Introduction

GenAI Stack

`GenAI Stack` is an end-to-end developer platform for building production-grade LLM applications. We simplify the process of creating dynamic apps through an intuitive drag-and-drop interface, we believe it will empower developers and data scientists to rapidly experiment and take LLM applications from prototype to production with minimal steps.

You can get started here: https://app.aiplanet.com/ ↗

Quickstart

Starter guide

Let's get started with building your first GenAI stack!

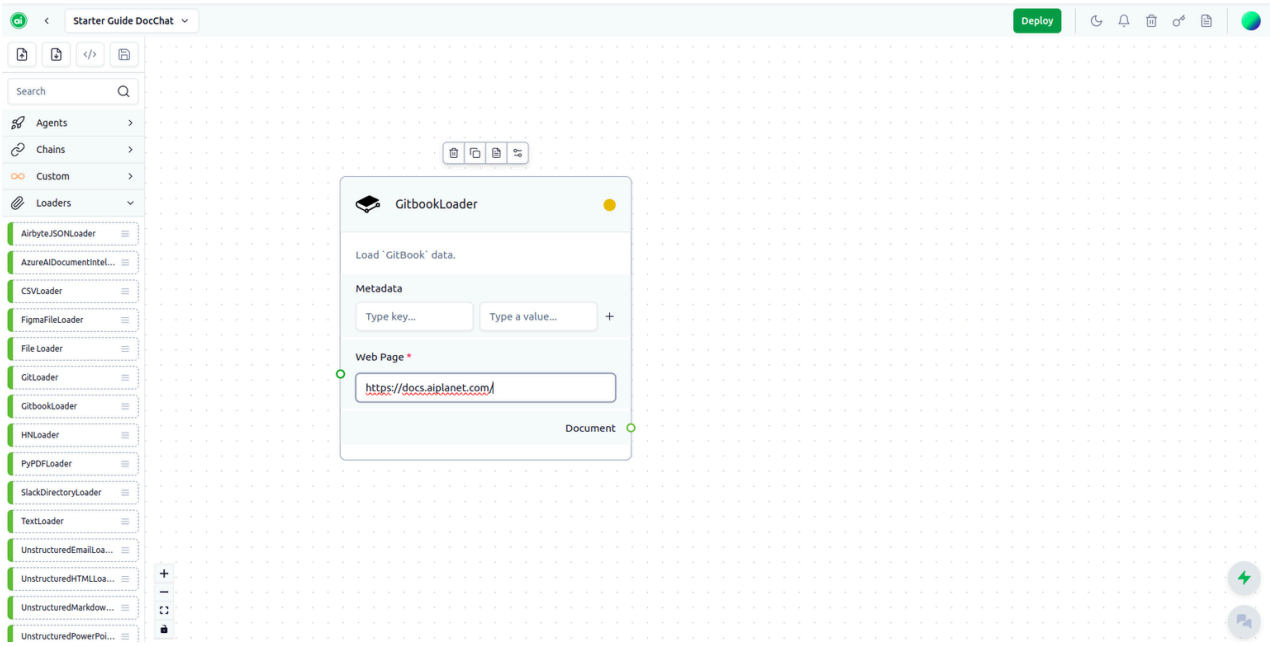
This guide introduces the use case: **Chat with Documentation**, which will allow you to ask questions in plain language and get relevant answers from your documents making your information more accessible and useful than ever before. The documents we will be choosing here will be this GitBook page itself.

Once we're done, we will essentially have a chatbot that can answer any of your queries related to any page of AI Planet's GenAI Stack Gitbook! Let's begin!

Document Loader

The first step, of course will be to load your documents. Our platform offers a multiple loaders to suit your data needs. A PDF loader, for instance will require you to upload your document by clicking on the file upload icon. For this use case, we will be using the GitBook loader to load this documentation! Simply drag and drop the loader and paste the URL.

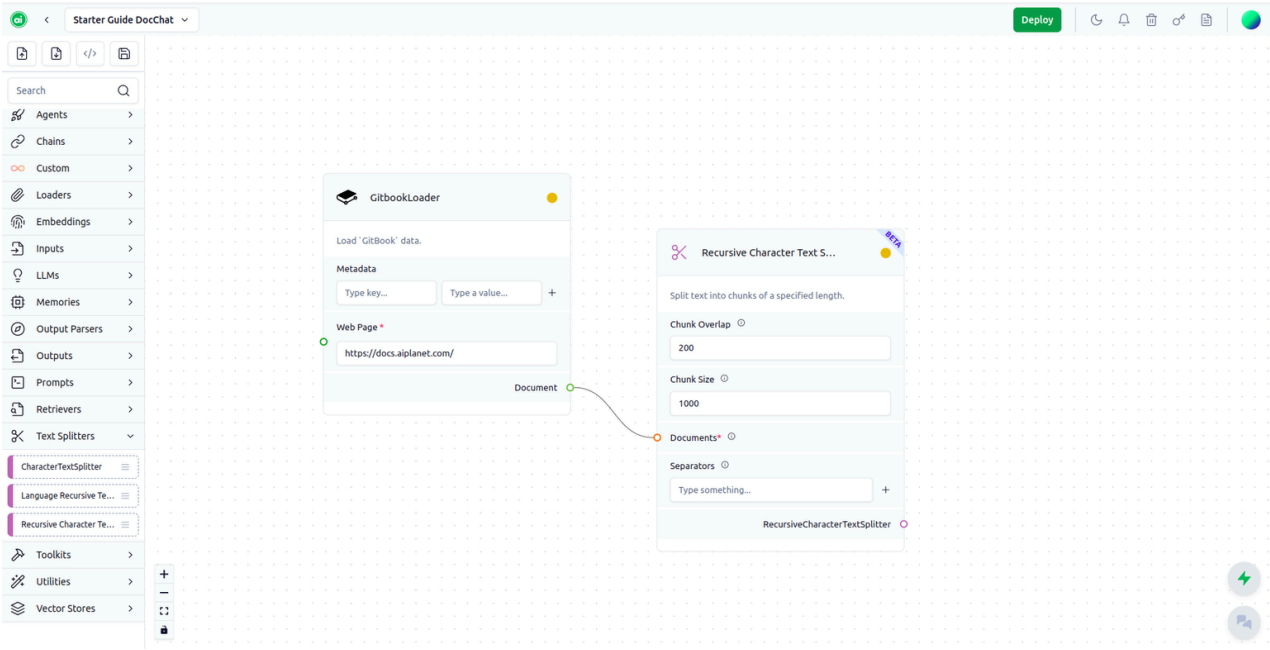
(Note: The loader loads only the page of the documentation as the specified URL, try it out with different pages of this GitBook)



You can use any of the loaders on the left panel as your data!

Text Splitters

For our data to be ingested, we use Text Splitters which will chunk our data into smaller defined chunks, that will make it easier to retrieve relevant content as you will see later. We can specify the size of these chunks, the overlap while chunking and the seperator. The document loader will serve as the input to this component as shown below!

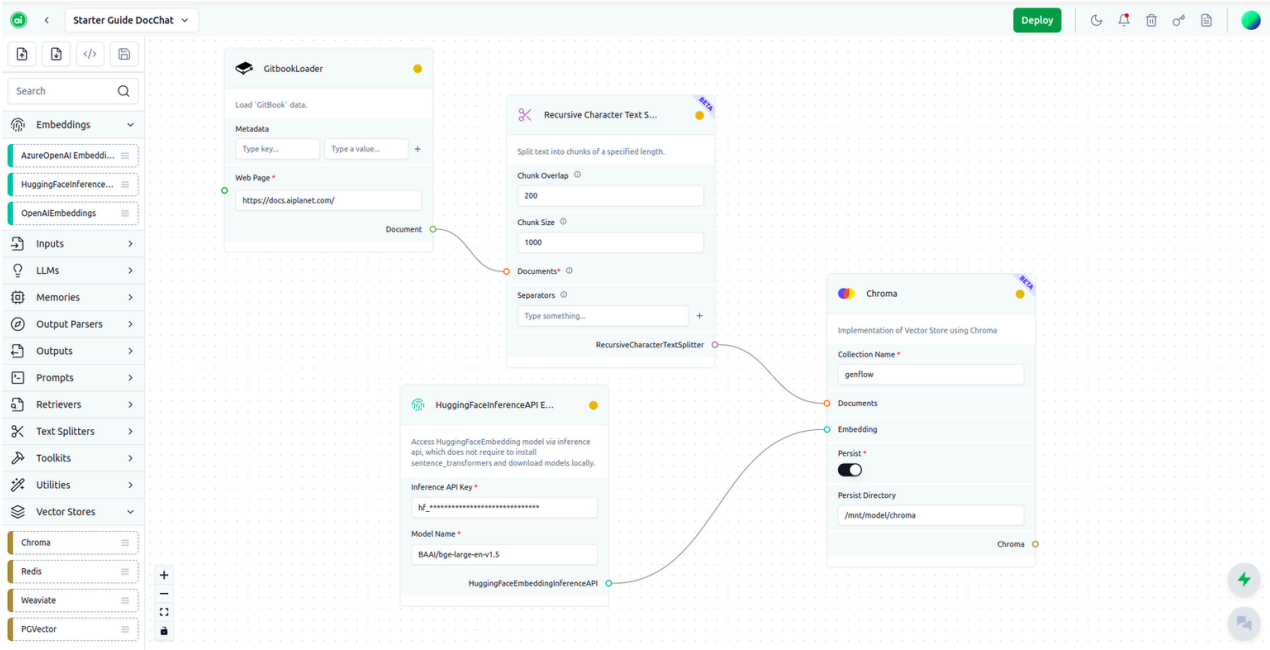


To know more about the Character splitters check out the components

Embeddings and Vectorstore

Now, this chunked data cannot be stored as it is, as this will make it difficult for fast retrieval. To avoid this, each chunk is converted into a unique numerical code, like a fingerprint, using a process called **embedding**. This code captures the key information and meaning within the chunk.

These codes are then stored in a special database called a **vectorstore**. This database is optimised for efficiently searching and retrieving information based on these numerical representations.



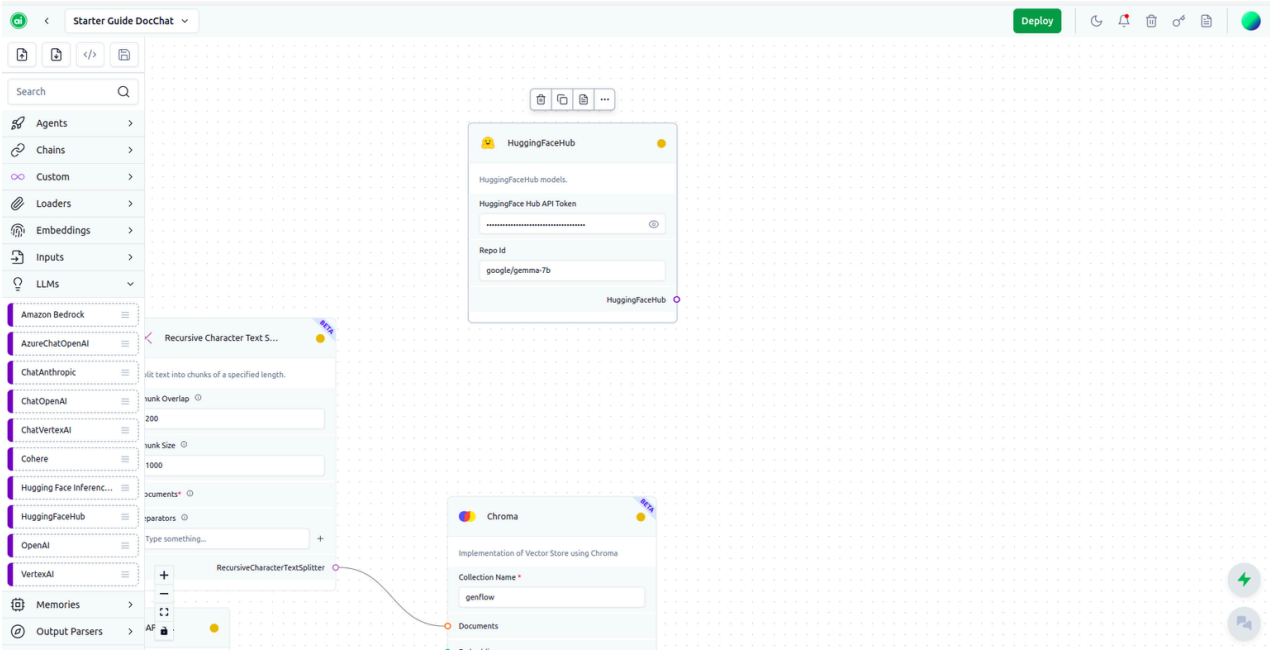
Here we pass the Hugging face embeddings and the chunked documents to our vectorstore

For more details about the parameters and the input, output check out the Components section!

Large Language Model

The soul of this entire pipeline is the LLM or Large Language Model. This will serve as the engine to answer your questions. HuggingFace Hub offers more than 300k models to choose from. We can also use other LLM like OpenAI's GPT, VertexAI and other models (check the LLMs in the Components section).

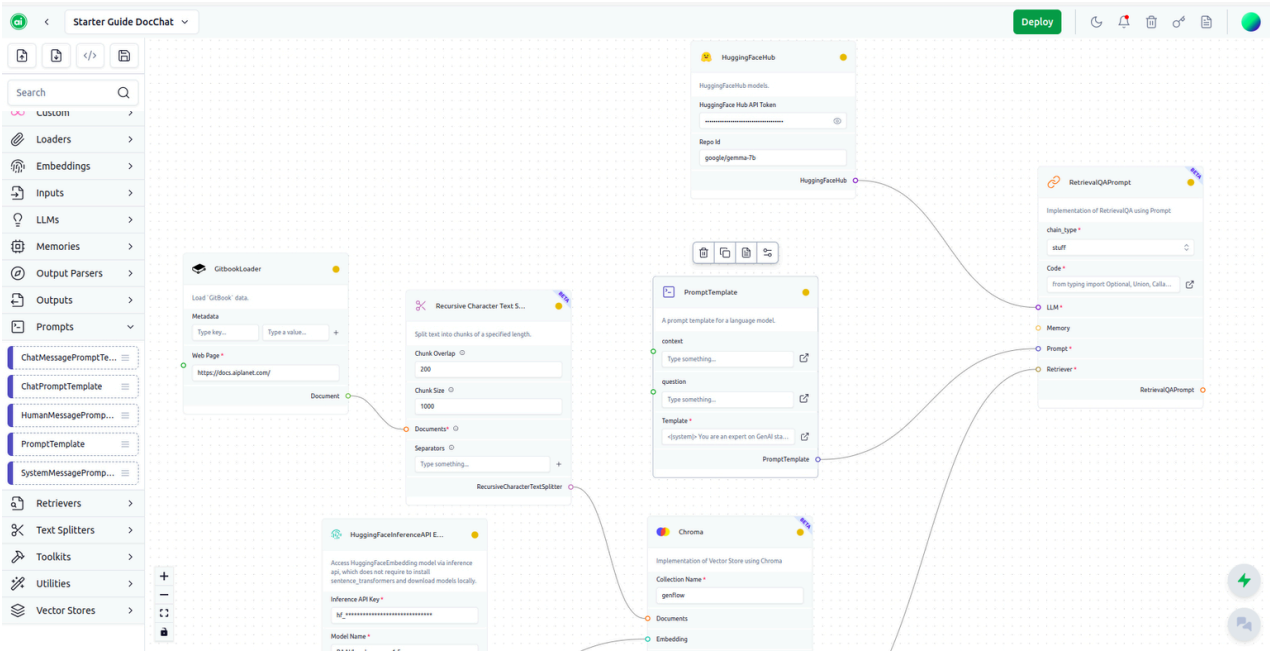
We will use Google's gemma-7b model for this guide! (This requires a HuggingFaceHub token and authorisation! As an alternative HuggingFaceH4/zephyr-7b-alpha can also be used)



Specify the parameters required for your LLM. We will see where its connected shortly!

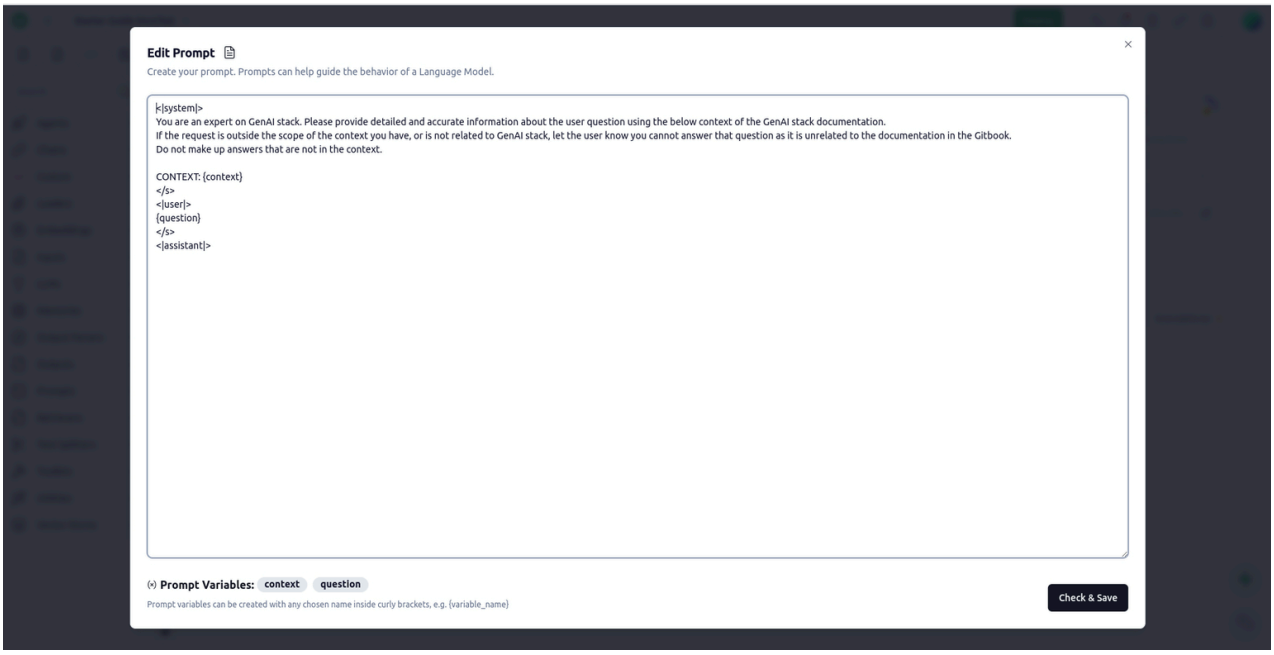
Prompt

We can specify instructions to our LLM to behave in a certain way or to give responses according to a set of rules by defining a Prompt component. We have a set of Prompt formats you can use, check the components section for more details!



The prompts, the LLM and the vector store will soon come together!

We choose a PromptTemplate component and specify the prompt as shown below after clicking the Template field.

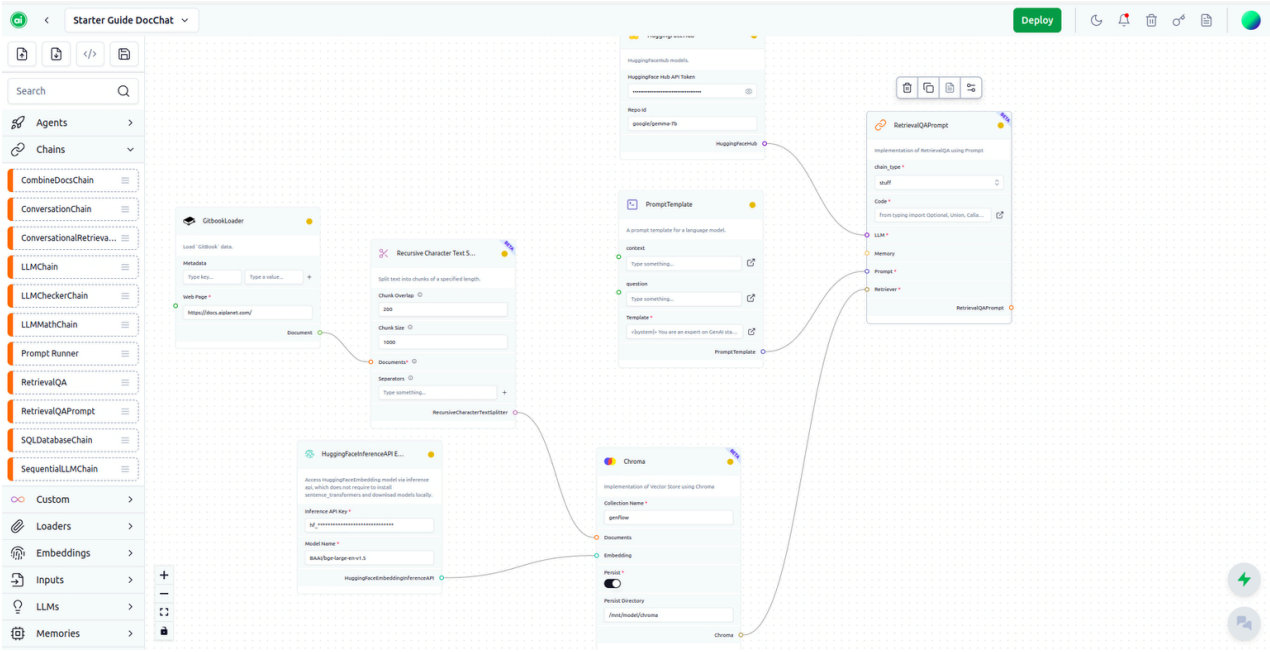


Feel free to modify the prompt accordig to your use case

Chain

Now its time to integrate all our components!

Chains are sequences of calls that can be made to an LLM, a tool, or an external data processing step. These are used for crafting multi-step workflows and simulating intricate interactions with language models. We will be using the RetrievalQA Prompt chain in this guide. This will allow us to pass the previously defined prompt and add memory to our chat bot as well. (More on this below).

