# GovSight

## Project Overview

**GovSight** is a persistent retrieval-augmented generation (RAG) system designed to offer ChatGPT-like assistance for municipal lobbying and government intelligence. It integrates multiple data sources and reasoning layers to answer user queries about local government information. At a high level, GovSight’s architecture combines:

* **Local Fact Store (Memory DB):** A SQLite database of facts and conversation history for quick recall of known information[[1]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=Features%3A%20,NEW%29). This memory layer supports structured subject-attribute-value triples and conversation logging.
* **Vector Search (Pinecone):** A semantic embedding index for retrieving relevant context or documents when local facts are insufficient.
* **Web Fallback (Live Search):** As a last resort, GovSight queries the web and summarizes results using an LLM to provide up-to-date information.

The system follows a cascading logic: it first attempts to answer using the local database, then checks the Pinecone vector index for relevant data, and finally performs a web search if needed[[2]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Memory%20Error%5D%20%7Be)[[3]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Web%20Query%20Error%5D%20%7Be). GovSight can be used via a command-line chat interface or through dedicated ingestion scripts that populate its knowledge base.

## Setup and Installation

**1. Environment Setup:** Clone the GovSight repository and ensure you have Python 3.10+ available. Create a virtual environment (if not using the provided venv/ folder) and install dependencies:

pip install -r requirements.txt

GovSight expects certain API keys to be set as environment variables (or in a .env file). These include **OpenAI API key** (OPENAI\_API\_KEY) for LLM services, and **Pinecone API key/ENV** (PINECONE\_API\_KEY, PINECONE\_ENV) for vector search. The provided .env file can be used to store these. The configuration loader will automatically load .env on startup[[4]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=import%20os%20import%20openai%20from,dotenv%20import%20load_dotenv).

**2. Database Initialization:** GovSight uses a SQLite database at govsight/data/memory.db for persistent memory. On first run, the necessary tables will be created automatically. You do not need to manually set up the database; the memory module bootstraps the schema if it’s missing[[5]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn). If you have an existing older memory.db, a one-time migration script (migrate\_memory\_db.py) is provided to update the schema – back up your DB and run this script if needed.

**3. Running the CLI:** After installing dependencies and setting environment variables, you can launch the GovSight chat interface. There are two entry points: - **Direct CLI (legacy engine):** Run the talk.py script to start an interactive Q&A session in the terminal. For example: python talk.py. This will initialize the memory database (if not already) and drop you into a prompt. - **Packaged CLI (new entry point):** GovSight also provides a wrapper CLI module: python -m govsight.cli.chat\_cli. This uses the same underlying logic by invoking the talk.py engine, but is structured for future updates. Both methods currently behave similarly.

When the CLI starts, you should see a welcome message and instructions. Type your questions or facts, and GovSight will respond by checking the local DB, Pinecone, and then the web in sequence[[6]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,continue)[[3]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Web%20Query%20Error%5D%20%7Be). Type “exit” or “quit” to end the session.

**4. Directory Layout:** The repository is organized as follows: - **Root** – High-level scripts and configs (e.g. talk.py, main.py, README.md, requirements.txt). - **govsight/** – Python package containing the core modules (memory, parser, retrieval, etc.). - **agents/** – Specialized agent scripts for particular tasks (e.g. resolving bill names, tagging). - **ingest/** – Data ingestion scripts for populating the knowledge base (e.g. ingesting bills, news, votes). - **utils/** – Standalone utility scripts (for Pinecone index management, scraping, etc.). - **tests/** – Basic tests for API connectivity and environment setup. - **data/** – Contains the SQLite database file (memory.db) and backups. - **logs/** – Contains runtime logs (e.g. govsight.log, web\_fallback.log).

## Codebase Structure and Module Analysis

The GovSight codebase is modular, with each Python module handling a specific aspect of the system. Below is a detailed breakdown of every major Python file, organized by functionality:

### Core Engine (CLI and Main Control)

* **talk.py** – This is the primary interactive CLI script (legacy engine). It creates a Memory instance and enters a loop to read user input[[7]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=def%20main)[[6]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,continue). For each input, it performs:
* **Local DB Check:** Uses search\_local\_facts() to find a fact in the SQLite DB. If found, returns it immediately[[6]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,continue).
* **Memory/Pinecone Check:** Attempts to retrieve an answer from the persistent memory layer (memory.search()), which currently checks the facts DB for a matching entry[[8]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,db_path%2C%20subject%2C%20attribute%29%20if%20answer)[[9]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=,green%5DGovSight%20%28Web%29%3A%5B%2Fgreen%5D%20%7Bsummary). *(Note: In the code, this is labeled as a Pinecone check, but the actual Pinecone query is not yet integrated in this step.)*
* **Web Fallback:** If no answer yet, calls the web reasoner (query\_web\_and\_summarize()) to search online and summarize results[[3]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Web%20Query%20Error%5D%20%7Be).

The talk.py script then prints the answer from whichever source provided it. It also stores any new fact provided by the user into the memory (for example, if the user says “The mayor of X is Y”, it will parse and store that fact)[[10]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,green%5DGovSight%20%28Memory%29%3A%5B%2Fgreen%5D%20%7Banswer). This script is marked as the “legacy” engine, and its functionality is being refactored into the modular govsight package.

* **govsight/cli/chat\_cli.py** – A thin wrapper that allows launching GovSight via python -m govsight.cli.chat\_cli. This module sets up logging and loads settings, then delegates to the legacy talk.main() function. It exists to provide a stable CLI entry point as the internal engine evolves. In the future, the implementation of chat\_cli can be changed to call a new engine without impacting how users launch the CLI. For now, it simply prints the welcome banner and invokes talk.main().
* **main.py** – A utility script at the root for running specific sub-agents. It defines an AGENTS mapping of agent names to their module import paths. When run with --agent <name> argument, it will dynamically import the corresponding module and execute its main() function. For example, python main.py --agent news will run the ingest/news\_agent.py script. If no --agent is provided, it lists available agent keys. This script is mainly for convenience during development or batch ingestion tasks.
* **web\_ui.py** – An experimental Streamlit app for a web-based UI (not fully integrated). It uses Streamlit to create an interactive Q&A interface in the browser. This UI loads the OpenAI and Pinecone API keys from config and performs an embed-search-answer cycle: it embeds the user’s query, queries the Pinecone index, constructs a context from the top results, and then asks GPT-4 for an answer. The answer and query are displayed in a chat history on the page. *Note: this component uses its own logic separate from the main GovSight pipeline and may not reflect the latest memory or parser improvements.* It’s primarily a proof of concept for a graphical interface.

### GovSight Core Modules

These modules reside in the govsight/ package and implement the core logic of memory storage, language parsing, retrieval planning, etc.

