CongressGPT

Executive Summary

We introduce CongressGPT, a web application that combines OpenAI's GPT models with a search engine to offer an intuitive, conversational interface for searching legislative documents. This tool leverages the most powerful capability of large language models (LLMs), the ability to answer complex questions and follow-ups in natural language, and extends it by allowing the LLM to autonomously retrieve and summarize text and metadata from a database of Congressional documents. This extends the knowledge available to GPT base models, and creates an additional layer of verifiability by allowing users to check claims made by GPT against the information from the original documents.

Table of Contents

1) Introduction

Overview

Congressional analysts face the challenging task of retrieving and synthesizing information from the large corpus of bills, resolutions, and other official documents produced by Congress over time. Currently, many analysts utilize Congress.gov, the official website maintained by the Library of Congress, which provides a comprehensive resource for performing keyword searches of Congressional documents.

There are several areas where a new product might improve upon Congress.gov. First, Congress.gov appears to rely on exact keywords matches for ranking results, which is the default configuration for the Apache Solr platform that Congress.gov utilizes. This is limiting, as documents that are conceptually related to a search query might be left out of the results if they do not contain the correct keywords. Second, Congress.gov users are limited by their knowledge of the topic they are searching for, and analysts who are researching a less-familiar policy subtopic might benefit from additional context and guidance.

In spring 2023, another team of students from Heinz College sought to address these shortcomings, conducting interviews with stakeholders at the Bipartisan Policy Center (BPC) and eventually developing a desktop application. The application provided an interface for searching for bills from the 117th Congress. Once these bills are retrieved, users can generate a summary using one of two LLMs.

The previous application lays the groundwork for improving the way analysts search Congressional documents; however, there are a few areas where it can be improved. It only provides access to a small corpus, being limited to documents from the 117th Congress. It also struggles to generate high quality summaries for long bills, which require significant processing time to complete, and are sometimes disjoined, as the bills are fed to the LLM in pieces. Finally, the application from the previous team is launched locally from the command line, making it challenging to utilize for non-technical users.

To address these issues, we introduce CongressGPT, a web application that provides a more intuitive interface for searching Congressional documents. CongressGPT extends OpenAI's GPT series of language models by connecting them to a search engine. Whenever the user has a specific question about Congressional documents to which GPT does not know the answer, it will autonomously query the search engine, retrieving relevant documents, passages, and metadata. GPT then summarizes and interprets this information for the user, and performs follow-up searches as needed.

This has the dual benefits of extending GPT's knowledge and capacity to answer user questions, while also providing an additional layer of verifiability, allowing the user to check claims made by GPT against the original text and metadata. The result is a highly intuitive method for searching and understanding the documents produced by Congress.

The Team

The capstone team is composed of students from the Heinz College of Public Policy and Information Systems at Carnegie Mellon University. The team members for the project, and their programs, are listed below:

- Kathy Chiang, MISM
- Jack Vandeleuv, MSPPM-DA
- Lily Chen, MISM
- Sylvia Zhang, MISM
- Leigh Li, MISM-BIDA

The Client

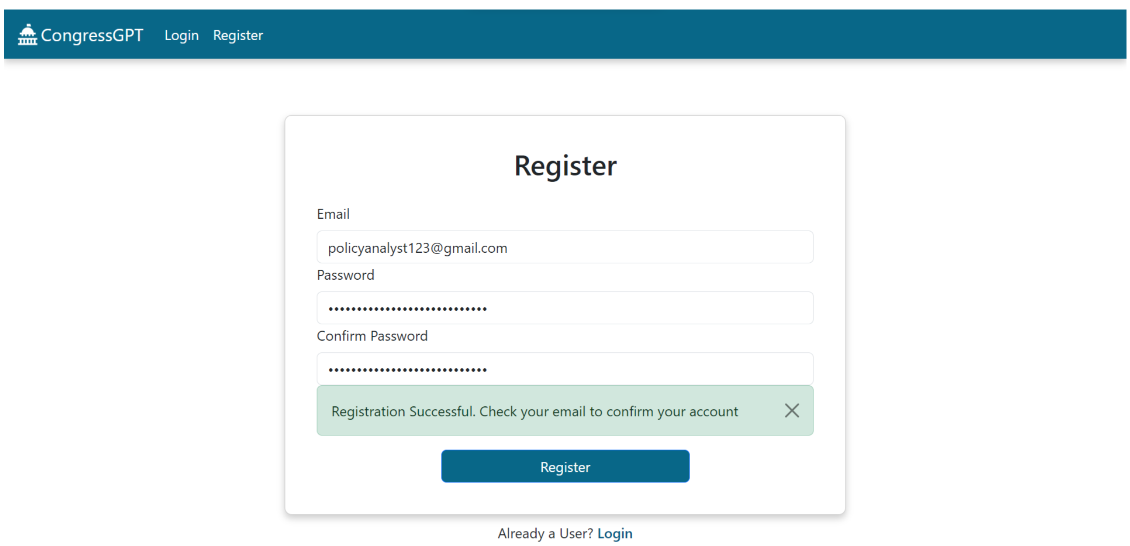
Our client, the Bipartisan Policy Center, is a think tank based in Washington, DC which is dedicated to fostering cooperation and actionable solutions across party lines. With a focus on a broad spectrum of policy areas ranging from healthcare and security to technology and energy, BPC is committed to merging the best ideas from both sides of the aisle to advance common goals. BPC's team of policy analysts is tasked with a diverse set of activities, which include coordinating discussions with key stakeholders, scrutinizing legislative texts, and producing insightful content such as articles, informational briefs, and in-depth policy analyses.

To execute these functions, they rely on a host of informational resources, notably Congress.gov, to stay informed about the latest legislative developments. Quick and easy access to accurate legislative information, which we aim to augment with this project, is therefore crucial for BPC analysts to perform their roles effectively.

2) User Journey

Login & Registration

A user visiting CongressGPT for the first time is required to sign up for an account in order to use the service.[1] On the registration page, users are prompted to enter their email address and create a password, which they must confirm by entering it a second time to ensure accuracy. Upon successful registration, a notification appears, advising the user to check their email to confirm their account, verifying the authenticity of the registered email and activating the account for use.

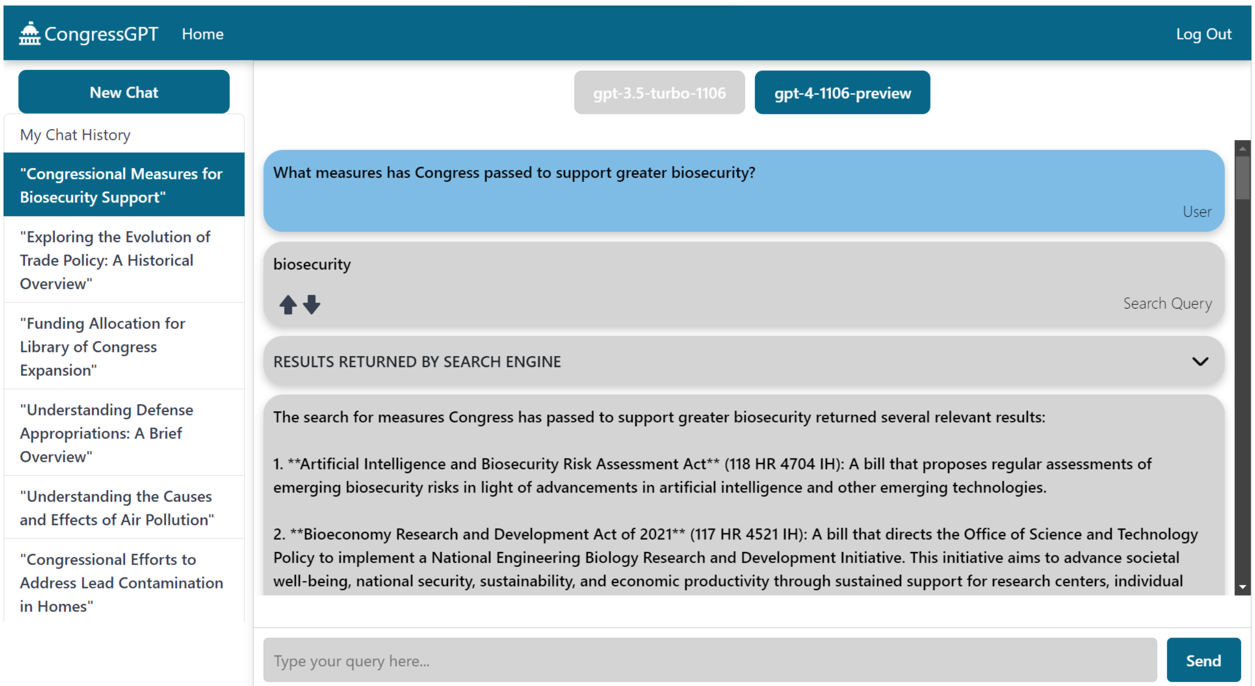


Chat and Search with GPT

CongressGPT's main functionality is its chat page, which is designed to resemble existing LLM-based chat applications. Before typing their message, users may wish to

[1] This is because CongressGPT supports a chat history feature which allows users to revisit and re-engage with old chats. This feature requires authentication in order to ensure that each user can only access their own chats.

