ChatGPT

PolicyGPT Integration Options: Document Management, Data Reference, Research Portals, and Meeting Summaries

PolicyGPT (PolicyCopilot) is a proof-of-concept AI assistant for drafting complex public sector policy artifacts (e.g. Treasury Board submissions, business cases). It uses a **FastAPI** backend with an OpenAI GPT-based engine. A key design goal is to integrate with systems that simulate the Government of Canada (GC) digital environment. We examine four integration categories – **Document Management**, **Data Reference/ Lookup**, **Research Portals**, and **Meeting Capture & Summarization** – identifying 2–4 viable product options for each. For each category, we assess technical feasibility (APIs, FastAPI integration), realism as a GC proxy, and pros/cons (UX, cost, auth, privacy). We then **recommend one product per category** that best balances realism with implementation speed. We compare the user experience of the recommended tool to its real GC equivalent (e.g. Google Drive vs GCdocs) to judge credibility for GC policy professionals. We also explain how each system supports PolicyGPT's mission of **document ingestion, search, structured drafting, collaboration, and compliant output** – particularly in the context of GC's IT **Project Gating Framework** (the PoC simulates users creating gate review artifacts for stages defined in the TBS guide [1]). Finally, for each recommended integration we provide a **FastAPI-compatible technical integration spec**: outlining a possible tool handler structure, key API endpoints/SDK usage, auth model, data format handling, and pseudocode/design for implementation.

Document Management Integration

Managing drafts and final documents is central to PolicyGPT. In the GC context, the official document repository is often **GCdocs** (OpenText Content Server) [2], used for records management and compliance. However, GCdocs is an internal system with no public API and is not designed for rapid iteration or collaboration (it's primarily for archiving and governance, not real-time co-authoring). For the PoC, we consider modern cloud document management services that could stand in for GCdocs or SharePoint, allowing the AI assistant to store, retrieve, and search policy documents. Key integration considerations are **API availability**, **ease of FastAPI integration**, **similarity to GC usage**, and **constraints** (auth and privacy).

Viable Options

- **Google Drive (Google Workspace)** – Google Drive offers a robust REST API and Python client libraries, making it quick to integrate with FastAPI [3]. It supports listing, uploading, downloading, and searching files, and even exporting Google Docs as text for AI consumption. **Feasibility:** High – the Drive API uses OAuth 2.0; a service account or a user auth consent can be set up easily in a test environment. Numerous guides and libraries exist (e.g. the `fastapi-cloud-drives` project supports Google Drive out-of-the-box [4]). **Realism:** Moderate – GC does not use Google Drive officially (data residency and security concerns), but it provides a generic cloud drive experience. **Pros:** Very fast to implement (well-documented quickstarts, minimal setup) [5]; excellent UX for collaboration (real-time editing, sharing) which can simulate a more user-friendly document system

than GCdocs. Free tier available for PoC. **Cons:** Data hosted on Google's cloud (privacy concern for real GC data); OAuth setup for multi-user might be needed; not a perfect analogue to GCdocs in terms of UI (Drive is more collaborative and less structured in records taxonomy than GCdocs).

- **Microsoft SharePoint / OneDrive (M365)** – SharePoint Online with OneDrive for Business is closer to what many GC departments use (as part of M365). Microsoft Graph API provides unified access to OneDrive/SharePoint document libraries [6]. **Feasibility:** Medium – Graph API integration is possible with Python (Microsoft provides Graph SDKs and REST endpoints) [6], but it requires registering an Azure AD app and handling OAuth with organizational accounts. This adds complexity for a PoC. **Realism:** High – Many GC users store working drafts on SharePoint or OneDrive and then transfer final copies to GCdocs for recordkeeping. SharePoint's structure (sites, libraries) and permission model align with GC contexts. **Pros:** Credible user experience (familiar Office 365 environment); data can reside in Canada if using the correct tenant; Graph API can access both personal drives and SharePoint sites with one code path [6]. **Cons:** Initial setup overhead (need a demo M365 tenant or developer tenant); Graph API has a learning curve (permissions scopes, tokens); slightly slower to prototype than Google's API. Also, for PoC use, one might rely on test accounts since actual GC tenant access is not available.

- **Box or Dropbox** – Enterprise cloud storage services like Box and Dropbox have RESTful APIs and are known for integrations. **Feasibility:** High – both have mature APIs (OAuth 2, SDKs). **Realism:** Low to Moderate – GC does not widely use these for official docs (there may be pockets of use or provincial agencies, but not standard for federal). **Pros:** Quick to integrate, with features like full-text search on documents (Box in particular focuses on enterprise document management). **Cons:** Less relevant to GC context; would still require storing data on external cloud. Cost may be an issue (free tiers are limited in storage).

- **Self-hosted ECM (Alfresco, Nextcloud)** – An open-source content management system could be stood up to emulate GCdocs more closely (structured folders, metadata). Alfresco has an API, and Nextcloud offers WebDAV/REST APIs. **Feasibility:** Low – standing up and configuring such a system for a PoC is time-consuming. **Realism:** High – if configured properly, could imitate a GC departmental repo, but the effort likely outweighs the benefit for a quick prototype. **Pros:** Full control over data (could be hosted on a GC-controlled environment); flexible. **Cons:** Significant setup and maintenance; custom integration needed (no ready-made FastAPI library); probably overkill for a PoC timeframe.

Recommended Option: Google Drive (Simulating GCdocs)

For the proof-of-concept, **Google Drive** is recommended as the document management integration. It strikes the best balance between **implementation speed** and a passable **proxy for GCdocs**. Google Drive's API is immediately accessible and well-documented, enabling us to build file operations into the FastAPI backend rapidly [5]. We can quickly implement endpoints to list and fetch documents for the GPT model to ingest or reference. While Google Drive is not used in the federal government, it provides a **cloud document repository** concept with sharing controls, which is sufficient to simulate how policy professionals manage drafts in a system. The **realism trade-off** is acceptable: the primary role of GCdocs (an official repository for final artifacts and a source of truth in gate reviews) can be mimicked by having a central Drive folder where PolicyGPT stores outputs and retrieves templates.

Justification: Google Drive's integration readiness means less time on infrastructure and more on demonstrating PolicyGPT's capabilities. It also supports collaborative editing if needed (though GCdocs itself isn't collaborative, GC users often collaborate in Word or SharePoint before finalizing records). Using Drive in the PoC allows quick iteration – e.g. uploading a TB submission template and letting PolicyGPT populate it. Additionally, Google's API allows content export (Google Docs to plaintext or PDF), enabling the AI to ingest document text easily. This helps fulfill PolicyGPT's mission of **document ingestion and search** – we can have the assistant pull in relevant pieces of existing documents from Drive to inform drafting.

From a **GC IT Project Gating** perspective, at Gate approvals (especially Gate 2: Business Case and Gate 3: Project Plan), there is an expectation of properly stored documentation. In the PoC, Google Drive will act as the repository where each stage's artifact (concept paper, business case, project plan, etc.) is saved and versioned, simulating the compliance of GCdocs. It ensures that PolicyGPT can **output structured documents and save them**, ready for a mock gate review.

User Experience: Google Drive vs GCdocs in a GC Environment

How credible will it feel to a GC policy professional if the PoC uses Google Drive? There are **differences** to consider in user experience:

- **Interface & Collaboration:** Google Drive is user-friendly, with easy sharing and real-time co-editing of Google Docs. GCdocs (OpenText) has a more rigid interface with check-in/check-out version control and no simultaneous editing. In practice, GC users often find GCdocs clunky [7] and use it mainly for final archiving, not day-to-day drafting. The PoC's use of Drive might actually feel *better* than GCdocs from a drafting perspective, albeit less familiar in look-and-feel. This discrepancy is acceptable since the PoC can clarify that Drive is a stand-in; users will understand it's a mock environment. The credibility comes from showing that documents are centrally managed and retrievable, which is a core concept of GCdocs.

- **Records Management vs Ease of Use:** GCdocs's strength is **records compliance** – enforcing metadata, retention schedules, and restricted access [2]. Google Drive in the PoC won't enforce complex metadata or classification out-of-the-box. We will simulate minimal metadata (perhaps using folder structure or naming conventions for gate documents). Policy professionals might notice the lack of mandatory file classifications that GCdocs would have, but for a PoC demonstration of AI capabilities, this likely isn't a focus. The advantage is that Drive allows quick search by content which PolicyGPT can leverage, whereas GCdocs search is notoriously poor; using Drive might actually *enhance* the demo by finding content quickly.

- **Authentication and Access:** In GCdocs, users are logged in via their enterprise account automatically. With Google Drive PoC, we might have a single pre-authorized service account or a demo Google account already connected, so the user in the demo isn't actively authenticating to Drive. This can be hidden behind the scenes. The experience will be that PolicyGPT just "has access" to the document repository. This is credible as an analogy to being an authorized GCdocs integration user (e.g., the AI has system access to relevant documents by design). We must ensure any UI or logging doesn't show Google branding overtly during the demo, to keep the immersion.