* **govsight/config/\_\_init\_\_.py** – Configuration loader. It reads environment variables (via dotenv) and defines a Settings class for all runtime settings. A global settings object is instantiated from this class. Key fields in Settings include paths (db\_path, log\_dir) and API keys (openai\_api\_key, pinecone\_api\_key, etc.). By default, db\_path is set to govsight/data/memory.db, and directories are created if missing. This centralized config allows the rest of the code to import settings and not worry about environment details. (There is also a settings-obsolete.py which contains a more complex dataclass approach and legacy config integration, but the project currently uses the simpler Settings in \_\_init\_\_.py. See **Redundant Components** below.)
* **govsight/memory/memory.py** – Implements the **Memory Layer**, which handles persistent conversation logs and fact storage[[1]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=Features%3A%20,NEW%29). Key points:
* **Schema Initialization:** On creation, Memory connects to the SQLite DB at settings.db\_path and calls bootstrap() to ensure tables exist[[5]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn).
* **Session Logging:** Methods like start\_session(session\_id) and log\_message(session\_id, role, message) insert conversation turns into the sessions and messages tables[[11]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20start_session%28self%2C%20session_id%3A%20str%29%20,%28session_id%2C%20int%28time.time)[[12]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=INSERT%20INTO%20messages%20,time), enabling chat history to be saved.
* **Fact Storage:** store\_fact(subject, attribute, value, source, confidence) inserts a new fact triple into the facts table[[13]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,%3F%2C%20%3F%2C%20%3F%2C). It also generates a slug for the subject (a normalized key) for easier lookup[[14]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,attribute%2C%20value%2C%20source%2C%20confidence%2C%20inserted_at). There is support for batch insertion via insert\_fact\_triples()[[15]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20insert_fact_triples%28self%2C%20triples%3A%20List%5Bdict%5D%29%20,).
* **Fact Retrieval:** search(query) performs a simple lookup in the facts table for any fact whose subject, attribute, or value matches the query text[[16]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20search%28self%2C%20query%3A%20str%29%20,result%20%3D%20cursor.fetchone)[[17]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20search%28self%2C%20query%3A%20str%29%20,fetchone%28%29%20if%20result). It returns the most recently inserted matching fact formatted as “Subject → Attribute: Value”. This allows quick retrieval of a known fact. (This functionality was recently fixed – see **Recent Updates** – to address a bug with missing slug columns and to ensure compatibility with the new OpenAI API usage.)

The memory module essentially provides a lightweight knowledge graph that persists across sessions. Its design and features are summarized in the header comment as supporting versioned persistence, slugification of subjects, conversation tracking, and NLP-based fact extraction[[1]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=Features%3A%20,NEW%29).

* **govsight/memory/records.py** – Defines simple dataclasses for in-memory representation of DB records. For example, FactRecord, SessionRecord, etc., mirror the columns of the facts and sessions tables. These make it easier to pass around query results in a structured way instead of raw tuples. (Currently, much of the code uses direct SQL and tuples, but these classes are available for future use.)
* **govsight/memory/schema.py** – Contains the SQL schema and migration logic for the memory DB. It defines bootstrap(conn) which creates tables (sessions, messages, facts) if they do not exist[[18]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn), and possibly a get\_schema\_version() to handle future migrations. This was used to upgrade the schema in the transition to the R2 memory design. The migrate\_memory\_db.py script in root uses this module to detect legacy table structure and port data if needed.
* **govsight/memory/watchlist.py** – (Planned, partially implemented) Intended to support a “watchlist” feature for tracking entities of interest. The memory manager notes mention maintaining a watchlist when a user expresses a “track/monitor” intent. However, the watchlist.py file is either empty or not included in the current build (it’s listed in the directory but not in the archive content). This indicates the feature is still a work in progress. We expect it to define data structures and methods to add items to a watchlist and perhaps query them later for updates. Currently, no active code references this module.
* **govsight/parser/parser.py** – Houses LLM-powered parsing functions to interpret user inputs. It provides:
* **parse\_fact\_from\_text(text)** – Uses an OpenAI GPT model to extract a triple (subject, attribute, value) from a sentence[[19]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=def%20parse_fact_from_text%28text%3A%20str%29%20,and%20value%20from%20a%20sentence)[[20]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D). It constructs a prompt asking for a single fact extraction and parses the model’s response into the tuple form[[21]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=print%28f)[[22]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=lines%20%3D%20response.strip%28%29.split%28,value%20except%20Exception%3A%20return%20None).
* **parse\_intent\_and\_facts(text)** – Uses GPT to classify the user’s intent (e.g. asking a question, providing a fact, casual chat) and optionally extract a fact[[23]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=def%20parse_intent_and_facts%28text%3A%20str%29%20,and%20extract%20any%20known%20facts)[[24]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D). The prompt asks for a JSON object output with fields intent, subject, attribute, value. The function returns a Python dict parsed from the model’s JSON answer[[25]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=response%20%3D%20chat_completion%28messages%2C%20model%3Dsettings).

These functions rely on an OpenAI chat model (configurable via settings.openai\_model). The parser is used at runtime to decide how to handle user input. For example, if a user’s input contains a factual statement, parse\_intent\_and\_facts will identify it as intent: provide\_fact and extract the triple, which talk.py then stores in memory[[26]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=try%3A%20,value)[[10]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,green%5DGovSight%20%28Memory%29%3A%5B%2Fgreen%5D%20%7Banswer). If it’s a question, the system knows to search for an answer. This module was recently updated to ensure compatibility with the latest OpenAI SDK and to refine the prompts.

* **govsight/parser/fact\_parser.py** – A supplemental parser that attempts lightweight **regex-based** extraction of facts before falling back to the LLM. It defines its own parse\_fact\_from\_text(text) that handles common patterns:
* Declarative statements like “The \<attr> of \<subject> is \<value>.”.
* Questions like “What is the \<attr> of \<subject>?”, or “Who is the \<attr> of \<subject>?”.

If the input matches these patterns, it returns a dictionary with subject, attr, and value (value may be None for questions). If none of the regex matches, it will call the GPT-based parser as a fallback. This combined approach improves efficiency by handling straightforward cases without an API call, and only using the LLM for complex or unexpected input. The govsight.retrieval.constraints module uses this to extract query constraints.