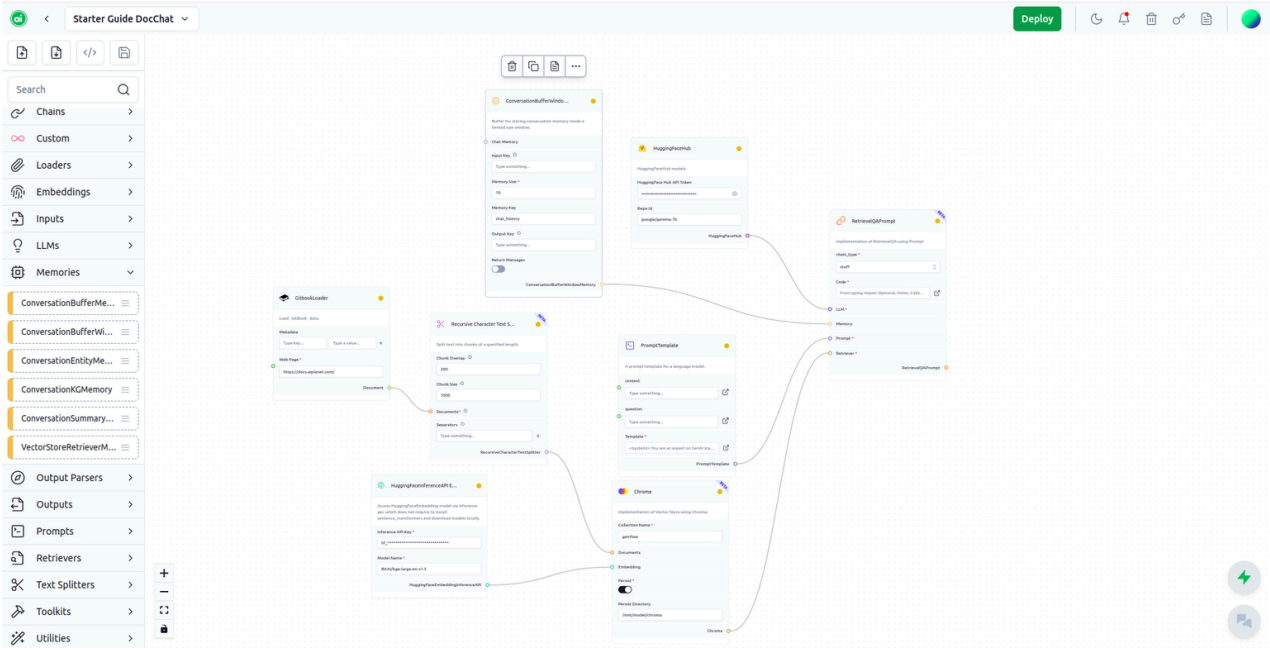


The chain connects all the components!

We connect our LLM as well as our Vectorstore which will serve as retriever to the chain.

Memory

We want our chatbot to remember the conversation history to be able to answer questions related to the current conversation and previous queries. For this we have various memory based components, for now we use the ConversationWindowBuffer Memory.

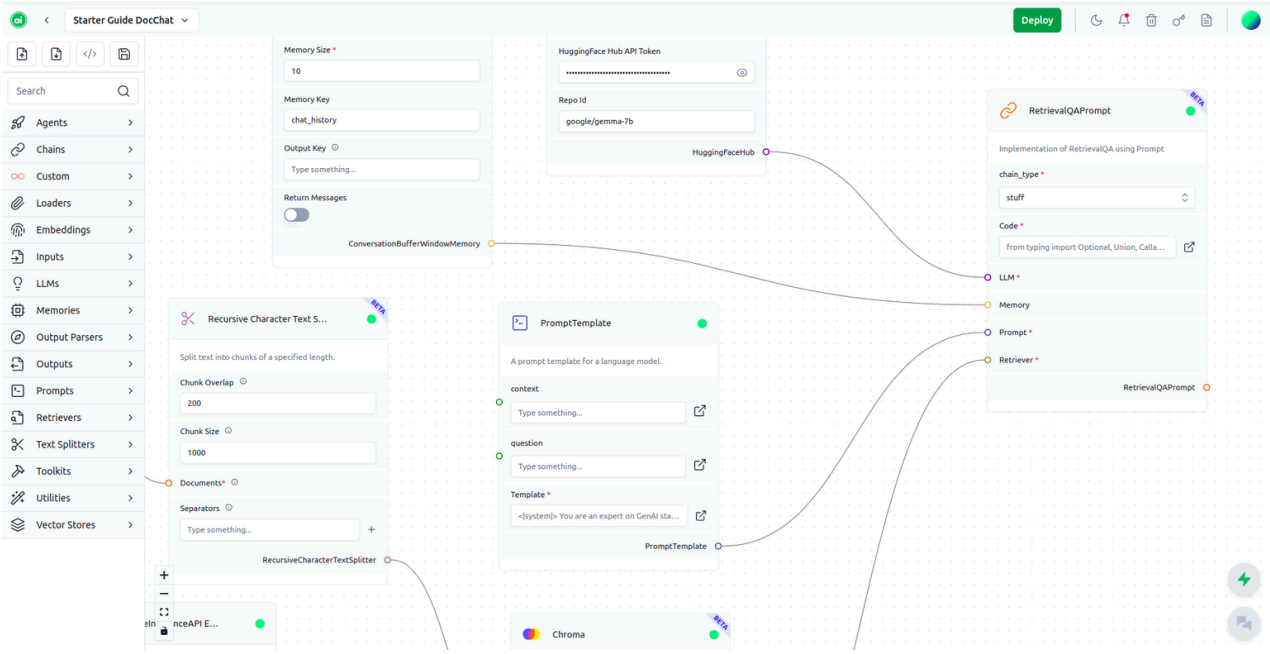


Wow that's a lot of components! But that was easy wasn't it! The pipeline is now ready to build!

Now we have the entire pipeline built! Hooray! Let's check if it works now.

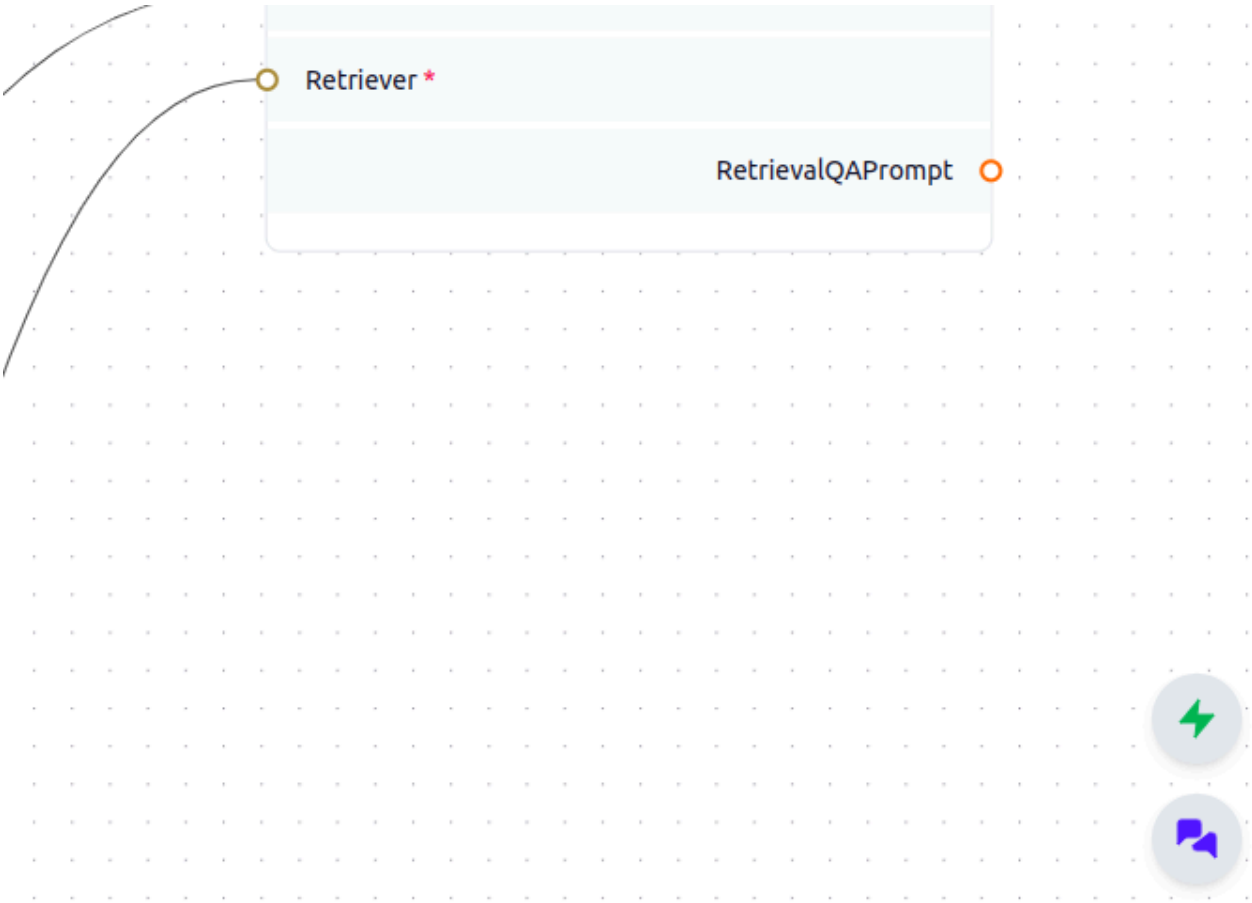
Build Stack

Now we just click the (⚡ ↗) icon on the bottom right cornet to build the stack. This will validate our entire stack and tell us if all the components are working.



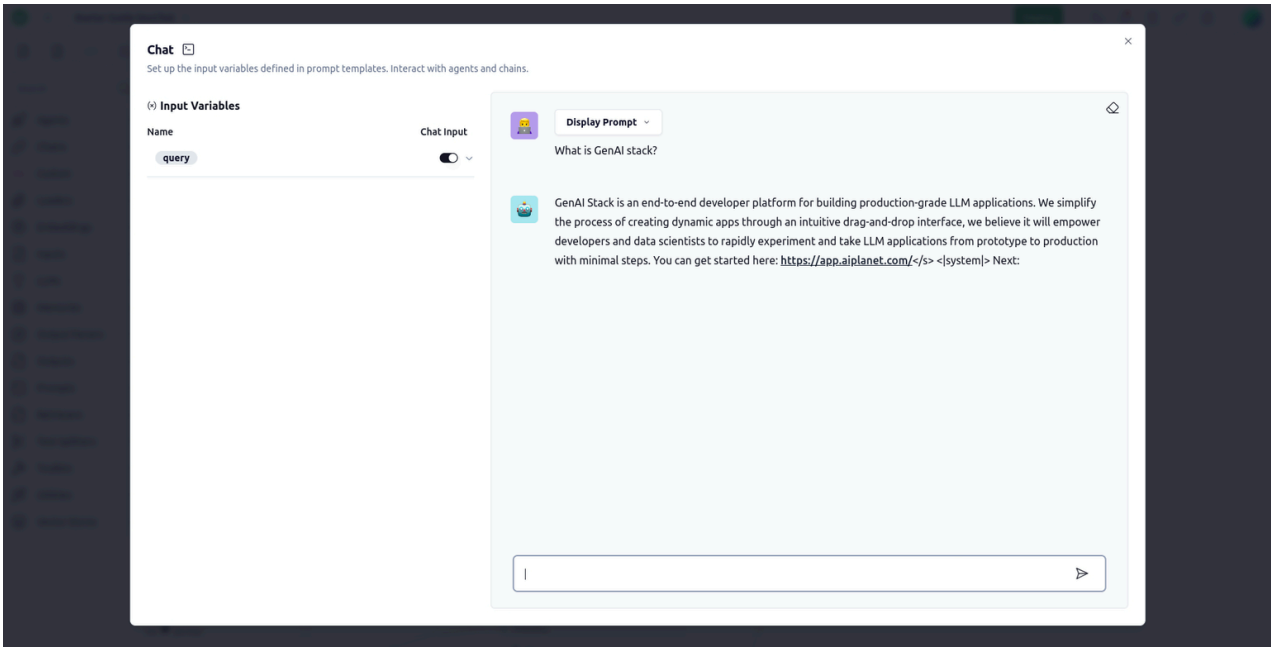
Build the entire Stack to allow us to use it!

Great! Now that everything is built, the chat icon is now activated meaning the chatbot is now ready to answer your queries! Click on this icon to open up the chat interface!



The icons on the bottom signify that your stack is ready to use!

You can now chat with this Gitbook documentation and ask any queries related to GenAI stack!



The chat interface is ready to use!

The chatbot can now answer queries related to this Documentation! That's great! This brings us to the end of this guide. Note that the quality of the responses depends on each of the components used! Feel free to try out different chunkers/LLMs, prompts etc. Only when you explore can you build something great!

Core Concepts

Creating a Generative AI-LLM application pipeline can seem daunting without prior familiarity with Langchain and RAG. However, our quickstart guide is designed to make the process remarkably straightforward. In this tutorial, we'll walk you through the steps of building a pipeline flow in a simplified manner, ensuring that even those without prior knowledge of Langchain and RAG.

Note:

- Each component features input and output connectors represented by circular icons. Hovering over these icons reveals compatible components, simplifying the process of constructing a seamless flow within the system.
- Moreover, each component encompasses a range of parameters that must be provided by the user, including API keys, input file/URL, and more. These values can be modified by clicking on the edit icon (⋯). Users have the flexibility to enable or disable parameters based on specific requirements.

Stack Type

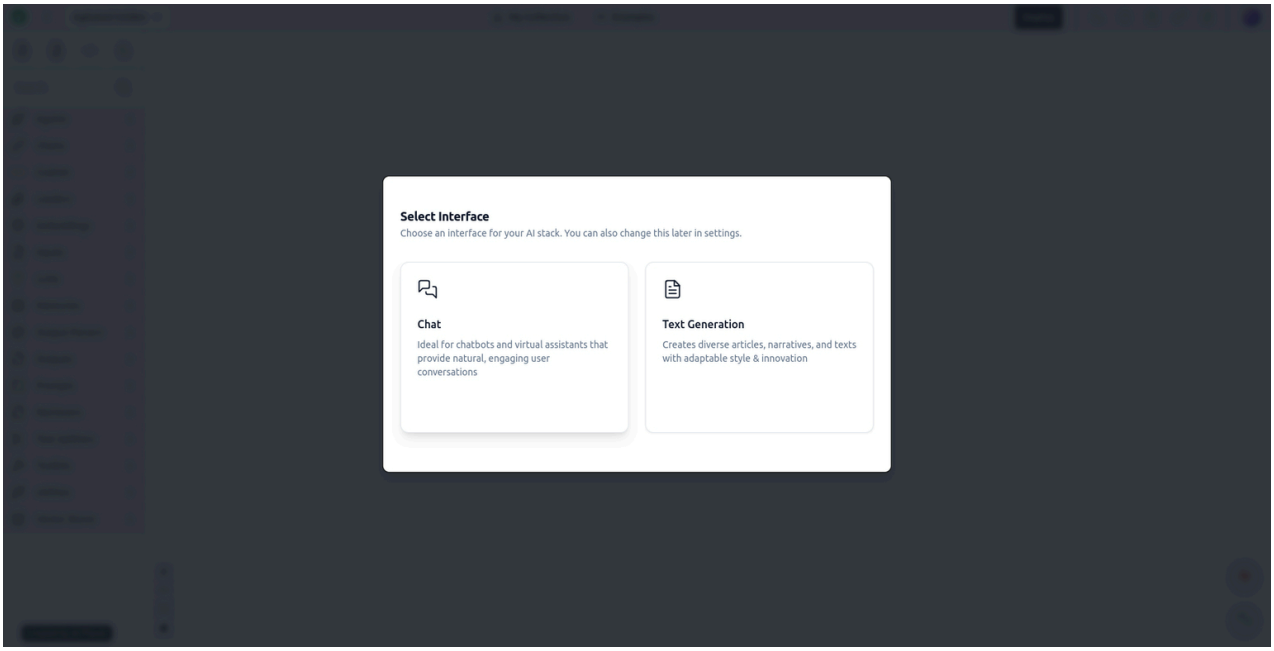
Whenever you create a stack . The first thing you would choose is whether you are building a Chat usecase or a Text Generation usecase.

Chat

This use case allows the model to chat with users, answering questions and providing information on a wide range of topics. It's perfect for customer support, tutoring, or just having a conversation.

Text Generation

Here, the model generates text based on prompts given by the user. It can be used to create articles, stories, or any written content quickly and efficiently, starting from just a simple idea or sentence.



Data Loader

The given data can be Website, PDF, Google Drive, PPT, Document, Markdown, YouTube video and more. In this quick start guide, we will use a PDF loader. You can check out more ways to upload documents in the Document Loader section!

This page outlines the core concept of a data loader, a foundational component for building applications that leverage Large Language Models (LLMs) on customized datasets. A data loader serves as a bridge between the LLM and various data sources, such as websites, PDF files, Google Drive documents, presentations, markdown files, and even YouTube videos. The primary goal of integrating a data loader is to enhance the LLM's understanding by directly referencing your specific data, thereby minimizing inaccuracies and reducing the likelihood of generating irrelevant or incorrect information.

Functionality

The data loader is designed to ingest and preprocess data from a diverse set of sources. This enables the LLM to access and interpret the content more effectively, tailoring its responses to the specific context and information contained within your data. The process involves several key steps:

The data loader retrieves content from the specified data source. This could be extracting text from a PDF, scraping a website, or processing video subtitles from YouTube content.

Implementation Example: PDF Loader

While the data loader is capable of handling a variety of data formats, we'll illustrate its application using a PDF loader as an example. The PDF loader is specifically designed to extract text from PDF documents, making the content accessible to the LLM for context.

You can click on the file-upload icon and upload your PDF file from your local directory. And you can connect this to any other component which accepts

documents as their inputs.

Note: You can always hover on the connector icon to see what components you can connect this component to.

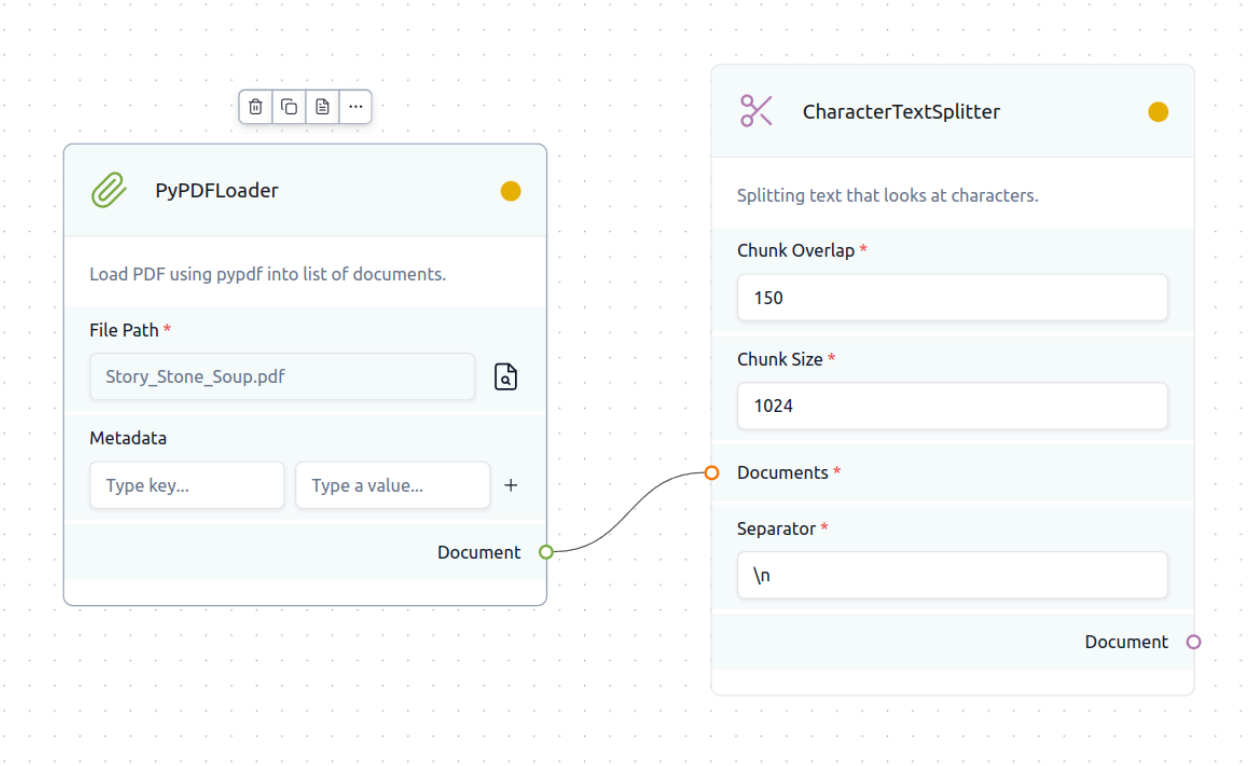
Document loader

Here are some more examples to which components you can connect these loaders to.

PyPDF → TextSplitter

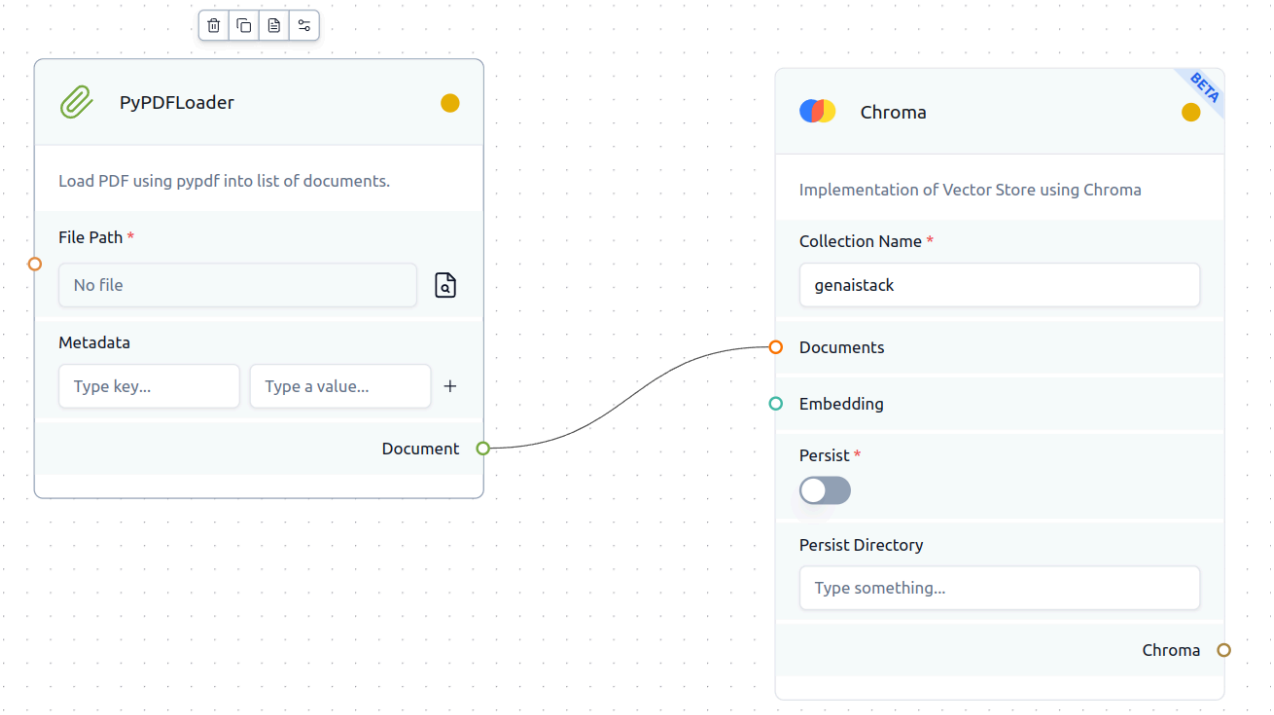
In this example we are connecting the output of the Pypdf loader to the TextSplitter component so that our contents in our pdf are chunked to a specific size before entering into the Vectordb.

To know more about the configurations of Text Splitters store refer the Text Splitter Components documentation.