Overall, using Google Drive should still feel like "there's a central document store" to the user. We will need to verbally or in docs explain the stand-in ("Think of this as representing your GCdocs repository") but the

core workflow – storing and retrieving policy documents – remains valid. Given the short timeline of a PoC, this is a **reasonable compromise**. The alternative, SharePoint via Graph, while more authentic, could slow down development and introduce Azure AD auth screens that might complicate the demo. We prefer to deliver a smooth demo where the AI seamlessly pulls up documents, even if under the hood it's Google Drive.

How Document Management Supports PolicyGPT's Mission

Integrating a document management system enables **document ingestion, search, and compliant output** for PolicyGPT:

- **Ingestion & Search:** PolicyGPT can ingest existing documents (e.g. previous TB submissions, policy papers) from the repository to inform its drafting. For example, if a user asks the AI, "Use the last Strategic Assessment document as a reference," the system can search the Drive folder for that file name or content. Google Drive's API supports file content search (for Google Docs and some text-based files) and metadata query. The assistant can retrieve the text and feed it into GPT to ground responses in real content. This mimics how a policy analyst would retrieve past documents from GCdocs or SharePoint to guide new writing.

- **Structured Drafting & Templates:** Many gate documents have templates (a business case template, a project charter template, etc.). Storing these in the document system means PolicyGPT can fetch a template and fill sections out. The AI could also save its drafted outputs back to Drive in the appropriate folder (e.g. a "Gate 2 – Business Case" folder), preserving structure. This supports **structured drafting** by keeping the content organized by project stage.

- **Collaboration:** While GCdocs is not a live collaboration tool, the PoC's use of Google Drive does allow multiple users (or user + AI) to technically edit a Google Doc simultaneously. We can use this to our advantage in demonstration: for instance, a user could watch PolicyGPT populating a Google Doc in real-time, or they could make tweaks and ask the AI to refresh the summary. This collaborative feel aligns with modern work practices (even if not exactly how GCdocs works, it's how policy teams *wish* they could work). It shows how AI and humans might co-create content.

- **Compliant Output & Gate Reviews:** At each gate, certain documents must be finalized and stored for decision makers [8]. By integrating with a repository, PolicyGPT ensures outputs aren't just generated and lost – they are saved in a central location with version control. This is crucial for credibility: a GC policy professional will expect that final documents live in an official repository for audit/tracking. Our PoC will simulate that by having all final AI-generated artifacts saved in Drive (with timestamps, version history via Google's native versioning). We can even mimic required approvals by showing that the document is in the repository ready for review (analogous to submitting it in GCdocs for a gate decision).

In summary, document management integration anchors PolicyGPT in the reality of government documentation processes, turning AI outputs into tangible files in a system of record.

Integration Technical Spec: Google Drive via FastAPI

We outline how to integrate Google Drive with the FastAPI backend to serve as PolicyGPT's document tool. The integration will use Google's REST API (via the Python Google API Client or direct HTTP calls) to perform actions like searching for files, reading file content, and writing new files. Below is a technical plan:

- **Authentication Model:** We will use **OAuth 2.0** with a Service Account or test user credentials. For simplicity in PoC, a service account can be created in a Google Cloud project with access to a specific Drive folder. The service account's JSON key can be stored securely on the backend. The Google API client library (e.g. `google.oauth2.service_account.Credentials`) will generate tokens for API calls. This avoids any interactive login during the demo. Alternatively, an interactive OAuth consent for a single Google account could be done once and the refresh token saved. (A service account is preferred for automation.)

- **API Endpoints & Methods:** We will implement a FastAPI router for document management, e.g. `@app.get("/documents/search")` and `@app.get("/documents/{file_id}")`.

- **Search Endpoint:** Accepts a query string (keywords). It calls Google Drive's Files.list endpoint with query parameters (e.g. `name contains 'business case'`) and fields to retrieve name, id, etc. For broader content search, we can query using the `fullText` search if the files are Google Docs or have text content. The endpoint returns a JSON list of matched files (with metadata like name, id, perhaps a snippet or description).

- **Get File Endpoint:** Given a file ID, fetches the file's content. If the file is a Google Doc, we use the Drive API's export functionality (e.g. export as plain text or PDF). If it's a PDF or Word document stored on Drive, we might first attempt to use Google Drive's ability to **automatically OCR/index** some files or use the Google Drive API's **drive.export** (for Google Docs) and possibly an external library for others (for PoC, we might restrict to Google Docs or plain text files for simplicity). The content (text) is returned, or if binary (PDF) we could integrate a PDF-to-text step (using an OCR or PDF parser library in Python).

- **Tool Handler Structure:** Within the FastAPI app (or as part of the GPT backend logic), we can create a `DocumentManager` class that encapsulates Drive operations. For example:

```
from googleapiclient.discovery import build
from google.oauth2 import service_account

SCOPES = ['https://www.googleapis.com/auth/drive.readonly', 'https://
www.googleapis.com/auth/drive.file']
creds =
service_account.Credentials.from_service_account_file('svc_account.json',
scopes=SCOPES)
drive_client = build('drive', 'v3', credentials=creds)

class DocumentManager:
    def search_files(self, query: str):
```

```
        results = drive_client.files().list(q=query, fields="files(id,
name)").execute()
        return results.get('files', [])
    def get_file_content(self, file_id: str):
        # For Google Docs, export as text
        file = drive_client.files().get(fileId=file_id, fields="mimeType,
name").execute()
        mime = file['mimeType']
        if mime == 'application/vnd.google-apps.document':
            # Export Google Doc as plain text
            text_content = drive_client.files().export(fileId=file_id,
mimeType="text/plain").execute()
            return text_content.decode('utf-8')
        else:
            # For other types (pdf, etc.), get download URL and fetch
content
            request = drive_client.files().get_media(fileId=file_id)
            fh = request.execute()  # this returns bytes
            return fh  # (Could pass to PDF parser if needed)
```

In FastAPI, we can instantiate this `DocumentManager` and use it inside endpoints or even directly in the GPT tool logic. For example, a FastAPI dependency could provide the manager to endpoints.

- **GPT Integration:** The GPT backend can use the document manager functions when needed. For instance, if the user asks "retrieve the draft from last week," the system (through a scripted tool invocation or an internal chain) uses `DocumentManager.search_files("last week draft")` to get an ID, then `get_file_content(id)` to retrieve text, which is then fed into GPT as context. We could also allow the user to explicitly invoke a tool (depending on how interactive the agent is). But likely, we'll have predefined triggers: e.g., if user requests a summary of a document, GPT knows to call the document API.

- **File Format Handling:** We'll likely store and work with Google Docs format for any AI-generated textual documents (so they are easily editable by users too). The integration will primarily deal in **text** – converting docs to text for the AI to read, and taking AI outputs (text) to put into Google Docs. Creating a new Google Doc via API is straightforward (files.create with MIME type Google Doc). For other content like images or complex formatting, that's out of scope; the focus is policy text. If needed, we can also fetch PDFs if some reference material is only in PDF – using something like PyPDF2 to extract text after downloading via Drive API.

- **Permissions & Security:** The service account or token will be limited to a specific folder (in Google Drive, one could restrict to a folder by set up, or simply not share other folders with that account). This way, even though Google is an external cloud, we host only non-sensitive or sample data in it. No personal identifiers are needed. For demo safety, we ensure nothing from the real GC environment is on Google – all content should be fictitious or publicly available data.

- **Error Handling:** The API should handle cases like file not found or no search results (return empty list or 404). Also rate limits (Google Drive API has generous quotas for read operations in a single account environment, likely not an issue in PoC).

FastAPI Endpoint Example:

```
from fastapi import APIRouter, HTTPException
router = APIRouter(prefix="/documents")

doc_manager = DocumentManager()

@router.get("/search")
async def search_documents(query: str):
    files = doc_manager.search_files(f"name contains '{query}'")
    return {"results": files}

@router.get("/{file_id}")
async def get_document(file_id: str):
    try:
        content = doc_manager.get_file_content(file_id)
    except Exception as e:
        raise HTTPException(status_code=404, detail="File not found or
unreadable")
    # Possibly post-process content (e.g., convert bytes to text if needed)
    if isinstance(content, bytes):
        # e.g., if PDF, attempt text extraction (not fully implemented here)
        content_text = extract_text_from_pdf_bytes(content)
        content = content_text or "[Unable to extract text]"
    return {"file_id": file_id, "content": content}
```

This specification illustrates how PolicyGPT's backend can interface with Google Drive. It provides the needed operations for the AI to search and retrieve documents, fulfilling the document management role in the PoC.

(Citations for this section: Google Drive API ease [5], multi-provider FastAPI support [4], SharePoint Graph API realism [6], GCdocs context [2].)