* **govsight/retrieval/constraints.py** – Defines a QueryConstraints dataclass and an extract\_constraints(query) function. This likely interprets a user query to figure out what known information can be used to answer it. For example, it might pull out the subject or attribute from a question to see if it’s in the local facts. It leverages the fact\_parser.parse\_fact\_from\_text to get structured data from the query. The output (if a subject/attribute is found) can guide the retrieval process – e.g., if the query is “What is the population of Grandview, TX?”, constraints might be subject: “Grandview, TX”, attribute: “population”.
* **govsight/retrieval/structured.py** – Uses QueryConstraints to fetch a known fact from memory. It provides get\_structured\_fact(constraints) which, if the constraints contain a subject and attribute, will query the Memory (or underlying DB) for a matching fact. This is essentially a structured lookup in the local knowledge base (the facts table). It imports Memory and FactRecord to perform this operation.
* **govsight/retrieval/semantic.py** – Stub for semantic search. It defines semantic\_search() which presumably will query Pinecone for embeddings relevant to the query. Currently it’s just a placeholder (likely returns nothing or dummy data), meant to be implemented to do vector similarity search in the Pinecone index.
* **govsight/retrieval/web\_search.py** – Placeholder for the web fallback layer. It defines web\_fallback() that should perform a web search and return results if the system has no answer locally. In this R0 version, it is not fully implemented. The actual web search logic is handled in the govsight/web\_reasoner module (see below). The existence of this stub suggests that in a future iteration, the retrieval planner might call web\_search.web\_fallback(query) as part of a unified pipeline.
* **govsight/retrieval/planner.py** – Orchestrates the above retrieval steps via a single retrieve(query) function. It calls extract\_constraints to parse the query, then tries get\_structured\_fact (for exact facts), then semantic\_search (vector/Pinecone), then web\_search.web\_fallback as last resort. It returns either a found fact or triggers the web fallback. In the current state, this integration isn’t fully wired into the CLI flow; the CLI calls the memory and web steps directly rather than using planner.retrieve. However, this module outlines the intended logic flow in a more modular way for future refactor.
* **govsight/llm/openai\_wrapper.py** – A simple wrapper around OpenAI API calls. It loads the API key via load\_dotenv() and configures openai.api\_key[[27]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=import%20os%20import%20openai%20from,dotenv%20import%20load_dotenv). It provides:
* get\_embedding(text, model="text-embedding-3-small") to obtain an embedding vector from OpenAI’s embedding API[[28]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20get_embedding,%F0%9F%94%B4%20Embedding%20error%3A%20%7Be).
* chat\_completion(messages, model="gpt-4") to get a chat completion from the OpenAI chat endpoint[[29]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20chat_completion,0%5D.message.content.strip). This function wraps openai.ChatCompletion.create (or similar; note the code uses openai.chat.completions.create which corresponds to the Chat API) and returns the assistant’s message text[[30]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=try%3A%20response%20%3D%20openai,%F0%9F%94%B4%20Chat%20completion%20error%3A%20%7Be).
* summarize\_web\_content(content, query="") to summarize a chunk of web page text in the context of a user query[[31]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20summarize_web_content,based%20on%20a%20user%20query)[[32]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=Return%20a%20brief%2C%20clear%20summary,prompt%7D%20%5D%20return%20chat_completion%28messages). It crafts a prompt instructing the model to act as a helpful web summarization assistant, providing relevant info from the page content for the given user query, then calls the chat\_completion helper.

This module centralizes all OpenAI calls so the rest of the code can call these high-level functions. The use of a small embedding model and a moderate temperature for chat is hard-coded here. In practice, these can be made configurable via settings.

* **govsight/llm/llm.py** (top-level in govsight) – Another OpenAI wrapper (legacy/alternate). It defines a single function chat\_completion() similar to the above, using openai and the global settings. In the current codebase, most references have been updated to use govsight.llm.openai\_wrapper instead. The presence of two wrappers is an artifact of refactoring. The openai\_wrapper.py is actively used (by parser and web\_reasoner), whereas llm.py might be a leftover. Future cleanup should remove or unify this to avoid confusion (see **Redundant Components**).
* **govsight/logging\_utils.py** – Sets up consistent logging for GovSight. It provides get\_logger(name) which configures a logger with a standard format and a rotating file handler. This ensures all modules can get a logger that writes to logs/govsight.log (by default) with the same format. Centralizing logging helps avoid the ad-hoc logging in earlier versions of talk.py. The CLI wrapper uses this utility to initialize logging early.
* **govsight/utils/slugify.py** – Contains a slugify(text) function for generating URL/filename-friendly slugs from strings. It likely lowercases the string, normalizes unicode, and replaces spaces or special chars (e.g., “City of New York” → “city-of-new-york”). The memory layer uses this to assign a slug key to each subject stored[[33]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,%3F%2C%20%3F%2C%20%3F%2C). *(Note: There is a duplicate slugify function in govsight/utils.py as well – see* *Redundant Components.)*
* **govsight/web\_reasoner.py** (top-level in package) – Implements the **web fallback querying**. It defines query\_web\_and\_summarize(query) which performs a live web search and then summarizes the results using the LLM:
* It likely uses an external search API (e.g., SerpAPI or similar) to get search results for the query, then fetches the content of the top result(s).
* Instead of returning raw text, it calls openai\_wrapper.summarize\_web\_content(page\_text, query) to condense the information.

This module references the updated OpenAI wrapper: it imports summarize\_web\_content from the wrapper and not the older internal methods. The logic here replaces an earlier approach that was in talk\_old.py/web\_reasoner folder, which had more complex steps for scoring and combining results. Now it’s streamlined to one function call for summarization. The CLI uses this function to provide an answer labeled as “(Web)” when neither local nor vector search yields an answer[[3]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Web%20Query%20Error%5D%20%7Be).

* **govsight/web-reasoner/** – This directory appears to be an *alternate or in-progress web reasoner implementation*. It contains its own query\_web\_and\_summarize.py and web\_reasoner.py:
* query\_web\_and\_summarize.py – Likely similar in purpose to the top-level one, possibly a newer design that uses requests and BeautifulSoup directly and maybe calls the govsight.llm.chat\_completion for summarizing.
* web\_reasoner.py (in this folder) – A more elaborate version of web reasoning with multiple functions: search\_web, scrape\_full\_text, summarize\_web\_results, and its own query\_web\_and\_summarize. It imports govsight.llm.llm.chat\_completion with a note to ensure it uses the new API. This suggests it was part of the refactor to adapt to OpenAI’s new SDK. It likely:
  + Uses a search API to get result URLs (search\_web),
  + Scrapes the text of those pages (scrape\_full\_text),
  + Summarizes or picks relevant parts (summarize\_web\_results),
  + Then returns a synthesized answer (query\_web\_and\_summarize).

**Important:** The package name here has a hyphen (web-reasoner), which is not a valid Python import without special handling. Indeed, the CLI does not use this module (it imports from govsight.web\_reasoner which resolves to the top-level file, not this folder). So this folder’s content is currently **unused**. It likely represents a newer approach that was partially implemented and set aside in favor of the simpler single-function approach. In the future, one of these should be chosen and the duplicate removed. For now, GovSight uses the top-level web\_reasoner.py for web queries.