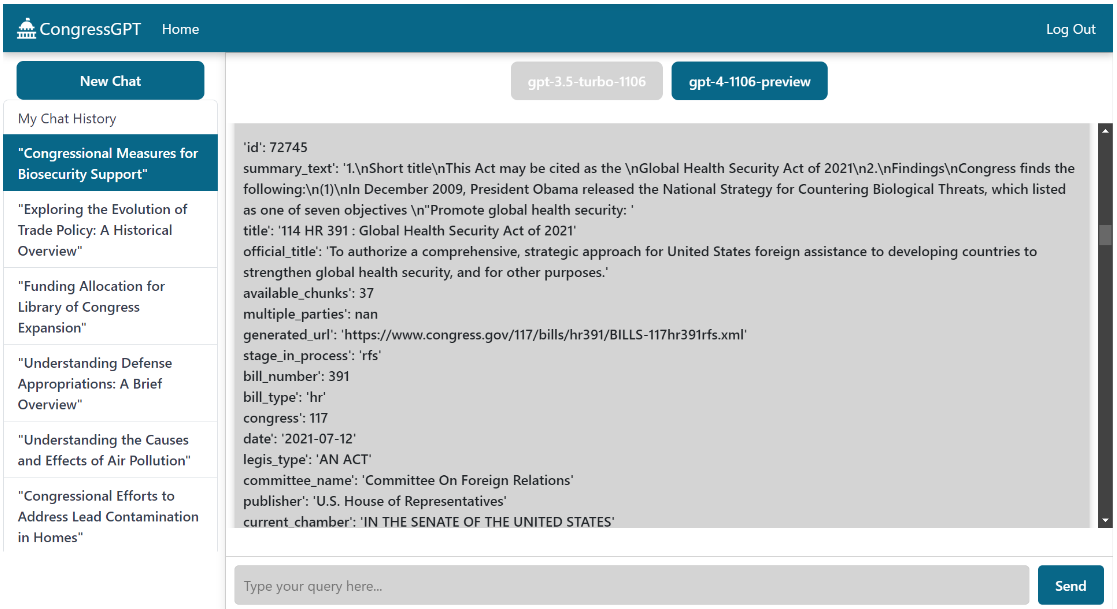
choose between the two available language models, GPT-3.5 and GPT-4.[2] In general, GPT-3.5 is quicker to respond, while GPT-4 produces higher quality responses and makes better use of the provided search functionality.[3] After users select their preferred language model, they can ask any question.



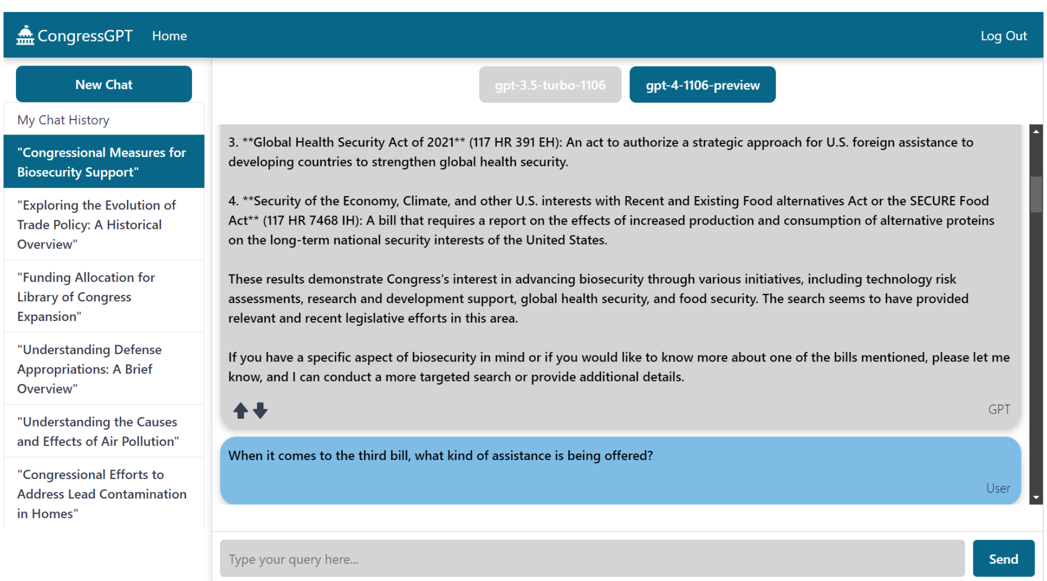
In this case, GPT immediately recognizes that the user's input requires a search, which it completes before giving its response. GPT selects "biosecurity" as the query, which is labeled as "Search Query" and displayed to the user. The top five search results are returned into the chat. They are hidden by default, but the user can click to expand them and see the raw text and metadata being read by GPT. The metadata includes fields like Congress session, date, chamber, and number of passages available to search in the full text.

[2] Whichever language model is selected when the user hits "Send" will answer the prompt. Users may switch models over the course of a chat if needed.

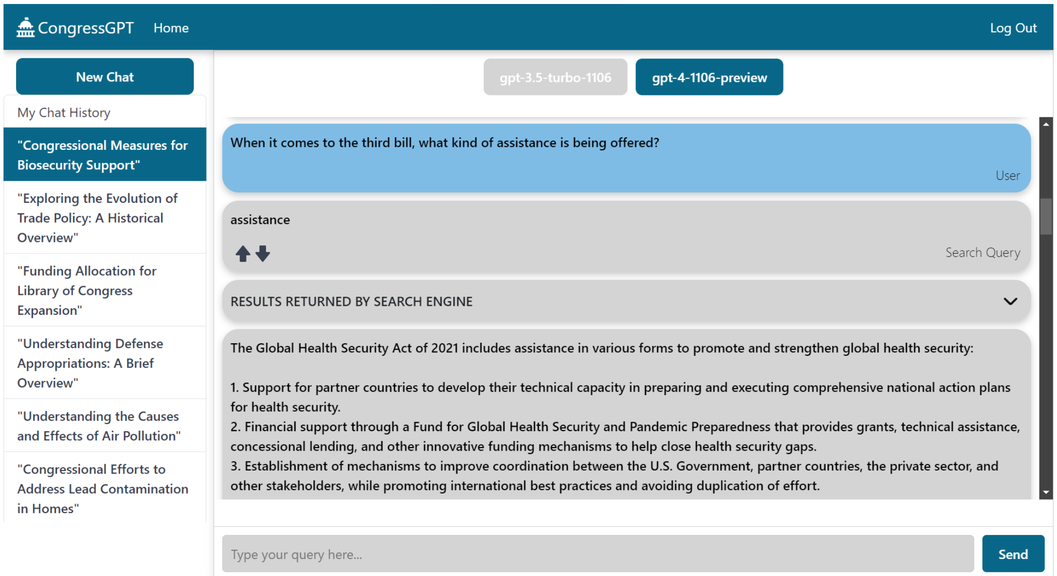
[3] GPT-4 also comes at a significantly higher cost-per-token generated.