Data Reference / Lookup Integration

Writing policy documents requires integrating **authoritative data and reference information** – such as statistics, lookup of program data, financial figures, or historical metrics. In a GC context, policy analysts pull data from official sources: e.g., **Statistics Canada** datasets, the GC **Open Data Portal**, internal databases, or budget and expenditure data (like the GC InfoBase). The PoC needs an integration that lets PolicyGPT query data to incorporate factual evidence into documents (supporting evidence-based proposals). This integration should allow the AI to retrieve specific data points or small datasets on demand.

Key considerations are the availability of open APIs for data, the relevance to typical GC usage, and how easily we can parse and feed data into the GPT model.

Viable Options

- **Statistics Canada Web Data Service (WDS) API** – Statistics Canada offers a public API for retrieving data series and tables [9]. **Feasibility:** Moderate – The API is publicly accessible (no auth needed for most data) and returns data in JSON or SDMX (XML) formats [10]. We would need to know the table or series identifiers to query specific data. The developer experience is a bit low-level (requires constructing queries with table IDs and vector IDs), but sample code exists. **Realism:** High – StatsCan is a primary source of official stats for Canadian policies. If an analyst needs, say, unemployment rates or population figures, StatsCan is the go-to. **Pros:** Direct access to official data; up-to-date releases (the API gives access to the latest data points daily). For PoC, we can hardcode or search for certain commonly used datasets (like CPI, population, etc.). **Cons:** The data requires understanding of the structure – we might need a mini lookup for table IDs unless we pre-select a few datasets. Also, the JSON structure of WDS can be complex (nested dimensions). Limited number of calls per second (though likely fine for PoC). Another con is that StatsCan API gives raw data, not explanatory text – the AI would need to interpret it (which it can, if prompted with the context).

- **Open Government Portal (CKAN API)** – The GC Open Data portal (open.canada.ca) provides a CKAN API to search and retrieve datasets metadata [11]. **Feasibility:** High – It's an open, **read-only API that needs no key for queries** [12]. We can use it to find datasets by keywords (e.g., "carbon emissions data") and possibly to fetch resource links. CKAN APIs typically allow full-text search on dataset titles/descriptions. **Realism:** Moderate – The portal is indeed a real GC resource, but policy analysts don't often directly use the API; they might browse the site. Still, it simulates an **open data lookup** which is credible if the policy work needs datasets (e.g., finding a relevant survey or GIS data). **Pros:** Easy to integrate (just HTTPS GET requests returning JSON); broad coverage of data across domains; no auth required for searching [12]. **Cons:** The API returns metadata; the actual data might be a CSV or PDF link that then has to be downloaded and parsed. This adds an extra step – but we can simplify by focusing on metadata or requiring the user to specify a known dataset. Also, many open data sets are large; we wouldn't fetch big data files in real-time. So likely we'd use it for *lookup* ("what dataset exists on X") rather than retrieving full numeric answers on the fly.

- **GC InfoBase or Internal DB (simulated)** – GC InfoBase is a repository of government expenditure and results data, but it doesn't have a public API for granular queries. However, we could simulate an **internal database or knowledge base** for key reference data (like standard demographic stats, economic indicators, etc.). For example, a small SQL or JSON store of common reference values (population of Canada, GDP, departmental budgets) that the AI can query. **Feasibility:** High – Because we'd design it ourselves. We could embed a small dataset and expose a simple lookup function (e.g., a dictionary of key stats). **Realism:** Moderate – In reality, analysts use multiple sources or ask internal data units; they don't query a single DB for all info. But having a curated data source in PoC could be a proxy for "the system knows important facts". **Pros:** Very fast queries, no external dependency; we control the content (can ensure relevance to our use cases). **Cons:** Not an actual product – more of a hardcoded solution. Lacks the expansiveness of real data sources. Also might feel less impressive unless the data is rich.

- **Wolfram Alpha API or Similar** – Wolfram Alpha is an external "computational knowledge" engine that can answer factual queries (e.g., "population of Ontario in 2020") via an API. **Feasibility:** High – API exists (AppID key needed), returns structured data or spoken answers. **Realism:** Low – GC analysts do not use WolframAlpha; it's not a known government system. However, it could cover a wide range of data in one interface, which is appealing for PoC (no need to pre-choose datasets). **Pros:** Broad knowledge base; natural language querying of facts. **Cons:** The answers may not always align with official Canadian sources, and relying on it might reduce the Canadian context credibility. It's also an external US service, which in a real GC scenario would raise flags (where is this info coming from?). Also cost: the free tier is limited.

Given these options, a combination might be ideal: using *specific GC data APIs for authenticity*, supplemented by a simpler lookup for general facts if needed. Let's decide on the primary integration.

Recommended Option: Statistics Canada API (for Authoritative Data)

We recommend using the **Statistics Canada Web Data Service API** as the primary data lookup integration, possibly supplemented by the Open Data portal for dataset discovery. StatsCan's API provides the realism of an official GC source and ensures that any data the AI pulls in the PoC can be traced to a credible origin (critical for policy work). Technically, integrating it is feasible within our FastAPI backend: no auth required, and we can retrieve JSON data for targeted series [9] .

To keep implementation manageable, we can pre-select a few relevant data series and build helper functions to query them. For example, if our policy scenario involves socio-economic data (unemployment rate, GDP, population), we identify the table IDs from Statistics Canada beforehand. The API can then retrieve the latest values so PolicyGPT can quote them in a business case or briefing.

Justification: This approach demonstrates that PolicyGPT can enhance **evidence-based drafting** by pulling live data, which will impress GC stakeholders. Using StatsCan aligns with public sector practice: analysts frequently cite StatsCan figures in submissions. It also addresses the **project gating context** – for instance, at Gate 2 (Business Case), one must justify a project with data on the current state or expected benefits. PolicyGPT could fetch relevant statistics (e.g., "X% of Canadians lack high-speed internet, according to StatsCan 2024") to strengthen the business case content.

The StatsCan integration is fairly quick to implement for a PoC because we can limit scope to a few API calls, and the data is open. Open Government Portal (CKAN) could be added to demonstrate dataset *search* capability: e.g., if user asks "Find data on electric vehicle adoption," the system could use CKAN API to suggest a dataset, though full retrieval might be outside PoC scope. This adds realism that the AI can point users to the right data sources.

User Experience: Simulated Data Lookup vs Real GC Process

In practice, how will a policy professional experience the data lookup integration in the PoC, and how does it compare to real life?

- **Speed of getting data:** In the demo, PolicyGPT might answer a query like "What's the latest unemployment rate?" by instantly providing the number and perhaps the source citation (StatsCan). This is likely **faster** than a human analyst's process, which might involve manually going to Statistics

Canada's website or internal data tables. The seamlessness will feel impressive, though it might also trigger questions: "Where did it get that number from?" We should ensure the AI cites the source or at least mentions "StatsCan" when using this data, to maintain credibility and transparency. GC professionals trust data that is sourced; if the AI just states facts without source, they might be skeptical. We can program the AI's prompt or response formatting to include "(Source: Statistics Canada)" when data is fetched.

- **Scope of data available:** The user might test the AI by asking various data questions. Our integration will handle some, but not all, queries. In reality, analysts have to hunt through multiple databases and sometimes the data isn't readily available. For PoC, we will likely constrain to a scenario (for example, if the policy is about a technology project, the data relevant to that). The AI might gracefully respond with "I don't have that data" or suggest where to find it if outside our integrated set. This needs to be managed to not break the illusion. We may preload some data points into the AI's knowledge (fine-tuned or in prompt) to cover gaps. Real GC experience is that sometimes you can't find data quickly – the PoC might actually **over-simplify** that by making it seem all data is an API call away. This is a known gap, but acceptable to showcase potential.

- **Data format and usage:** StatsCan API gives raw numbers or tables. The AI in PoC will likely extract a number or trend and then incorporate it into narrative. For example, if StatsCan returns a JSON with a series of monthly unemployment rates, we might code the tool to pick the latest value. The user experience is that they get a nicely written sentence with the figure. This is great for demo, but users might wonder if the AI correctly interpreted the table. We should pick straightforward data to avoid misinterpretation. Perhaps use API calls that directly return a single aggregated value (some StatsCan tables have summary indicators or we can fetch "latest value of series X").

- **Credibility of source:** Using "Statistics Canada" as the source will resonate well; it's a gold standard for data. If we also use the Open Data portal, the AI might say "according to an open dataset from Natural Resources Canada…". This also sounds realistic (GC has many open datasets). We must ensure any such data is indeed from a GC department or agency to maintain credibility. We wouldn't want the AI quoting random data from Wikipedia or WolframAlpha in the demo, as that reduces the GC-flavor. So, sticking to GC sources in output is important.

In summary, the PoC user experience for data lookup will feel like having a smart research assistant that instantly provides official statistics. This is a bit idealized compared to the real slog of data gathering, but it demonstrates the potential efficiency gain.