### Database and Local Search

* **govsight/db/core.py** – Provides simple helpers for directly interacting with the SQLite database outside the Memory class context. It sets the DB\_PATH to the govsight/data/memory.db file and defines:
* search\_local\_facts(query) – Opens the SQLite DB, selects all facts, and performs a brute-force search for the query string in the concatenated “subject attribute value” of each fact. If a match is found, it returns a string like "Subject attribute: value". This is a lightweight search (O(n) scan) and could be optimized, but given the expected scale of facts it’s acceptable. This function is used in the CLI as the first step for answering[[6]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,continue).
* insert\_fact(subject, attribute, value) – Inserts a new fact directly into the facts table. This might be used by ingestion scripts to store facts without going through the Memory class.

There is mention of an ensure\_db\_and\_table() in the CLI, but **Note:** there is no ensure\_db\_and\_table function in this module (the CLI imports it but it doesn’t exist in the final code[[34]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=from%20govsight,parser%20import%20parse_intent_and_facts)). The database initialization is effectively handled by Memory.bootstrap on demand. This appears to be a small oversight in the CLI code – calling ensure\_db\_and\_table() has no effect since it’s not defined. In practice, this doesn’t break anything (the memory DB is created when Memory is instantiated), but it is a known minor issue.

### Vector Store (Pinecone) Integration

* **govsight/vector/search.py** – Handles Pinecone vector operations. It uses the new official Pinecone Python client (pinecone.Pinecone class for Serverless) and OpenAI’s embedding API:
* On import, it loads environment keys and initializes the OpenAI API key and Pinecone client. If required keys are missing, it raises an error early.
* get\_embedding(text, model="text-embedding-3-small") – Generates a 1536-dim embedding for the given text using OpenAI’s embedding endpoint. (It calls openai.Embedding.create, which is updated syntax for the API.)
* search\_pinecone(query, top\_k=5) – Embeds the query using the above, then queries the Pinecone index for the top\_k most similar vectors. It returns the list of matches (each containing an id, score, and metadata if any). The index name is defaulted to "govsight-index" in this module, which should correspond to the Pinecone index that ingestion scripts populate.

This module represents the **semantic memory** of GovSight. The CLI currently imports search\_pinecone but does not actually call it in the main loop (the code path is present but commented or bypassed) – likely due to final integration not being complete in R0. In future versions, after checking local facts, GovSight will use search\_pinecone to find relevant context and either present it or use it to help answer questions.

### Specialized Agents and Ingestion Scripts

The **agents/** and **ingest/** directories contain scripts for populating the GovSight knowledge base and performing domain-specific tasks. These are generally run on-demand (via main.py --agent or manually) rather than as part of the interactive chat loop. Below is an overview:

* **agents/bill\_resolver.py** – Likely takes an ambiguous bill name or number and resolves it to a canonical form or fetches details. For example, if a user mentions “Infrastructure Bill”, this agent might clarify which specific bill is meant. It could use external APIs to search bills and then insert results into the database or Pinecone.
* **agents/query\_pinecone.py** – A simple utility to query the Pinecone index from the command line. It loads the OpenAI and Pinecone keys, creates a Pinecone client, and likely allows the user to input a query to see the raw matches (for debugging vector search). This is useful to inspect what data is stored in Pinecone and how queries are matching.
* **agents/store\_user\_profile.py** – Possibly intended to store information about the user (e.g., name, organization, interests) as facts, so the system can personalize responses or recall preferences. It might parse a provided profile and insert facts into the memory DB.
* **agents/tagger\_agent.py** – Perhaps uses an LLM to auto-tag or categorize some text. For instance, after ingesting a news article, this agent could assign tags (like topics or sentiment) which could then be stored as metadata in Pinecone for better search filtering. Without the code content it’s unclear, but the name suggests an NLP tagging function.
* **ingest/communication\_agent.py** – Could ingest official communications (e.g., press releases, newsletters, emails) into the system. It might connect to an email inbox or scrape a website for communications and then store the content.
* **ingest/congress\_agent.py** – Likely interfaces with a Congressional API (or a dataset) to fetch details on specific bills, amendments, or votes from U.S. Congress. Given the presence of **documentation files** like BillEndpoint.md and AmendmentEndpoint.md in the repo, GovSight likely uses the Congress.gov API or a similar service. This agent might retrieve data for a single bill or Congress member on demand.
* **ingest/congress\_bulk\_ingest.py** – As the name suggests, it probably performs a bulk import of congressional data. For example, it could fetch all bills from a session, all amendments, or all voting records and insert them into Pinecone or the local DB. The code snippet shows use of requests and pinecone.Pinecone and a loop with time delays, implying it calls an external API repeatedly and indexes results.
* **ingest/news\_agent.py** – Ingests news articles, likely focused on local news or specific RSS feeds. The script uses feedparser (to read RSS feeds) and trafilatura (to extract text from HTML). It probably monitors certain news feeds (e.g., local government news, press releases) and for each new entry, scrapes the content, summarizes it (possibly with OpenAI), and upserts it into Pinecone with metadata. The ingestion log files (news\_ingestion\_log.json) likely track which articles have been processed.
* **ingest/press\_agent.py** – Possibly similar to news\_agent but specifically for official press releases. It might scrape press release pages of government officials or agencies and index those. Alternatively, it could monitor feeds or email for press statements.
* **ingest/vote\_history\_agent.py** – Fetches historical voting records. It uses requests, pinecone.Pinecone, and openai.OpenAI. Likely it pulls data like council or congressional vote records, then perhaps summarizes or annotates them via OpenAI before storage. This could populate the knowledge base with facts like “On date X, person Y voted YES on Z”.
* **ingest/vote\_tracker\_agent.py** – Possibly a continuous tracker for new votes. It might run periodically to get the latest voting outcomes (for example, city council meetings or congressional votes) and update the database. It could also compare against a watchlist (if certain votes or topics are being tracked for changes).

Each ingest script typically follows a pattern: load API keys with load\_dotenv(), fetch data from an external source, possibly do some cleaning or use LLM to summarize, and then store results. They often initialize a Pinecone client and OpenAI client at the top, and their end goal is to call Pinecone’s .upsert to add vectors or call the Memory’s insert\_fact. The **logs** directory has web\_fallback.log and govsight.log which likely capture errors or status from these agents (e.g., if a web fallback query failed or an ingestion succeeded).