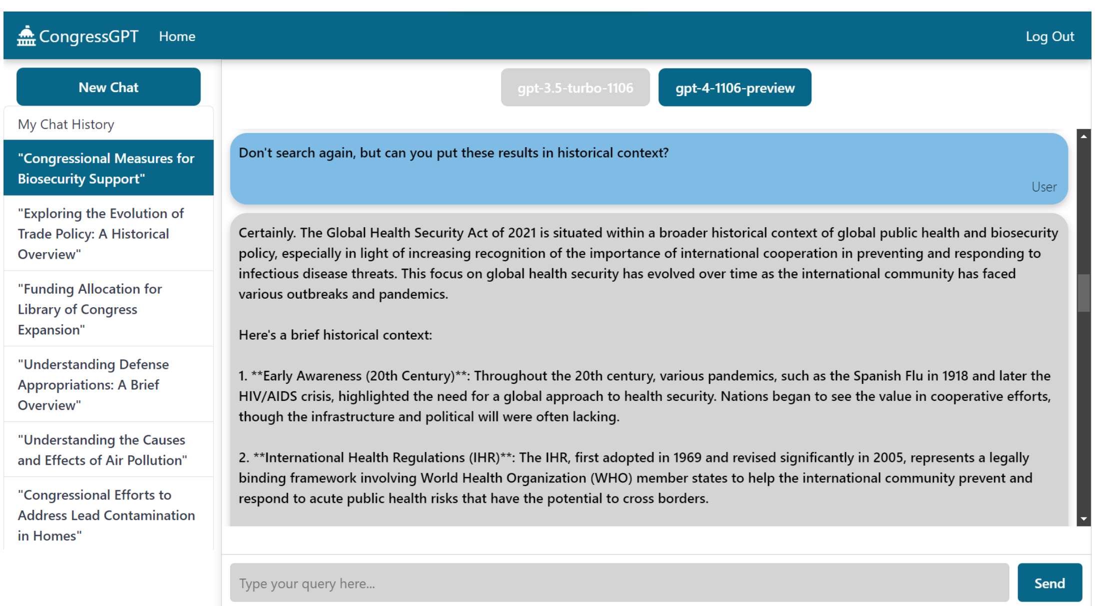
The search results for this query covered a wide range of subtopics related to biosecurity, including international cooperation, biological engineering research, and AI-related biosecurity threats. For each result, GPT automatically provides a summary and writes one or two sentences to describe them. We will ask GPT to tell us more about the third bill, which deals with global health security. The bill mentioned providing assistance to developing countries to improve health security, so we will ask for more information about that aspect of the bill.



Upon receiving our follow-up question, GPT performs another search, this time within the specified bill. Because GPT flagged the request internally, the search engine returned the passages most related to the new search query, in this case “assistance,” from the bill GPT requested, in this case the Global Health Security Act of 2021.

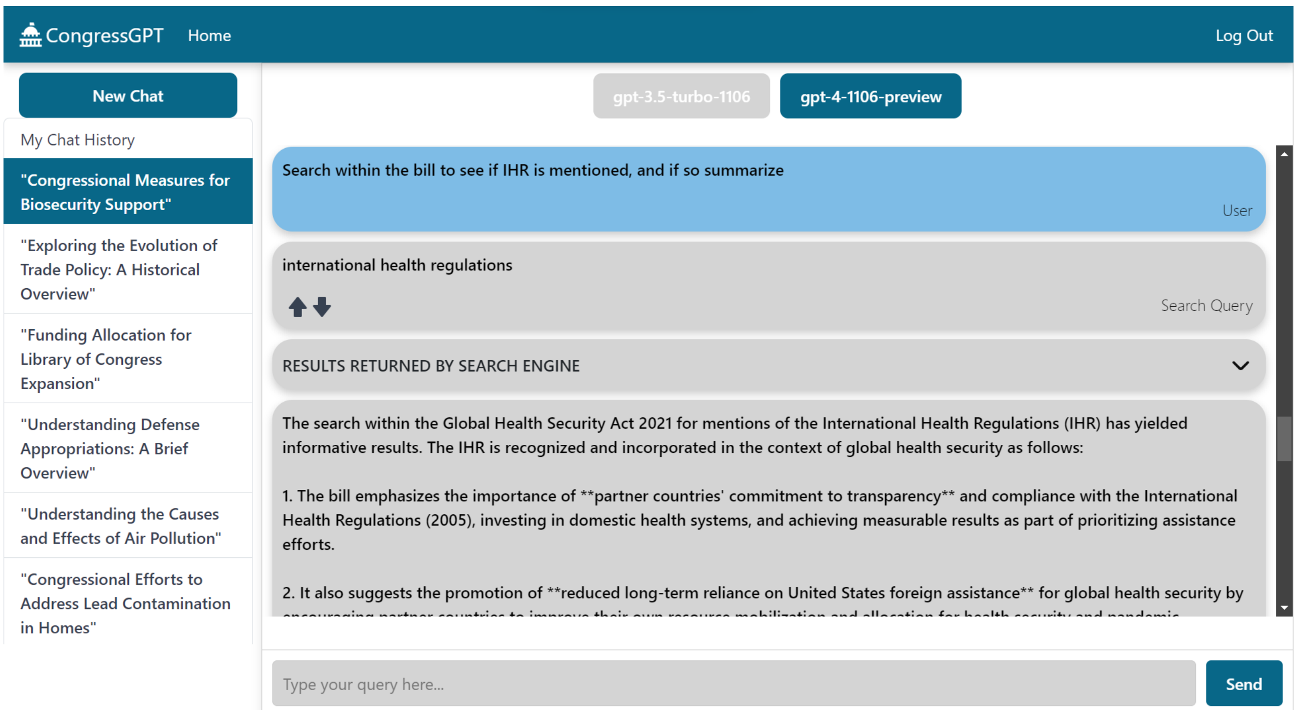


GPT responds again, this time summarizing the text chunks returned by the search engine. It mentions correctly that the bill supports the development of national action plans, a Fund for Global Health Security and Pandemic Preparedness, and mechanisms for intergovernmental cooperation.

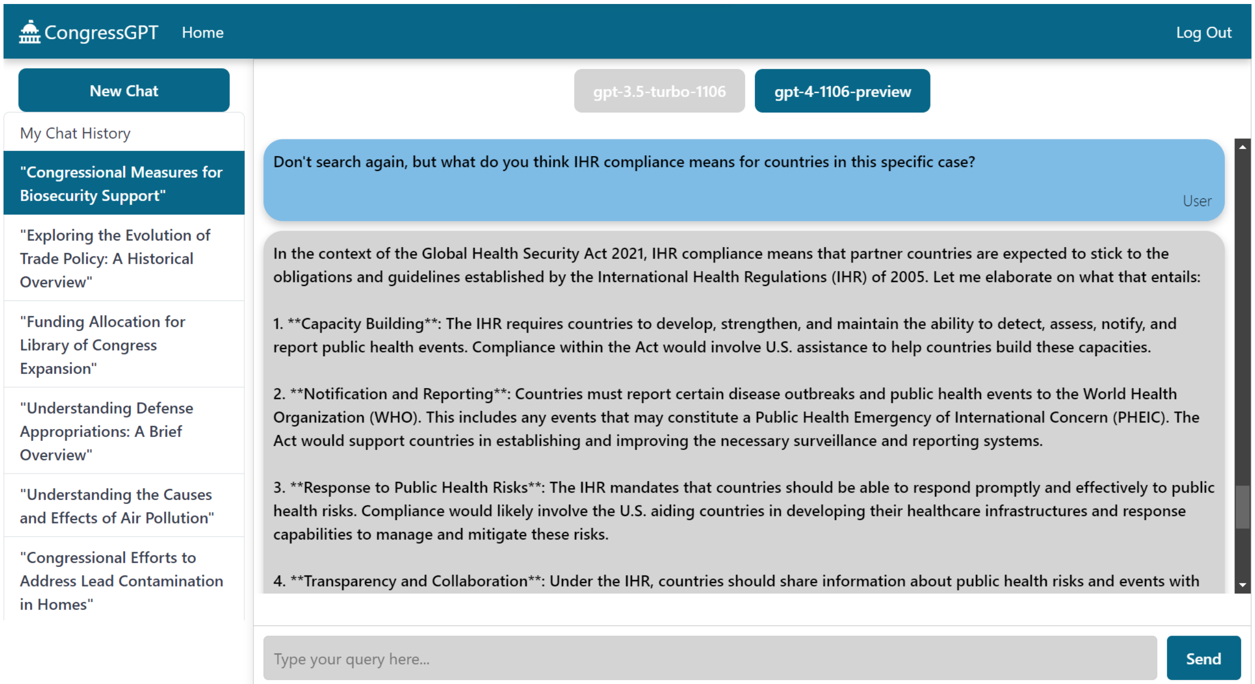


At this point, the user might want GPT to add additional context, rather than just summarizing the contents of the bill. Because GPT is relatively eager to perform new searches, which can lead to it getting off track if this is not the desired outcome, we explicitly ask it not to perform another search.