How Data Lookup Supports PolicyGPT's Mission

Data lookup integration directly furthers the **knowledge enrichment** aspect of PolicyGPT's mission:

- **Evidence-Based Drafting:** By pulling in quantitative evidence, the AI helps create robust policy documents. For example, in a **Treasury Board Submission**, there's often a "Analysis/Rationale" section requiring data (how many people will be affected, what the current trend is, etc.). PolicyGPT can fetch those numbers so the content is stronger. This supports GC's push for evidence-based policy. It also ties to **gate requirements**: a gate review will scrutinize if the proposal is well-supported. Having the right data (and even citing it) shows a credible artifact.

- **Reference Consistency:** In policy shops, there are often "one source of truth" reference numbers (for example, the official population count must come from StatsCan, not an arbitrary source). Our integration ensures PolicyGPT draws from the canonical sources, maintaining consistency. This aligns with compliance – using authorized data avoids conflicting figures. If PolicyGPT were to generate numbers without reference, it could undermine trust. Instead, by integration, it effectively has a lookup table of approved figures.

- **Automation of Mundane Lookups:** Policy professionals spend time looking up standard info (like departmental strategic outcomes, previous funding numbers, etc.). If we simulate some of these via our data integration, PolicyGPT can auto-fill those parts of documents. For instance, a **business case's** cost section could reference "According to the 2024 Budget, $X million was allocated to related initiatives" if we pre-load such data. This automation allows the human to focus on analysis rather than hunting down numbers.

- **Accuracy and Compliance:** From a compliance perspective, using official data sources means the document output is more likely to pass reviews (no incorrect data). It also provides traceability – e.g., if challenged, one can show which dataset the number came from (in a future iteration, we could even include dataset IDs as metadata). This is important for the **gating framework**, where gate reviewers ask "Where did this figure come from?". PolicyGPT could answer or footnote it, demonstrating an audit trail, which is a core part of project approval processes.

Thus, data lookup integration empowers PolicyGPT to be not just a writer but a *research assistant*, aligning with how real policy development interweaves writing with data gathering.

Integration Technical Spec: StatsCan API via FastAPI

To implement the Statistics Canada (StatCan) data reference integration, we will create a service client that can query the StatCan Web Data Service (WDS) API and possibly the Open Government CKAN API for dataset search. The integration involves making HTTP requests (REST) and parsing JSON responses. Key points of the technical design:

- **Authentication:** StatCan's API is open for reads – no API key or auth is required for retrieving data [13] (the same goes for open.canada.ca CKAN search). This simplifies our integration; we just call the endpoints directly. We should be mindful of polite usage (though our volume will be low).

- **StatCan API Basics:** StatCan's WDS has specific endpoints (often with URLs containing table identifiers). For example, a typical call might be:
`GET https://api.statcan.gc.ca/rest/getAllCubesList/en` – to list data tables, or
`GET https://api.statcan.gc.ca/rest/data/tableId/<parameters>/JSON` – to get data.
We will likely use the JSON service for a specific table. If we know the table ID (e.g., **14-10-0327-01** for unemployment rate), we can retrieve JSON data. The response structure includes metadata and an array of observations.

- **Designing Query Functions:** We can implement a `StatCanClient` class in Python:

```python
import requests
import urllib.parse

class StatCanClient:
    BASE_URL = "https://api.statcan.gc.ca/rest/data"
    def get_latest_value(self, table_id: str, vector_id: str = None):
        # Construct URL for JSON data; vector_id can specify a series
within the table
        url = f"{self.BASE_URL}/{table_id}/JSON"
        if vector_id:
            # Using vector or coordinate to filter a specific series if
needed
            url = f"{self.BASE_URL}/{table_id}/
{urllib.parse.quote(vector_id)}/JSON"
        resp = requests.get(url)
        resp.raise_for_status()
        data = resp.json()
        # Assume data['data'] holds observations in chronological order
        obs = data.get('data', [])
        if not obs:
            return None
        latest = obs[-1]  # last observation
        value = latest.get('value')
        ref_period = latest.get('referencePeriod')
        return value, ref_period
```

In practice, we might need to specify parameters for the API call if the table is large, like limiting to the latest period. StatCan's API has methods like `getDataFromCubePidCoordAndLatestNPeriods` [14] which could be utilized to directly get the last N periods for a series. We might use those more advanced endpoints to avoid downloading full tables.

- **Dataset Search (CKAN):** If implementing dataset search, we use the Open Government **CKAN API**. For example:
  `GET https://open.canada.ca/data/en/api/3/action/package_search?q=electric+vehicles`
  This returns a JSON with a list of datasets matching the query. We can parse the result to retrieve titles, descriptions, and links. A `OpenDataClient.search_datasets(query)` function can wrap this. Since this is more for demonstrating capability, we don't necessarily need deep integration; just enough to show the AI can find a relevant dataset title.

- **FastAPI Endpoints:** We can expose a couple of endpoints in our API for data lookup, or call the clients internally from the GPT logic:

- `@app.get("/data/{table_id}")`: Return latest data (or specific slice) from a StatCan table. For example, `/data/14-10-0327-01` might return the latest unemployment rate value and date.

- `@app.get("/data/search")`: Query the open data catalog. E.g., `?q=term`. Returns dataset metadata results.

Example implementation:

```
data_router = APIRouter(prefix="/data")
statcan = StatCanClient()
@data_router.get("/{table_id}")
def get_data(table_id: str, series_id: str = None):
    try:
        value, period = statcan.get_latest_value(table_id, series_id)
    except Exception as e:
        raise HTTPException(status_code=400, detail="Error fetching data")
    if value is None:
        raise HTTPException(status_code=404, detail="No data found")
    return {"table": table_id, "latest_period": period, "latest_value": value}
```

For the search:

```
import requests
@data_router.get("/search")
def search_datasets(q: str):
    url = f"https://open.canada.ca/data/en/api/3/action/package_search?
q={urllib.parse.quote(q)}"
    resp = requests.get(url)
    results = resp.json().get('result', {}).get('results', [])
    simplified = [{"title": ds["title"], "desc": ds.get("notes",""), "id":
ds["id"]} for ds in results]
    return {"results": simplified}
```

We might not expose these directly to end-users in the UI, but rather let the AI agent use them. For instance, if the AI receives a query requiring data, it could call an internal function that uses `StatCanClient`.

- **GPT Integration:** The GPT backend can have tools/functions like `get_statcan_data(table_id)` or even more semantically, `find_statistic(concept)` that under the hood knows which table to query. A simple mapping could exist, e.g., concept "unemployment rate" -> table 14-10-0327-01. For the PoC, we can hardcode a few mappings or require the user to specify the needed data explicitly. Alternatively, the AI prompt can be engineered to ask for data by table if it knows it (which is unlikely unless we prime it with such knowledge). A straightforward approach: the user might instruct “Insert the latest unemployment rate (StatCan table 14-10-0327-01).” PolicyGPT could detect the pattern and call the tool. It’s a bit manual but effective.

- **Data Format Handling:** The StatCan API JSON needs parsing. We’ll extract values and perhaps units if needed (some API responses include a unit or note). The output we provide to the AI will be text or

a structured snippet it can turn into text. For example, we might return `"6.8 (percent, April 2025)"` or a full sentence ready to insert. We should ensure numeric values are turned into strings with appropriate formatting (no excessive precision, etc.). If needed, convert to human-readable form (e.g., 0.068 -> 6.8%). This is straightforward in Python.

- **Error and Edge Handling:** If API fails or data missing, the system should handle gracefully – maybe the AI says "(data not available)" or uses a placeholder. We can mitigate by testing the calls beforehand with known working series. Also, network calls should ideally be async in FastAPI, but given PoC scale, sync is fine or we can use `httpx` with async.

- **Performance:** The data volumes we fetch are small (just the latest few records), so performance is good. We should cache results within a session to avoid repeated API calls if the same data is needed multiple times in one conversation, but it's not critical.

Using this spec, implementing the data reference integration is relatively lightweight. We leverage existing open APIs and just parse results for the AI. With StatsCan JSON providing the raw figures and the CKAN API enabling search of metadata, PolicyGPT will be able to both *retrieve facts* and *point to datasets* when needed, strengthening its utility in drafting informed policy documents.

(Citations for this section: StatCan API description [9], Open Data API open access [12].)

Research Portals Integration

Policy development is research-intensive. Analysts gather background from policy reports, academic studies, past governmental reports, and best practices from other jurisdictions. In a GC setting, this could involve searching internal knowledge repositories (like GCpedia, departmental libraries), external research portals (Library of Parliament, Policy Horizons Canada publications), or simply general web searches (since much policy info is public). For PolicyGPT to assist in drafting narrative and context, it should be able to **fetch relevant reference material or information** from a research source. The integration here acts like the AI's "research assistant": finding and possibly summarizing relevant documents or articles.

We need integration options that allow querying a body of knowledge and retrieving text or summaries. Considerations include the breadth of information available, API availability, and alignment with how government folks do research.

