml`, indicating this is a to-do. Additionally, the generation chain currently drafts all sections in sequence (the log shows context gather and synth/refine repeated five times for five sections, all in one go). There's a potential gap in allowing the user to generate sections one-by-one vs. all at once; the system should support both flows. *Modernization opportunity:* Incorporating an interactive review loop where the LLM itself suggests improvements or checks against a checklist (almost a self-critique step) could enhance section quality before user feedback. Also, using advanced techniques like few-shot examples for section refinement (to enforce tone consistency across sections) or employing a second AI "critic" to review each section against the requirements could bolster quality (the team discussed an advocate vs. critic model for drafting [27] [28]). Finally, better integration of the ArtifactSection database model with version control – e.g. storing multiple revisions and using a diff to highlight changes – would improve traceability of revisions.

Toolchain Orchestration (Planner and Execution Flow)

- **User Story:** *As a user, I want the AI to guide me through the document creation process, automatically deciding which tools or steps to invoke based on what I ask for (e.g. uploading inputs, drafting content, finalizing the document).*
- **Design & Implementation:** The GovDoc Copilot is built as a **tool-enabled GPT planner**. Instead of a monolithic prompt, it has a library of tools and chains that the AI (or a backend orchestrator) can call to perform specific tasks. The mapping from high-level user intents to toolchain is largely rule-based in this implementation. The **Planner Orchestrator** (in `planner_orchestrator.py`) listens for certain trigger phrases or commands. For example, when the user says "Let's start drafting the Problem Statement," the planner interprets this as an intent to generate a section and routes it to the `generate_section_chain`. In development notes, they explicitly planned to extend `planner_orchestrator` such that a planner call `generate_section` triggers the `compose_and_cite` chain (the internal name for section drafting) [29]. This implies the orchestrator has a mapping: *generate_section* → call memory_retrieve + section_synthesizer + section_refiner, etc. Similarly, a user request to "finalize the document" triggers the `assemble_artifact_chain` (which calls format/merge/finalize tools). The **ToolRegistry** plays a key role in orchestration: it loads all tool definitions from `tool_catalog.yaml` and can instantiate the tool classes on demand [30]. This allows the planner to call tools by name dynamically. Only certain tools are exposed to the GPT's decision-making (marked `GPT_facing: true` in the catalog, although not explicitly shown in our snippet, presumably to hide internal tools). The orchestrator's flow aligns with the user journey phases: Phase 1 – select artifact (calls `getArtifactRequirements` to load template from YAML), Phase 2 – ingest inputs (`IngestInputChain` for uploading files) [31], Phase 3 – alignment and research (`alignWithReferenceDocuments`, `record_research` if user asks), Phase 4 – drafting sections (`generate_section_chain`), Phase 5 – iterative review (`revise_section_chain`), and Phase 6 – final assembly (`assemble_artifact_chain`). Each of these may be a multi-tool sequence executed either by the AI agent or by deterministic backend code. In this sandbox branch, much of the sequencing is implemented in Python (e.g. `IngestInputChain.run` explicitly calls upload tool

then profile engine [1] [2]). For drafting, there isn't a single `ComposeAndCiteChain.py` in the repository, but the WP17b exit report confirms it was implemented as a chain with the steps we saw [32]. The **inference pattern** here is that the LLM is invoked for tasks requiring natural language generation or understanding (like generating content or parsing feedback), whereas the orchestrator and tools handle deterministic or API tasks (database queries, file uploads, formatting). This separation is evident in code: e.g., `ToolRegistry.get_tool` loads Python classes (APIs, DB, etc.) while within a tool like SectionSynthesizer, the actual call to OpenAI's API is made via `chat_completion_request` [33].

