ChatGPT

GovDoc Copilot: Technical Implementation

1. Prompt Ingestion and Memory Architecture

User Stories: As a policy analyst, I can upload project documents (text, files, or links) so that the system ingests and summarizes relevant inputs. As a user, I want my project profile (ID, title, dates, budget, scope, risks, etc.) to be automatically extracted and updated in memory, so that all subsequent sections use the latest context [1] [2].

Design Artifacts: The system uses an **IngestInputChain** (in `app/engines/toolchains/IngestInputChain.py`) that maps the `input_method` ("text", "file", "link") to tools like `uploadTextInput`, `uploadFileInput`, or `uploadLinkInput`. These tools extract raw text and metadata (e.g. project_id). The chain then calls the `ProjectProfileEngine` (an LLM-based extractor) to populate a structured `project_profile` JSON from the text [1]. Missing fields are set to null, and date/budget fields are normalized. The updated profile is saved to the database via `ProjectProfileEngine().save_profile(...)` [3]. Each call logs its usage for audit (via `log_tool_usage`) including summaries of inputs and outputs. Inputs are also stored in a **PromptLog/Redis memory** so that content persists across the session [2].

Test Outlines:
- *Valid Inputs:* Test uploading each input type (text/file/link) with minimal and maximal fields. Check that `project_id` is required, and errors are raised if missing or malformed [4].
- *Edge Cases:* Upload empty text or unsupported file types to ensure graceful errors. Provide malformed dates or numbers to trigger the cleaning logic (e.g. "June 32, 2025" or non-numeric budget) and verify they reset to null [5].
- *Memory Tests:* After multiple uploads, retrieve the project profile via `load_profile` to ensure memory was merged correctly. Test that duplicate `project_id` merges fields (existing values are not overwritten by nulls) [6].
- *Gaps:* Validate that very large files don't break token limits (splitting may be needed) and that Redis/memory has eviction policies for long sessions.

Modernization Opportunities: Use a dedicated vector-based memory store (e.g. Redis+Vector or LangChain memory) so that past "PromptLog" entries can be semantically searched, not just chronologically stored. Automating metadata extraction (using OpenAI or embedding models) could further enrich the profile. Additionally, containerize the ingestion chain to scale on demand, and integrate data validation libraries (like Pydantic models) to enforce schema rigorously.

2. Reference Alignment (Embedding Flows + alignWithReferenceDocuments)

User Stories: As a document owner, I want each section drafted to include evidence from official sources and meet gate criteria. For example: "Ensure my strategy section quotes the 2025 Mandate Letter and key

policy guidelines." The system should allow uploading and indexing **reference documents** (e.g. policies, mandate letters) so it can align outputs accordingly [7].

Design Artifacts: The flow here is not explicitly implemented in the code we saw, but the design spec provides guidance. There are likely tool wrappers for `uploadReferenceDocument`, `listReferenceDocuments`, and an alignment tool `alignWithReferenceDocuments`. In practice, uploaded reference docs would be sent to an embedding service, stored in a vector database, and tagged by gate or artifact. At generation time, the Planner or section tool can query this vector store to retrieve relevant excerpts. While the current code base does not show these classes explicitly, the OpenAPI guide indicates a call to `alignWithReferenceDocuments` that "extract[s] relevant excerpts, citations, and summaries" [7]. This implies a reference alignment module that queries embeddings of criteria or official docs against the draft content.

Test Outlines:
- *Reference Retrieval:* Given a draft query (e.g. "BC Announces first strategy"), ensure `alignWithReferenceDocuments` finds the correct passage in a reference doc. Test with multiple references where only one matches.
- *Empty/No-Reference:* If no relevant reference exists, ensure the system returns a null result or polite fallback (e.g. "No specific reference found for this content").
- *Mismatch Cases:* Supply conflicting reference documents (e.g. outdated policy vs latest). Test that the tool prefers the most recent or gate-appropriate version.
- *Edge Cases:* Very short draft text might retrieve irrelevant references. Ensure a minimum snippet length or additional context is requested.

Modernization Opportunities: Integrate a semantic knowledge graph of government policies (e.g. Wikisource or Legislation API) so the system can query official data sources directly. Use advanced embedding models (like GPT-4 Turbo embeddings or Llama-index) for more accurate semantic matching. Automate continuous syncing of new references (e.g. pulling from government databases) and use enterprise search (ElasticSearch with vector plugin or Pinecone) for scalable, real-time reference alignment.

3. Web Research Workflows (webSearch, record_research, goc_alignment_search)

User Stories: As an analyst, I may need external research (e.g. best practices, risk examples) beyond our uploaded inputs. I want GovDoc Copilot to formulate good search queries, call a web search, and log useful findings. For instance: "Search for recent AI strategy publications for citations," then save the findings into my project memory for use [8] [9].