GPT follows our instructions, giving a historical overview, during which it mentions the establishment of International Health Regulations (IHRs), a legally binding framework for WHO member states.



Because we are now interested in IHRs, we ask GPT to make one more full text search to learn how they impact the U.S.'s approach as outlined by the bill.



The results are informative. GPT identifies multiple areas where IHR compliance is relevant to the bill in question, including that the bill is designed to help developing countries identify and report ongoing health crises to the international community, which is a key requirement of the IHR.

Overall, this example chat is a good representation of a typical user journey through CongressGPT. The platform allows for an ongoing dialogue between the user and AI assistant. The user can ask GPT to search the whole corpus of Congressional documents, or retrieve detailed information about just one document. The user can also instruct GPT to analyze the results from a particular angle, or add context using its own understanding. Because each chat is stored, the user can navigate back to a prior chat at any time and pick up where they left off.

3) Implementation Details

CongressGPT is a full-stack application built using React.js, Django, Supabase, and the OpenAI API. Basic implementation details of the application will be outlined in this section, and further expanded upon in the technical appendix.

Frontend

CongressGPT utilizes React.js as a frontend framework. React supplements standard JavaScript, making it easy to dynamically rerender individual parts of a web page if something changes, without requiring a full page reload. This makes it highly responsive and well-suited to a chat application.

The React frontend handles user authentication using the Supabase client library. Supabase is a Backend-as-a-Service, which provides a fully-managed PostgreSQL database in the cloud. Crucially, Supabase implements its own authentication system on top of PostgreSQL's row-level security features, which allow the database to securely limit access to individual database rows based on user credentials. Using Supabase this way allows us to avoid directly handling password information or creating our own authentication procedure, which might expose us to security flaws.

Once users enter their username and password, the frontend obtains an encrypted JSON Web Token using the Supabase client library, which can then be used as a credential for requesting user-specific information from Supabase.

In most cases, the frontend retrieves data from the Django web server, which itself communicates with Supabase; however, to increase efficiency, chat history information is retrieved by the frontend directly from Supabase. This reduces network latency and simplifies the code for switching between chats.

Backend

The backend is composed of a Django web server that communicates between the different internal and external services required to make the application function. Most interactions between the backend and frontend are structured like a dialogue. The frontend sends a user's chat message to the backend, along with the associated chat ID and other parameters if necessary, and the backend returns a response from GPT, and other information like search results, if requested.

As part of processing a request for a new message, the server retrieves any prior messages in the same chat from Supabase and sends them together with the new user message to the OpenAI API. Because Supabase only allows each user to access their own data, a JWT token must be sent by the frontend to the backend to allow for database requests. OpenAI does not keep a history of previous messages, so each new request must contain a history of previous messages.[4]

To make the response more immediate for the user, our application breaks some requests into multiple pieces. In particular, anytime GPT requests a search, the backend processing is split into two parts. First, the initial message where GPT requests a search is returned to the frontend so it can be displayed to the user right away. Then, the frontend makes a second request to the backend, which engages GPT again to summarize and explain the search results for the user.[5]

If GPT requests a search, the server queries the search engine. The search results are appended to the chat history, and another request is made to the OpenAI API in order to generate GPT's reaction to the search results.

---

[4] In the current implementation at most 15 prior messages are given to GPT in order to avoid increasing financial and computational cost too greatly.

[5] A future team may wish to further increase the number of relays between the frontend and backend to allow, for example, search results to be returned immediately without waiting for GPT to respond to the results. This will impose a small network overhead to overall processing time, but will give the user much more immediate feedback when making searches. It may also be desirable to stream in the response from OpenAI one token at a time using server-sent events.

After the server is finished processing its response, it begins asynchronously posting any user messages, GPT messages, or search engine responses to Supabase for long term storage. For responses involving a search query, this process may not be complete by the time the frontend makes the request, so there is a procedure in place to retry multiple times before concluding that the search request was not formatted correctly.

Search Engine

CongressGPT implements two search engines, one for retrieving summaries across all bills, and one for retrieving relevant paragraphs within one bill. The implementation of these two search engines is similar in principle. Both utilize a two-stage retrieval system, where the first stage uses a computationally-efficient algorithm in order to quickly scan through all potential matches and filter down to the top results, and the second stage uses a slower, higher quality algorithm to determine the final ranking of search results.

For the first stage, the summaries search engine utilizes SQLite3's implementation of BM25, a popular bag-of-words search algorithm.[6] BM25 relies on a few key statistics to assign a score to a given query/document pair. For each word in the search query, BM25 calculates the number of times that word appears in each document. It also makes use of inverse-document frequency, which gives more weight to words that are rare across the whole corpus, as well as document-length normalization, which prevents longer documents from scoring highly only because they are long.

This first stage produces an initial set of matching documents. These top documents are then reordered in the second stage using LEGAL-BERT, which is a variant of the Bidirectional Encoder Representations from Transformers (BERT) model developed by Google in 2018.[7] BERT processes text by representing each word (or sometimes fragments of words) as a vector in high-dimensional space. This method excels at

[6] For more details, see the SQLite3 documentation: https://www.sqlite.org/fts5.html

[7] "LEGAL-BERT: The Muppets straight out of Law School," Chalkidis et al. https://aclanthology.org/2020.findings-emnlp.261.pdf

capturing the relationships between words and the different meanings of a word in different contexts. This capability can then be leveraged for search. By using BERT to embed the search query and text from each document in the same vector space, we can use a distance metric like inner product to find which documents are most similar to the query according to BERT's understanding.

LEGAL-BERT further refines the BERT base model's understanding by fine-tuning the model on legal and legislative documents. In particular, LEGAL-BERT was trained on E.U. and U.K. legislative and judicial documents, and on U.S. court and contract documents. This enables it to achieve better performance on tasks like text classification (where BERT attempts to predict what outcome is associated with a given legal document), and imbues the BERT base model with greater understanding of legal language.

For the summary search engine, BERT embeddings are precomputed ahead of time, as the embedding process is computationally intensive and cannot be completed quickly without significant hardware resources. After BM25 retrieves the initial set of bills, the search query is embedded by BERT, and the stored embeddings for the selected subset of bills are compared against the query embedding using a simple inner product calculation.

The search engine for retrieving matching passages within a bill utilizes a similar strategy. However, unlike the summary search engine, embeddings for each passage are computed during the retrieval process. This saves on disk space, avoids a lengthy precomputation stage, and allows for flexibility in terms of changing passage size, or trying different passage formation strategies. It is also reasonably quick, as only the top passages are reranked. For the initial retriever, this second search engine uses cosine similarity, an efficient and easy-to-implement retrieval model that treats each unique word as a dimension in a high-dimensional vector space, and the frequency of each word in a document as the distance to move along that dimension.

Data Scraping

The current data pipeline for CongressGPT relies on GovInfo.gov, a resource provided by the U.S. Government Publishing Office. This is a transition from the previous team's project, which utilized the Congress.gov API. We made the switch primarily to speed up the data scraping process. GovInfo allows for bulk data downloads, which unlike Congress.gov's API does not impose rate limits. This made it simple to quickly download all bills and bill summaries from the past 10 years.

However, there are several advantages to the Congress.gov API which may make it an appealing data source for a future team wishing to create a more automated pipeline. Most importantly, it offers more structured data, eliminating the need for complicated XML parsing, which is required in order to extract metadata from the GovInfo documents. Furthermore, it provides a more structured method for identifying different versions of the same bill, which proved challenging with the GovInfo dataset.