- **Testing:** Orchestration can be tested at a high level with scenario-based tests. For instance, a test that simulates the entire user flow: provide a gate selection, upload a document, then issue a "draft section" command, and finally "finalize." The expected outcome is that the Planner calls the correct sequence in order. In absence of a full autonomous agent in this version, these would be broken into unit tests for each chain. For example, a planner test might call a function with input "start Gate 0 draft" and verify it returns a structured plan or directly invokes `generate_section_chain`. The design documents (e.g. WP12 planner phase spec) suggest the team was moving toward a more declarative mapping of phases to tools [28], which implies tests to ensure each phase's entry and exit criteria are handled. Additionally, one could test `ToolRegistry.list_tools()` to ensure only intended tools are exposed to GPT (to avoid it calling low-level or dangerous functions). The logs from an integrated run provide evidence of proper orchestration: e.g., after the user uploaded content, we see a gap, then a series of context tools and drafting in rapid succession, then formatting and Drive upload at the end – indicating the system moved through the phases in order. Each transition (ingest → draft → finalize) corresponded to a user action that the planner correctly caught.
- **Gaps & Improvements:** *Implementation gap:* The orchestrator in this branch is likely simplistic – it may not handle complex mixed intents or error recovery well. There is no evidence of a sophisticated dialog management (for example, if the user asks to regenerate a section with different parameters, does it clear the old one or version it? Likely manual). Also, some planned tools (e.g. `manualEditSync` for incorporating human edits, `listReferenceDocuments`) are not yet implemented, limiting the planner's reach. *Modernization opportunity:* Adopting an advanced agent framework (such as LangChain or AIPlugin integration) could allow the LLM itself to choose tools more flexibly rather than relying on hard-coded triggers. However, given the sensitive and structured nature of this workflow, a controlled planner might be preferable. Improving the state management – for example using a finite state machine for document stages – could make the orchestration more robust (ensuring that, say, you can't finalize before all required sections are drafted, etc.). Another improvement is better logging and traceability: the design mentions a **ReasoningTrace** YAML log for chain-of-thought [25]. Implementing that (perhaps by capturing the planner's decision sequence and tool inputs/outputs in a structured file) would greatly help with debugging and audits. In summary, the orchestration is functional but could evolve into a more adaptive "copilot" agent that still respects the government workflow constraints.

Final Artifact Assembly and Formatting

- **User Story:** *As a user, once my sections are ready, I want the assistant to assemble a complete, polished document (with cover page, table of contents, and consistent formatting) and provide it to me as a shareable file.*
- **Design & Implementation:** After all sections have been drafted and reviewed, the **assemble_artifact_chain** is invoked (e.g. when the user says "Let's finalize"). This chain handles the

collation and output of the document. First, it loads the latest version of each section – likely from the ArtifactSection DB table or from the in-memory review cache (Redis). In the logs, we see multiple `fetchReviewSection` calls right before finalization, which suggests the system pulled each section's content (perhaps to ensure it had the latest revisions) and then `formatSection` calls to apply uniform formatting [34]. The **formatSection** tool takes a section's text, ensures it has the appropriate Markdown heading and an anchor tag, and outputs a formatted section string [35] [36]. (It strips any user-provided heading to avoid duplication [35], then prepends `## Section Title` with an `<a id="section_id"></a>` for hyperlink anchors.) Next, the chain uses **mergeSections** to join all the formatted sections in the correct order. The `mergeSections` tool simply concatenates the list of section strings with blank lines in between [37]. This produces the full document body in Markdown. Then **finalizeDocument** is called to add front-matter. This tool creates a cover page with the document's title, version, artifact ID, gate number, and a timestamp [34]. It also auto-generates a **Table of Contents** by iterating over sections and listing each with a link to its anchor [38]. The result is a complete Markdown text (`final_markdown`) ready for export. Finally, the chain calls **storeToDrive** to output the document. The `storeToDrive` tool converts the markdown to PDF (using `markdown2` and WeasyPrint for HTML-to-PDF rendering) [39] and uploads it to Google Drive via the Drive API [40]. It organizes the file in a structured Drive folder: `PolicyGPT/<Project>/<gate_X>/<artifact_name>` as per WP20 design [41] [42], creating subfolders if needed [43]. The uploaded file is given a versioned name (e.g. `investment_proposal_concept_v0.3_20250604T045208.pdf`) and the Drive link is captured [44] [45]. The link is then returned to the user (and logged) so that they can access the final document. The system also shares the file with predefined emails or makes it "anyone with link can view" according to config [46], which is important for collaboration. At this point, the document is considered committed/finalized, and a **DocumentVersionLog** entry would be made in the database noting the Drive URL (the WP20 plan notes updating this log with the final link [47], though the exact code isn't shown in this branch).