### Utility Scripts

* **add\_column.py** – (Seen in root listing) Possibly a one-off script to alter the SQLite schema (e.g., to add a new column to facts). Could have been used during development migrations. Not usually needed unless modifying the DB structure.
* **memory\_manager.py** – An earlier comprehensive implementation of memory handling (now largely replaced by the modular govsight.memory components). Its docstring describes many features (session summarization, fact extraction via GPT, watchlist, embedding to Pinecone, etc.) – effectively a superset of what the new modules do. It includes methods to summarize a conversation session with GPT and extract facts from an assistant’s answer. It also dealt with embedding messages and facts into Pinecone (\_embed\_text and \_upsert\_embedding) and managing a watchlist. This file was part of the monolithic R1 design and is now obsolete. Its logic has been split: conversation logging is in memory.py, fact extraction in parser, summarization in openai\_wrapper.summarize\_web\_content, etc. If restarting development, this file can likely be removed after ensuring all needed functionality is ported, because it’s not used by the current CLI.
* **serp\_client.py** – A simple client for search engine results. It contains functions serp\_search(query), parse\_serp\_results(json), fetch\_url\_text(url), and serp\_search\_and\_fetch(query). This suggests it was using an API like SerpAPI or Google Custom Search (since it references parsing results JSON). It’s used by the older web\_reasoner.py (top-level) to perform the actual search and retrieval of web content. In the current refactor, direct use of requests and BeautifulSoup in web\_reasoner.py might replace this. If the new approach is stable, serp\_client.py could be deprecated. However, it’s a useful standalone utility for retrieving search results in JSON form, so it might be kept for specialized queries or further development of the web fallback.
* **smoke\_memory\_test.py** – A quick test script to verify that the memory layer works as expected. It loads settings and instantiates Memory, then performs a series of actions: start a session, log a couple of messages, insert a fact, and attempt to retrieve it. It prints out results to confirm each operation’s success. Developers can run this to ensure the memory DB schema is correct and the core functions aren’t breaking. This is especially useful after a refactor or when setting up on a new machine.
* **Utility scripts in utils/:**
* **embed\_and\_upsert.py**, **pinecone\_init.py**, **pinecone\_inventory.py**, **reset\_pinecone\_index.py** – Various maintenance tools for Pinecone. For example, pinecone\_init.py might create an index or namespace, pinecone\_inventory.py could list all vectors/ids stored, and reset\_pinecone\_index.py would delete or recreate the index (careful with this in production!). embed\_and\_upsert.py probably takes a document or text input, produces an embedding (using OpenAI), and inserts it into the Pinecone index – useful for manual indexing outside the main ingestion flows.
* **scrape\_record\_playwright.py** – A specialized scraper that likely uses Playwright to retrieve content from sites that require dynamic loading or JS (maybe for difficult pages like certain government sites or PDFs). This could be used to fetch things like scanned meeting minutes or complex pages and then feed them into GovSight.
* **Tests (tests/):** There are a few minimal tests:
* test\_env.py simply loads the .env and prints whether API keys are present – a sanity check that your environment is configured.
* test\_congress\_api.py and test\_vote\_api.py likely attempt to hit the Congress API or whichever sources for votes to verify connectivity and perhaps that expected data comes back. These might require valid API keys and internet access, hence more integration tests than unit tests.

## Redundant or Obsolete Components

During the recent refactoring, several files became outdated or duplicated. Here are elements that appear to be no longer needed and could be removed or merged:

* **Legacy Memory Manager and Talk:** memory\_manager.py and talk\_old.py in the root (and their archived versions in archive/) represent the previous monolithic design. The new code splits responsibilities into the modular files described above. For instance, session summarization and watchlist logic from memory\_manager.py are not currently active but can be re-implemented using the new structure. The talk\_old.py (legacy CLI engine) is superseded by the streamlined talk.py – the new version omits the complex web search logic in favor of calling query\_web\_and\_summarize and uses the new parser pipeline[[35]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=def%20main%28%29%3A%20print%28,dim%5DType%20%27exit%27%20to%20quit.%5B%2Fdim%5D%5Cn)[[36]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=,green%5DGovSight%20%28Web%29%3A%5B%2Fgreen%5D%20%7Bsummary). These legacy files in archive/ can be kept for reference but are not executed. Going forward, they should either be removed or clearly marked as deprecated to avoid confusion.
* **Duplicate Config Files:** There is govsight/config.py (a module) and a govsight/config/ package. The package’s \_\_init\_\_.py provides the current Settings class and global instance, while config.py (module) appears to hold an older global config (likely using SimpleNamespace and reading .env). This can lead to ambiguity in imports (govsight.config could load the package or the module). In practice, the code uses govsight.config.settings, which refers to the package’s output. The config.py module is thus obsolete and should be removed to prevent import confusion. All configuration is now handled through the Settings class and environment variables.
* **LLM Wrapper Duplication:** As noted, govsight/llm.py (top-level) duplicates functionality of govsight/llm/openai\_wrapper.py. The project should standardize on one. Given that openai\_wrapper.py also includes the summarization function and is referenced by the parser and web reasoner, it makes sense to keep that and remove llm.py. If any module still imports govsight.llm.llm, it should be changed to use openai\_wrapper. This will avoid confusion and ensure all OpenAI calls go through the same code path.
* **Slugify in Two Places:** The function slugify() exists in both govsight/utils.py and govsight/utils/slugify.py. The memory module imports govsight.utils.slugify (which, due to how Python resolves imports, actually imports the **function** from utils.py because utils is a module in that context, not the package). This is a bit confusing. The dedicated slugify.py should be the single source of truth; the function in utils.py could be removed, and utils/\_\_init\_\_.py could import slugify from the submodule for convenience. This is a minor cleanup, but it will tidy up the utils.
* **Web Reasoner Overlap:** There are two implementations of query\_web\_and\_summarize (top-level vs. web-reasoner/ package). Only one is needed. The top-level one is currently in use and is simpler (one-step summarize), whereas the package version is more granular but unused. To reduce confusion, the project should remove or rename the unused one (perhaps move the advanced logic into the main flow if needed). As is, having a directory name with a hyphen is problematic for imports, so this is clearly an experimental artifact.
* **Ensure DB Initialization:** The reference to ensure\_db\_and\_table() in talk.py is vestigial since no such function exists in the current db.core (it was perhaps part of search\_local.py in older code). This call can be removed or replaced with a proper check. The Memory class ensures the DB schema, so an extra step isn’t needed on chat startup. Removing that call would clean up a harmless but confusing line.
* **summarization\_agent.py:** Listed in the directory but not actually present in code, this might have been intended for a future agent to summarize large texts or documents (maybe as part of ingestion). Since it’s not implemented, any references to it should be pruned to avoid confusion in the module listing.
* **Archive folder:** Contains memory\_manager1.py, talk1.py, etc. These seem to be intermediate backups from the refactor process. They are not imported anywhere. They can be safely archived outside the main codebase or deleted once the team is confident the new implementation covers all needed features.

In summary, a number of files are carry-overs from the refactoring (memory\_manager.py, talk\_old.py, config.py module, llm.py, etc.). Cleaning these up will make the codebase leaner and reduce the chance of someone editing the wrong file by mistake.