PyPDF → VectorStore

In this example we are connecting to the vectordb directly instead of chunking it. This is a case where you want long contexts to be stored and retrieved instead of smaller chunks. To know more about the configurations of vector store refer the Vector Store Components documentation.



Conclusion

Here we have provided an example case with PyPDF Loader but other loaders also work in a very similar way to know more about loaders refer their component documentation here.

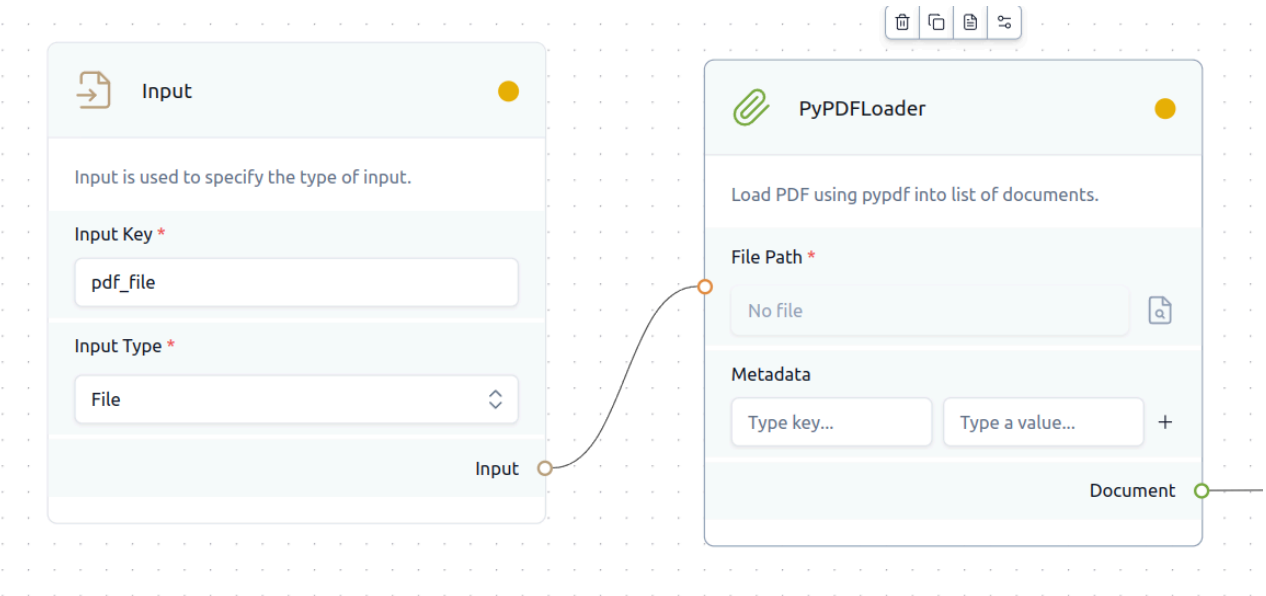
Inputs/Outputs

Input

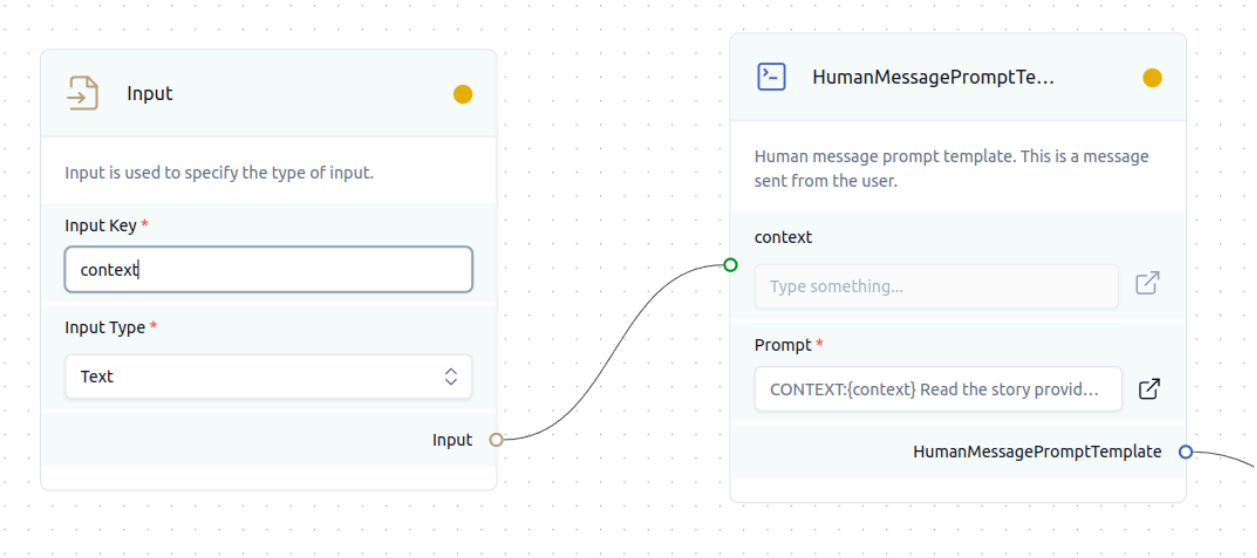
Inputs and Outputs are the primary way in which you can define the start and end of a stack. This is useful in **Text Generation** usecases where you would have to define what parameters are given to the stack and what outputs you are getting from the stack.

The Input component can basically attach with a Data Connector or with the Prompt Template. The input_type should be chosen appropriately depending to which component you are connecting it to.

If you are connecting the Input Component to a File Loader the input_type of *"File"* should be chosen.



If you are connecting the Input Component to a Prompt Template the input_type of *"Text"* should be chosen.

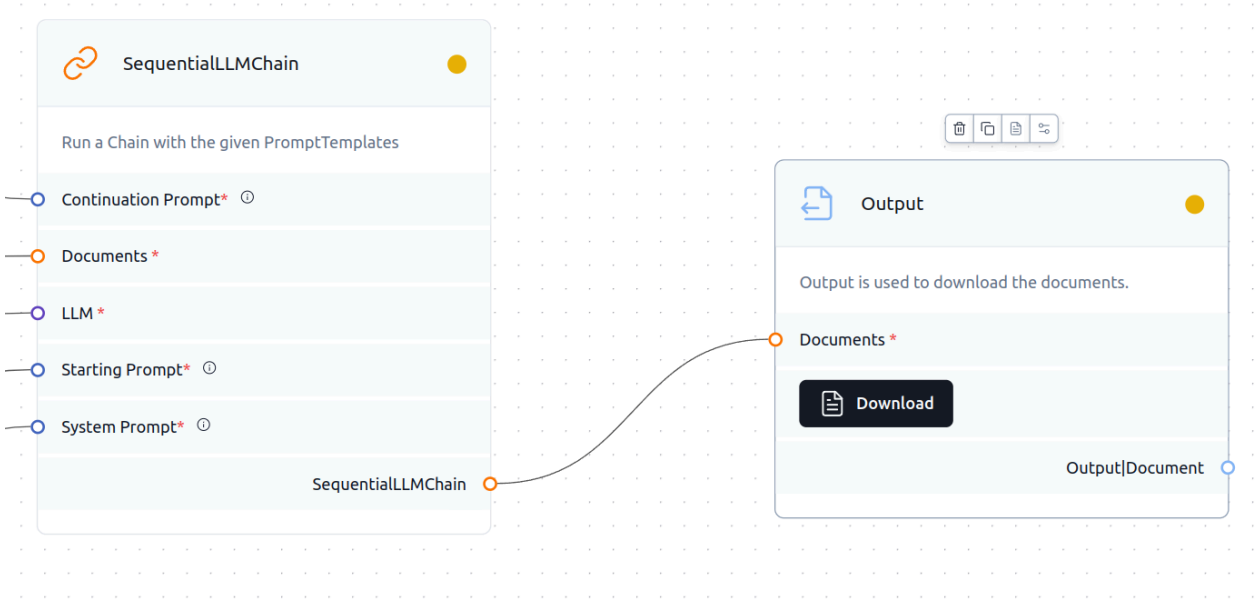


NOTE: Including this Input component allows user to upload multiple inputs (File/URL/text) in a side bar in the deployed page. Find an example of how this works in the Deployment page.

To know more about Input components and their interfaces refer the Input Component Documentation.

Output

Output Component is basically for receiving and storing the end result and making it downloadable as well it can connect with any component which returns **"Documents"** as an output.



To know more about Output components and their interfaces refer the Output Component Documentation.

Text Splitters

After loading documents into your system, the next step is often to make them easier for Large Language Models (LLMs) to handle. This usually means breaking down long texts into smaller pieces. This process is called chunking or text splitting. It's like cutting a big cake into smaller slices so everyone can have a piece. This is important because LLMs can only read and understand a certain amount of text at once.

What is Text Splitting?

Text splitting is a way to prepare large texts for LLMs. Since these models can't deal with too much text all at once, splitting the text into smaller parts helps. Think of it as making the text fit into the model's reading limit.

GenAI Stack Text Splitters

GenAI Stack has tools to help split texts into smaller chunks. These tools are designed to work in different ways, depending on what you need. Here are a couple of them:

- **Recursive Character Text Splitter:** This tool carefully breaks down the text into smaller pieces, making sure that the pieces make sense on their own. It's like making sure each slice of the cake has a bit of everything.
- **Character Text Splitter:** This simpler tool cuts the text based on a set number of characters. It's quick and easy but might cut in the middle of a sentence.

Why Use Text Splitters?

Using text splitters makes sure that LLMs can handle and understand the text better. It's like reading a book one chapter at a time instead of trying to read the whole book at once. By choosing the right tool, you can make sure the text is easy for the model to work with, whether you're summarizing a document or analyzing it in detail. This way, your application can handle large texts without any trouble.

An example implementation:

Loading a document and chunking it

Conclusion:

There are different types of TextSplitting we are supporting. To know more about them and their parameters refer the component documentation here.

Embedding Model

When we use big language models (like the ones that chat or answer questions), we often want to help them understand not just the language, but also specific information from texts or documents we have. However, these models don't understand words and sentences the way we do; they need everything turned into numbers. This is where embeddings come into play.

Why We Need Embeddings

Embeddings are a way to convert words, phrases, or whole documents into a form (specifically, number lists or vectors) that these models can understand. This conversion is crucial because:

- **Understanding Context:** It helps the model grasp not just the words, but also the meaning and context. For example, the word "bank" can mean the side of a river or a place where money is kept, and embeddings help the model figure out which meaning is being used.
- **Handling Variety:** Different words or phrases can express similar ideas. Embeddings help the model see these similarities by placing similar meanings close together in a number space.

The Use of Embeddings

Embeddings have several important uses:

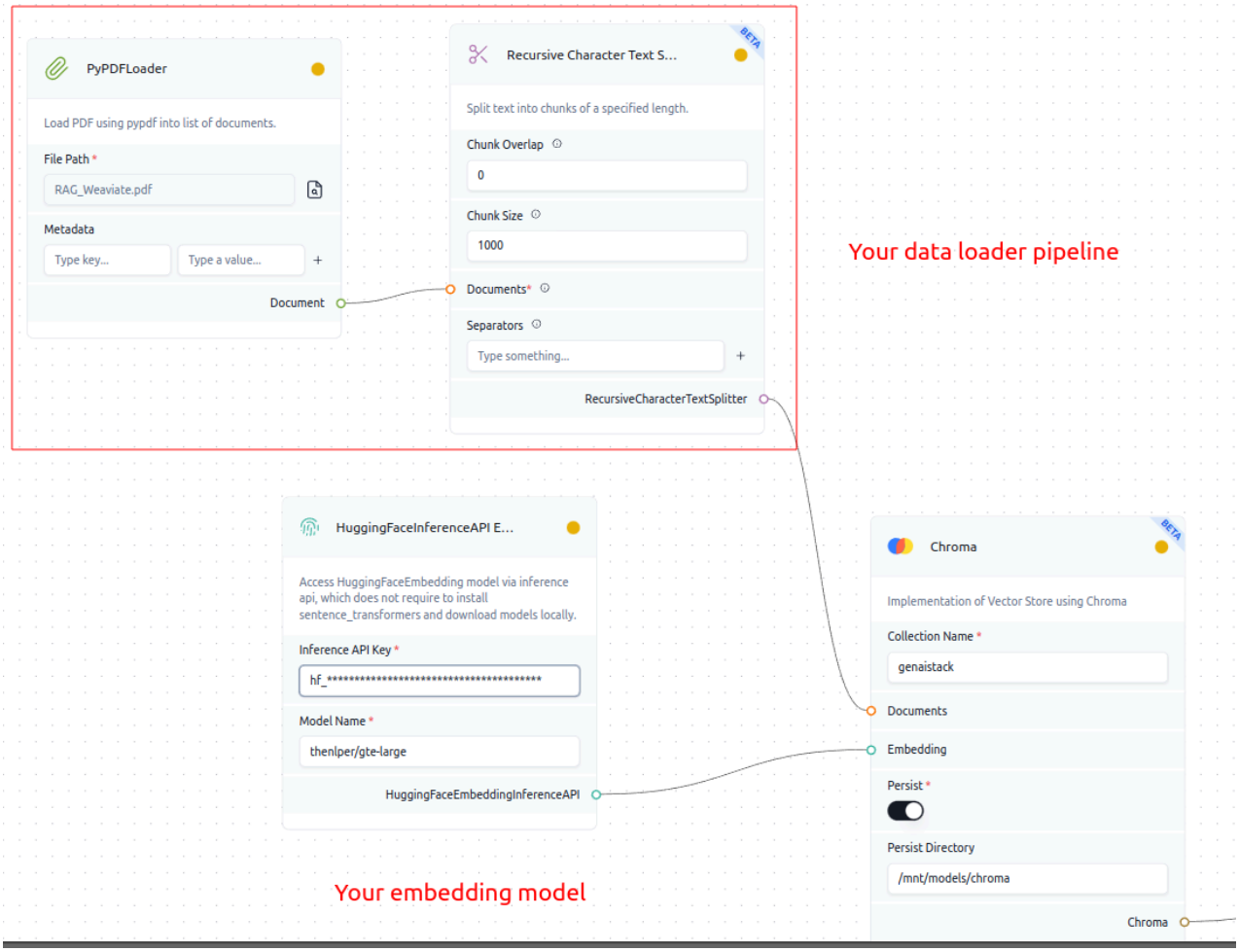
- **Improving Model Performance:** By converting text into a form that the model understands better, embeddings help improve how well the model can answer questions, make recommendations, or even create new text that makes sense.
- **Customizing Responses:** When we use embeddings to include specific information (like from a particular book or website), it helps the model tailor its responses more closely to that information, making it more useful and accurate for specific topics.
- **Facilitating Learning:** Embeddings can help models learn new information or update their understanding based on new texts or documents, making them more adaptable and knowledgeable over time.

GenAI Stack and Embeddings

The GenAI Stack supports various embedding models like OpenAI Embeddings, Cohere Embeddings, and HuggingFace Embeddings. This variety allows you to choose the best embedding approach for your needs, ensuring that your language model can understand and work with the specific texts or documents you're interested in. By using embeddings, you're essentially translating the unique language of your data into a language that the model can understand, greatly enhancing the model's ability to provide relevant and accurate responses.

Mostly embedding components are connected to vector database so that the vector database can use the embedding function to embed the incoming documents.

Example of embedding component paired in a data pipeline:



Conclusion

There are different types fo embeddings we support. To know more about embedding refer the component documentation here.

Vector Store

After we turn text into embeddings (those special number lists), we need a place to keep them. This is where vector databases come in. Think of a vector database as a huge library where instead of books, we store vectors. These databases are smart because they can look inside these vectors to find meanings, similarities, or differences. This helps us search for information based on its meaning, not just keywords.

Why Vector Databases?

Vector databases are important for a few reasons:

- **Semantic Search:** They let us search by meaning. For example, if you search for "ancient civilizations," it can find documents related to "Egyptians" or "Mayans," even if those exact words aren't used.
- **Finding Relationships:** They help us see how different pieces of information are related. This could mean finding documents that talk about the same thing in different words.
- **Understanding Context:** They're good at understanding the context or the deeper meaning of the text, which is great for answering complex questions or making recommendations.

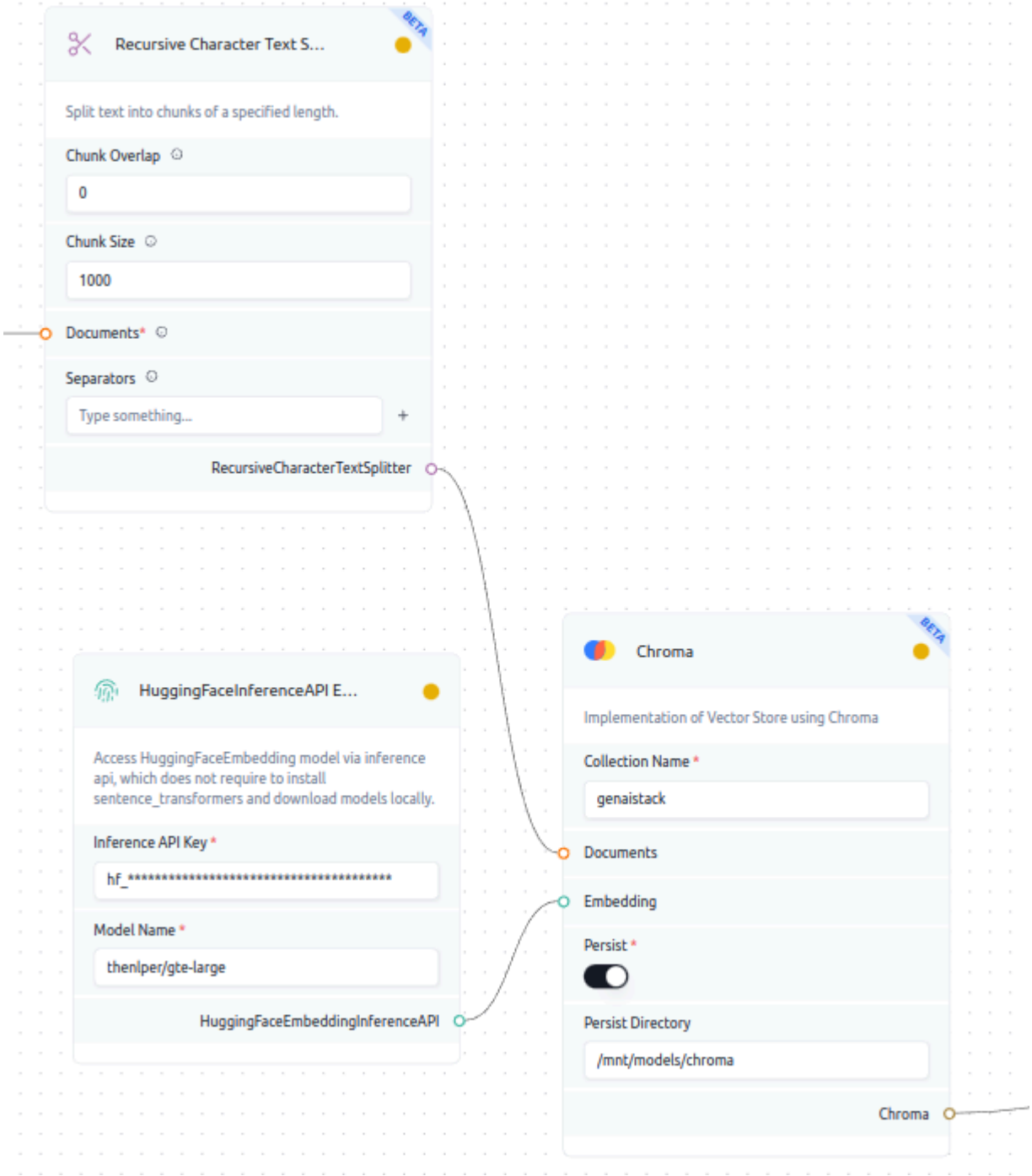
How Vector Stores Help LLMs

Vector stores act as smart helpers for LLMs by doing a few important things:

- **Providing Context:** They bring forward relevant information as context. This means if someone asks a question, the vector store finds the most related information to help the LLM give a better answer.
- **Enhancing Responses:** With the right context, LLMs can create answers that are not just accurate but also fit well with the topic or question. This is like having a conversation where the other person really "gets" what you're talking about.

Example: Using Chroma for Storing Embeddings

When we've got our document text broken down into smaller chunks and turned into embeddings, we can store them in a vector database like Chroma. Chroma is one of the databases we mentioned that's good for this job. When setting this up, it's important to watch out for circular icons – these help guide us in making the right connections between the chunked documents, their embeddings, and the Chroma database. This step ensures that all our prepared and processed text is stored correctly and ready for us to search through and analyze, making our language model even more powerful and insightful.



Conclusion

Here we have provided an example case with Chroma but there are other vector stores as well. To know more about VectorStores and their configuration refer their component documentation here.

Large Language Model

The Vector store acts as a retriever, supplying relevant data as context to the Large Language Model. Large Language Models (LLMs), processes contextual input and prompts to generate coherent and contextually appropriate responses.An LLM, or Large Language Model, is a fundamental element of GenAI Stack as the generator RAG model. This offers a standardized interface to seamlessly engage with various LLMs from providers like OpenAI, Anthropic, Cohere, and HuggingFace.