- **Testing:** To test final assembly, one would ideally populate the system with a set of completed section drafts (in ArtifactSection or as dummy text), then call the `assemble_artifact_chain`. In a unit test, you can directly call `formatSection` on sample inputs to ensure it produces the expected markdown (check that each section string starts with the correct anchor and heading level [36]). Then test `mergeSections` by giving it a list of small strings and verifying the joined output matches the intended format (with newlines). For `finalizeDocument`, provide a dummy title, version, etc., and a short `document_body` markdown – the output should contain a header with those fields and a "## Table of Contents" with the sections linked [38]. The TOC generation can be tested by varying the sections list (including edge cases like no sections, or section titles needing escaping). The `storeToDrive` tool is more complex to test (involves external API calls); in a test environment one might mock the Google Drive service. But one can test the conversion logic locally: feed a simple markdown with known content, run `markdown2.markdown` and `HTML.write_pdf()`, and assert that the resulting PDF bytes are not empty. In integration, after `storeToDrive` runs, the log should show a `"drive_url"` in the output summary and the HTTP response from Drive should indicate success (this is evident in our runtime log – the `storeToDrive` output_summary contains a Google Drive link). Finally, an integration test would confirm that the final PDF indeed has all sections, correct headings, and is accessible via the link (this last step might be manual or done in a staging environment for security).
- **Gaps & Improvements:** *Implementation gap:* The finalization process in this branch assumes the document is in English and relatively short. Features like bilingual output or splitting a very large document into parts (to avoid Drive size limits) are mentioned but not yet implemented (WP20 notes

chunking if needed [48]). Error handling during export is another gap – e.g., if the Drive API fails or credentials are missing, the user should be informed gracefully (currently, `storeToDrive` will raise and the user might get no clear message). Also, while the code sets sharing permissions, it does so unconditionally to a fixed email and public; a more secure approach might integrate user-specific sharing settings. *Modernization opportunity:* A potential enhancement is to allow output to multiple formats – for instance, generating a Word document (.docx) or HTML in addition to PDF. The architecture could be extended with a `finalizeDocument(format=...)` parameter or separate tools (e.g. `exportToWord`). Another improvement is tighter integration with the project repository: for example, after finalizing, the system could commit the markdown to a version control system or send it via email – this would fit government record-keeping practices. Additionally, implementing **DocumentVersionLog** (as intended) would allow the system to keep track of all versions committed, enabling rollbacks or comparisons between versions. In terms of formatting, using a standardized converter like Pandoc could increase fidelity (especially for Word/PDF with tables of contents). Overall, the final assembly does achieve a key goal – producing a ready-to-use document – and with some refinements in error handling, format options, and trace logging (e.g., log the Drive file ID in the database for auditing), it can be made even more robust for production use.

Sources: The analysis above is based on the active code in the `sandbox-curious-falcon` branch of the *ai-delivery-sandbox* repository and associated design documents. Key files include the tool implementations for uploading and drafting (e.g. `IngestInputChain.py`, `memory_retrieve.py`, `section_synthesizer.py`, etc.) and the prompt templates and design specs like `policygpt_user_flow.md` which outline the intended workflow. For example, the tool definitions in `tool_catalog.yaml` map high-level GPT tool names to Python classes [30], and the WP17b design plan describes the compose-and-cite chain structure [20]. The runtime log (`log_tool_usage.json`) provided a trace of tool invocations, confirming how these components interact in practice (e.g., memory retrieval pulling uploaded text [5], research being recorded with citations [18], and final document assembly producing a Drive link [44]). These sources were used to validate each step of the reverse-engineered flow.

---

[1] [2] [3] [4] [8] [31] IngestInputChain.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/engines/toolchains/IngestInputChain.py

[5] [6] [7] memory_retrieve.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/memory_retrieve.py

[9] spillover_tracker.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/spillover_tracker.md

[10] [11] [12] [13] [33] section_synthesizer.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/section_synthesizer.py

[14] query_prompt_generator.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/tool_wrappers/query_prompt_generator.py

15 21 22 generate_section_prompts.yaml
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/prompts/generate_section_prompts.yaml

16 17 18 record_research.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/record_research.py

19 20 25 26 29 WP17b_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP17b/WP17b_design_plan.md

23 24 section_refiner.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/section_refiner.py

27 28 32 memory.yaml
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/memory.yaml

30 tool_registry.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_registry.py

34 38 finalizeDocument.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/finalizeDocument.py

35 36 formatSection.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/formatSection.py

37 mergeSections.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/mergeSections.py

39 40 42 43 44 45 46 storeToDrive.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/sandbox-curious-falcon/app/tools/tool_wrappers/storeToDrive.py

41 47 48 WP20_design_plan.md
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/project/build/wps/WP20/WP20_design_plan.md