4) Evaluation

Evaluating Search Accuracy

One goal of CongressGPT is to improve the quality of search results compared to existing systems like Congress.gov. Making this comparison rigorously is challenging, as it requires a set of relevance judgements. Relevance judgements are quality scores assigned to query/document pairs. For example, imagine we search with the query "biosecurity regulations," and in response 100 bills are returned. For each bill, we would ideally want a score indicating how relevant that bill is to the query.[8] Across many queries, this would allow us to make a direct comparison between two search engines.

We obtain relevance judgements for Congressional documents by deriving implicit judgements from Congressional Research Service (CRS) reports. CRS is the non-partisan research arm of Congress, which regularly produces reports on policy topics of interest to Congress. As part of these reports, CRS frequently cites bills that it considers to be relevant to the policy area under consideration. For each CRS report we consider, we create search queries using the report title, and then run those queries through both our search engine and Congress.gov. Then, we look to see where the bills mentioned by that CRS report (which we consider relevant to the query) are ranked in the respective search results as a proxy measure of result quality.

Our evaluation considers three metrics, each of which tells us something different about the relative performance of CongressGPT and Congress.gov. The first metric is reciprocal rank. Reciprocal rank is $\frac{1}{R}$, where R is the position of the first relevant document in the search results. This metric is somewhat unstable, as it only looks at the first relevant document and therefore discards a significant amount of information about the overall rankings. Nevertheless, it has become popular in recent years as many search applications place an increasingly high premium on returning relevant results at

[8] Relevance judgements may be binary, e.g. "relevant" or "not relevant," or multivalued, e.g. rated on a scale from one to five.

the very top of the rankings. Our second metric, Precision@K, measures the proportion of documents in the top K results that are relevant. Our third metric, Recall@K, compliments this by measuring the proportion of relevant documents that are ranked in the top K results.

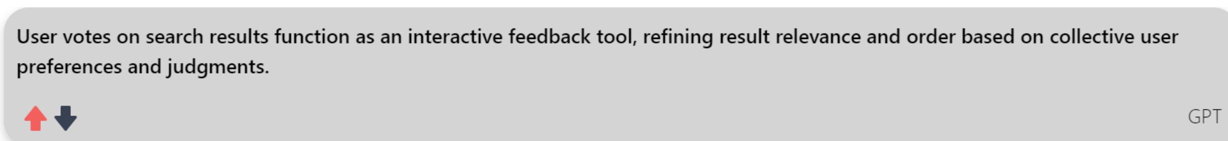
	Congress.gov	CongressGPT	% Difference
Reciprocal Rank	0.07	0.20	+174%
Precision@50	0.03	0.04	+12%
Precision@500	0.02	0.03	+2%
Recall@50	0.19	0.15	-22%
Recall@500	0.47	0.48	+2%

Overall, CongressGPT's search algorithm is much better at ranking at least one relevant document highly, as indicated by the reciprocal rank score. This is particularly critical for our application, where it is only practical to return ~1-10 results at a time into the chat for GPT to summarize. CongressGPT also improves on Precision@50, but does noticeably worse in Recall@50. This is concerning, as it indicates that there are relevant bills which are not being picked up by our search engine. In the future, more work is required to further refine the underlying search algorithm to try and improve both recall and precision-based metrics.

We caution that this evaluation technique can only serve as a lower bound on retrieval accuracy, as CRS reports do not aim to exhaustively specify every bill that is relevant to a policy topic. However, this technique remains useful for comparing between search engines, as we have no reason to think that the bills which happen to be mentioned by CRS favor one retrieval system over the other.

Data for Future Evaluation

CongressGPT produces several sources of user data which may be useful in improving and evaluating search quality in the future. The first of these is manual ratings. For each response returned by CongressGPT, including both search results and messages written by GPT, the user may choose to upvote or downvote. These votes are registered in Supabase and can be easily retrieved by those with database admin permissions. This provides a source of feedback which can indicate when GPT's response is low quality, or when search results are not relevant to the user's query.



A second source of data is the chat history stored in Supabase. Making use of this data requires additional processing compared to the aforementioned rating data, but it has the advantage of not relying on users to manually indicate search result quality. Here are some examples of circumstances that may indicate a bill was a good match for a user's query:

- The user instructs GPT to make a search within the bill to retrieve the most relevant paragraphs.
- GPT chooses to summarize the bill first, even though it was returned lower in the search results.
- The user stops searching after the bill is returned.

Similarly, there are some circumstances which may indicate that a bill was not a good match for the user's search query.

- GPT chooses not to summarize a bill, even though it was returned in the search results.
- GPT autonomously decides to perform a second search, which may indicate that the first set of results were not a good fit.
- The user instructs GPT to search again using a similar but different query.

Hallucination & Safety

LLMs are prone to hallucination, where the model confidently gives incorrect information. This is a critical barrier standing in the way of fully reliable LLM-based information retrieval.

Our application helps address this issue by asking GPT to rely on trustworthy external sources, namely Congressional documents, as a source of ground truth. This provides an extra layer of verifiability by giving the user the option to check the claims made by GPT against the original documents.

However, the user will not always be willing or able to manually verify GPT's claims, so hallucination remains an issue. During early testing, we observed GPT-3.5 hallucinating fake results on several occasions.[9] In each case, this occurred after an early version of the search engine failed to return relevant results, prompting GPT-3.5 to make up its own.

A future team may wish to implement an additional safeguard system that checks to make sure GPT is not inventing fake bills. This might involve a regular expression match that pulls out any substrings formatted like bills in GPT's response, followed by a fuzzy text match against the ground truth search results. A warning could then be displayed to the user if a bill is mentioned which does not appear in the search results.

Another limitation of our system is that it does not implement any safeguards on GPT's response beyond the content filters already imposed by OpenAI. This is a critical area in which a future team may seek to improve.

Privacy

CongressGPT stores all user messages, GPT responses, and search results in Supabase. Each message is attached to a user, whose credentials are stored securely by Supabase's

[9] GPT-4 has been shown to have fewer hallucinations, a finding which is in line with our anecdotal experience.

authentication system, which only gives each user access to their own data. This allows the application to provide a chat history, where users can look at their previous message history and pick up where they left off in old chats. It also allows us to collect data that can lead to the long term improvements of the application.

This comes with some privacy trade-offs, however. The content of each message is visible to the development team, as well as the associated email address (passwords are encrypted and are only exposed to the user). If anonymity is desired, a user could use a VPN and sign up with a new email address that is not otherwise associated with them. However, this is a burdensome work around and in the long term it is desirable to find another solution.

A future team might wish to consider privacy measures which can more thoroughly blind user information from the development team. This is not possible with the fully-managed Supabase service that the application currently relies on, so a different solution for authenticating users and storing their data would be required.

5) Recommendations for Future Work

Improving Existing Functionality

The current implementation of CongressGPT extends GPT's knowledge base using in-context learning, where new information (in our case, search results) is included in GPT's prompt. This has the advantage of creating a way for the user to trace the information summarized by GPT to its source, providing an additional layer of verifiability.[10]

The downside of this approach is that there is a computational and financial cost to having GPT summarize a large amount of text data. This approach also does not make use of the broad knowledge base which makes GPT such a powerful tool. Fundamentally, a CongressGPT that relies on in-context learning will be limited to the knowledge contained in the summaries and passages returned by the search engine, as well as the innate knowledge of the base GPT 3.5 and GPT-4 models. A fine-tuned GPT model would understand more about the workings of Congress, and could provide more detailed background information compared to the base models. It could also potentially provide better guidance about what types of searches might yield the best results.

Rather than fine-tuning GPT on Congressional bills themselves, a future team might seek to leverage existing expert commentaries about Congress, such as CRS reports, Government Accountability Office reports, or Congressional Hearing testimonies. Fine-tuning on these data sources could train CongressGPT to be more knowledgeable about Congress and give it a broad understanding that may extend the current in-context learning approach.