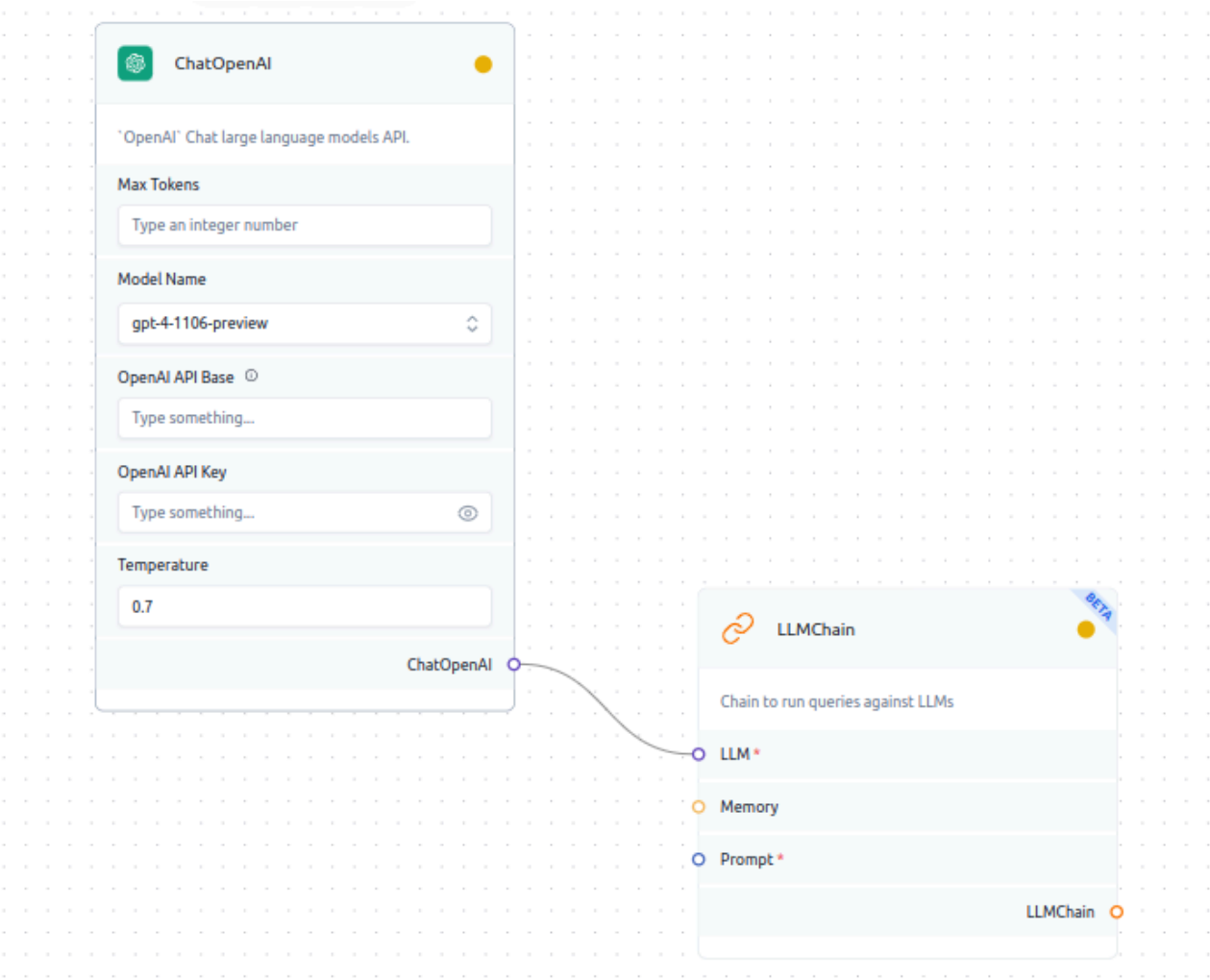
Universal Interface for LLMs

GenAI Stack doesn't keep its own LLMs in-house. Instead, it has something like a universal remote control that lets it connect with LLMs from big names like OpenAI, Anthropic, Cohere, and HuggingFace. This means you can pick and choose the best brain for the job without worrying about if it will fit with your setup. The LLM class in GenAI Stack is this universal remote, making sure that no matter which LLM you're talking to, the conversation stacks smoothly.

Why This Matters

This setup is great because it gives you flexibility. You're not stuck with just one LLM; you can use different ones for different tasks, all through a simple, consistent interface. This makes your AI smarter and more adaptable, ready to tackle a wide range of questions with the best knowledge available. Whether you're building a chatbot, a research assistant, or something entirely new, the GenAI Stack makes it easier to bring your ideas to life with the power of LLMs.

Example: ChatOpenAI LLM → Chain



Conclusion:

In this example we have shown only the OpenAI LLM but there are a bunch of other LLMs you can use as well . To know the list of LLMs supported by GenAI Stack you can refer the component documentation here.

Memory

While optional, memory stands out as a crucial component in this particular pipeline, especially when constructing a chatbot. It ensures that all conversations are stored in a buffer, enabling the chatbot to leverage chat history for more contextually relevant responses and facilitating the ability to ask follow-up questions.

Why We Need Memory

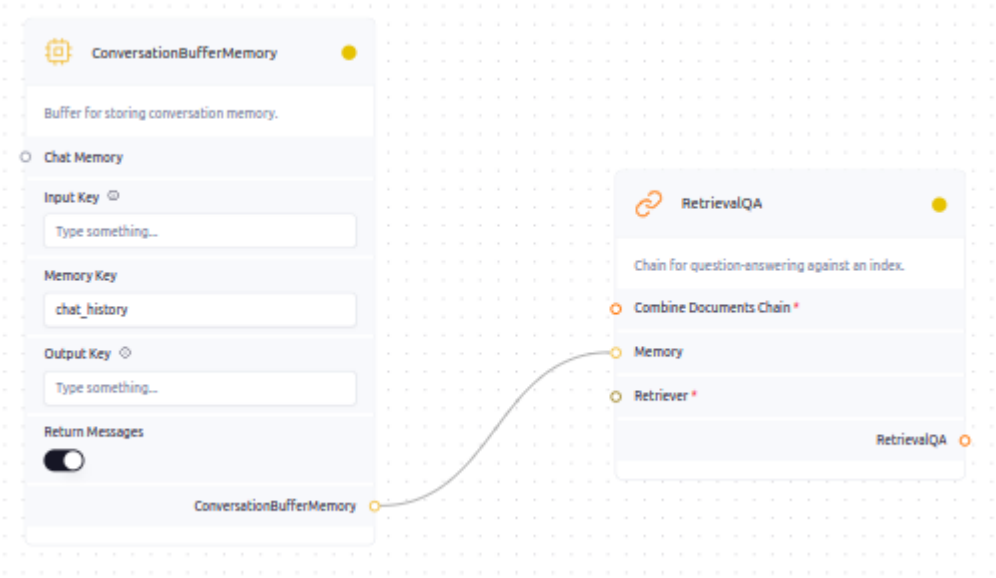
Memory plays a vital role in this pipeline, particularly in chatbot construction. It ensures that all conversations are stored in a buffer, allowing the chatbot to utilize chat history. This stored context enables the chatbot to provide responses that are more relevant to the ongoing conversation and facilitates the capability to ask follow-up questions.

The Use of Memory

Memory serves several crucial purposes:

- Enhanced Contextual Responses: By retaining previous conversation history, memory enables the chatbot to better understand the context of current interactions.
- Seamless Conversation Flow: With access to past interactions, memory helps maintain continuity in conversations. The chatbot can recall previous topics discussed and smoothly transition between related subjects, creating a more natural and engaging user experience.
- Improved User Engagement: Memory enables the chatbot to build rapport with users by recalling previous interactions and incorporating this knowledge into ongoing conversations.

Memory is an optional part in almost all the chains. In order to store the history of the conversation, we make return messages to be true and directly connect to either RetrievalQA or LLMChain chain.



Conclusion

For more information on Memory, check the documentation here.

Chain

Chains in GenAI Stack are cohesive assemblies of interconnected and easily reusable components. They encapsulate a sequence of calls to various components such as models, document retrievers, other chains, etc., offering a streamlined and user-friendly interface to navigate through this sequence.

Why We Need Chains?

Chains serve as a mechanism to organize and connect various components within a system, facilitating the flow of data and instructions. This structured approach is crucial for the following reasons:

Understanding Context: Chains enable the system to comprehend not only individual components but also their interrelationships and contextual significance. By linking components in a chain, the system can grasp the broader context of the task at hand, enhancing its overall understanding and performance.

The Use of Chains

Chains serve as a mechanism to organize and connect various components within a system, facilitating the flow of data and instructions.Chains offer several key benefits in system design and operation:

Improving Workflow Efficiency: By organizing components into a chain, the system can streamline its workflow and optimize resource utilization. This structured approach enhances the efficiency of data processing and task execution, leading to improved overall performance.

Customizing Responses: Chains allow for the customization of system behavior and responses based on specific requirements or input conditions. By configuring the sequence and interaction of components within a chain, the system can tailor its responses to better suit the desired outcomes or user preferences.

Now, it's time to integrate all the remaining components into the RetrievalQA chain. To construct this chain, connect the Memory, Large Language Models (LLM), and Retriever. It's important to note that direct connection of LLM to the RetrievalQA chain is not possible; instead, we employ an additional chain known as CombinedDocsChain for this purpose.

Integrate all the components using chain

Conclusion

Here we have mentioned a simple chain flow, To know more about chains please refer the documentation here .
To know more about usecases please look at the Usecases section here.

Testing Stack

Build Stack

After ensuring proper connection of all components, initiate the stack by clicking on the run icon (⚡) to compile the connections.

Build Stack

Upon a successful build, navigate to the chat icon to commence interaction.

Interact with your stack

Chat Interface

GenAI Stack's chat interface provides a user-friendly experience and functionality to interact with the model and customize the prompt. The sidebar brings options that allow users to view and edit pre-defined prompt variables(query). This feature facilitates quick experimentation by enabling the modification of variable values right in the chat.

To see the full details of the prompt in the format it was originally submitted, simply click on "Display Prompt." This option allows you to view the prompt in its exact original structure.

Feel free to thoroughly test your stack by asking as many questions as possible before proceeding to deploy and share the application with others.

Deployment

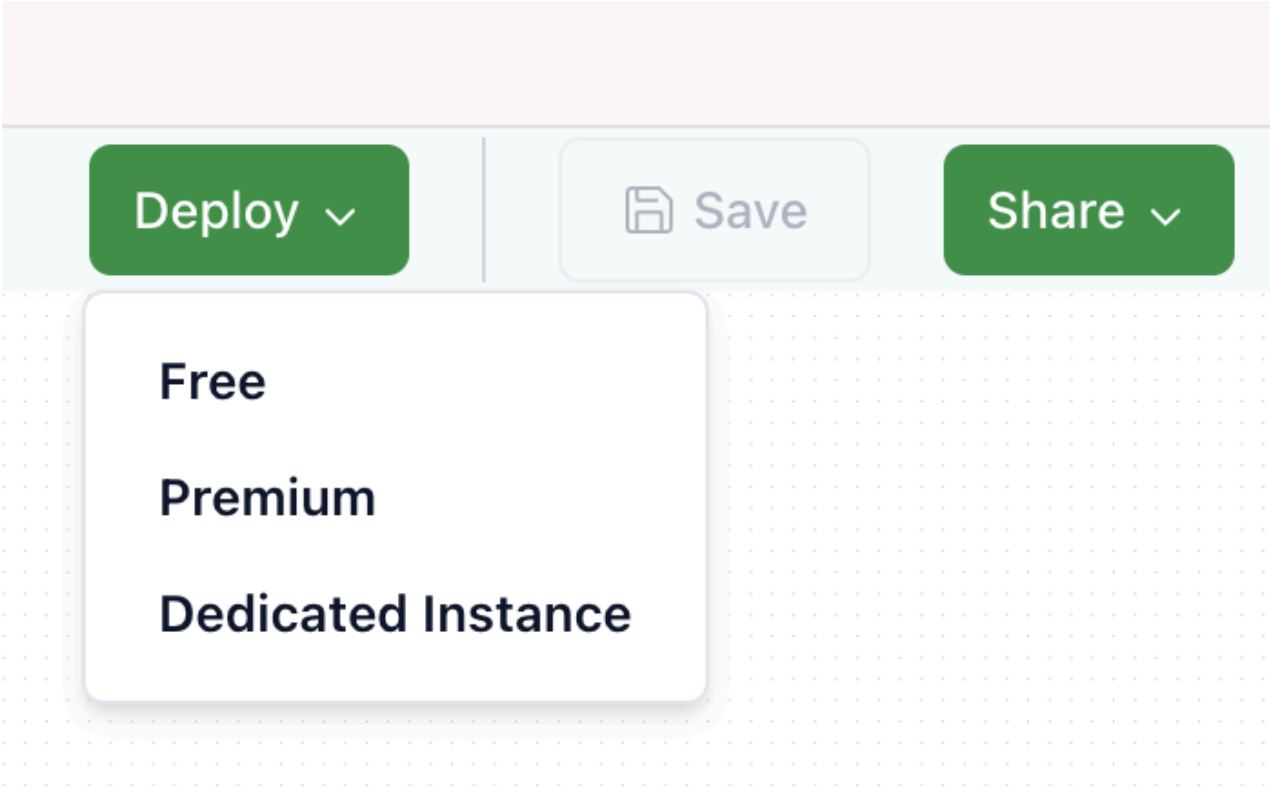
Deploying your stack is a crucial step in transitioning your chatbot application from the testing phase to live usage. This process involves setting up the application on a server and making it accessible to users through the web.

How to Deploy Your Stack

1. **Initiate Deployment**:
    - After thoroughly testing your application through the Chat Interface, click on the "Deploy" button.

2. **Deployment Options**:

When you click on the "Deploy" button, a dropdown menu will appear with three options:



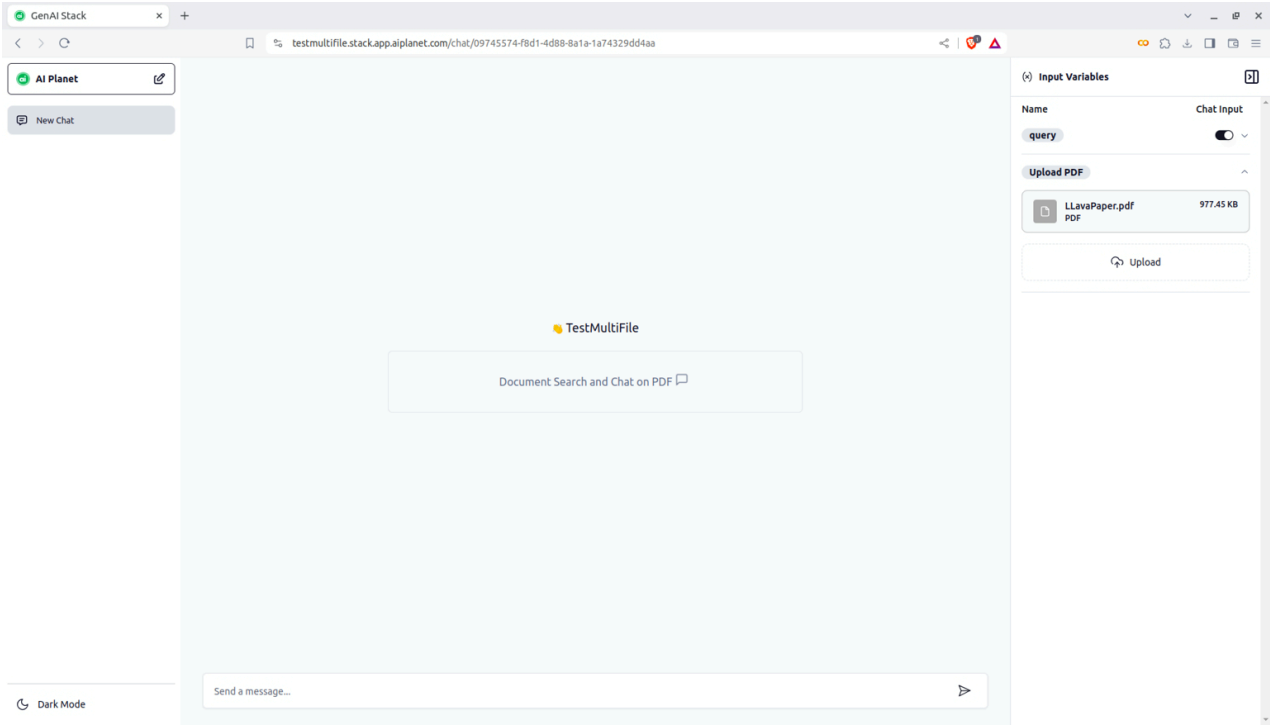
- **Free:**
  - The stack will be deployed and will automatically shut down after 2 hours.
  - Suitable for short-term testing and demonstrations.
- **Premium:**
  - The stack will be deployed and will remain active indefinitely.
  - Ideal for continuous use without worrying about downtime.
- **Dedicated:**
  - The stack will be deployed on a dedicated server.
  - You can choose a custom domain.
  - Autoscaling options are available, allowing the application to handle varying loads efficiently.
  - This option is best for high-traffic applications needing robust performance.

3. **Generate and Share Link**:

- Upon successful deployment, a shareable link will be generated.
- You can share this link to allow others to access and interact with your application easily.

Deploying your chatbot application effectively ensures it is accessible, reliable, and ready for user interaction. Choose the deployment option that best fits your needs and get started today!

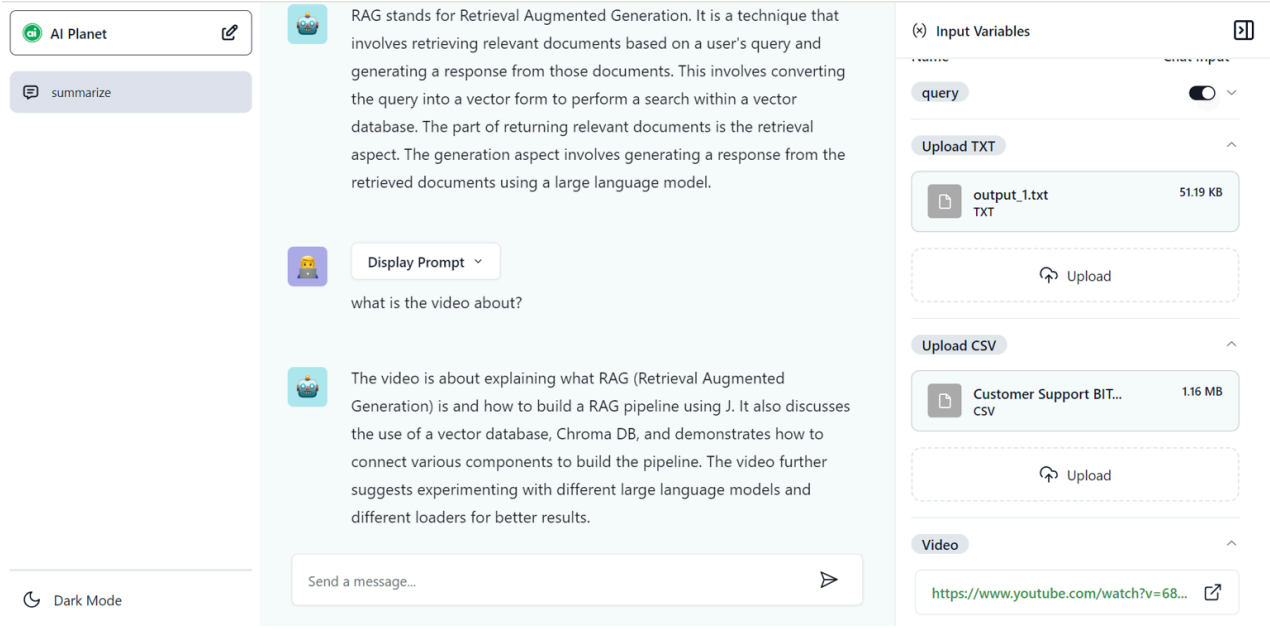
Chat



Chat interface with upload sidebar

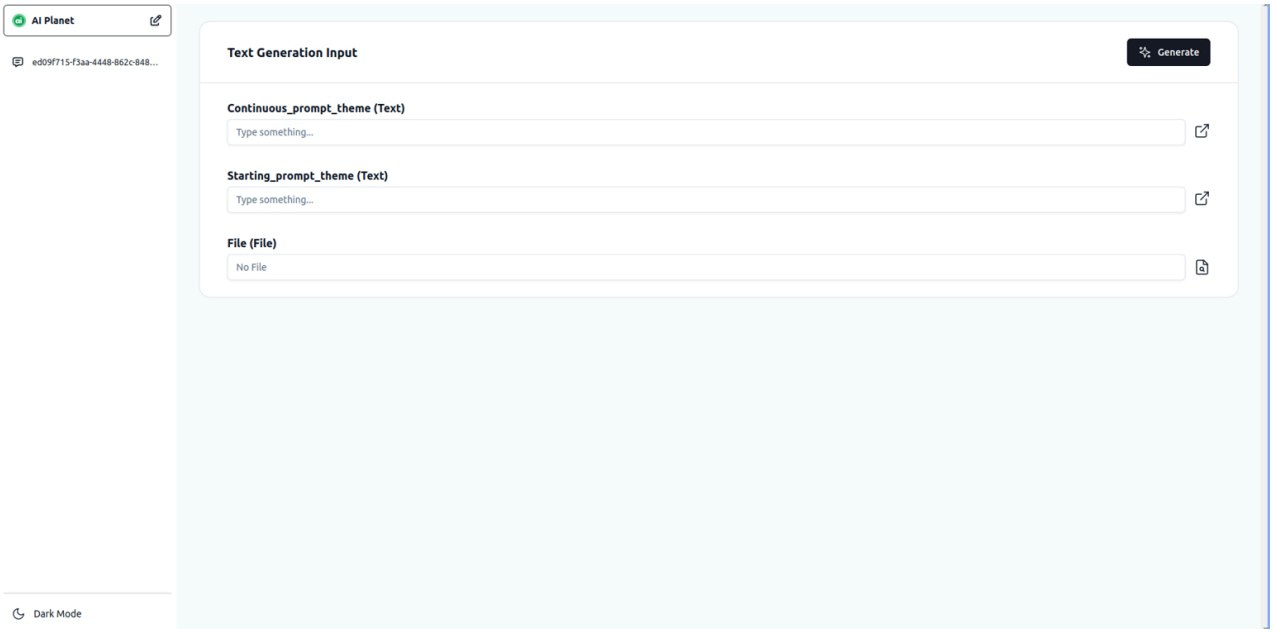
As mentioned in Input components, if you want to allow user to upload their own Inputs and the Input Component is attached to the Document Loader when you build the stack, you will see an Input Upload sidebar. Your own Inputs can be uploaded here before beginning the chat!

The Input Variables of the Chat are the **Queries** inputted by the user for chat and the **Multiple Files** that are to be uploaded by the user for the same, as shown below:



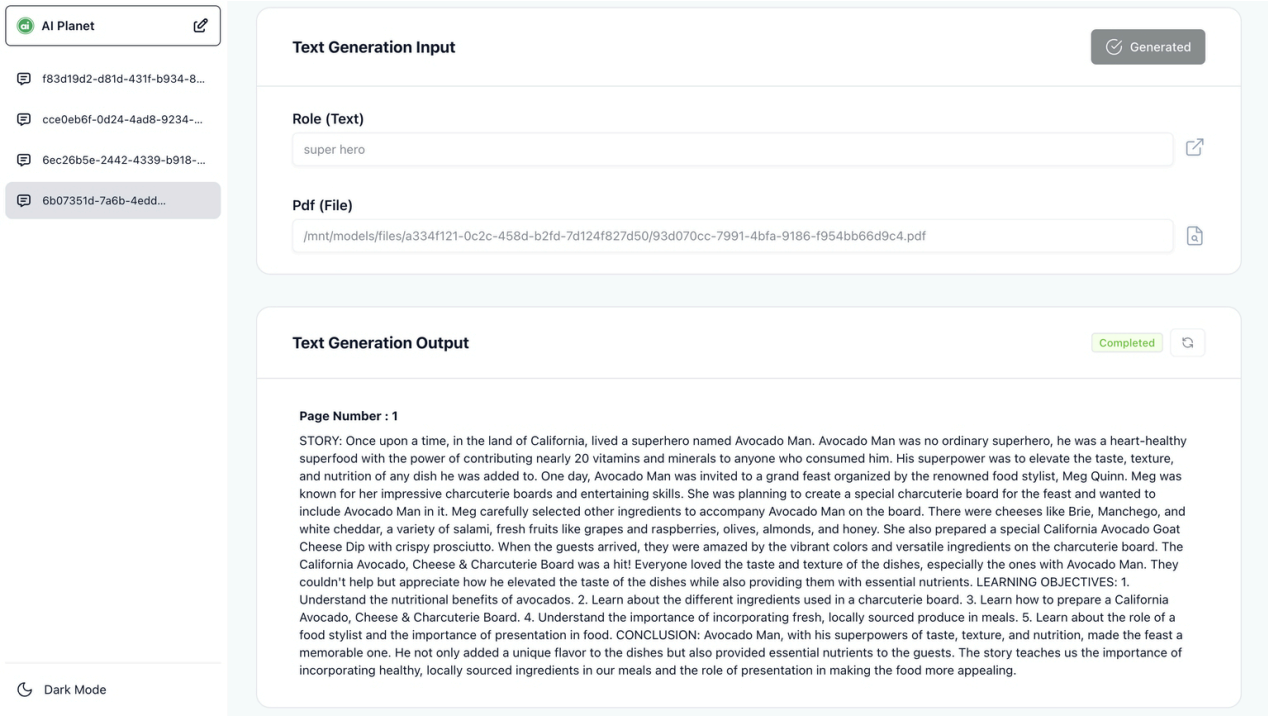
Multiple Inputs can also be uploaded one by one! Multiple types of Files can also be uploaded at the same time in this interface like PDFs, Text, URLs, CSVs, Videos etc to create the final chat. As shown in the Image below, multiple instances of each type of file can be uploaded by the user in the Upload section.