Design Artifacts: The code includes a **Query Prompt Generator** tool (`query_prompt_generator.py`) which takes the project profile and existing memory to craft a Google/Web search query [10]. It fetches a prompt template (via a raw GitHub URL) and uses `chat_completion_request` to generate the query. The system likely has a `webSearch` tool wrapper (not shown in code) that calls an external search API or web scraper. Once results are obtained, a `record_research` tool would save selected snippets and sources into the project memory/log. The guide confirms this flow: after any web insight, call `record_research` to save notes and sources [9]. There may also be a specialized

`goc_alignment_search` for searching Government of Canada sites, possibly by targeting official domains.

Test Outlines:
- *Query Generation:* Given a sample project profile and memory, verify the generated query is relevant. For example, profile includes "AI policy," memory includes "talent," ensure the query mentions "AI talent Government of Canada" or similar. If memory is empty, the tool should still create a sensible query from profile.
- *Search API Call:* Mock the webSearch tool to simulate API failures (timeout, quota exhausted) and ensure the system handles it gracefully (with an error message or fallback plan). Test that safe search filters are respected and that only reliable sources (e.g. .gc.ca domains if required) are returned for `goc_alignment_search`.
- *Record Functionality:* After generating results, verify `record_research` properly logs the title, URL, and snippet into the `ReasoningTrace` or memory. Edge case: duplicate records should not be re-added. Ensure markdown links or plain text are formatted correctly.
- *Privacy/Security:* Confirm that webSearch does not leak sensitive project data in the query (logs should not store full user content).

Modernization Opportunities: Instead of raw web queries, integrate with a news/legislation API (e.g. Google Custom Search, GNews API) for structured results. Use the OpenAI browsing tool or ChatGPT's browsing for richer context. For `goc_alignment_search`, incorporate official government datasets or APIs (e.g. Canada.ca content API). Caching common searches and building a small knowledge base of relevant links could reduce repeated queries and speed up response times.

4. Section Generation and Revision (generate_section_chain, revise_section_chain, section_synthesizer, etc.)

User Stories: As a user, I want the assistant to draft each document section (e.g. "Problem Statement", "Background") one at a time. If I provide feedback or a revised draft, the system should incorporate it and refine the section. For example: "Generate the Problem Statement based on my inputs," then later, "Reword this draft for clarity" [11].

Design Artifacts: The guide names tools: `generateSectionDraft` / `generate_section_chain` (to create a section) and `revise_section_chain` (to refine with feedback) [11]. Internally, these likely correspond to **toolchains** such as **ComposeAndCiteChain** (per WP17b design) which sequentially runs: (1) `memory_retrieve` tool to fetch context, (2) `section_synthesizer` tool to draft content, and (3) `section_refiner` tool to apply stylistic edits or criteria [12]. The chain outputs an `ArtifactSection` record with text, citations, and status. Review steps involve a feedback loop: the user gives comments or a new version, and the chain re-invokes the LLM (possibly with different system/user prompts) to produce an improved draft [13]. This likely leverages `log_tool_usage` to track every generation/revision.

Test Outlines:
- *Content Quality:* Seed a known small input (e.g. a bullet list of facts) and test that `generate_section_chain` returns a coherent paragraph covering those points. Ensure format matches expected style (headers, tone).
- *Chunking:* If inputs are too long, verify the chain splits them or fails gracefully. E.g. a very large list of inputs

should not blow token limits.
- *Revision:* Simulate a review with comment “add more on risk.” The `section_refiner` tool should include extra risk content. Check that original text is not lost (see WP17b that a diff summary might be stored in `section_review_fetcher`).
- *Versioning:* Ensure that each saved `ArtifactSection` has versioning metadata (e.g. **version** field or commit tags). Test that the final assembly process picks up the approved version.
- *Edge Cases:* If the user asks for a different section than expected (e.g. skipping an outline), the planner should not confuse chains. Also test cancellation or changing gate mid-generation.

Modernization Opportunities: Use advanced LLM features like GPT-4 Turbo’s “function calling” to structure sections. Integrate a retrieval-augmented generation (RAG) approach: fetch related data points (from memory or web) at generation time to improve factual accuracy. Employ automated style checking (e.g. with specialized policy language models) to enforce consistency. Finally, allow collaborative editing by merging user-changed text and AI suggestions (Git merge-like tool) for smoother integration of external edits.

5. Toolchain Orchestration Logic (Planner, Registry, Inference Patterns)

User Stories: As a developer or system architect, I want the assistant to automatically route user requests to the correct sequence of tools. For example, when the user says “Draft section X,” the system should invoke the *generate_section_chain*. If the user says “the user is uploading inputs,” it should invoke the *IngestInputChain*. This ensures a coherent, end-to-end workflow without manual intervention.

Design Artifacts: The core of orchestration is a **Planner/Orchestrator** (likely in `app/engines/api_router.py` and a planner class not explicitly shown in our code excerpts). Based on user intent or API endpoint, it picks a **toolchain** from `app/engines/toolchains/`. The `ToolRegistry` (used in `IngestInputChain`) maintains a catalog of individual tools (wrappers). The OpenAPI guide lists endpoints grouped by purpose (e.g. *Input Prep*, *Drafting*, *Reviewing*, *Finalizing*) [14]. Under the hood, the API router accepts a chain invocation (like POST `/toolchain/generate_section_chain`) which triggers the corresponding Python class. The system uses a unified `chat_completion_request` pattern for any LLM step [15]. All tools follow a standard interface (`validate`, `run_tool`) so the registry can dynamically load them.