Besides improving GPT itself, there are many directions in which a future team may seek to improve the underlying search engine used by CongressGPT. One option might

[10] This also makes it possible in principle to create an automated system to verify the bill titles summarized by GPT against the results returned by the search engine, although our current application does not implement this.

be to switch the language model used for search result reranking. The current model, LEGAL-BERT, is showing subpar performance compared to what is often achieved by neural reranking methods. One possible reason for this is that the underlying tasks that LEGAL-BERT is fine-tuned on differ from passage similarity comparison. It may be useful to experiment with other fine-tuned models like Sentence-BERT, which is specifically trained to produce vectors that can provide passage-to-passage similarity scores.[11] A future team could also use CRS report data or user data to fine-tune their own BERT model to perform search specifically on the corpus of Congress data.

A future team may also wish to experiment with replacing the current two-stage retrieval system with a vector database. This would further reduce computational time for each search, and would allow CongressGPT to bypass the initial BM25 stage by using an Approximate Nearest Neighbor (ANN) search to directly retrieve the bill passages which are most similar to the search query vector. ANN is built into most popular vector database solutions, such as pgvector, Pinecone, and Milvus. This would also allow matching passages to be returned directly alongside bill summaries.[12]

Scalability

A significant challenge in scaling this application is efficiently managing prior chat messages. The process of generating a GPT response inevitably involves reading all previous messages, leading to escalating time and cost as the chat context expands. Our current solution to this issue is to only feed the 15 most recent messages in a chat back to GPT. However, this has the disadvantage of causing GPT to completely forget earlier interactions in long chats.

A future team could work to develop a method to compress or summarize prior messages before feeding them back to GPT; an enhancement currently absent in our application. This is an active area of experimentation in large-language model-based

---

[11] "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," Reimers and Gurevych. https://arxiv.org/pdf/1908.10084.pdf

[12] Currently, this is not supported because BERT embeddings are only precomputed for the first five passages for each bill.

chat applications which does not yet have a definitive solution. One common technique which a future team may wish to explore is feeding the chat history periodically into GPT-4, generating a summary, and using that summary in-place of the full history.

As the application gains more users, it may also become desirable to scale up processing and memory resources. Digital Ocean allows for easy vertical scaling of servers up to 8 GB of memory and 4 CPUs, after which a more expensive account subscription is required.

Eventually, if there are enough users, increasing the power of the virtual machine running the server will become insufficient, and horizontal scaling will be required. However, our database solution, Supabase, provides a fully-managed solution for scaling as the application grows, meaning that the only bottleneck will be the processing performed by the Django web server.[13]

Future Features

Ideally, CongressGPT would be able to automatically ingest and process new Congress data as it becomes available. Having access to the most recent information is critical to many Congressional staffers and analysts, who are often closely tracking fast-moving developments within Congress. Our current code forms a complete data pipeline; however, more work needs to be done to package this code into a robust script that can be run periodically to update CongressGPT with the latest information.

Another challenge with our dataset is that it relies on parsing documents from GovInfo.gov, which are returned in XML format. This approach can be inconsistent and may lead to missing or malformatted information. In implementing an automated pipeline, a future team might find it easier to move back to the Congress.gov API, which returns results more slowly, but in a more structured format. Because our current search implementation requires pre-computed BERT embeddings for each bill, an

---

[13] While the application is currently using Supabase's free tier, at a larger scale a paid plan will be required. Learn more about Supabase's pricing options on their website: https://supabase.com/pricing

automated pipeline may also choose to perform this lengthy processing step at the same time a new bill is downloaded from the Congress.gov API. This will result in a pipeline which is slow, but which can run intermittently in the background as new data from Congress becomes available.

6) Conclusion

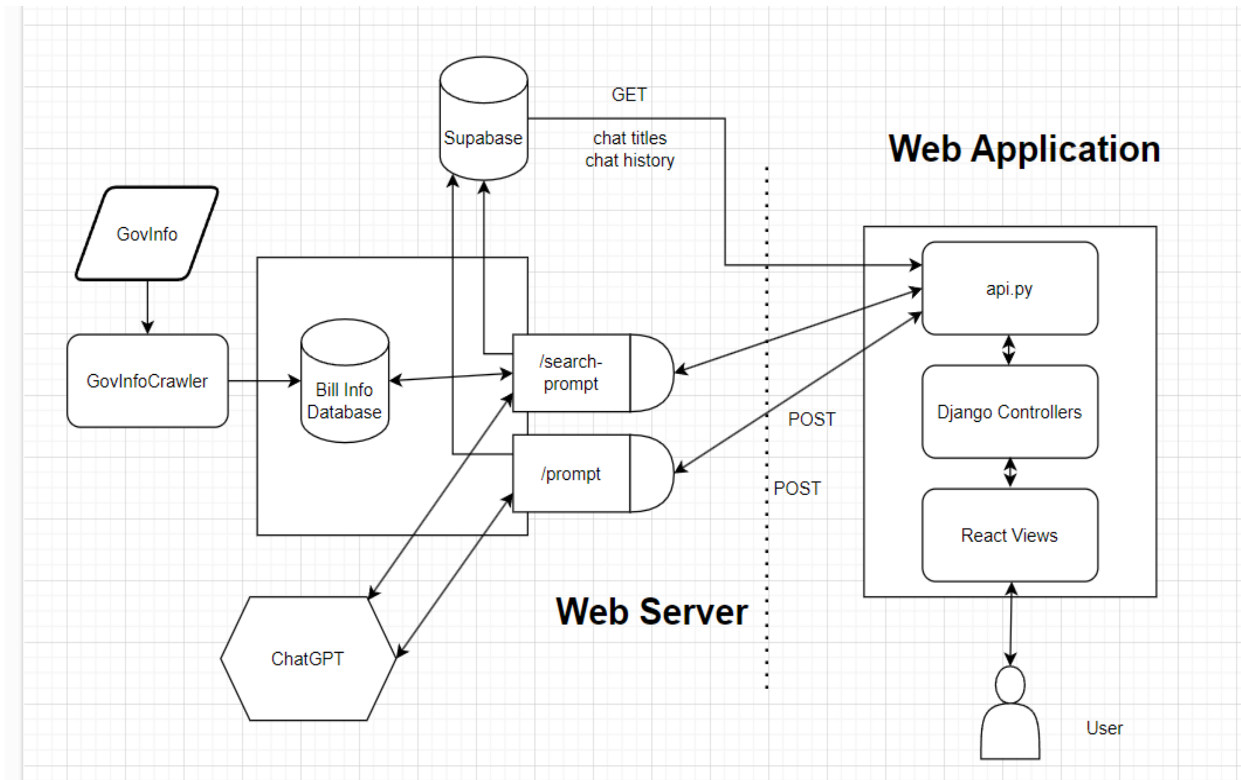
CongressGPT provides an intuitive interface for retrieving information about Congressional documents. By extending OpenAI's GPT series of models with a search engine, we are able to provide concise, context-aware summarization of complex legislative documents, leading to a higher quality search experience.

Furthermore, by structuring user interactions as a dialogue between the user and an LLM, we allow for in-depth follow-up questions and tailored analysis. Our application also addresses a core limitation of LLMs by injecting raw bill text and metadata directly into the chat, thereby providing a method for verifying claims made by the LLM as part of the search process.

The system is currently in the prototype stage, and is ready to be used by analysts and others who wish to experiment with this new method of exploring Congressional documents. As previously outlined, we have also created a roadmap for continuous improvement based on user feedback and qualitative evaluation, which will make CongressGPT a better product over time.

Appendix

Technology Stack



Our project can be divided into two different parts: the web application and web server. The user interacts with the web application, which either retrieves information directly from Supabase through the project API, or makes a request to the Django web server. The server is in charge of coordinating between the services involved in processing user chat requests, including the search engine, the OpenAI API, and Supabase.

supabase

Supabase is a Backend-as-a-Service, which provides a fully-managed, scalable cloud database and authentication service. We use Supabase to store all data particular to individual users, i.e. authentication credentials and chat histories.