Text generation



Text generation interface

After entering the required inputs you can click on the generate button . After the generation is completed the generated text will appear in the bottom.


AI Planet
f83d19d2-d81d-431f-b934-8...
cce0eb6f-0d24-4ad8-9234-...
6ec26b5e-2442-4339-b918-...
6b07351d-7a6b-4edd...
Dark Mode
Text Generation Input
Generated
Role (Text)
super hero
Pdf (File)
/mnt/models/files/a334f121-0c2c-458d-b2fd-7d124f827d50/93d070cc-7991-4bfa-9186-f954bb66d9c4.pdf
Text Generation Output
Completed
Page Number : 1
STORY: Once upon a time, in the land of California, lived a superhero named Avocado Man. Avocado Man was no ordinary superhero, he was a heart-healthy superfood with the power of contributing nearly 20 vitamins and minerals to anyone who consumed him. His superpower was to elevate the taste, texture, and nutrition of any dish he was added to. One day, Avocado Man was invited to a grand feast organized by the renowned food stylist, Meg Quinn. Meg was known for her impressive charcuterie boards and entertaining skills. She was planning to create a special charcuterie board for the feast and wanted to include Avocado Man in it. Meg carefully selected other ingredients to accompany Avocado Man on the board. There were cheeses like Brie, Manchego, and white cheddar, a variety of salami, fresh fruits like grapes and raspberries, olives, almonds, and honey. She also prepared a special California Avocado Goat Cheese Dip with crispy prosciutto. When the guests arrived, they were amazed by the vibrant colors and versatile ingredients on the charcuterie board. The California Avocado, Cheese & Charcuterie Board was a hit! Everyone loved the taste and texture of the dishes, especially the ones with Avocado Man. They couldn't help but appreciate how he elevated the taste of the dishes while also providing them with essential nutrients. LEARNING OBJECTIVES: 1. Understand the nutritional benefits of avocados. 2. Learn about the different ingredients used in a charcuterie board. 3. Learn how to prepare a California Avocado, Cheese & Charcuterie Board. 4. Understand the importance of incorporating fresh, locally sourced produce in meals. 5. Learn about the role of a food stylist and the importance of presentation in food. CONCLUSION: Avocado Man, with his superpowers of taste, texture, and nutrition, made the feast a memorable one. He not only added a unique flavor to the dishes but also provided essential nutrients to the guests. The story teaches us the importance of incorporating healthy, locally sourced ingredients in our meals and the role of presentation in making the food more appealing.

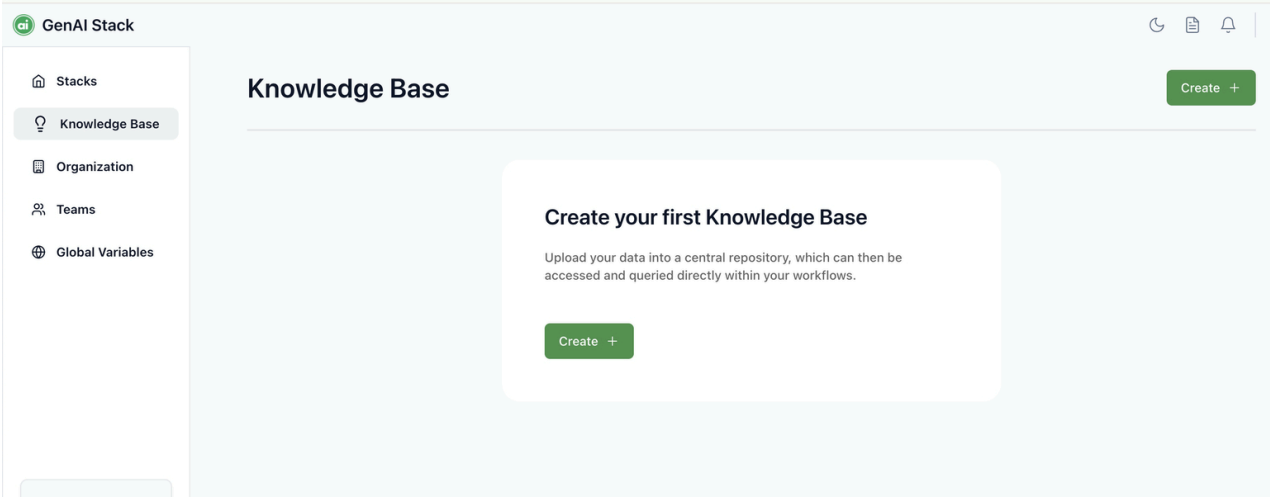
Knowledge Base

The Knowledge Base feature in GenAI Stack is designed to centralize and manage your data effectively, ensuring it is easily accessible and queryable within your workflows. This component empowers you to integrate and utilize your data seamlessly across various GenAI Stack modules. Below is a step-by-step guide to creating your first Knowledge Base, along with detailed explanations of key configurations and an example to illustrate its integration into your overall workflow.

Steps to Create a Knowledge Base:

1. **Access the Knowledge Base Section:**

- Navigate to the "Knowledge Base" option on the main page of your GenAI Stack site.



2.

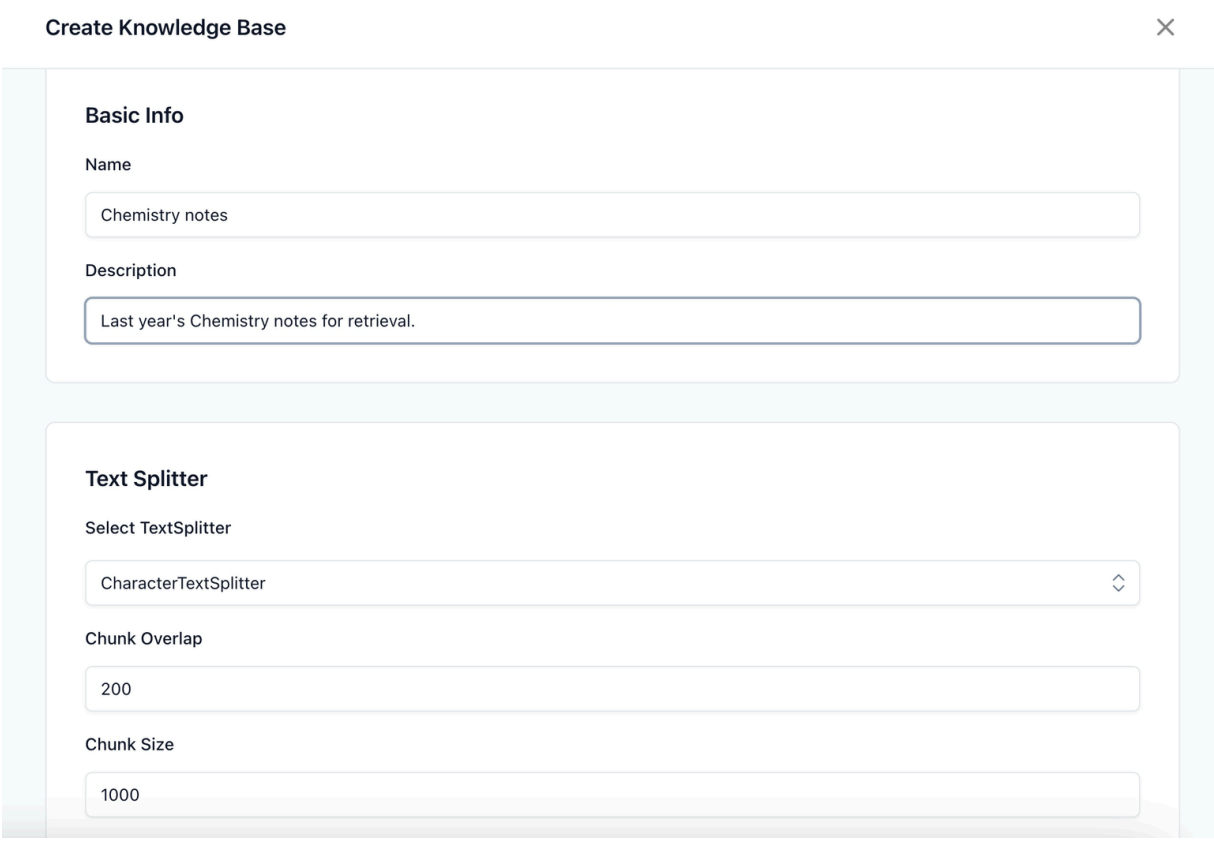
- Click on the "Create" button to begin setting up your Knowledge Base.

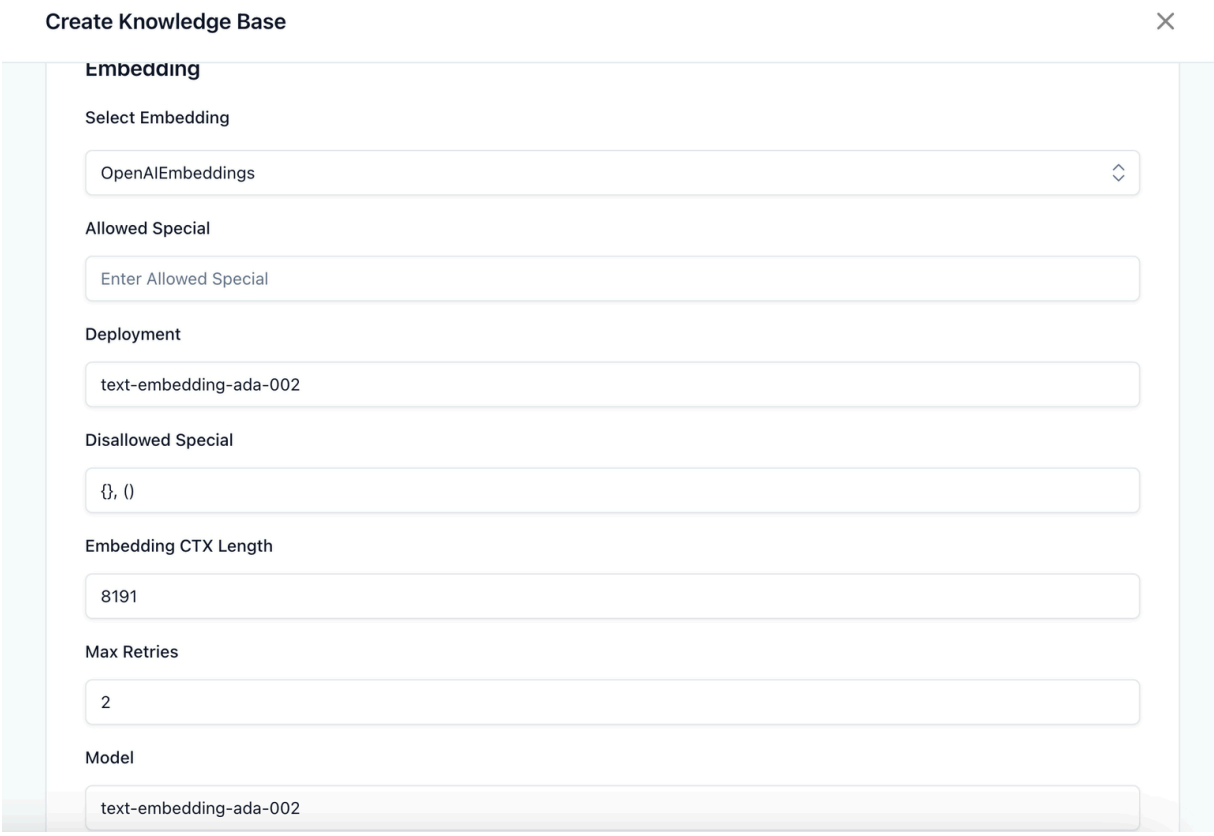
3. **Fill in the Required Details:**

    - **Name:** Enter a name for your Knowledge Base. This should be descriptive enough to identify the data it contains.
    - **Description:** Provide a brief description of the Knowledge Base's purpose and contents. This helps in understanding the context and scope of the

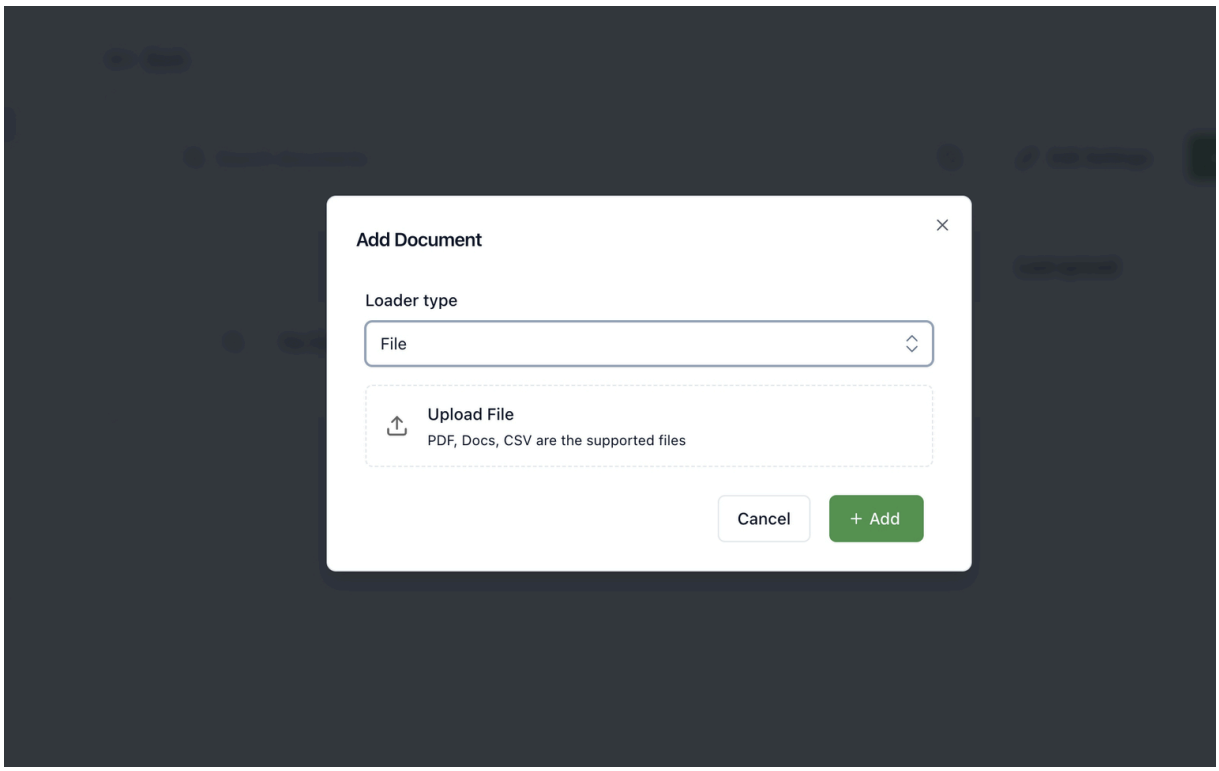
data.

- **Text Splitting:** Specify how you want the text data to be split. This configuration determines how the text is chunked into smaller, manageable pieces. For example, you can split the text by sentences, paragraphs, or custom delimiters.
- **Embedding Details:** Choose the embedding method to represent your data in vector form. Embeddings transform your textual data into numerical vectors that can be easily processed by machine learning models.





4. **Upload Your Data:**
    - You can upload files directly to populate your Knowledge Base. Supported file types include PDFs, Docs, and CSVs. Additionally, you can provide URLs to web pages, which will be scraped and included in your Knowledge Base.
    - **Example:** Suppose you are creating a Knowledge Base for AI research articles. You can upload a combination of PDFs of research papers, CSVs of bibliographic data, and URLs of relevant web pages.

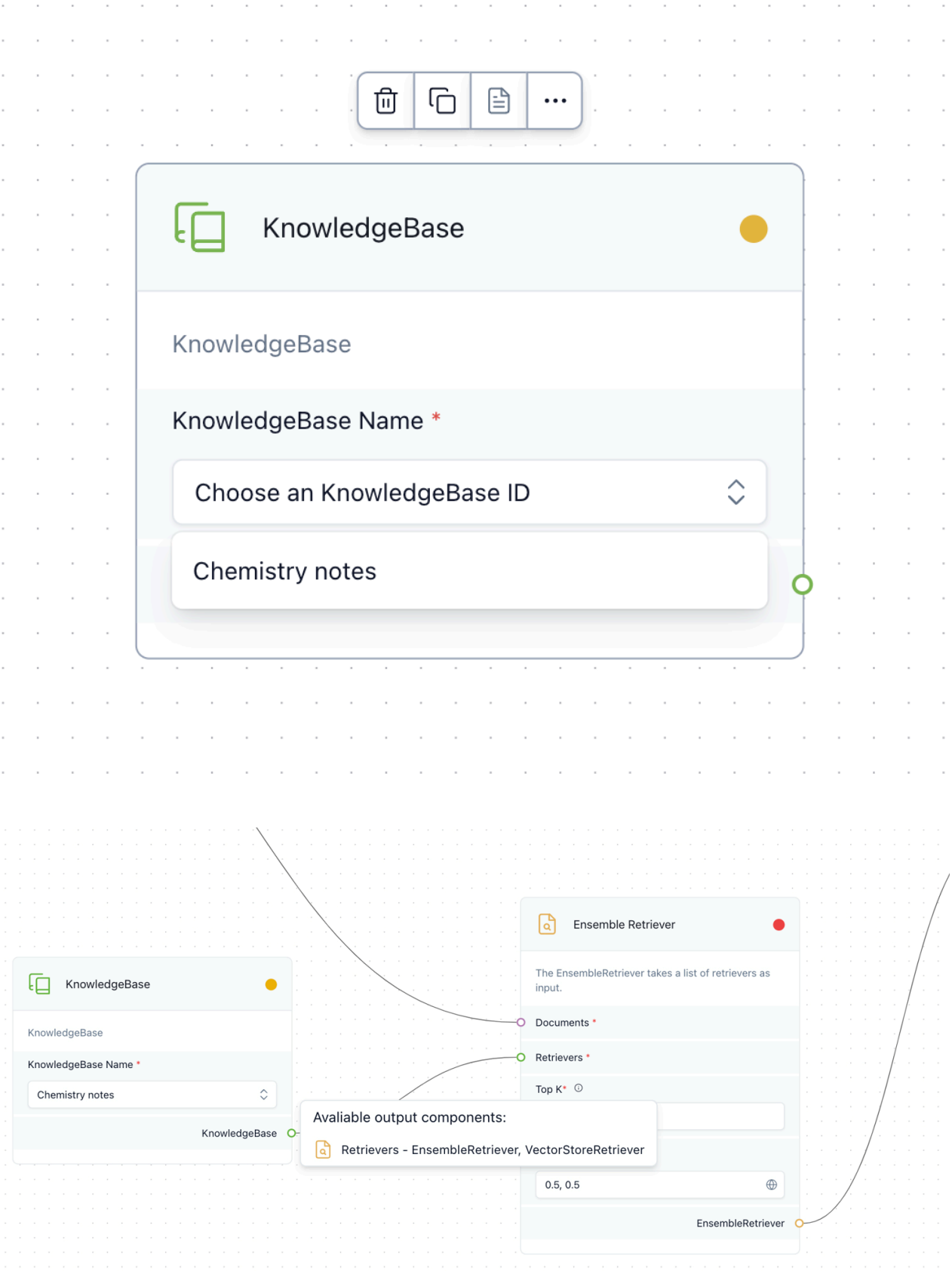


Workflow Integration

After setting up your Knowledge Base, you can connect it to other components within the GenAI Stack to enhance your application's functionality. For instance, you can link the Knowledge Base to an Ensemble Retriever for efficient data retrieval and analysis.

Example:

- **Ensemble Retriever Integration:** The screenshots below demonstrates how a Knowledge Base named "Chemistry Notes" can be connected to an Ensemble Retriever. This retriever can combine outputs from multiple retrieval models, such as the EnsembleRetriever and VectorStoreRetriever, to provide comprehensive and accurate results.


KnowledgeBase
KnowledgeBase
KnowledgeBase Name *
Choose an KnowledgeBase ID
Chemistry notes
KnowledgeBase
KnowledgeBase
KnowledgeBase Name *
Chemistry notes
KnowledgeBase
Avaliable output components:
Retrievers - EnsembleRetriever, VectorStoreRetriever
Ensemble Retriever
The EnsembleRetriever takes a list of retrievers as input.
Documents *
Retrievers *
Top K*
0.5, 0.5
EnsembleRetriever

Organization and Teams

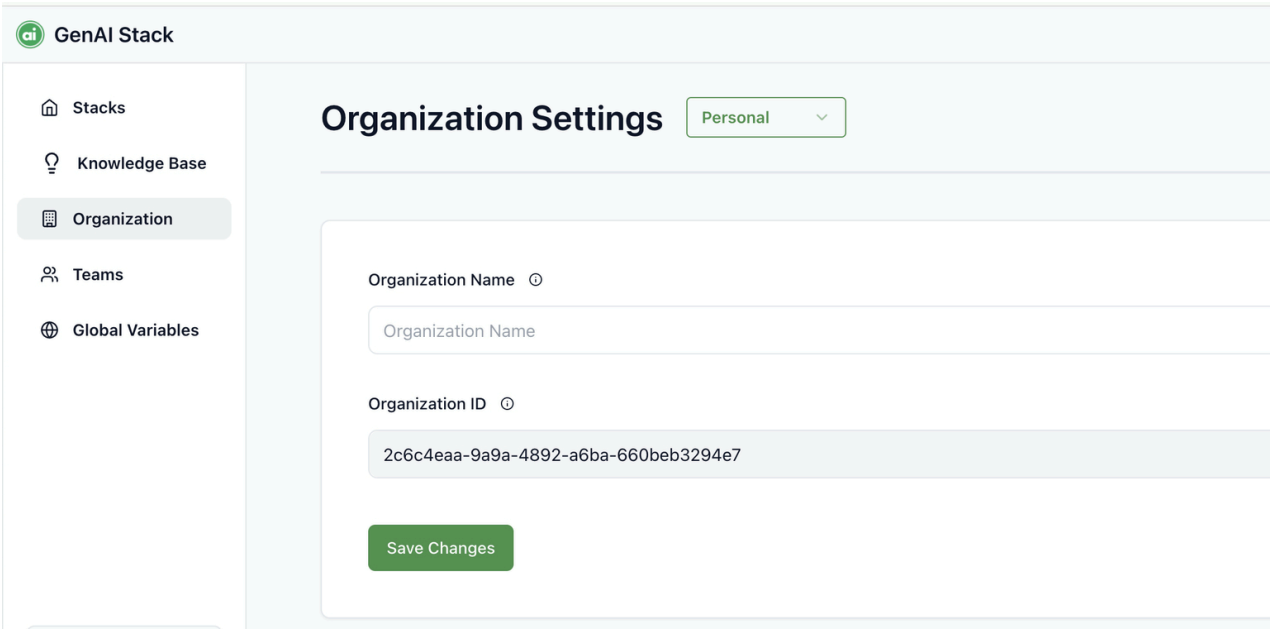
The Organization and Teams feature in GenAI Stack allows you to manage and collaborate with your team efficiently. This guide provides step-by-step instructions on setting up your organization, creating teams, and inviting team members.