## Summary of Recent Updates

The GovSight project underwent significant changes in its latest iteration (R2 refactor). Key updates include:

* **Memory Layer Bug Fixes:** The Memory.search() function was fixed to properly query the facts table and return a result[[17]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20search%28self%2C%20query%3A%20str%29%20,fetchone%28%29%20if%20result). Earlier, there was an issue with a mismatched column (attr\_slug) and usage of an older OpenAI call within memory. Now, Memory.search simply looks for any fact containing the query and returns the latest one, which has proven to be more reliable. Additionally, the SQLite schema was updated to the new format (with a slug field for subjects, etc.), and any legacy data can be migrated via the provided script. These fixes ensure the fact store retrieval is working as intended – the system can recall facts given during the conversation or ingested from data.
* **Working Fact Store & Retrieval:** With the above fixes, the local fact store is now fully functional for recall. The pipeline of parsing user input for facts and storing them is in place – e.g., if the user provides a fact, GovSight will parse it and confirm storage[[26]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=try%3A%20,value)[[10]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,green%5DGovSight%20%28Memory%29%3A%5B%2Fgreen%5D%20%7Banswer). The retrieval planner (retrieval.planner) was introduced to coordinate between structured (DB) search, semantic search, and web search, laying groundwork for a more robust query handling. While the semantic search integration (Pinecone) is not yet active in the chat flow, the code for it (embedding and querying) is written and tested separately.
* **Parser Rework:** The natural language parsing was improved by splitting fact\_parser and parser. Now, simple factual statements or questions are handled by regex in fact\_parser.py for efficiency. This reduces unnecessary API calls for straightforward inputs. The GPT-based parser in parser.py has been refined with clearer prompts and JSON output for intent classification[[23]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=def%20parse_intent_and_facts%28text%3A%20str%29%20,and%20extract%20any%20known%20facts)[[24]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D). Debug printouts (e.g., showing the selected OpenAI model or settings) were added to assist in development[[37]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=from%20govsight,openai_wrapper%20import%20chat_completion%20import%20json)[[20]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D). Overall, parsing is more robust and easier to maintain.
* **Settings and Configuration Refactor:** The configuration approach was cleaned up. Instead of using a global config.py with module-level constants, GovSight now uses a Settings dataclass (in govsight.config) with environment variable overrides. All modules that need config (OpenAI model, DB path, etc.) import this unified settings object. This makes it simple to manage profiles or change defaults. The settings-obsolete.py was a transitional step that combined legacy config with env vars; now the simpler approach is in place. The .env file is loaded at startup of config and vector modules, so keys are picked up seamlessly[[27]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=import%20os%20import%20openai%20from,dotenv%20import%20load_dotenv). This update also addressed compatibility with updated OpenAI SDK calls (ensuring the correct classes/method names are used).
* **OpenAI API Compatibility:** The code was updated to accommodate OpenAI’s newer API versions. For instance, earlier code might have used deprecated endpoints; the refactor uses openai.ChatCompletion.create via the wrapper and openai.Embedding.create for embeddings. Error handling around these calls was also improved (printing out errors but not crashing the program)[[38]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=model%3Dmodel%2C%20messages%3Dmessages%2C%20temperature%3D0,Error%20processing%20request). These changes fix breaking issues that occurred when OpenAI made changes to their Python library.
* **Pinecone Integration Updates:** Pinecone’s API also evolved. The old usage of pinecone.init(...) and global index was replaced with the new Pinecone class and client object. In vector/search.py, we see the new pattern: pc = Pinecone(api\_key=...) and then pc.Index(index\_name) for operations. This was a necessary update, as noted in the development notes, to remove deprecation warnings and use the latest Pinecone serverless client. As a result, vector search code is now up-to-date, though integration into the chat loop is pending.
* **Simplified Web Fallback:** The web querying was consolidated into the query\_web\_and\_summarize function, removing previous multi-step complexity. The current approach fetches the page content and directly summarizes it with a single prompt. This made the web fallback more reliable (fewer moving parts) at the cost of always using the top search result. In future, this can be expanded to consider multiple results or use the more advanced planner, but as of now it fixed cases where the old logic sometimes failed to return an answer at all. Logging of web queries and errors was also added (to web\_fallback.log) for monitoring issues in that part of the pipeline.
* **CLI Enhancements:** The chat CLI experience was improved. Redundant prompts or echo issues were removed (in older versions, the assistant’s answer might have been unnecessarily re-printed or certain prompts were visible; those have been cleaned up). The output now clearly labels the source of the answer: e.g., “GovSight (Local DB): ...”, “GovSight (Memory): ...”, or “GovSight (Web): ...” so the user knows how the question was answered[[39]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=from%20,universal%20slug%20generator)[[40]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn). The welcome message and exit handling were also standardized in the CLI.

In summary, these updates collectively make GovSight more stable and maintainable: the fact memory works reliably, the parser is smarter, the configuration is unified, and external service calls (OpenAI/Pinecone) are using current best practices.

## Known Issues and Limitations

Despite the progress, there are a few known issues and areas for improvement:

* **Web Fallback Specificity:** The current web fallback sometimes returns very general summaries. Because it takes the first search result and summarizes the page, if the query is niche or the top result isn’t directly answering the question, the summary might not contain the specific detail asked. For example, asking a very specific data point might yield a summary of an entire article that doesn’t highlight the answer. To address this, the web reasoner may need to search multiple results or explicitly extract the answer (perhaps via follow-up questions to the LLM). For now, users might get an answer like “According to the page, various information about X is discussed…” rather than the exact figure they wanted. This is an area to refine (potentially by leveraging the more advanced web-reasoner module or using a Q&A chain on the retrieved content).
* **Incomplete Pinecone Usage:** While the code for Pinecone vector search is present and functioning, the chat logic is not yet utilizing it in step 2 as intended. In talk.py, after checking the local DB, the code actually calls memory.search() (which hits the SQLite again) instead of vector.search\_pinecone()[[41]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=continue). This is likely a placeholder to avoid complexity while Pinecone data is sparse. It means semantic search over ingested documents isn’t happening during chat. A developer picking up the project should integrate search\_pinecone() into the flow and combine its results with the answer (or use them to formulate a better answer). As ingestion agents populate Pinecone, enabling this will greatly improve answer recall for questions that the local DB can’t answer but the indexed documents can.
* **Unused Imports & Code Artifacts:** Due to the refactoring, some files import things that are no longer used. For example, talk.py imports search\_pinecone but doesn’t use it, and talk\_old.py (if one were to run it) imports a config that isn’t defined in the new structure properly. These unused imports don’t cause runtime errors in the main execution path, but they should be cleaned to avoid confusion. Similarly, some debug print statements (e.g., in parser.py printing the settings content[[42]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=from%20typing%20import%20Optional%2C%20Tuple%2C,openai_wrapper%20import%20chat_completion%20import%20json) or in openai\_wrapper printing error messages) could be toggled or removed in production or replaced with proper logging.
* **Ensure DB Initialization Call:** As mentioned, ensure\_db\_and\_table() is called but not implemented, which could mislead someone reading the code. It doesn’t currently break functionality, but it is an inconsistency to be resolved. A related issue is that the memory DB path is duplicated in db/core.py and in settings.db\_path. Currently they point to the same file, but this dual definition could diverge. Ideally db/core.py should use settings.db\_path instead of constructing its own path string, to avoid any mismatch if the path is configured differently.
* **Concurrency and File Locking:** The SQLite usage in Memory is protected by a threading lock for that process, but if multiple processes or threads try to access the DB (for example, running an ingest script while the chat is active), there could be contention. SQLite can handle some concurrency but it’s limited. This hasn’t surfaced as a major issue yet, but it’s something to keep in mind if expanding to multi-threaded contexts or running a web server plus background ingestions. A possible improvement is to use a connection per thread or an async queue for DB writes.
* **Watchlist and Follow-ups:** The concept of a “watchlist” (tracking when new information on a topic comes up) is only partially implemented in design notes. Currently, if a user asks GovSight to “monitor” something, there is no functional mechanism to do so. The groundwork (like an empty watchlist.py and some prompts in memory\_manager.py) is there, but this feature is not active. It’s a known missing piece; any mention of it in the UI would be misleading. Implementing the watchlist would involve storing the topics to watch and periodically checking for updates (perhaps via the ingestion agents or scheduled web queries).
* **Model and Parameter Configurability:** Right now, model names and parameters (like settings.openai\_model or Pinecone index name) are hard-coded defaults or lightly configurable via env. For instance, the parser uses settings.openai\_model, but settings doesn’t actually have openai\_model defined in the current Settings class (it was present in the obsolete settings). This means it might be defaulting to some empty value or using a fallback in code (indeed, parser.py prints settings.openai\_model[[20]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D) which might be None). In practice, the wrapper defaulted to "gpt-4". This is a minor bug – either the Settings class should include an openai\_model field, or the code should be updated to use a constant or the environment variable for model (e.g., GOVSIGHT\_MODEL). Similarly, the Pinecone index name is defined in multiple places (INDEX\_NAME in vector/search vs. default in obsolete settings). Consolidating these configurations is needed to avoid confusion.
* **Documentation Mismatch:** The provided Markdown and PDF documentation in docs/ might not reflect the latest code behavior. For example, BillEndpoint.md presumably describes an API usage for bills, but if the code changed how it fetches bills, the docs may be outdated. Ensuring the documentation (especially any developer hand-off guides or design status notes) matches the code is an ongoing task. The presence of development notes files (like “Working List of Improvements Needed.docx”) indicates awareness of issues; those should be checked off against this Known Issues list to verify what has been fixed and what remains.
* **Testing Gaps:** Tests are very sparse. The few provided tests do not cover the complex functionality (no tests for the parser, memory layer, or agents). This means regressions can happen unnoticed. Increasing test coverage, even with simple scenarios (e.g., fact insertion and retrieval, parser extracting a known format, etc.), would greatly help reliability.

## Next Steps and Roadmap

Moving forward, here are the recommended next development tasks for GovSight, in rough priority order:

* **Integrate Pinecone Search into QA Loop:** Enable semantic vector retrieval in the chat pipeline. After local DB search, use search\_pinecone() to fetch relevant context documents. The assistant can then either present that info or use it to craft a better answer. *Plan:* retrieve top 1-3 Pinecone matches, and either automatically return a summary or feed those into a prompt for the LLM to generate a final answer. This will allow GovSight to answer questions from ingested data (e.g., news articles or bills) that are not explicitly stored as facts.
* **Improve Web Query Strategy:** Enhance the query\_web\_and\_summarize process to ensure specific answers. Options include: searching multiple results and combining their info, using a targeted question-answering prompt (like “Based on the above content, the answer to the question is \_\_\_”), or using a tool like the more advanced web reasoner logic with scoring. Implementing the govsight/web-reasoner module fully or integrating an external QA model on retrieved text will make web fallback more precise. Logging should be monitored to identify cases where the web answer was insufficient.
* **Watchlist Feature:** Complete the watchlist functionality. Define a way for users to add topics to a watchlist (perhaps a command in the CLI like !watch Grandview mayor). Store these in a new watchlist table via the watchlist.py module. Implement a background process or extend ingestion agents to periodically check for new info on these topics (e.g., search news or web for the topic). When new information is found, GovSight could proactively alert the user in the next session or log it. This would fulfill the “track/monitor intent” capability hinted in the design.
* **User Profile and Personalization:** Flesh out the store\_user\_profile.py agent or similar functionality. The system could store facts about the user (e.g., user’s city, interests, roles) so that queries can be answered in a more personalized way (for example, preferring local city info if the profile says the user works in City X). Implement a structured way to save user data (possibly as special facts with a reserved subject like “<Username>”) and retrieve those when needed. Also consider privacy and security of storing personal data.
* **Refine Fact Storage and Confidence:** Introduce a mechanism to handle confidence levels for stored facts. Currently, any parsed fact from the user or web is stored without validation. In the future, adding a confidence score or source attribution (the source field is already in the schema) and using that to decide whether to auto-use the fact in answers would be wise. For instance, facts gleaned from the web with low confidence might not be automatically trusted. The groundwork is there (the source and confidence fields), so utilizing them would improve reliability.
* **Remove Redundant Code:** Clean up the obsolete files as identified. Remove duplicate functions (slugify, extra config, old CLI, old memory manager) to reduce confusion. This also involves updating references: for example, ensure all code uses govsight.config.settings and not any legacy config, and use one OpenAI interface throughout. A thorough sweep for TODOs or # type: ignore flags (like in chat\_cli importing talk) can also reveal code that can be refactored or removed.
* **Expand Test Coverage:** Develop unit tests for critical components:
* Memory store: test inserting and retrieving facts, and that sessions logging works.
* Parser: feed known inputs to parse\_fact\_from\_text and parse\_intent\_and\_facts and assert the correct output (this may require mocking OpenAI API – consider dependency injection or a debug mode for parser).
* Vector search: if a Pinecone mock or test environment is available, test that embedding and query return expected structure.
* Web reasoner: could use a stubbed requests.get to test that summary is generated as expected.

Having these tests will catch issues early as new changes are made.