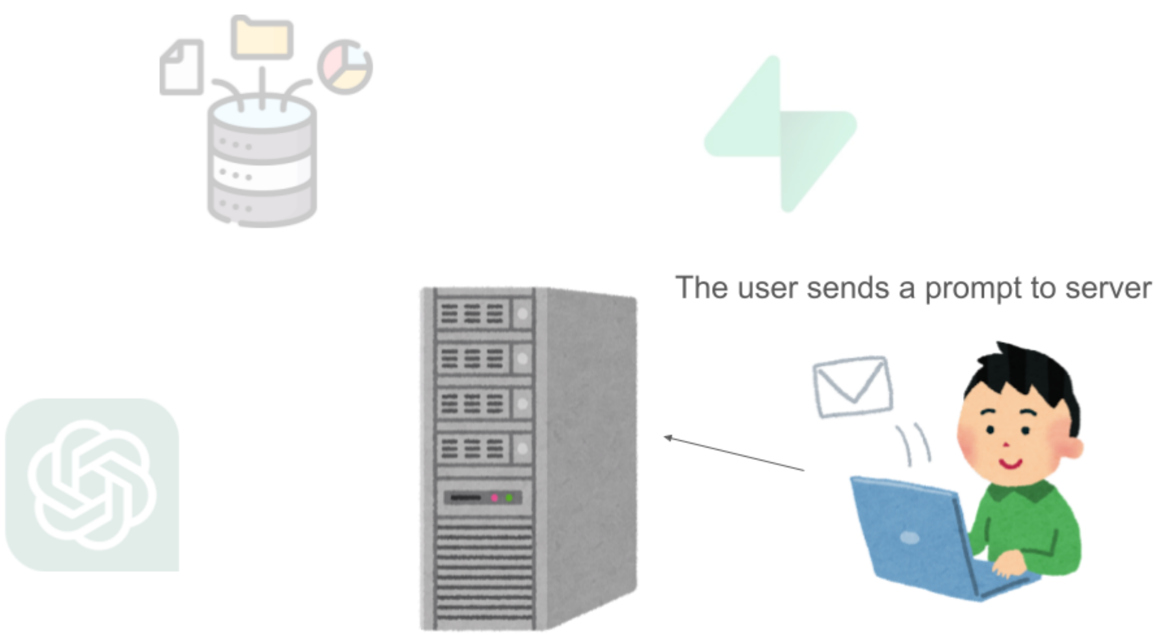
LEGAL-BERT is a fine-tuned variant of the Bidirectional Encoder Representations from Transformers (BERT) model developed by Google in 2018. We used LEGAL-BERT to produce embeddings for the first five passages of each Congressional document in our database. These embeddings are each a vector which represents a text passage in LEGAL-BERT's vector space. At runtime, the search engine creates embeddings for each new query and then compares the query vector with pre-computed vectors from the top documents that were identified by the first stage ranker.

The current implementation uses inner product to compare the query and document vectors. For each document, the first five passages are each embedded, and the final score is the mean inner product score (between the query and passage vectors) across all five passages.

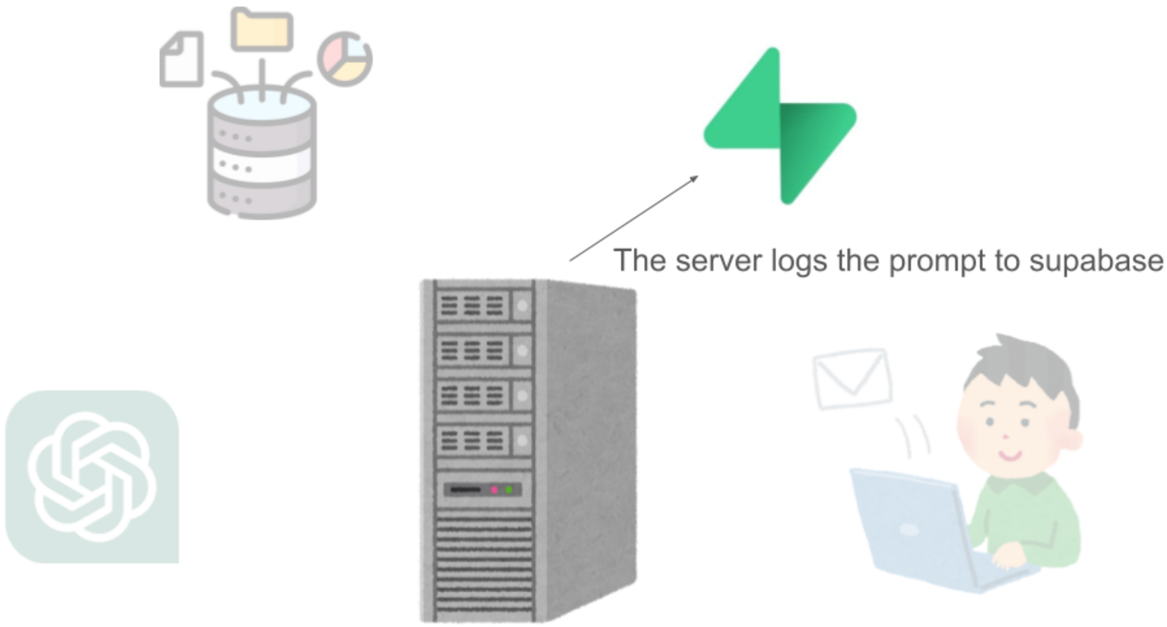
We use React.js and Django as the frontend and backend frameworks respectively. When the frontend needs information from the backend, it sends a request to the RESTful APIs exposed by Django.

While user data is stored in the cloud using Supabase, information about Congressional documents is stored alongside the web server in a separate SQLite3 database. This was done for three reasons. First, housing SQLite alongside the web server (which contains the search engine) allows for quick communication without the overhead of multiple HTTP requests to Supabase. Second, about 6 GB of space is required to store 10 years of Congressional documents (including indices and embeddings), and this would push the application beyond the limits of the Supabase free tier. Third, the search engine leverages SQLite's built-in support for BM25, which is not supported by PostgreSQL. A future team may wish to experiment with alternate search engine architectures which make use of pgvector, the vector database implemented in PostgreSQL and Supabase.

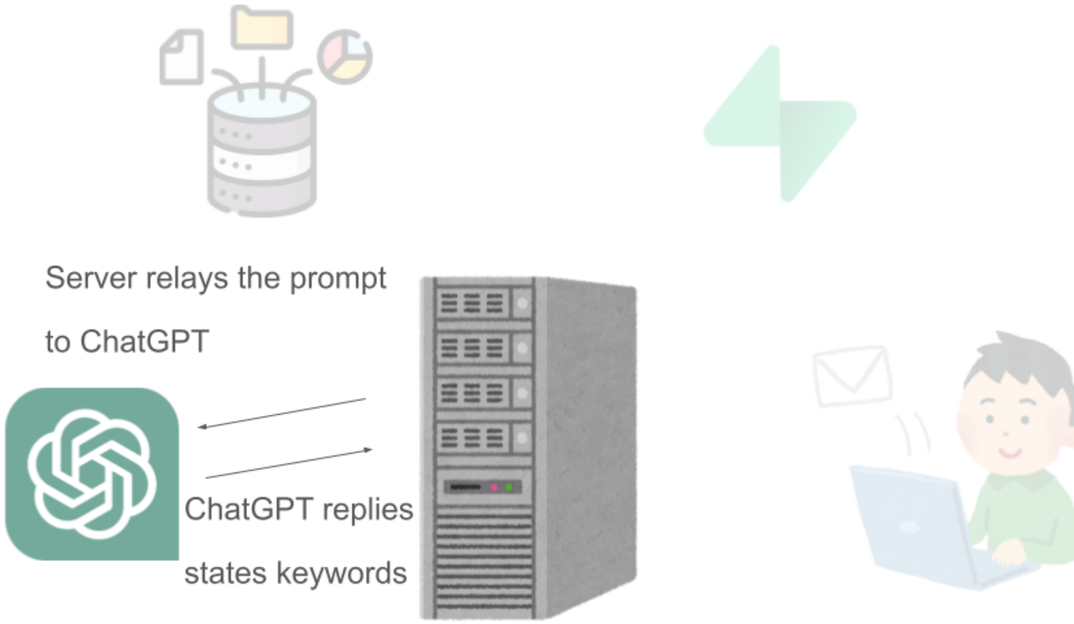
Lifecycle of a Prompt



A prompt is any input from the user, often a question related to Congressional documents. To generate a response, the frontend makes a request to the web server.