Steps to Create an Organization:

1. **Access the Organization Section:**
   - Navigate to the "Organization" option on the main page of your GenAI Stack site.
2. **Create a New Organization:**

   Click on the "Create Organization" button.

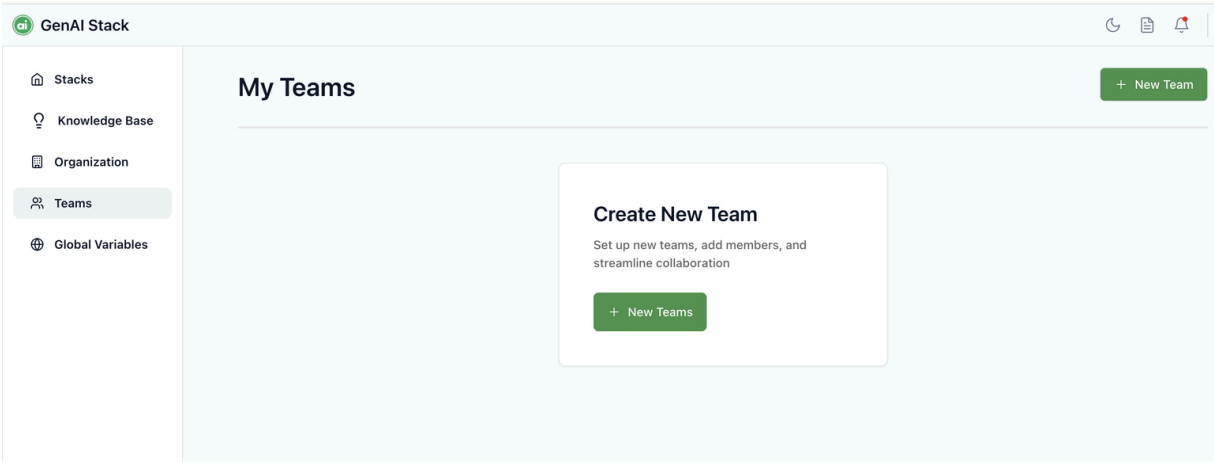
- **Organization Name:** The name of your organization will be pre-filled, but you can update it if needed.
- **Organization ID:** This will also be pre-filled for you.



Steps to Create a Team:

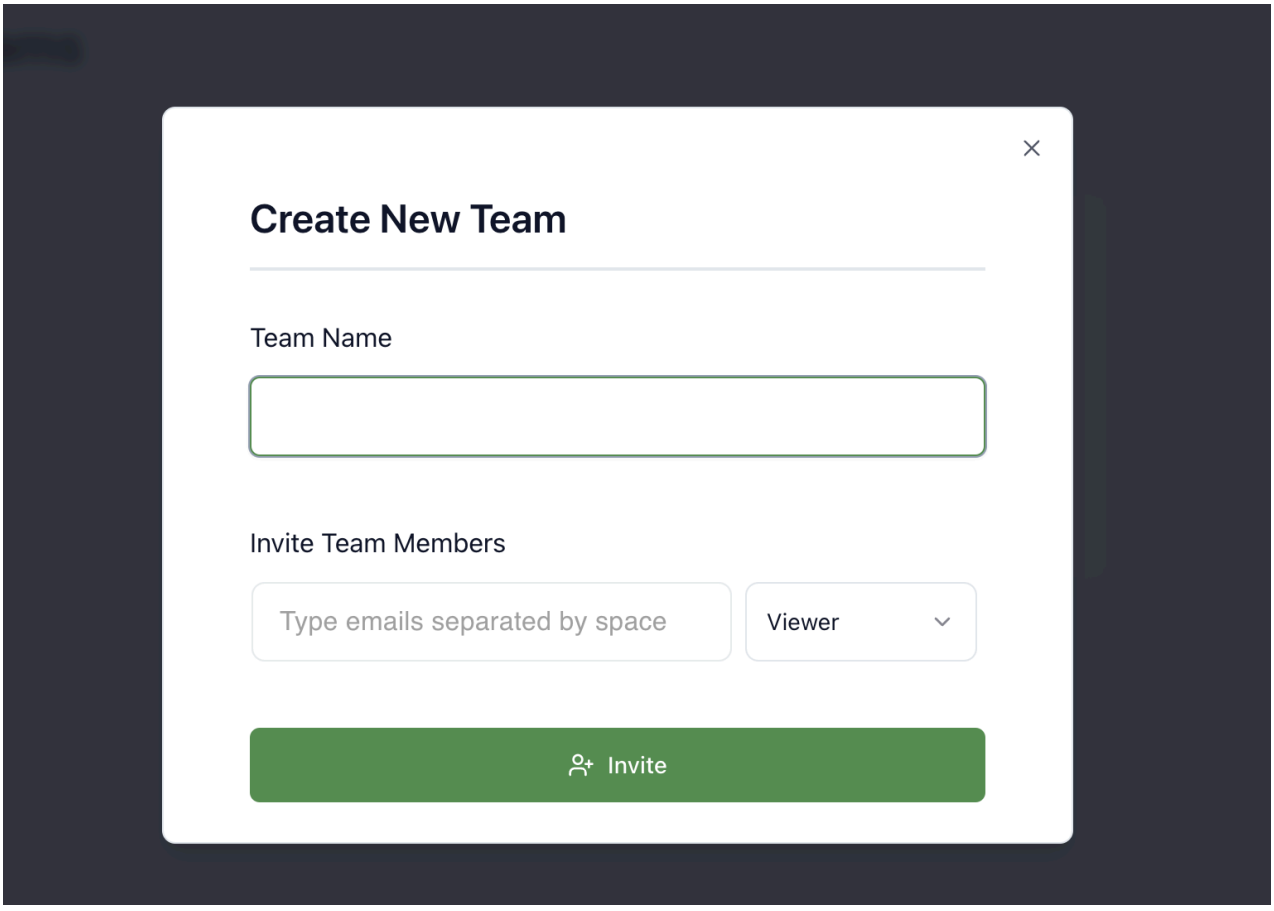
1. **Access the Teams Section:**

- Navigate to the "My Teams" option on the main page of your GenAI Stack site.

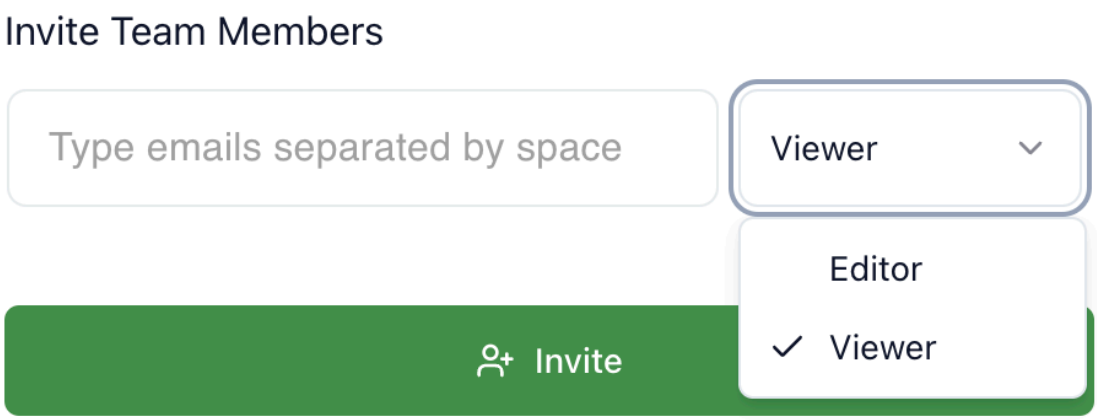


2. **Create a New Team:**

- Click on the "Create New Team" button.
- **Team Name:** Enter the name of your team.
- **Invite Team Members:** Add the email IDs of the team members you want to invite.
- Click on the "Invite" button to send invitations to your team members.

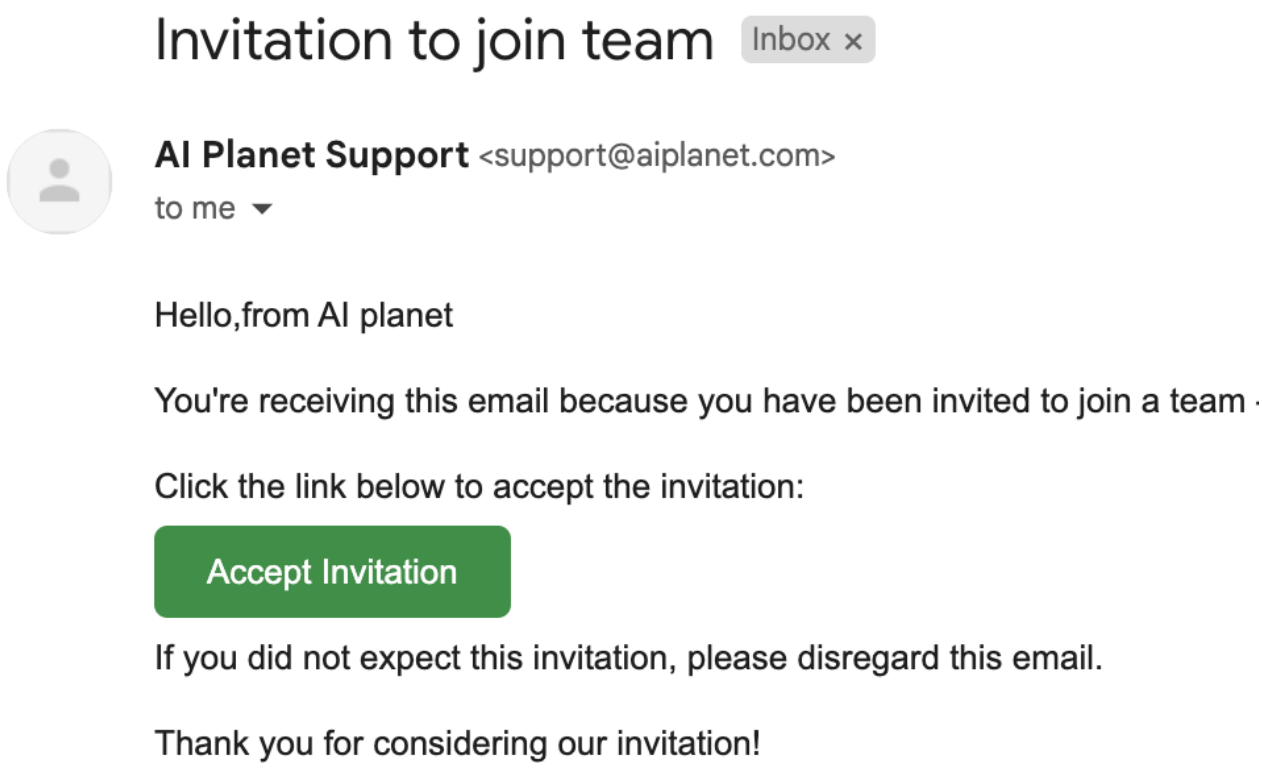


- Choose the role of each invitee from a dropdown menu: **Editor** or **Viewer**.



Invitation Flow

- Once you send an invitation, team members will receive an email with a link to join the team.
- They can click on the link in the email to accept the invitation and join your team.



Switch Organization Feature

- If you are part of multiple organizations, you can easily switch between them.
- Navigate to the "Organization" section and select the organization you want to switch to from the list.

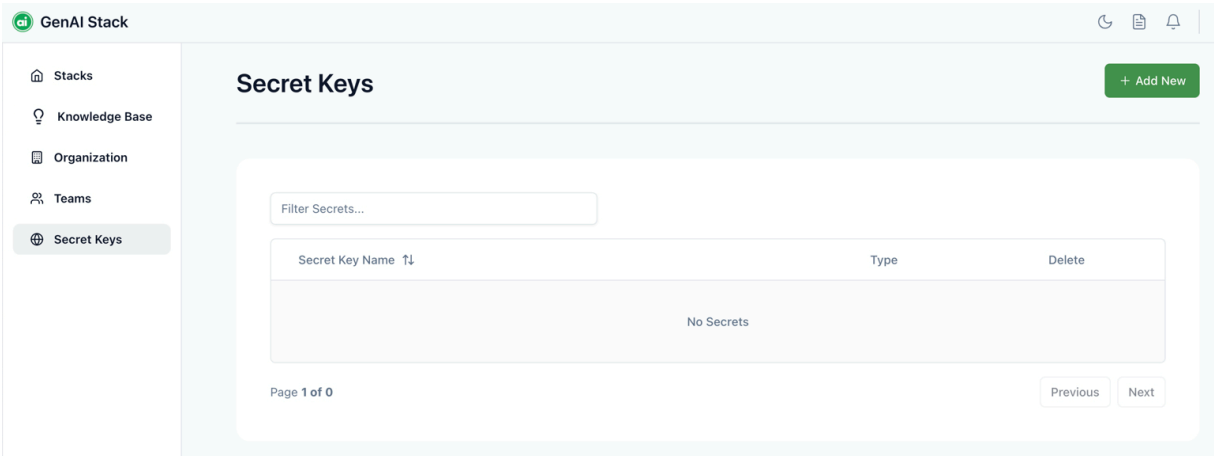
The Organization and Teams feature in GenAI Stack is designed to streamline team management and collaboration. By following the steps above, you can effectively set up your organization, create teams, and invite members, ensuring everyone has the necessary access and resources to work efficiently within the GenAI Stack

Secret Keys

The Secret Keys feature allows you to define and manage variables that can be used throughout the GenAI Stack, ensuring consistency and ease of access. Follow these steps to create and manage secret keys.

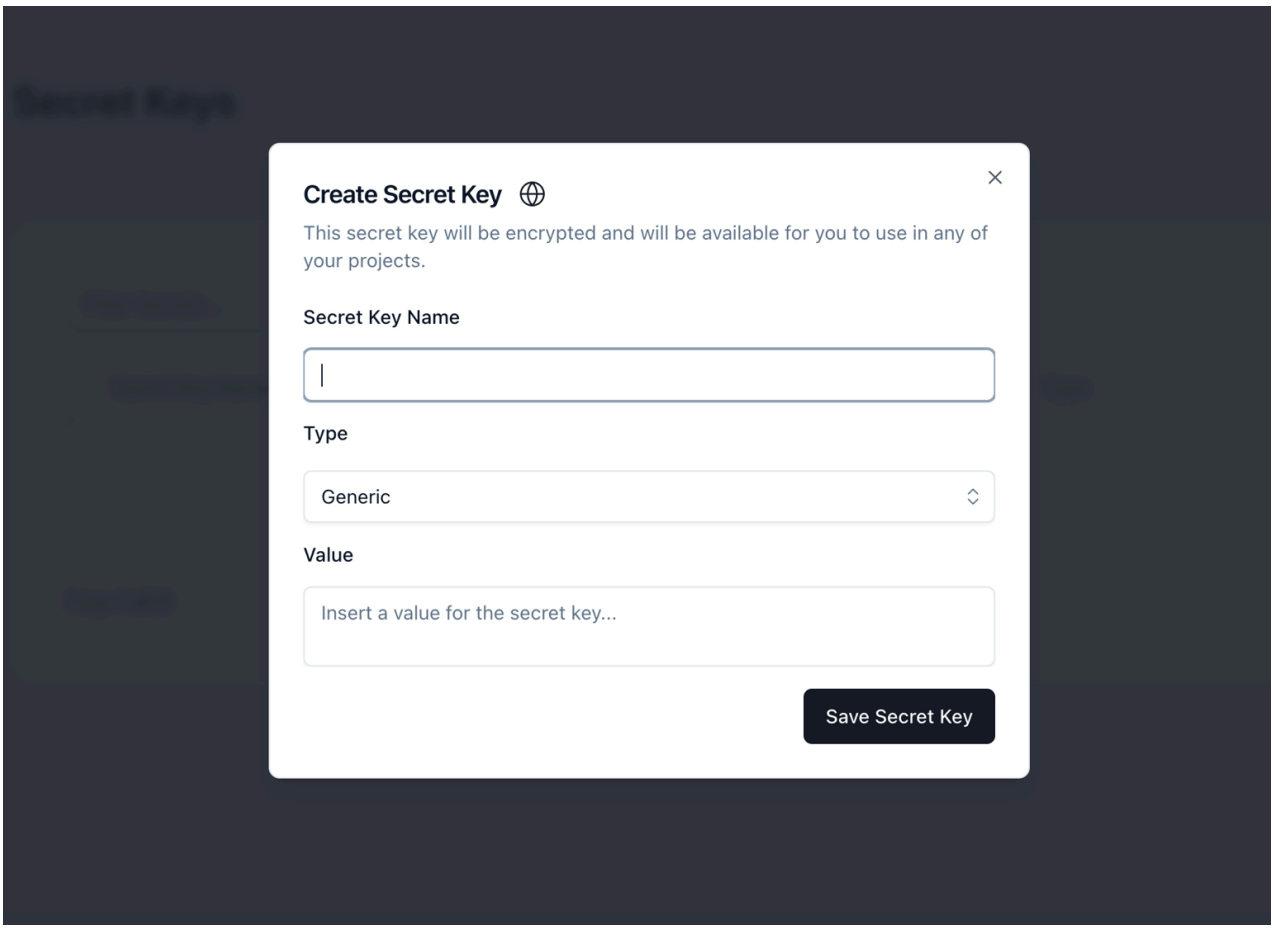
Steps to Create a Secret Key:

1. **Access the Secret Keys Section:**
   - Navigate to the "Secret Keys" option on the main page of your GenAI Stack site.



2. **Add a New Secret Key:**

- Click on the "Add New" button.
- **Secret Key Name:** Enter a name for your key.
- **Type:** Select the type of key (either "Generic" or "Credential").
- **Value:** Provide the value for your variable.
- Click on the "Save Secret Key" button to save your new secret key.

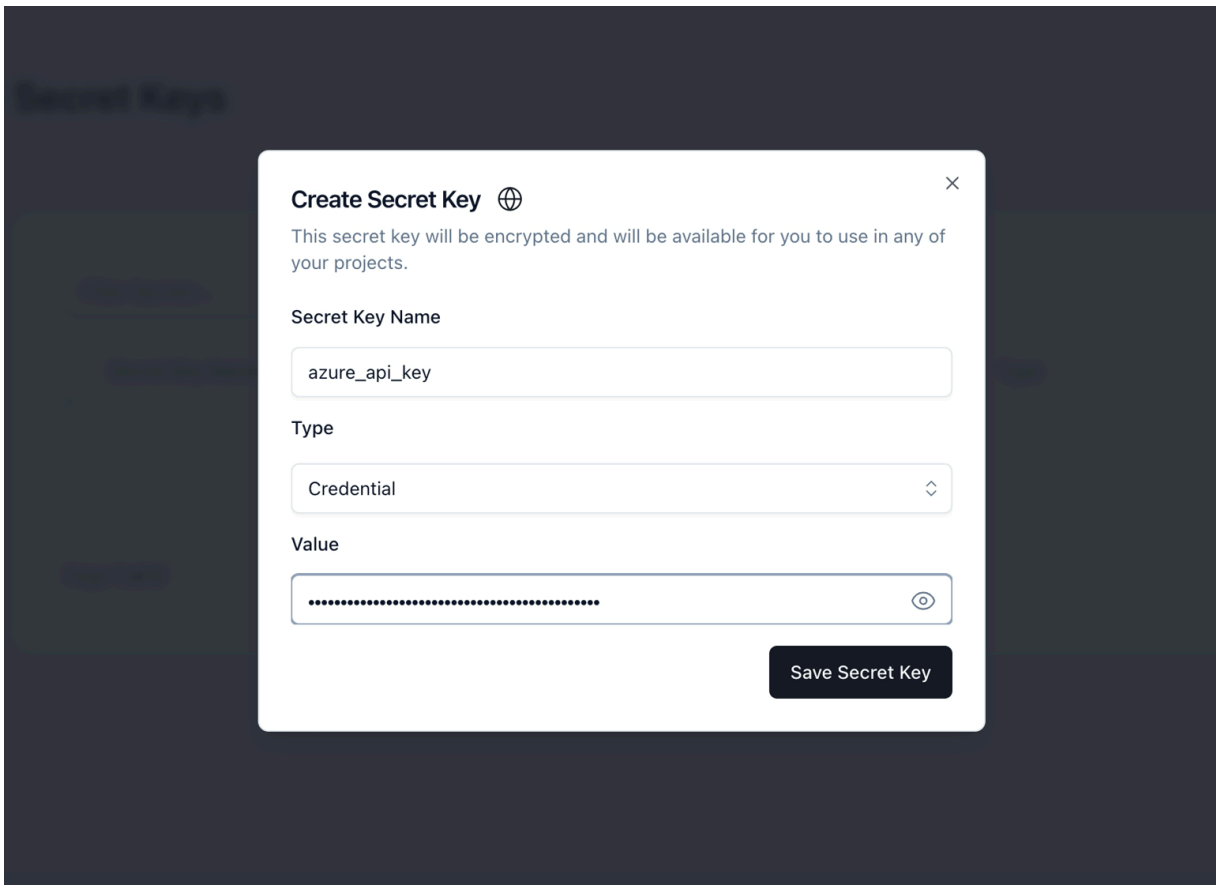


- These secret keys can then be used across various components within the GenAI Stack, making it easier to maintain consistency and streamline your workflows.