* **Documentation and Examples:** Update the README (this document) and other markdown guides to reflect the current state. Add usage examples in the README for common tasks (e.g., “How to add a fact”, “How to ask a question about a new bill”). Possibly provide a small sample dataset or ingestion example in the repository so new developers can see how data flows from ingestion to answer. Also, maintain the developer guide (the docx notes) in markdown so it’s version-controlled and easier to update.
* **Multi-turn Conversation Improvements:** Currently, conversation history is logged but not actively used in generating answers (the system doesn’t feed previous turns into the LLM context, except what’s retrieved as facts). Consider utilizing recent conversation context to clarify pronouns or follow-up questions. For example, if the user asks “What about its population?” after asking about a city, GovSight should use the last subject (“the city”) from memory. This could be implemented by caching the last query’s subject in the session, or by pulling the last few messages via memory.get\_messages() and including them in the LLM prompt. This will make the chatbot feel more coherent over multiple turns.
* **Feature: Summarization Agent:** Implement the missing summarization\_agent (if needed). This could be an agent that takes a long text (like a lengthy bill or report) and produces a concise summary stored in the knowledge base. It would use the OpenAI API to generate summaries chunk by chunk, then save those as either facts or as a “summary” field in Pinecone metadata. Such summaries can make answering questions about long documents more efficient.
* **Error Handling and Resilience:** Make the system more resilient to external failures. For instance, if OpenAI API quota is exhausted or Pinecone is unreachable, the system should handle it gracefully (maybe falling back to web only, or informing the user of a temporary issue). Currently, exceptions are caught in the CLI loop and logged[[43][44]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=except%20Exception%20as%20e%3A%20logging.exception%28,red%5D%E2%9D%8C%20Error%3A%5B%2Fred%5D%20%7Be), but more specific messaging might help. Similarly, ensure that ingestion scripts can continue past network errors or rate limits by catching exceptions and waiting/retrying appropriately.

By addressing these tasks, GovSight will become a more complete and robust platform. The refactoring has set a solid foundation – a modular architecture with clear separation of concerns. The next steps are to fully activate each part of that architecture (especially the vector and retrieval pipeline) and to polish the user experience and reliability. With these improvements, GovSight will be well on its way to serving as a powerful assistant for municipal data and beyond.

[[1]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=Features%3A%20,NEW%29) [[5]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn) [[11]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20start_session%28self%2C%20session_id%3A%20str%29%20,%28session_id%2C%20int%28time.time) [[12]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=INSERT%20INTO%20messages%20,time) [[13]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,%3F%2C%20%3F%2C%20%3F%2C) [[14]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,attribute%2C%20value%2C%20source%2C%20confidence%2C%20inserted_at) [[15]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20insert_fact_triples%28self%2C%20triples%3A%20List%5Bdict%5D%29%20,) [[16]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20search%28self%2C%20query%3A%20str%29%20,result%20%3D%20cursor.fetchone) [[17]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20search%28self%2C%20query%3A%20str%29%20,fetchone%28%29%20if%20result) [[18]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn) [[33]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=def%20store_fact,%3F%2C%20%3F%2C%20%3F%2C) [[39]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=from%20,universal%20slug%20generator) [[40]](file://file-MZxrbavjHXPqCJdeUZ12ix#:~:text=self,conn) memory.py

<file://file-MZxrbavjHXPqCJdeUZ12ix>

[[2]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Memory%20Error%5D%20%7Be) [[3]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,Web%20Query%20Error%5D%20%7Be) [[6]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=,continue) [[7]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=def%20main) [[34]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=from%20govsight,parser%20import%20parse_intent_and_facts) [[41]](file://file-NKzGmVXe8t8QeevyZKH2MN#:~:text=continue) talk.py

<file://file-NKzGmVXe8t8QeevyZKH2MN>

[[4]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=import%20os%20import%20openai%20from,dotenv%20import%20load_dotenv) [[27]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=import%20os%20import%20openai%20from,dotenv%20import%20load_dotenv) [[28]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20get_embedding,%F0%9F%94%B4%20Embedding%20error%3A%20%7Be) [[29]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20chat_completion,0%5D.message.content.strip) [[30]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=try%3A%20response%20%3D%20openai,%F0%9F%94%B4%20Chat%20completion%20error%3A%20%7Be) [[31]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=def%20summarize_web_content,based%20on%20a%20user%20query) [[32]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=Return%20a%20brief%2C%20clear%20summary,prompt%7D%20%5D%20return%20chat_completion%28messages) [[38]](file://file-CfgQYtawQZ1XcFXVmGgcuw#:~:text=model%3Dmodel%2C%20messages%3Dmessages%2C%20temperature%3D0,Error%20processing%20request) openai\_wrapper.py

<file://file-CfgQYtawQZ1XcFXVmGgcuw>

[[8]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,db_path%2C%20subject%2C%20attribute%29%20if%20answer) [[9]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=,green%5DGovSight%20%28Web%29%3A%5B%2Fgreen%5D%20%7Bsummary) [[10]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=if%20subject%20and%20attribute%20and,green%5DGovSight%20%28Memory%29%3A%5B%2Fgreen%5D%20%7Banswer) [[26]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=try%3A%20,value) [[35]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=def%20main%28%29%3A%20print%28,dim%5DType%20%27exit%27%20to%20quit.%5B%2Fdim%5D%5Cn) [[36]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=,green%5DGovSight%20%28Web%29%3A%5B%2Fgreen%5D%20%7Bsummary) [[43]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=except%20Exception%20as%20e%3A%20logging.exception%28,red%5D%E2%9D%8C%20Error%3A%5B%2Fred%5D%20%7Be) [[44]](file://file-9GthUXkdP3gRs7jedTcsk6#:~:text=except%20Exception%20as%20e%3A%20logging.exception%28,red%5D%E2%9D%8C%20Error%3A%5B%2Fred%5D%20%7Be) talk.py

<file://file-9GthUXkdP3gRs7jedTcsk6>

[[19]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=def%20parse_fact_from_text%28text%3A%20str%29%20,and%20value%20from%20a%20sentence) [[20]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D) [[21]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=print%28f) [[22]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=lines%20%3D%20response.strip%28%29.split%28,value%20except%20Exception%3A%20return%20None) [[23]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=def%20parse_intent_and_facts%28text%3A%20str%29%20,and%20extract%20any%20known%20facts) [[24]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=messages%20%3D%20%5B%20%7B,prompt%7D) [[25]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=response%20%3D%20chat_completion%28messages%2C%20model%3Dsettings) [[37]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=from%20govsight,openai_wrapper%20import%20chat_completion%20import%20json) [[42]](file://file-PgU5RJRJQ9Z9CCSQnazUjw#:~:text=from%20typing%20import%20Optional%2C%20Tuple%2C,openai_wrapper%20import%20chat_completion%20import%20json) parser.py

<file://file-PgU5RJRJQ9Z9CCSQnazUjw>