Viable Options

- **Web Search API (Google or Bing)** – Using a web search API to query the internet for relevant information. **Feasibility:** High – Both Google's Custom Search JSON API and Microsoft Bing Search API can be used to get search results in JSON format [15] [16]. We can then fetch specific pages if needed. **Realism:** Moderate – In reality, analysts do use Google to find info (e.g., similar policies, news articles, etc.), though they are mindful of authoritative sources. Using a general web search might bring in a lot of noise or non-credible sources, so we'd need to focus it. Google's Programmable Search allows restricting to certain domains (for example, we could make a custom engine that searches only `.gc.ca` websites, or known think tanks). **Pros:** Huge range of information, up-to-date content (especially if looking for recent developments or cross-jurisdictional

info). **Cons:** Potentially too broad – risk of irrelevant or unvetted info. Also, reliance on external search raises privacy (the queries might reveal internal questions) and cost (free tier limits: Google gives 100 queries/day free [17]; Bing's free tier is being retired soon, moving to paid Azure service). But for PoC the cost for a few queries is trivial.

- **Academic/Policy Research API (Semantic Scholar, etc.)** – Use an API to search academic papers or policy documents. Semantic Scholar's API, for example, can search publications by keywords and return paper titles, abstracts, etc. [18]. **Feasibility:** Moderate – Semantic Scholar API is free but has rate limits (100 requests/5min) and requires an API key for higher volumes. It returns JSON with paper metadata. **Realism:** High for certain contexts – If the policy involves evidence from research (e.g., a health policy referencing a medical study), an academic search is relevant. Also, organizations like Policy Horizons or think tanks sometimes publish papers that are indexed academically. **Pros:** Focuses on high-quality sources (peer-reviewed or institutional publications). **Cons:** Might miss non-academic but important sources (like government reports that aren't in academic databases). Also, linking directly to academic content might be less directly useful unless the AI can read and summarize the found papers – which could be heavy.

- **Internal Knowledge Base (Simulated GCpedia or Library)** – GCpedia is an internal wiki where public servants share knowledge; many departments also have curated libraries or archives of past policy documents. We could simulate this by creating a **corpus of relevant documents** and indexing it (using ElasticSearch or even just storing as text). Then provide a search tool over that corpus. **Feasibility:** Moderate – It requires assembling content in advance. For PoC, we might include, say, a handful of past TB submissions, policy frameworks, or policy guides as sample docs. We could index them with a simple full-text search or vector search for semantic similarity. FastAPI could then query this index. **Realism:** High – This mimics an internal portal where one searches for related documents or precedents. If we include real public documents (like portions of TBS policies, etc.), it can feel like the AI is knowledgeable about GC policy context. **Pros:** Controlled content – we know the quality and can ensure relevance. Good for offline demo (no external call needed once set up). **Cons:** Limited scope – only as good as the docs we load. Lacks the breadth of a live web search. Also additional setup to create the index (though something like Whoosh or simple Python search could suffice for a small set).

- **Specific Policy Portals** – For example, the Library of Parliament publishes research reports on legislative issues, or the Canada School of Public Service has a digital library. However, these typically don't have open APIs. Another is **Policy Commons** (a platform aggregating think tank reports) but that's a subscription service, not sure about API. These are likely not easily integrated within PoC constraints.

Given these, a **web search with domain filtering** plus a **small internal corpus** might together cover the needs. Let's pick an approach that gives us quick wins and authenticity.

Recommended Option: Custom Web Search (Focused on GC and Reputable Sources)

We recommend integrating a **Google Programmable Search Engine (Custom Search JSON API)** configured to prioritize GC and related domains. This effectively gives PolicyGPT the ability to "Google" in a controlled fashion. We will create a custom search engine that includes relevant domains such as `*.gc.ca` (Canadian government sites), perhaps `*.canada.ca`, and possibly known think tank domains

(e.g., `policyoptions.irpp.org`, `conferenceboard.ca`) or international orgs (OECD, World Bank) if relevant to our scenario. This ensures search results are **biased towards credible, policy-relevant sources**. The Google Custom Search API then allows us to retrieve top results in JSON [15], and we can fetch the content of those pages if needed.

Justification: This option is quick to implement and **broadly mimics how a policy analyst would search for information**. In real life, they might Google a topic and then sift for authoritative sources – by pre-filtering domains, we save the AI from sifting misinformation. Technically, using Google's API is straightforward (just an API key and a search engine ID, no OAuth needed). The results come with snippets, which the AI could use to decide relevance. For the PoC, we might not go as far as fully reading external webpages (which could be complex due to needing web scraping and summarization), but we can demonstrate the AI suggesting sources or pulling short bits of info. For example, if asked "Has any policy similar to this been done before?", the AI could perform a search and answer, "A 2022 report by Policy Horizons Canada explored a similar idea [19]," referencing what it found.

Using the custom search aligns with **GC gating stages** especially in early phases (Gate 0/1 Idea and Gate 2 Analysis) where scanning the landscape of existing policies and research is critical. PolicyGPT with this integration can help users identify precedent documents or lessons learned from other departments, which adds realism to the PoC scenario.

User Experience: AI Web Search vs Traditional Research

In the PoC demo, when a user asks a question that requires outside knowledge, PolicyGPT will leverage the search integration. Here's how that will feel and how it compares to a real analyst's process:

- **Scope of answers:** The AI, via the custom search, might present *sourced information*. For instance, user asks, "What are the key barriers to telehealth adoption identified in past studies?" The AI triggers a search behind the scenes for "telehealth adoption barriers Canada policy study". It then might respond with a synthesized answer: "According to a 2021 report by the Canadian Institute for Health Information, key barriers include internet access, digital literacy, and privacy concerns [20]." In reality, an analyst might find the CIHI report after some googling and reading. The AI doing this in seconds is a significant efficiency boost. As long as the information is accurate and sourced, a policy professional will find it credible. We should ensure the AI cites or names the source (the citation might be visible in the demo interface as per our system, but we can also have it say "CIHI 2021 report"). This practice mirrors how analysts footnote their sources.

- **Quality control:** One risk is the AI might pick up a snippet out of context. Real researchers critically evaluate sources, whereas the AI might not. To keep trust, we might have the AI share the source document or link for the user to verify if needed. For PoC, maybe we don't actually click out to the web (since it might not be allowed live), but we can have pre-fetched relevant snippets ready. The user experience might be somewhat scripted here – e.g., we know some questions that will be asked and have pre-loaded the answers from known good sources via the search integration so it appears seamless.

- **Familiarity:** Many GC policy folks use internal tools like InfoGov or GCpedia, but also just Google. Seeing the AI quote a Government of Canada webpage or a known think tank will resonate ("Oh, it found something on Bank of Canada's site"). It will feel like the AI has access to a vast library (which it

essentially does via the web). This is powerful, but we will need to contain it to relevant stuff. If the user deliberately asks something obscure or off-topic, the AI might still try a search – we could limit the demo to avoid weird queries.

- **Comparison to internal systems:** If a GC user asks "Do we have anything on GCpedia about this?", in real life they might go to GCpedia (which is intranet only). We don't have that in PoC, but the web search might find public analogues or nothing. We might just focus on publicly available knowledge. The user might not notice the difference if we cover the expected info. Also, if needed, we could simulate an internal wiki by including some dummy GCpedia content in our custom search index (for example, hosting a page and letting Google index it, but this is probably unnecessary detail). The key is delivering useful reference info, not the exact portal used.

Overall, the experience will be of an AI that can pull relevant background info on-the-fly, which is something policy analysts often spend days doing. That's a compelling narrative for the PoC.

How Research Integration Supports PolicyGPT's Mission

The research portal/search integration is crucial for **knowledge-augmented drafting** and **collaboration** in the sense of sharing knowledge:

- **Informing Content with External Knowledge:** When drafting, say, a business case (Gate 2 deliverable), an analyst needs to include context like "what have others done?" or "strategic alignment with existing policies." PolicyGPT, via research integration, can insert relevant context. For example, linking a proposal to a current Government priority by quoting a Minister's statement or an existing policy framework (found online on Canada.ca). This makes the AI's output more robust and anchored in reality, rather than just generic text. Essentially, the AI can bring in *domain knowledge* that wasn't part of its base prompt or training, by actively retrieving it.

- **Speeding up Literature Review:** In early gate stages (idea and initiation), there's often a literature review or jurisdictional scan. The integrated search allows PolicyGPT to do a mini literature review in seconds. It might present the user with a few key sources or even summarize them. This supports the collaboration between the user and AI – the user can ask the AI to gather info, and the AI delivers summaries or quotes, which the user then uses to make decisions or craft arguments. This is collaborative in a research sense.

- **Ensuring Accuracy and Credibility:** By pulling from recognized sources, PolicyGPT avoids the trap of hallucinating facts. The integration essentially forces it to fetch factual content when needed. For compliance, this is important – any factual assertion in a policy document must be defensible. If PolicyGPT writes "90% of Canadians use smartphones," that must come from somewhere authoritative. Through our integration, that claim would come with a source (e.g., CRTC report). This increases the confidence reviewers (in a gate meeting) would have in the AI-generated document.