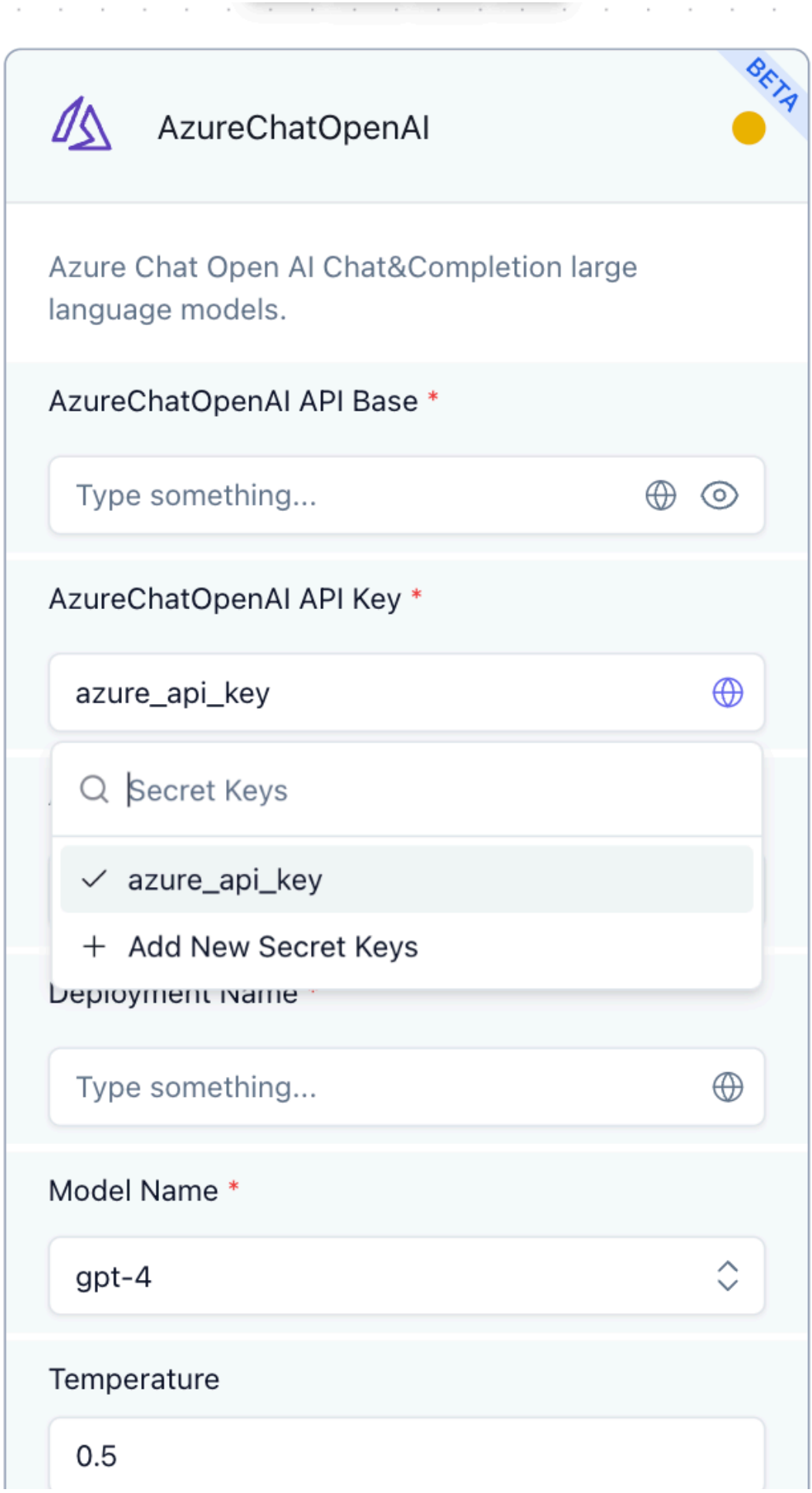
Using Secret Keys in Flows

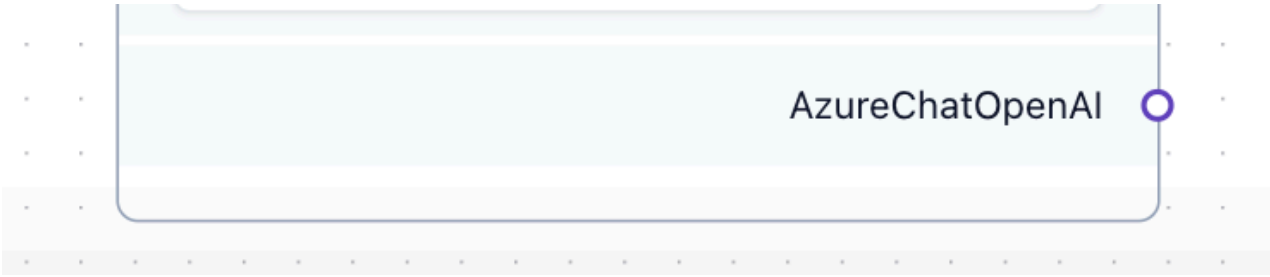
Example: Using Secret Keys for API Keys

1. **Storing API Keys:**
   - Navigate to the "Secret keys" section and create a new key with a name like `azure_api_key`.
   - Set the type to "Credential" and input your Azure API key as the value.
   - Save the variable.



2. **Accessing Secret Keys:**
    - In your deployment or configuration settings, when prompted to enter the API key, click on the globe icon next to the input field.
    - This will open a dropdown menu where you can select the pre-defined secret key (`azure_api_key` in this case).


BETA
AzureChatOpenAI
Azure Chat Open AI Chat&Completion large language models.
AzureChatOpenAI API Base *
Type something...
AzureChatOpenAI API Key *
azure_api_key
Secret Keys
azure_api_key
Add New Secret Keys
Deployment Name *
Type something...
Model Name *
gpt-4
Temperature
0.5



By using secret keys, you can easily update the value in one place, and it will automatically propagate to all components that reference this variable. This ensures consistency and simplifies management, especially when dealing with sensitive information like API keys.

More Use Cases:

1. **Configuration Settings:**
   - **Scenario:** You need to manage configuration settings that are used across multiple components.
   - **Secret Key Setup:** Create secret keys of type "Generic" to store configuration values like server URLs, timeout settings, or feature flags.
   - **Flow Integration:** Access these keys in different parts of your application to ensure consistent behavior and easy updates.
2. **Default Values for User Inputs:**
   - **Scenario:** You want to pre-fill forms or set default values in user interfaces.
   - **Secret Key Setup:** Create secret keys for default form values.
   - **Flow Integration:** Reference these keys in the UI components to automatically populate fields with the default data.

Managing Secret Keys:

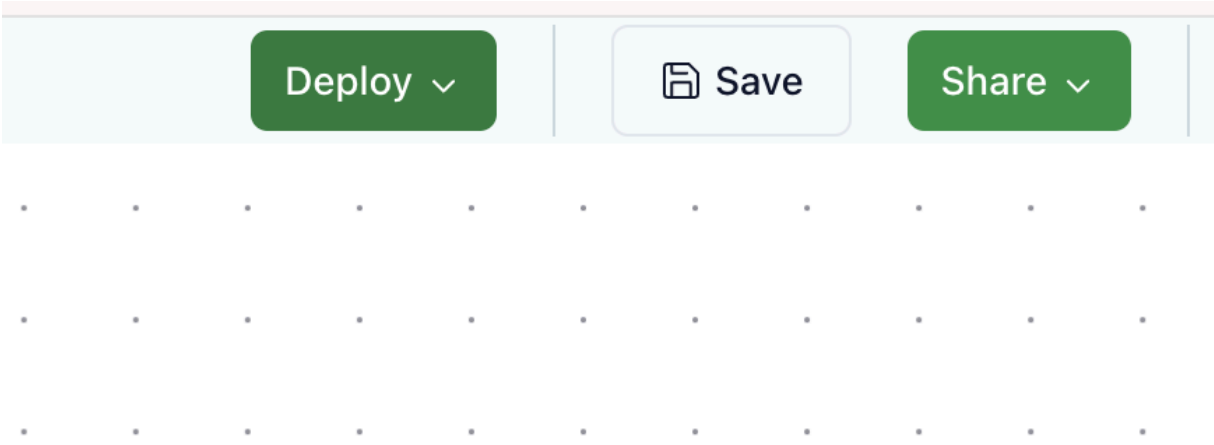
1. You can edit or delete secret keys as needed by navigating to the "**Secret Keys**" section and selecting the keys you want to manage.
2. This ensures your application stays up-to-date with the latest values without the need to change individual components.

Logs

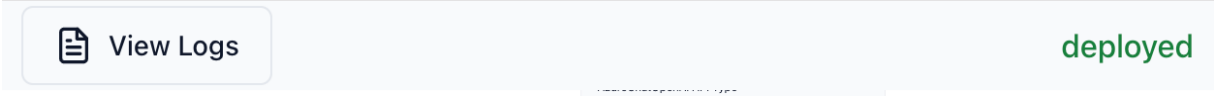
The GenAI Stack provides a comprehensive logging feature that allows users to view and manage logs generated during the deployment and execution of their AI workflows. These logs are essential for monitoring performance, debugging issues, and ensuring the smooth operation of your AI stacks. This documentation covers the steps to access and understand the logging features in GenAI Stack.

Accessing Logs

1. **Deploying Your Stack:**
   - After configuring your workflow and ensuring all components are correctly set up, click on the "Deploy" button to deploy your stack.



   - Once the deployment is successful, the status will change to "Deployed," as indicated in the top bar of the interface.
2. **Navigating to Logs:**
   - In the top bar of the GenAI Stack interface, you will see an option labeled "View Logs." This button becomes available once your stack is deployed.

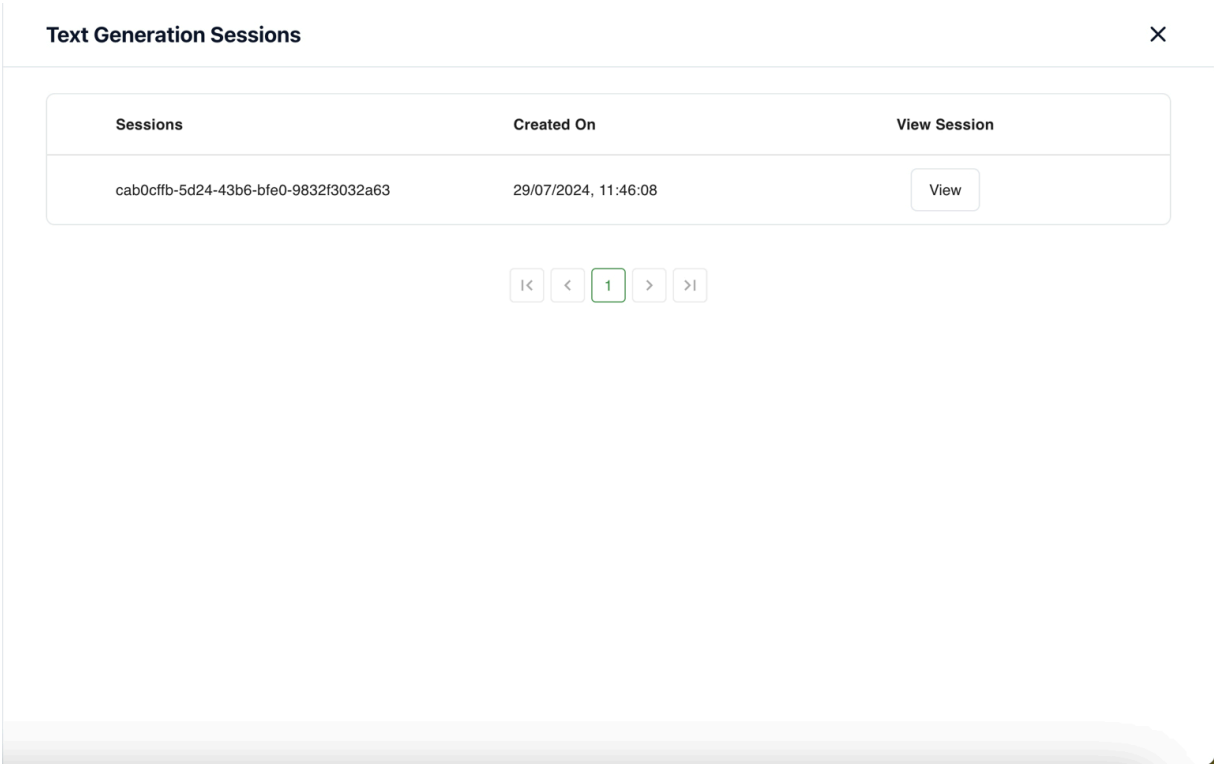


   - Click on "View Logs" to access the logs for the deployed stack.

Using the Logs Feature

1. **Log Overview:**

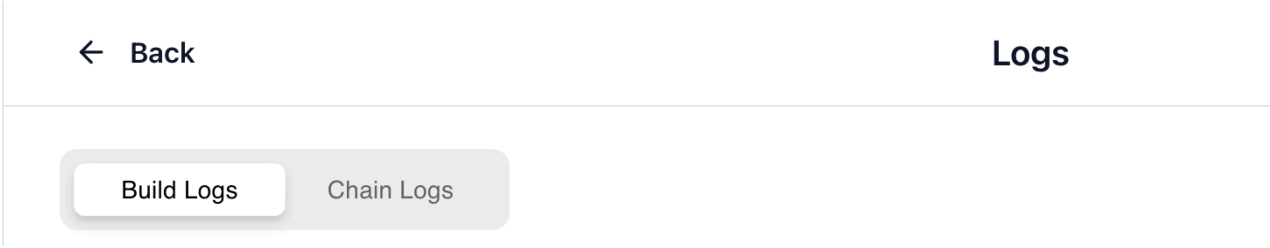
- The logs interface provides a comprehensive view of all activities and events that occur during the execution of your stack.
- You can see details such as timestamps, event types, and any error messages that might have been generated.



2. **Session Management:**
   - The logs are organized by sessions. Each session represents a distinct execution of your deployed workflow.
   - You can view the list of sessions in the "Text Generation Sessions" section, where each session has a unique identifier and a timestamp of when it was created.
3. **Viewing Specific Sessions:**
   - To view the details of a specific session, click on the "View" button next to the session ID.
   - This will open a detailed log of that particular session, where you can analyze the execution flow, inspect input and output data, and identify any issues.

Log Interface

The log interface is divided into two main sections:



1. **Build Logs**: These logs provide information about the build process of your AI workflows. It includes details about each component's initialization, data loading, and any errors or warnings encountered during the build process.
2. **Chain Logs**: These logs provide information about the execution of your AI workflows. It includes details about the data processing, model inference, and any outputs generated by the workflow components.

Summary

By providing detailed logs and an intuitive interface, GenAI Stack ensures that users can efficiently monitor and manage their AI workflows. The logs feature is crucial for maintaining the performance and reliability of your AI stacks, offering insights into both the build and execution phases. With the ability to view detailed session logs, users can better understand their workflows and address any issues that arise.

By following this guide, users can leverage the logging capabilities of GenAI Stack to their full potential, ensuring a smooth and effective AI development process.

Components

Inputs

Input

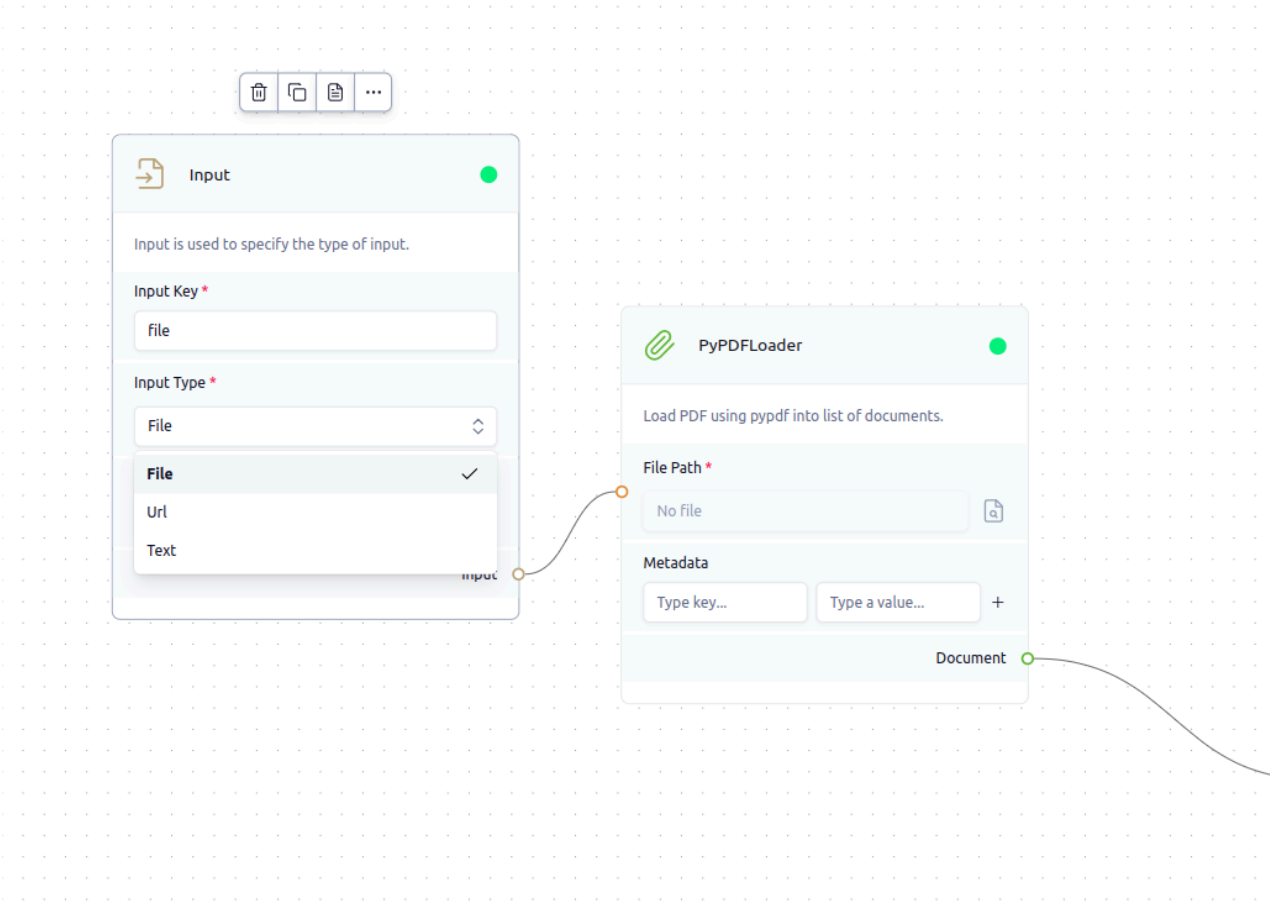
The Input serves as the entry point and allows users to specify the input. It provides flexibility by supporting three different types of input values, including Text, File, and URL.

Parameters

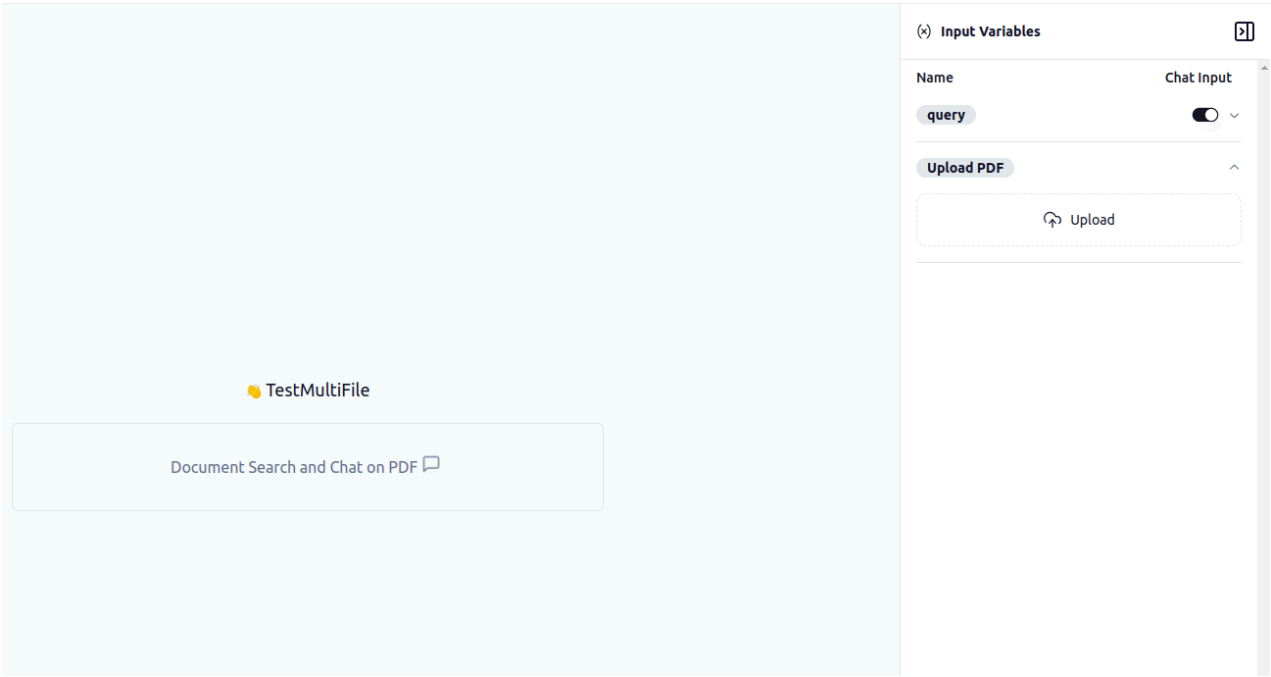
- **Input Key:** A unique identifier for the input value.
- **Input Type**: Specifies the type of input value, such as Text, File, or URL.
- **Input Value**: The actual input value, which can be a File path, URL, or Text. This parameter is optional.

Connecting an Input component will allow you to upload your file/URL/text(allows multiple files/URL/text as well) in your deployed chat interface. This feature won't be available unless the Input component is attached here. This is also required for the Text Generation stack type.

Adding an Input component in your Stack results in the Upload File option appearing in the Input variables sidebar on deployment.