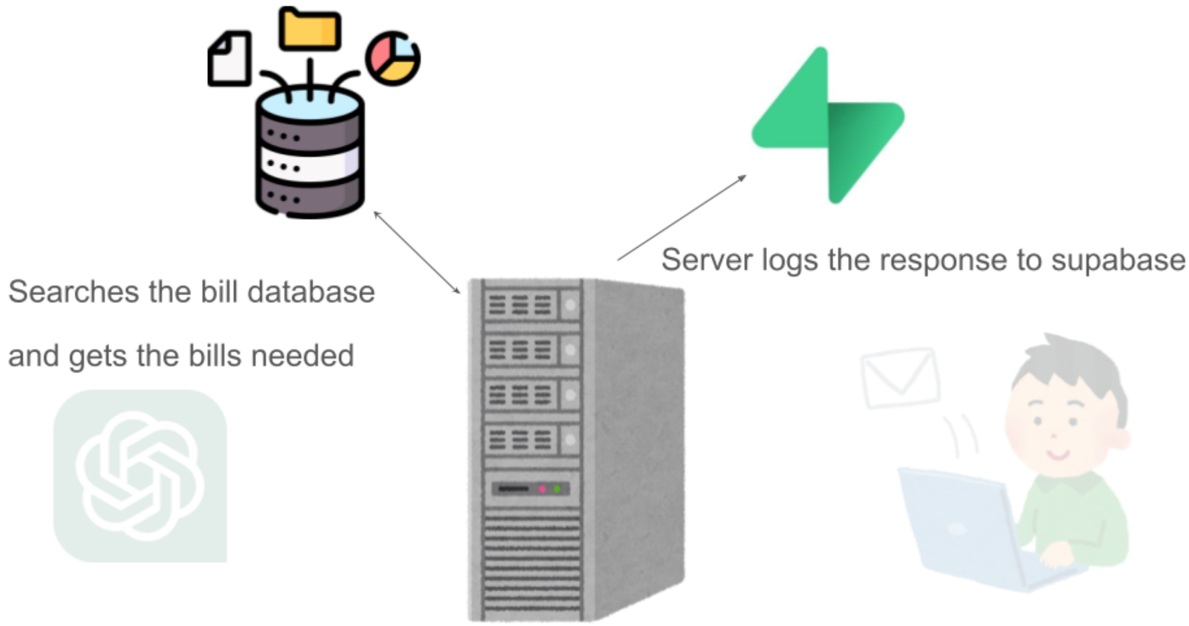


The server, after receiving a prompt/message from the user, posts it to Supabase. This allows the user to return to this ongoing chat in the future. Also, because the web server is structured as a set of RESTful APIs, which do not keep track of state, the prior chat history must be retrieved from Supabase everytime a new message is processed.



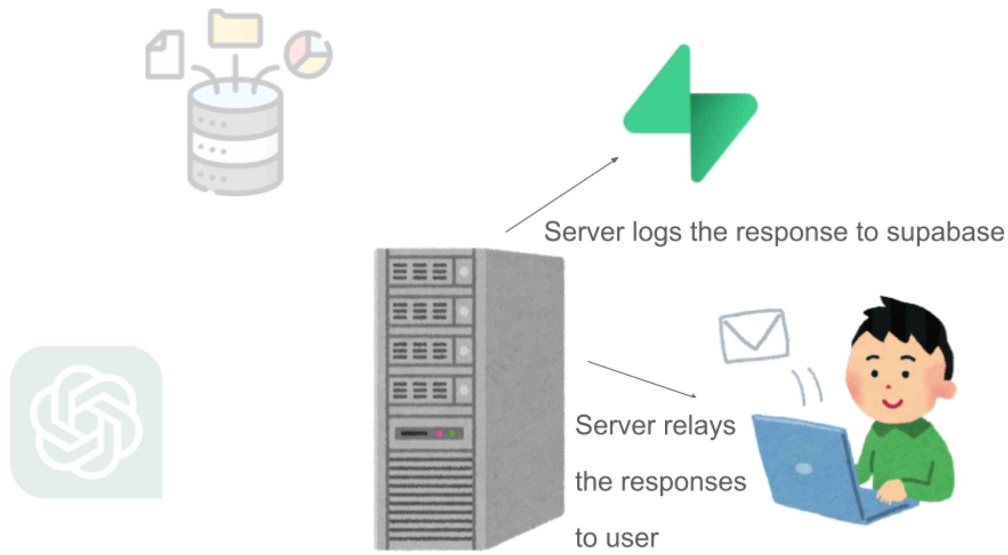
The server works as a middleman, relaying the prompt to GPT through the OpenAI API. If GPT believes the user wants it to search for Congressional documents, it will invoke a function. Functions are a type of structured interaction supported by the OpenAI API. Each time a request is made to GPT, a description of available functions is sent with it. If GPT deems it appropriate, it will flag that it wants to perform a function and supply the required parameters.

CongressGPT provides GPT with two functions, search summaries and search full texts. Search summaries searches across all available bills and returns a summary and metadata for each, whereas search full texts searches within one bill to find the most relevant passages. Because searching full texts requires GPT to supply a bill's ID, which it will only know from reading previously returned search results, a summary search must be performed before a full text search.



Whenever GPT invokes a search function, it will also generate a search query that it believes matches the user's information needs. This query is sent immediately to the frontend (so it can quickly display feedback to the user), where it is labeled a "Search Request" in the chat interface. Following this, the frontend makes another request to the backend, this time querying the "search" endpoint. A loading animation is displayed to the user, and the full search request and GPT response is processed.

When the search endpoint is called, the server will retrieve the previous messages in the chat again, this time checking to make sure that the most recent message is flagged as a search request. If it is, it queries the search engine, and sends the updated chat with the search results to GPT again for a summary. After this processing is complete, it returns a response to the frontend and asynchronously posts all new messages to Supabase.



Deployment

The application is hosted in two places, Supabase and Digital Ocean. Supabase stores user chat messages and provides authentication. Currently, the database is small enough that it does not exceed the maximum size for the free tier. In the future, it may be desirable to upgrade to a paid plan to allow for a larger database.

The rest of the application is currently hosted using Digital Ocean, a cloud provider which offers low-cost Linux virtual machines.[14] This includes the Django web server and the ~6 GB database of Congressional documents, which are stored in a SQLite3 database and indexed ahead of time to allow for rapid searching.

The application works well with 2 GB of memory and a single Intel CPU, costing $12/month. This configuration allows it to run multiple asynchronous workers using Gunicorn and gevent. These workers allow for a non-blocking event loop that can free up computational resources to handle additional tasks while the application is waiting for a response from an external source. This is critical for running the application

[14] The exception is Supabase, which is a fully-managed service that maintains its own servers, to which we make network requests.

effectively, as the backend spends most of its processing time waiting for responses from OpenAI and is therefore heavily I/O bound.

CRS Report-Based Evaluation

Within each CRS report, bills are identified programmatically by combining their code (e.g. H.R. 1234) with their associated session (e.g. 116th Congress). Bill codes can be easily isolated using regular expression matching. However, codes change every Congress, and are therefore ambiguous. To get around this, we perform a proximity search around each bill code, looking for the closest substring that is formatted like a valid Congress session. This method may not always be accurate; however, CRS reports are quite consistent in mentioning a Congress session alongside a bill code, and our manual inspections suggested that this method is reasonably successful.

We experimented with multiple methods of query formation, including removing stopwords and other common report-related terms (e.g. “overview”) from the title, and using GPT-3.5 to generate queries based on report text. Our current implementation relies on manual query formation, with 1-5 queries for each CRS report title. We chose a mix of long and short queries, as the traditional bag-of-words search algorithm employed by Congress.gov likely performs better on shorter queries, whereas the BERT-based search algorithm employed by our system likely performs better on longer queries.[15] The current evaluation process uses 20 CRS reports, although future work could easily expand the number of reports tested.

---

[15] “Understanding searches better than ever before,” Pandu Nayak. https://blog.google/products/search/search-language-understanding-bert/