- **Knowledge Management:** In a broader sense, this feature turns PolicyGPT into a tool that not only writes but also helps manage knowledge. It could potentially index and recall *internal* policy documents if extended. For the PoC we use external publicly available ones (since we can't connect to GC intranet), but the principle shown is that an AI assistant can leverage a knowledge base. This

aligns with how the GC could one day integrate such a tool with their own internal research portals, multiplying its utility. We can articulate that vision in the PoC.

Thus, the research integration ensures that PolicyGPT's outputs are **context-rich and evidence-informed**, aligning with the needs of policy development at each stage.

Integration Technical Spec: Google Custom Search API via FastAPI

Here we detail how to implement the recommended custom search integration. We will use Google's Custom Search JSON API, which requires setting up a search engine and obtaining an API key. The steps and design:

- **Google Programmable Search Setup:** We will create a Programmable Search Engine (PSE) in Google's control panel, specifying the domains to include. For example, we add `*.gc.ca` and possibly other domains. We get a **Search Engine ID (cx)** for this PSE. We also generate an **API key** from Google Cloud Console for the Custom Search API. These will be stored in our FastAPI app configuration.

- **Search Query Endpoint:** Implement an endpoint like `@app.get("/research/search")` that takes a query string. This endpoint will call Google's Custom Search API endpoint:
  `GET https://www.googleapis.com/customsearch/v1?key=API_KEY&cx=SEARCH_ENGINE_ID&q={query}`
  This returns a JSON with search results (items with title, snippet, link, etc.).

Example code:

```
import os, requests
GOOGLE_CSE_ID = os.getenv("GOOGLE_CSE_ID")
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY")

class ResearchClient:
    def search(self, query):
        url = ("https://www.googleapis.com/customsearch/v1"
               f"?key={GOOGLE_API_KEY}&cx={GOOGLE_CSE_ID}
&q={requests.utils.requote_uri(query)}")
        resp = requests.get(url)
        resp.raise_for_status()
        data = resp.json()
        return data.get("items", [])  # list of results
```

- **Optional: Page Fetch & Summarization:** If we want the AI to actually read an article and summarize, we would need to fetch the page content. We might not do full web scraping due to complexity and potential variability of sites (also risk of hitting anti-scraping measures). Instead, as a simplified approach, we can rely on the snippet provided by Google (which is a few lines of context around the query terms). For PoC, using the snippet might be enough for the AI to glean some info.

Alternatively, we identify one or two key sources in advance and handle them specifically (like if we know a Policy Horizons PDF is crucial, we could pre-download and summarize it offline, then provide that summary when appropriate).

- **FastAPI Endpoint Implementation:**

```
research_router = APIRouter(prefix="/research")
research_client = ResearchClient()

@research_router.get("/search")
def search_research(q: str):
    results = research_client.search(q)
    # We can truncate or simplify results to what we need
    simplified = []
    for item in results[:3]:  # limit to top 3 for brevity
        simplified.append({
            "title": item.get("title"),
            "snippet": item.get("snippet"),
            "link": item.get("link")
        })
    return {"results": simplified}
```

This will give the top 3 results with snippet and link. The AI agent can then decide what to do – possibly present these to the user or use the snippet info in its response.

- **AI Usage Pattern:** We might integrate this as a **tool the GPT model can use** when it doesn't have an answer from its internal knowledge. One approach is using an **agent pattern** (like ReAct: the model decides "I need to search for X," calls the search tool, sees results, then formulates answer). OpenAI function calling or LangChain-like approaches could facilitate this. But even without full agent complexity, we can have some triggers. For example, if user query contains certain keywords ("find information" or "what do studies show about..."), we route it to a search function first, then feed both the query and results to GPT in a formatted prompt. E.g.,

```
Human: "What are the latest trends in AI policy in other countries?"
(System triggers search for "AI policy other countries latest trends")
(System gets results and creates a context like:
 "Result1: Title... Snippet...; Result2: ...")
System (to GPT): "Using the following research results, answer the
question..."
```

This way GPT has some info to go on. We will have to curate how we present the snippet to GPT to ensure it uses it correctly and cites if needed.

- **Auth & Rate Limits:** Google's API key usage has the daily free quota and then per-query cost. Given we'll only demonstrate a few queries, we are safe under free limits. We will ensure the API key is not

exposed (store server-side). If offline demo with no internet, we might instead prepare a stub: e.g., have a cached JSON of search results for expected queries and have the `ResearchClient` return that (to avoid actual external calls in a closed environment). This is an implementation detail depending on demo constraints.

- **Alternate API (Bing)**: If we went with Bing, the code would be similar: it requires an Azure subscription key, and the endpoint `https://api.bing.microsoft.com/v7.0/search?q={query}`. The JSON structure differs slightly. Given that Bing's service might be changing (per the note of retirement [21]), Google's is safer for now.

- **Content Filtering:** We will keep the search queries general and trust the domain filtering to avoid any NSFW or irrelevant content. Since we restrict to GC and similar domains, we shouldn't hit problematic content. Otherwise, we'd use the search API's safe search parameters or filter out results that aren't from allowed domains if needed.

- **Response Integration:** The final answer generation by GPT might include quotes from sources. We can programmatically append source info in the answer (like our conversation interface might automatically attach references). But in the simplest form, GPT could just mention the source by name as part of its answer, using the snippet info. This may be sufficient to satisfy users' curiosity without us building a full citation system (though our UI appears to support references, as evidenced by our citation format, but that's meta in this explanation).

To illustrate, here's pseudocode tying it together in a pseudo-agent flow:

```
user_query = "Are there examples of similar initiatives in other countries?"
if needs_research(user_query):  # some keyword or classifier
    search_results = research_client.search(user_query)
    context = summarize_snippets(search_results[:3])
    prompt = f"User asked: '{user_query}'. I searched relevant sources and
found:\n{context}\nUsing this information, answer the question comprehensively."
    answer = openai_completion(prompt)
else:
    answer = openai_completion(user_query)
```

Where `summarize_snippets` might combine snippets into a brief background paragraph for GPT.

This technical setup will empower PolicyGPT to extend beyond its base knowledge and incorporate up-to-date, specific information, making its responses far more valuable in a policy context. It effectively connects our AI to the wealth of public domain policy knowledge, similar to how an actual policy team would operate by consulting various resources.

(Citations for this section: Google Custom Search JSON API [15], Bing API JSON response [16], Semantic Scholar API overview [18].)

Meeting Capture & Summarization Integration

Policy development is a team sport – it involves meetings for brainstorming, consultations with stakeholders, and governance boards (like Gate review meetings). Capturing the content of these meetings and summarizing action items or decisions is essential. The PoC envisions that PolicyGPT could ingest and summarize meeting discussions, then use that information to inform documents (for example, summarizing a consultation session and including key points in a briefing note). Thus, an integration is needed to capture **meeting audio or transcripts**, and generate summaries.

We look at options that allow us to either get a transcript from a meeting platform (Teams, Zoom, etc.) or to process audio recordings with AI, then summarize. Considerations: integration complexity with meeting software, availability of recorded data, and maintaining confidentiality (meetings often contain sensitive info).

Viable Options

- **Microsoft Teams (Graph API for Transcripts)** – Since GC uses Microsoft Teams extensively, leveraging its ability to record and transcribe meetings is most realistic. Microsoft Graph has APIs to fetch meeting transcripts and recordings **after the meeting** [22]. **Feasibility:** Moderate to Low for PoC – To use Graph, one needs proper app permissions and it typically works in an enterprise context. For a PoC, setting up a dummy M365 tenant and scheduling meetings might be heavy. Additionally, Graph transcript API was recently made available and requires that the meeting had transcription enabled and the organizer's consent, etc. **Realism:** Very High – If implemented, this is exactly how a real solution would capture meeting data in GC (with all security of their cloud). **Pros:** Transcripts are accurate (Teams uses Azure Cognitive Service for transcription), speaker-attributed, and no data leaves Microsoft's ecosystem. We could then feed the transcript text to GPT for summarization. **Cons:** Setup and credentials – the PoC team would need to automate Teams meetings or use existing transcripts. Also, Graph calls are "metered" (possibly require payment or special allowances) [23]. Likely not trivial to implement fully within a short PoC timeframe.

- **Zoom API + Transcription** – Zoom, while less common in GC, provides cloud recording and transcript features for paid accounts. An API can retrieve the transcript file of a recorded meeting [24]. **Feasibility:** Moderate – It requires a Zoom account with enabled cloud recording. But Zoom's API is straightforward (OAuth or JWT app). **Realism:** Low – GC departments generally avoid Zoom for sensitive meetings, though some use it for public consultations. It's not as standard as Teams. **Pros:** If the team already uses Zoom, it's an easy way to get a transcript without building speech-to-text ourselves. **Cons:** Introducing Zoom in the PoC might break the illusion for GC folks, as they'd expect Teams. Also, dealing with Zoom auth and downloading files is similar complexity to Graph, but with less relevance.