Test Outlines:
- *Intent Routing:* Issue API calls or simulated chat messages that correspond to different pipeline stages (e.g. “align references” vs “finalize artifact”). Verify the right chain is invoked. Test unknown or ambiguous intents to ensure clean errors.
- *Registry Discovery:* Try to load a tool not in the catalog to see if the registry fails as expected. Add a dummy tool and confirm it’s callable via the registry.
- *Concurrency:* Simulate multiple sessions (different project_ids) hitting the system simultaneously to check that tool instances don’t share state incorrectly. Ensure session IDs keep contexts separated.
- *Resilience:* If one tool in a chain fails (e.g. an exception during `section_synthesizer`), test that error is caught, logged, and does not crash the API (and returns a clear message to user).

Modernization Opportunities: Adopt a workflow or orchestration engine (e.g. Temporal, AWS Step Functions, or Apache Airflow) to manage complex toolchain state and retries. Leverage FastAPI's dependency injection to cleanly separate planning logic from tool implementations. Define the `tool_catalog` as a YAML or JSON so new tools can be added without code changes. Implement versioning for toolchains (canary vs production chain logic). Lastly, introduce monitoring dashboards that visualize active flows and tool usage for debugging and optimization.

6. Final Assembly and Drive Export (storeToDrive, finalizeArtifact)

User Stories: As a user, I want the fully assembled document saved to Google Drive automatically when I'm done. After all sections are approved, running `finalizeArtifact` should merge sections, apply final formatting, and upload the PDF or DOCX to Drive, returning a sharable link [16]. I should receive the Drive URL with version metadata in the chat.

Design Artifacts: The final step is an **AssembleArtifactChain** (not explicitly shown, but referred to in the user guide). This chain would collect all `ArtifactSection` entries (in order), concatenate them with metadata (version, gate ID, table of contents), and then call a tool `storeToDrive`. The design plan (WP20) mentions a `storeToDrive` script that uses Google API to upload the final document and updates a `DocumentVersionLog` with the Drive URL [17]. The `log_tool_usage` shows an example entry with tool=`storeToDrive` producing a Drive URL [18]. Another tool `finalizeDocument` (shown in logs) saved the markdown content prior to upload. The guide instructs that `finalizeArtifact` will return a download link (likely the Drive file link) to present to the user [19].

Test Outlines:
- *Drive Integration:* Test using a mock or test Google Drive API credentials: ensure `storeToDrive` creates the folder structure (e.g. by gate) and uploads the file. Verify that if a file with the same name exists, a new version is created or a unique name is used.
- *Link Correctness:* After upload, check that the returned URL is valid (e.g. Google Drive "view" link). If the Drive folder is missing or credentials expired, handle the error cleanly and notify the user.
- *Final Document Assembly:* Provide three dummy `ArtifactSection` entries with known content; verify that the final assembled document contains each section in order with proper headings and a Table of Contents (as the log example shows) [20]. Ensure that metadata (version, date) matches expectations.
- *Edge Cases:* If some sections were never generated (skipped), decide whether to insert placeholders or error out. Test finalizing an empty document.

Modernization Opportunities: Beyond Google Drive, allow exporting to other destinations (Microsoft OneDrive, SharePoint) by abstracting the storage tool. Use Google Docs API instead of static file upload, enabling the user to edit the final document collaboratively online. Implement automatic document formatting (e.g. normalizing fonts, applying templates) before upload. Finally, send notifications (e.g. email or Slack) with the link for broader collaboration, and integrate webhook callbacks for approval status updates.

Architecture Overview and Data Flows: The above components interconnect as follows (see diagram below). The user interacts via chat or API, which routes requests through the **API Router** and **Planner** into specific **Toolchains**. Toolchains coordinate **Tool Wrappers** (for LLM calls, web search, DB I/O) and interact with the **Database/Memory Store**. The final assembly interacts with external Google Drive for export. Data

(inputs, memory entries, reference docs, vector embeddings) flow between these modules so that each LLM step has the necessary context.

21 *Figure: Architecture overview of GovDoc Copilot (components and high-level flow).*

22 *Figure: Example toolchain execution flow (Input ingestion and section generation).*

23 *Figure: Data flow sketch (embeddings of inputs and references, retrieval for drafting).*

---

1 3 4 5 6 IngestInputChain.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/engines/toolchains/IngestInputChain.py

2 7 9 11 13 14 16 19 policygpt_custom_gpt_guide.md
file://file-ApW2AcaAKsZUzq4tzzp6Sd

8 10 query_prompt_generator.py
https://github.com/stewmckendry/ai-delivery-sandbox/blob/10b087a2f19c936c82bb5a755c935b8165f2856a/app/tools/tool_wrappers/query_prompt_generator.py

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

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

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

18 20 21 22 23 log_tool_usage.json
file://file-E4HzBpfN8sFAtSRx2Ue1WH