- **Third-Party AI Transcription Services (AssemblyAI, Otter.ai, etc.)** – These services let you upload audio and get a transcript and summary back using AI. **Feasibility:** High – For example, AssemblyAI provides an API where you submit an audio file URL, and it returns a transcript and even an automatic summary [25]. Integration is simple HTTP with an API key, and Python SDKs are available [26]. Otter.ai and others are more end-user apps, not open APIs (Otter has some integration but not a public API). **Realism:** Moderate – GC wouldn't use an external service for real classified meetings (privacy concerns), but for a PoC with dummy data, it's fine. Many individuals are familiar with these

tools conceptually (automatic meeting notes). **Pros:** Easiest to implement – no need to orchestrate actual meetings; we could use a recorded audio or even a text script and feed it to the service. Some services (AssemblyAI, OpenAI Whisper via API) can achieve high accuracy. The summarization can be done by the service or by feeding the transcript to GPT. **Cons:** Requires sending data to a third party – not an issue for fake data, but we should note it's not production-suitable for GC. Cost could be a factor if processing long recordings, but AssemblyAI offers some free credits [26]. Summaries might not capture nuances unless tuned.

- **In-House Transcription (Open Source)** – Use an open source speech-to-text model (like OpenAI's Whisper) locally to transcribe recordings, then summarize with GPT. **Feasibility:** Medium – Whisper models (especially the large version) are heavy but could be run on a local machine (or a smaller model for speed). This avoids external API and costs. Summarization can be done with our existing GPT model. **Realism:** High in principle – GC could host such models internally for privacy. But for PoC, doing this integration may be more effort (setting up audio processing pipeline) than using a ready API. **Pros:** No external dependencies, complete control; demonstrates potential for fully on-prem solution. **Cons:** Implementation time, need for decent hardware if using a big model. For PoC, probably not worth it unless we already have a small audio file to test on.

Considering these, the fastest route is to use a third-party transcription+summary API (we can always say this simulates what an internal service would do). Meanwhile, to maintain GC flavor, we can narrate that "This could be done via Teams in practice." So a hybrid approach: We'll implement with an **AI transcription service** but frame it in the context of Teams meetings.

Recommended Option: AI Meeting Transcription Service (AssemblyAI) to Simulate Teams

We will use **AssemblyAI's API** to handle meeting transcription and summarization in the PoC. AssemblyAI allows us to submit an audio file and get back a transcript and an AI-generated summary of that transcript [27] [25]. This significantly reduces integration complexity (just a REST call with our API key). We can record a sample meeting (or even have a scripted conversation) relevant to our policy scenario, and use that as input. The output summary can then be fed to or presented by PolicyGPT.

Justification: This approach gives a quick win: we can demonstrate the capability of turning a raw meeting into useful text for policy writing. It balances realism (the concept of an automatic meeting summary is becoming common, e.g., Teams has a "Intelligent Recap" feature in development by Microsoft) with speed (we don't have to build or heavily configure enterprise tools in the PoC timeframe).

We will clarify that in a real GC implementation, this function would integrate with **Teams** (since Teams now has APIs to fetch transcripts [22]), but for the demo we are using a stand-in service. The output – a concise meeting summary – supports the **gating process**: e.g., capturing decisions from a Gate review meeting or summarizing stakeholder feedback from consultations (which might occur between Gate 2 and 3). PolicyGPT could then automatically incorporate these decisions and feedback into the next version of the document.

User Experience: Meeting Summaries in PoC vs Real Life

How will the user interact with this feature in the demo, and how does it map to their real experience?

- **In the demo:** We might present it as, “PolicyGPT can ingest your meeting recordings and summarize them.” For example, we can have a scenario where a team had a meeting discussing policy options. The user can either upload an audio file (if we expose that in the UI) or simply instruct the AI, “Summarize the last stakeholder consultation meeting.” If we have preprocessed that meeting with AssemblyAI, PolicyGPT can then display the summary. Alternatively, the AI might already have the summary in its context and use it to answer questions like “What were the key concerns raised by stakeholders?” In the demo, the summary might look like a short bullet list or paragraph: “Stakeholders raised concerns about cost and timeline, and emphasized the importance of user training. They agreed the policy should include a phased implementation…” etc.

- **Real GC process:** Traditionally, someone would take meeting minutes or notes. Often those notes are lengthy and need distillation. The idea that AI can give an **instant executive summary** of a meeting is quite appealing. Microsoft is actually integrating such features in Teams (which some forward-looking GC users might have heard of). So seeing it in PoC will feel quite cutting-edge but plausible. The user doesn’t have to slog through transcripts; they get the gist. If the PoC meeting summary is well-crafted, policy professionals will appreciate how it frees them from note-taking and lets them focus on content.

- **Accuracy and completeness:** A potential concern is whether the AI summary misses nuance. In real life, a policy analyst might distrust a fully automated summary until they verify it. In the PoC, since we control the input, we can ensure the meeting content is simple enough that the summary is obviously correct. For instance, we wouldn’t test it on a highly technical or contentious meeting where summary might oversimplify. We’ll use a scenario where the meeting had clear, simple outcomes (for demo clarity).

- **Integration with writing:** The real value is that the summary or even full transcript can feed into writing. For example, after summarizing the consultation, the user could ask PolicyGPT to update the business case’s “Consultation section” with info from that summary. The AI can do that since it has the summary text. This shows a seamless pipeline: meeting -> summary -> updated document. In GC, that pipeline is currently manual (someone writes the summary then copy-pastes it). The PoC experience thus demonstrates efficiency.

- **Privacy considerations:** In real usage, recording and transcribing meetings raises privacy issues (one must inform participants, etc.). GC has rules about this. For PoC, we sidestep that by using dummy content and presumably all participants “consent” by design. If a GC user asks “Would this be allowed?”, we can note that by the time of a real implementation, the integration would use approved internal services (like Teams on GC cloud) to ensure data stays secure. Since we highlight that our PoC uses an external service only for convenience, that should satisfy concerned viewers.

Overall, the meeting summarization feature will appear as a helpful add-on where PolicyGPT not only writes but also listens and learns from discussions, making it a well-rounded assistant.

How Meeting Capture Supports PolicyGPT's Mission

Integrating meeting capture and summarization contributes to PolicyGPT's goal of supporting **collaboration** and ensuring that **no information is lost** in the policy development process:

- **Collaboration & Knowledge Sharing:** Meetings are a key collaborative activity. By capturing them, PolicyGPT acts as a second pair of ears. Team members who weren't in a meeting could ask PolicyGPT what was discussed, or use the summary to get up to speed. This is particularly relevant in a GC context where projects often have turnover or involve multiple departments. It also encourages transparency – the PoC could show that even if a manager misses a meeting, the AI can brief them. In gating terms, before a Gate review, there might be many prep meetings; the AI can ensure the outputs of those prep meetings (decisions, agreed changes) are reflected in the documents that go to the actual Gate approval.

- **Documenting Decisions:** One of the purposes of project gating is to have documented decisions at each gate. A meeting summary integration means that if the Gate Review Board meets (which is a meeting) and outlines conditions to proceed [28], PolicyGPT can capture those and help incorporate them into the updated project plan or record them for compliance. For example, if Gate 2 decision was "proceed with conditions X, Y, Z," the AI could automatically log that and later ensure the next gate submission addresses them. This makes the gating process more rigorous and traceable.

- **Reducing Administrative Burden:** Taking minutes is an admin burden often falling on policy analysts or project officers. Automating it with PolicyGPT's help means those staff can focus more on substantive work. The AI-generated summary might even be good enough to use as official record (with minor edits). This value proposition can be highlighted in the PoC narrative.

- **Input for AI Drafting:** The content of meetings (especially consultations or expert briefings) often contains qualitative insights that need to be written up in policy docs. Having an integration to capture those means the AI has additional **contextual knowledge** to draw from. For instance, if stakeholders expressed a concern, the AI can ensure the policy recommendation addresses it. Without this, the AI would rely only on what the user explicitly tells it. This integration broadens the AI's situational awareness.

- **Timeliness:** Summaries can be generated in near-real-time after a meeting. So if immediately after a meeting the team wants to update a document, they can ask PolicyGPT right away. In GC, often there is a lag – someone writes minutes days later and by then some context is lost. PolicyGPT could eliminate that lag, making the process more agile (which is something GC's gating sometimes lacks, as gating is seen as bureaucratic – this shows how AI might streamline it).

All told, meeting capture & summarization integration ties the loop between *discussions* and *documents*, a loop that is currently very manual. It reinforces PolicyGPT as a comprehensive copilot through all aspects of policy development.

Integration Technical Spec: AssemblyAI (Transcription & Summary) via FastAPI

We outline how to implement the meeting summarization integration using AssemblyAI's API. The process involves uploading an audio file (or providing a URL to it), then retrieving the transcription and summary from the API. Key steps and design:

- **AssemblyAI API Key:** We will obtain a free API key from AssemblyAI [26] and store it in the backend config. The service offers a unified endpoint where you can send an audio and request features like auto-summarization.

- **Submitting Audio for Transcription:** We have two ways:

- **Direct Upload:** The AssemblyAI API allows posting raw audio data, but a simpler way is to host the audio file somewhere (even a presigned URL or a public link) and just send the URL to AssemblyAI.
- **Live Record -> File:** For PoC, likely we'll have a pre-recorded meeting saved as an MP3. We can include that file in the project and maybe serve it via a static route or just send it from disk.

The API call (per their docs) is a `POST https://api.assemblyai.com/v2/transcript` with JSON:

```
{
  "audio_url": "https://link.to/meeting.mp3",
  "summarize": true,
  "summary_type": "bullets"
}
```

We can ask for different summary styles (bullets, paragraph, etc.). Bullets are nice for clear key points.

- **Polling/Callback:** AssemblyAI's transcription is async. After submitting, one must poll an endpoint `/v2/transcript/{id}` until the status is `completed`, then get the results including `text` (full transcript) and `summary_text` if summarization was requested. For PoC, since the audio might be several minutes, this could take some time (a few minutes to transcribe depending on length). We might simulate or speed this up by using a short audio. Alternatively, do the transcription offline and store the results, returning them immediately in demo (to avoid waiting).

- **FastAPI Implementation:** We can create an endpoint `@app.post("/meetings/summarize")` which accepts an audio file upload or a reference ID. In a simple design, we might not even expose it to the frontend – we could preprocess and just have the summary ready. But to be thorough:

```
import requests, time

AAI_API_URL = "https://api.assemblyai.com/v2/transcript"
AAI_HEADERS = {"authorization": os.getenv("ASSEMBLYAI_API_KEY")}

@app.post("/meetings/summarize")
def summarize_meeting(meeting_file: UploadFile = File(...)):
```

```
    # Save the file locally
    file_location = f"/tmp/{meeting_file.filename}"
    with open(file_location, "wb") as f:
        f.write(meeting_file.file.read())
    # Option 1: upload to AssemblyAI's hosted endpoint for file (they have
an upload endpoint too)
    upload_url = upload_to_assemblyai(file_location)
    # Request transcription with summary
    data = {"audio_url": upload_url, "summarize": True, "summary_type":
"bullets"}
    response = requests.post(AAI_API_URL, json=data, headers=AAI_HEADERS)
    transcript_id = response.json().get('id')
    # Poll for completion (in real scenario, better to use WebSockets or
callback)
    for _ in range(60):  # e.g., poll up to 60 times (approx 60*2 sec = 2
min)
        status_resp = requests.get(f"{AAI_API_URL}/{transcript_id}",
headers=AAI_HEADERS)
        status = status_resp.json().get('status')
        if status == 'completed':
            summary = status_resp.json().get('summary_text')
            return {"summary": summary}
        elif status == 'error':
            raise HTTPException(status_code=500, detail="Transcription
failed")
        time.sleep(2)
    raise HTTPException(status_code=202, detail="Transcription in
progress")
```

Here `upload_to_assemblyai` might use AssemblyAI's specific upload endpoint to send the file and get a URL (they provide that to avoid hosting files externally).

In practice, we might not want to do the upload via FastAPI if the file is large; instead, instruct the user to record and store it accessible. But as a PoC, an upload is fine for a short file.

- **Using the Summary in PolicyGPT:** Once we have the summary text, what do we do? We can:
- Show it to the user (as an output of the above API call).
- Internally store it (perhaps attached to a meeting ID or date) so that later the user can ask questions referencing that meeting. For example, keep a dict like `meeting_summaries = {"stakeholder_meeting_2025-05-01": summary}`. The AI model could then be prompted with that summary if needed.
- Or even directly feed the summary into the GPT conversation context if we know the user is going to move to writing based on it.

One likely use-case: After summarizing, the user says, "Incorporate the consultation findings into section 4 of the business case." PolicyGPT then uses the summary as input to generate that section. So the backend should make the summary accessible to the drafting function. Possibly we store it in memory or in the

document management system (imagine saving it as a text file on Google Drive for traceability). For simplicity, keeping it in a variable or passing it along is fine.

- **Auth & Privacy:** AssemblyAI API key is secret; audio content goes to their servers (hosted presumably in the US). We only use mock data, so privacy is not a concern for PoC. In a real-case discussion, we note that the integration could instead use Azure's cognitive services in the GC cloud to do similar transcription (which would be analogous to using Teams' built-in capabilities).

- **Error Handling:** We should consider what happens if audio is poor quality or the summary isn't great. Possibly ensure our sample audio is clear. If AssemblyAI returns an error or a bad summary, we could have a fallback: either just show the raw transcript or have a manually written summary to use. Given this is a demo, controlling the inputs tightly is advisable.

- **Demo Flow Consideration:** Possibly we won't actually record a live meeting during the demo (that would be complicated and slow). Instead, we might prepare one in advance. So the demonstration might be: "PolicyGPT already processed yesterday's meeting," and the summary is readily available to ask questions on or to include in text. This way we skip the waiting in front of the audience. The integration is still there under the hood (we did the work), but we present it as something that has happened. If we wanted to show it live, we might use a very short audio clip (like 30 seconds) so that transcription returns in a few seconds. This could be risky if networking or API is slow.

Given these steps, we have a clear plan to integrate meeting transcription and summarization. It provides an API endpoint for uploading and summarizing an audio recording, and then we can leverage the result in subsequent GPT operations – fulfilling the vision of turning meetings into actionable content.

(Citations for this section: Microsoft Graph transcripts availability [22], AssemblyAI summarization capability [25].)

Conclusion: By integrating these four system types – **Document Management, Data Lookup, Research Search, and Meeting Summaries** – into PolicyGPT, we create a realistic environment in which an AI assistant can operate similarly to a GC policy analyst's digital workspace. Each recommended product was chosen to maximize our PoC agility while maintaining credibility: Google Drive for document management (simulating GCdocs' repository function with a more demo-friendly API) [2], StatsCan's API for data (ensuring factual accuracy from official sources) [9], a Google-based custom search for research (broad access to policy knowledge with focus on authoritative domains) [15], and AssemblyAI for meeting notes (demonstrating automated insight extraction akin to upcoming Teams features) [25]. Together, these integrations enable PolicyGPT to ingest and output information across the policy development lifecycle: from initial research and consultations through drafting and refinement to final documentation for gate approvals. This comprehensive approach will make the PoC experience feel authentic to GC professionals and clearly showcase how AI can streamline the creation of policy artifacts in alignment with the **IT Project Gating Framework**'s emphasis on informed decision-making and documented evidence at each gate [8] [29].

---

1 8 28 Guide to Project Gating - Canada.ca
https://www.canada.ca/en/treasury-board-secretariat/services/information-technology-project-management/project-management/guide-project-gating.html

2 Software Introduction –OpenText GCDocs™ | Institute of Certified Records Managers
https://www.icrm.org/node/2963

3 5 Python quickstart | Google Drive | Google for Developers
https://developers.google.com/drive/api/quickstart/python

4 fastapi-cloud-drives · PyPI
https://pypi.org/project/fastapi-cloud-drives/

6 Access OneDrive and SharePoint via Microsoft Graph API - OneDrive dev center | Microsoft Learn
https://learn.microsoft.com/en-us/onedrive/developer/rest-api/?view=odsp-graph-online

7 Stop misusing GCDocs, it's terrible : r/CanadaPublicServants - Reddit
https://www.reddit.com/r/CanadaPublicServants/comments/pdfzim/stop_misusing_gcdocs_its_terrible/

9 10 14 Application Program Interface (API)
https://www.statcan.gc.ca/en/microdata/api

11 12 13 Open Government API - Open Government Portal
https://open.canada.ca/data/en/dataset/2d90548d-50ef-4802-91f8-c59c5cf68251

15 17 Custom Search JSON API | Programmable Search Engine | Google for Developers
https://developers.google.com/custom-search/v1/overview

16 21 Bing Web Search Python client library quickstart - Bing Search Services | Microsoft Learn
https://learn.microsoft.com/en-us/bing/search-apis/bing-web-search/quickstarts/sdk/web-search-client-library-python

18 Semantic Scholar Academic Graph API | Semantic Scholar
https://www.semanticscholar.org/product/api

19 Canadian Open Science Repositories
https://guides.library.ualberta.ca/open-science/cnd-repositories

20 Create custom Google search engine for your domain(s) and fetch ...
https://dev.to/jochemstoel/create-custom-google-search-engine-for-your-domains-and-fetch-results-as-json-69o

22 23 29 Fetch Meeting Transcripts & Recordings - Teams | Microsoft Learn
https://learn.microsoft.com/en-us/microsoftteams/platform/graph-api/meeting-transcripts/overview-transcripts

24 7 APIs to get Zoom transcripts: A comprehensive guide - Recall.ai
https://www.recall.ai/post/7-apis-to-get-zoom-transcripts-a-comprehensive-guide

25 26 Summarize meetings in 5 minutes with Python
https://www.assemblyai.com/blog/summarize-meetings-python

27 Summarization | AssemblyAI | Documentation
https://www.assemblyai.com/docs/audio-intelligence/summarization