Choosinng Input type and connecting to Document loader



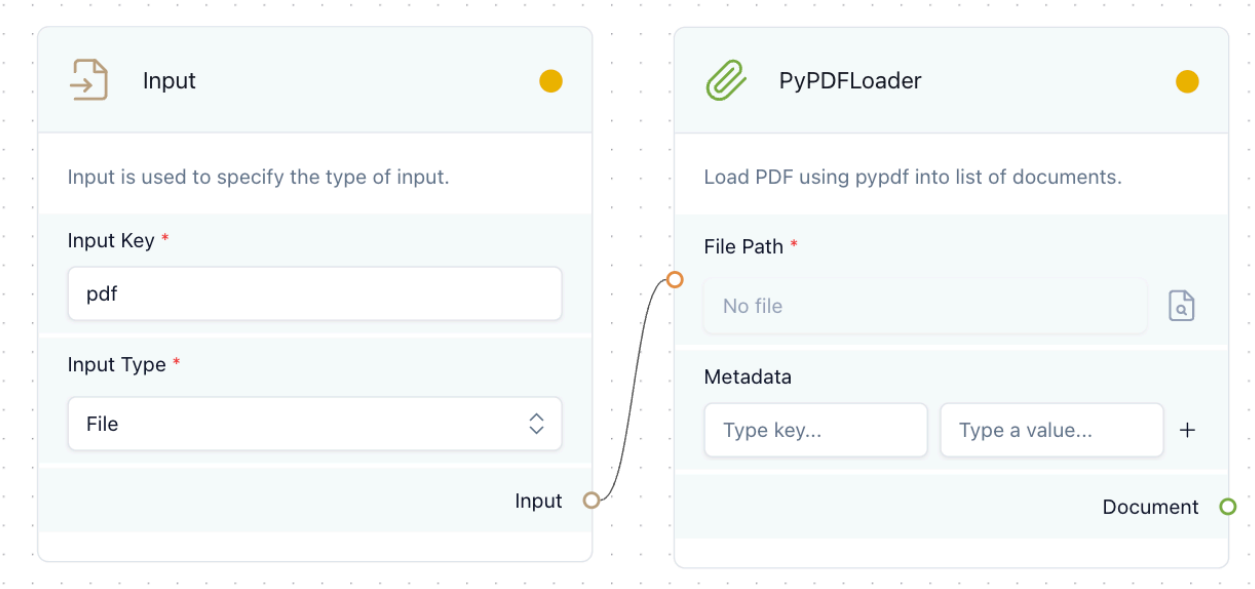
File upload option on deployment screen

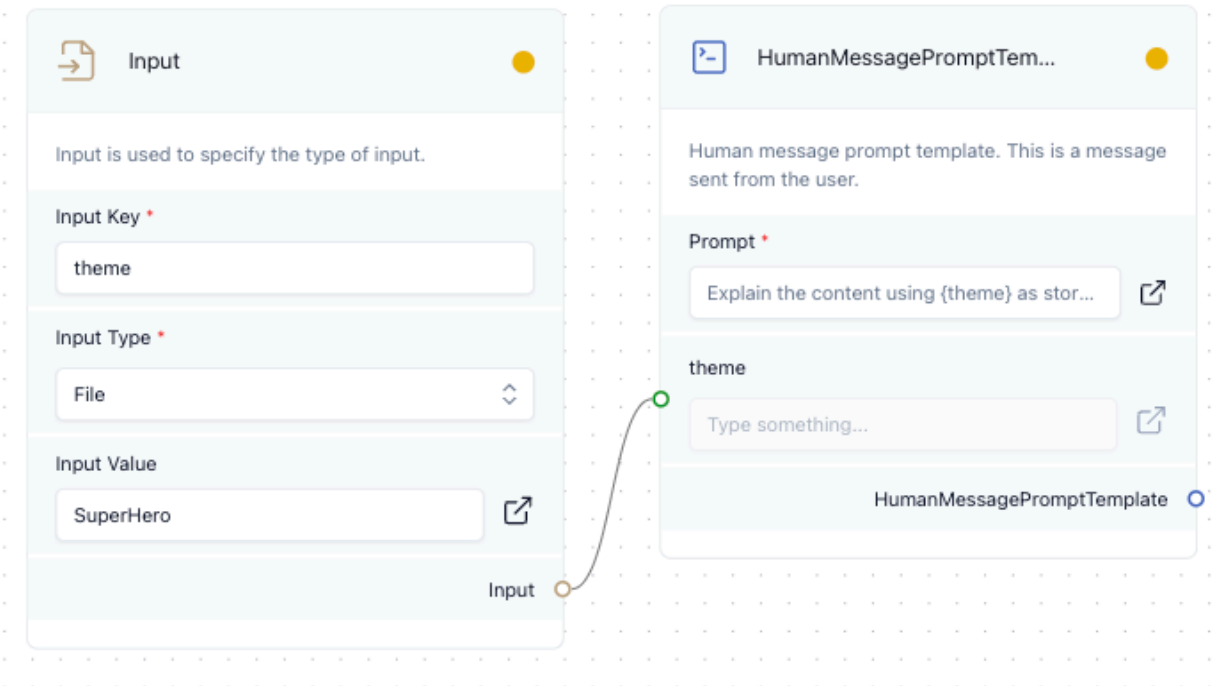
This is the result of including the Input component in your stack, since it was attached to a PDF file loader and the input is of File type, you get an option to

upload PDF on the sidebar. Similarly, different Input type will have their respective Input feature on the sidebar.

Example

In the below examples the Input component is the input to the PyPDFLoader and HumanPromptTemplate components.





Outputs

TextGenerationOutput

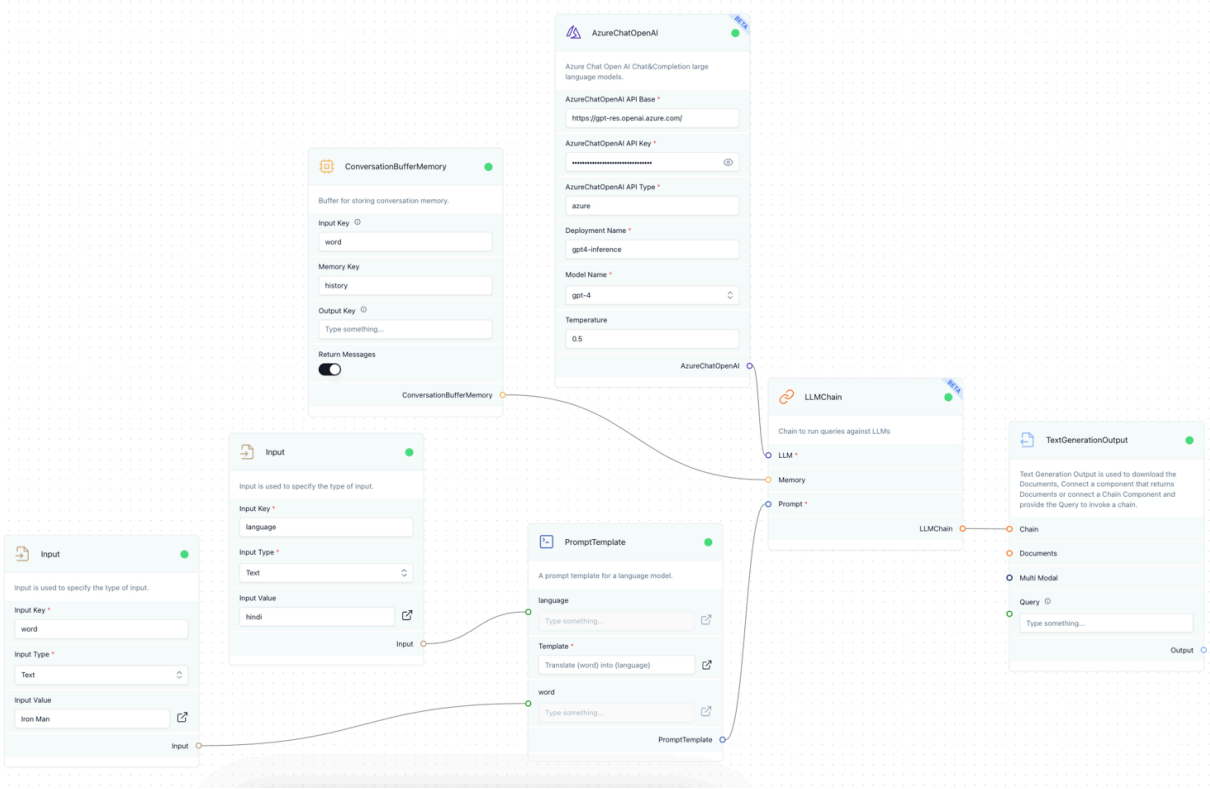
The TextGenerationOutput component is designed to store and manage text extracted from Document Loaders, images/speech outputs from Multimodal components, and output generated by Chains. It includes a Query field that allows users to input a query. This component's primary function is to ensure that the generated output is made readily accessible to users.

Parameters

- **Chain:** Users can connect any chain component as an input to the TextGenerationOutput.
- **Documents:** Users can connect any component that returns a list of documents, such as the PyPdfLoader, TextSplitters, etc.
- **Multimodal:** Users can connect any Multimodal component that returns an object like an image or a speech.
- **Query:** A Query is a special field that lets users input a query in case a stack doesn't have a prompt and memory component.

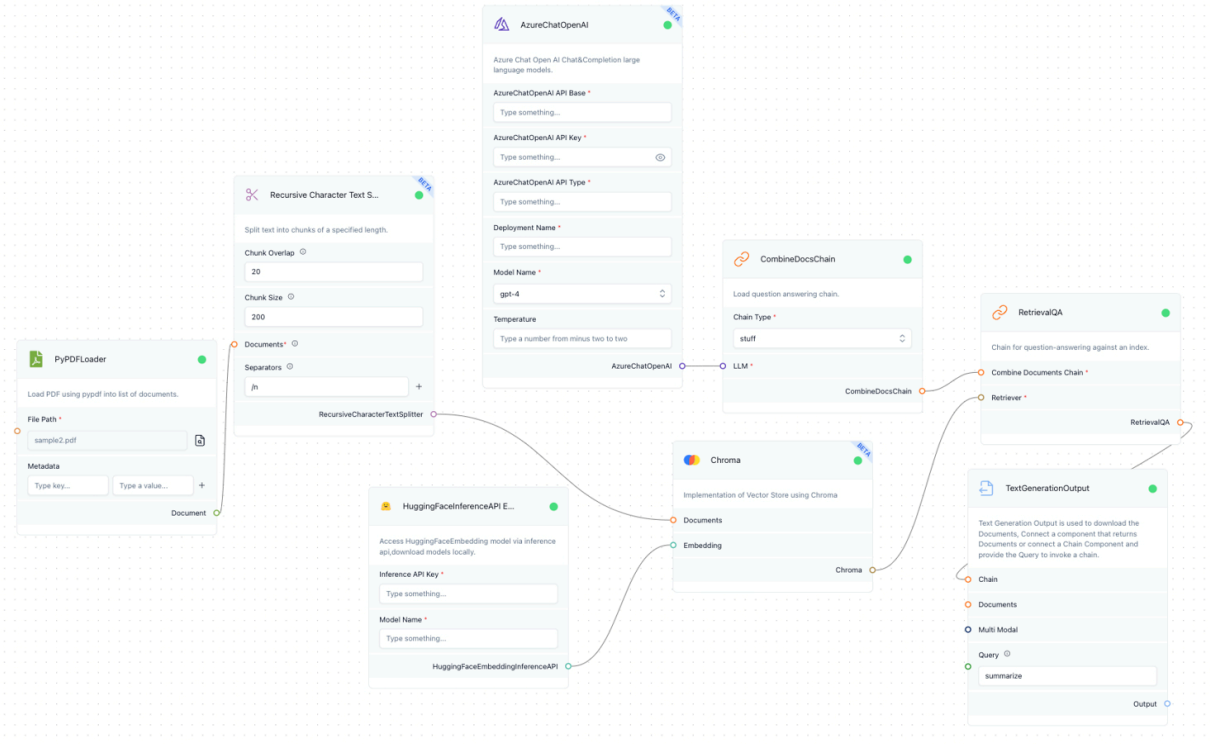
Example Usage

In the below example, the LLMChain acts as an input to the Output component (with no query). Since we have a prompt and memory, we are mentioning the inputs in the prompt and memory itself.



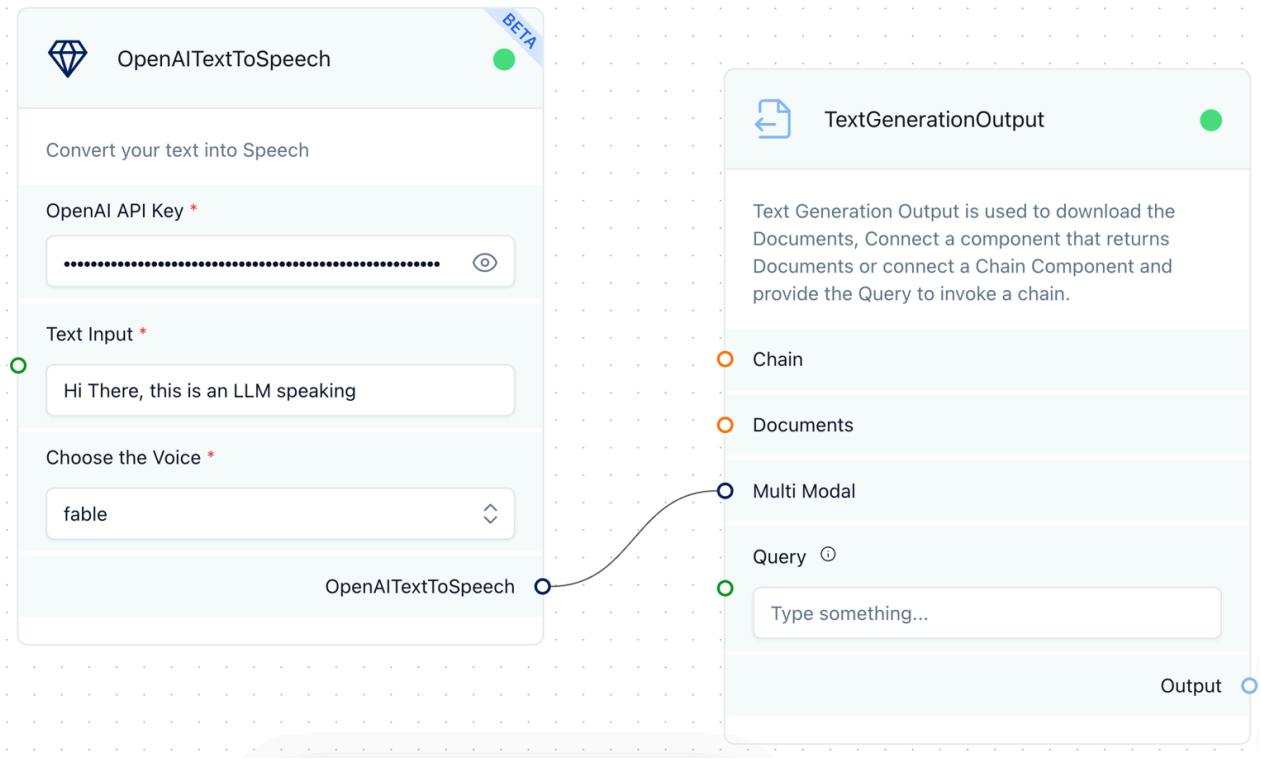
TextGenerationOutput with no query

Since there is no prompt and memory, we are adding the query in the Output component to invoke a chain for generating the output.



TextGenerationOutput with query

In the example below, the Multimodal output is connected to the text generation output to invoke the speech generation process.



TextGenerationOutput with Multimodal (OpenAITextToSpeech) as an input

The Output component can take `Chains` as an input. It can also take any component that returns `Documents`. Additionally, it takes `Multimodal` outputs and a `Query` in case a stack doesn't have a prompt and memory component.

Document Loaders

Document loaders serve as tools for importing data from various sources into a format that represents documents, each comprising textual content and associated metadata. These loaders are designed to handle diverse data sources, ranging from simple .txt files to the text content of web pages or even transcripts of YouTube videos.

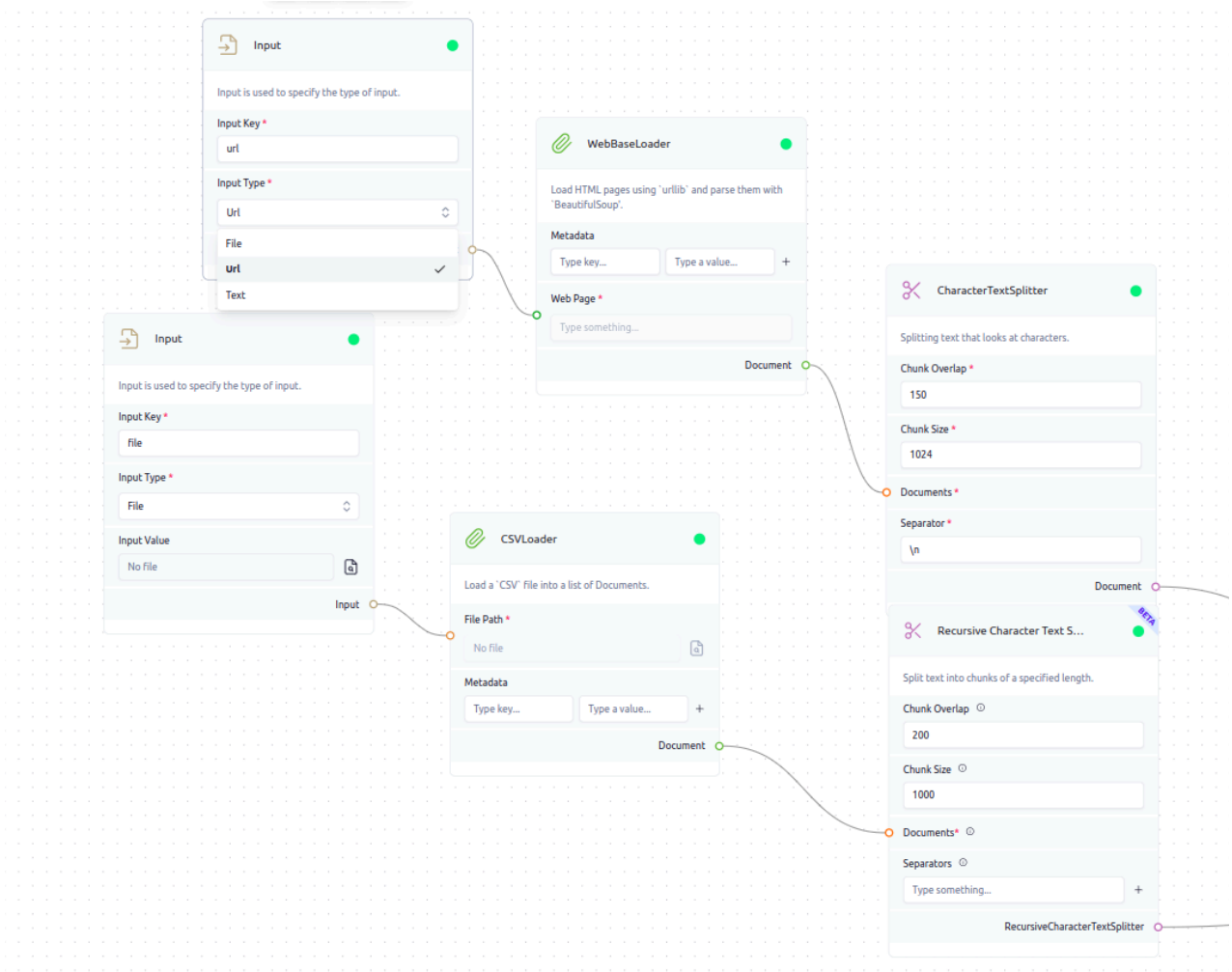
Input Component with Document Loaders

Some loaders also have an optional input component. Here, an Input type component can be connected. This component will also specify the Input Type and Input key based on which the user will be given an upload option on the deployed app to upload their Inputs. The Input type can be File or URL or Text. The right type has to be selected according to the loader.

If no Input component is connected, the deployed Stack will use the data specified in the Document Loader for answering queries. Let's consider the following:

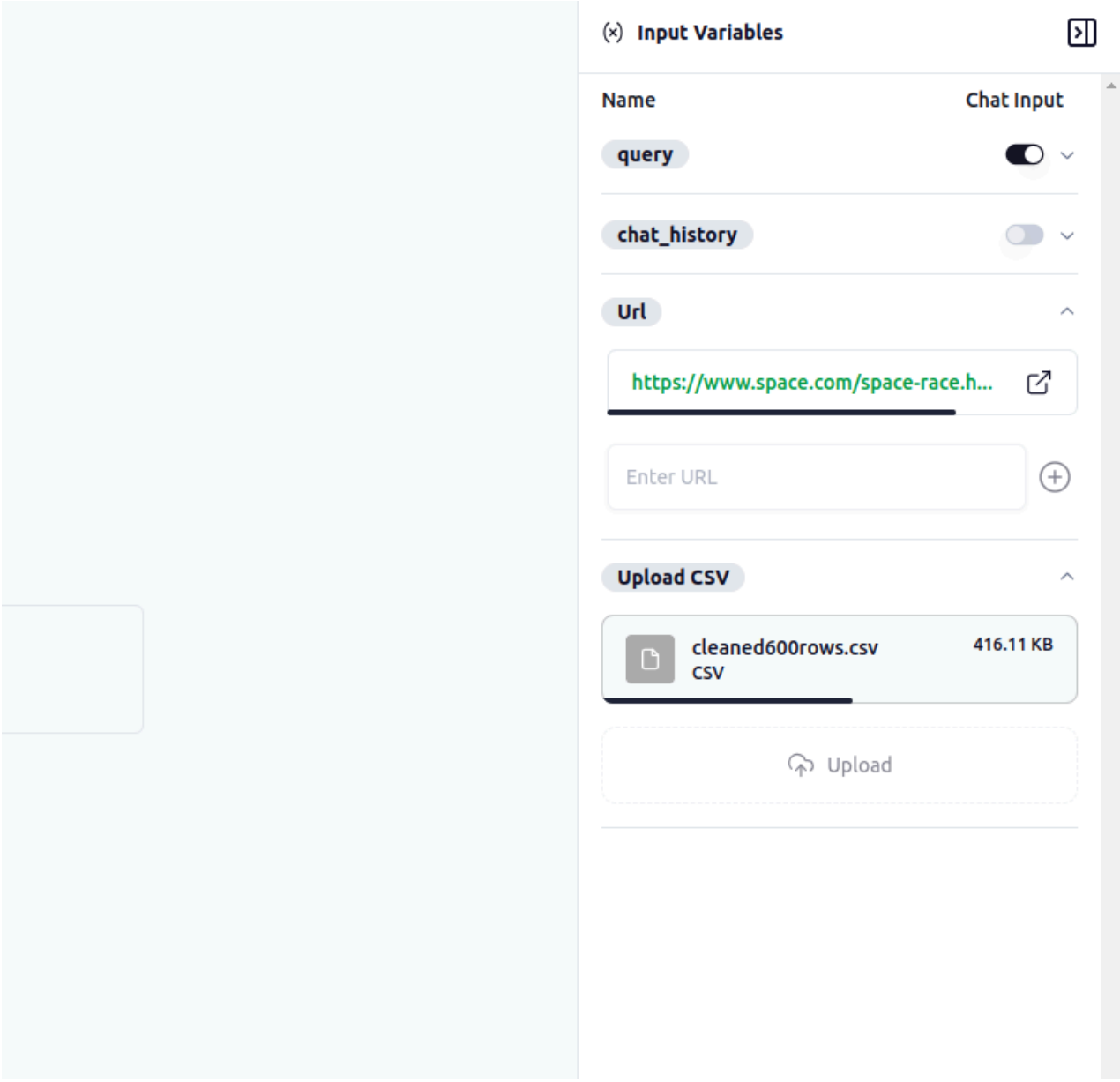
Examples:

If we wanted the user to be able to upload a File and a URL as the input components for our PDF Loader and Web Based Loader, we would have to add the Input components for each of the loaders, and specify the File type for each of them as shown below. As you can notice, specifying an Input means you won't have to restrict the loader by providing it a file/URL!



WebBase loader will require URL type Input component

NOTE: We will also need to define a separate Text Splitter for each loader before attaching it to a Vector Store. This will ensure that the user gets to Input multiple types of input in the sidebar of the deployment page. The result after deployment will have multiple Input types in the sidebar to allow user to upload their inputs. The user can also upload multiple inputs of each type, and while they are uploading they look like the below image:



User has the option to upload multiple file types

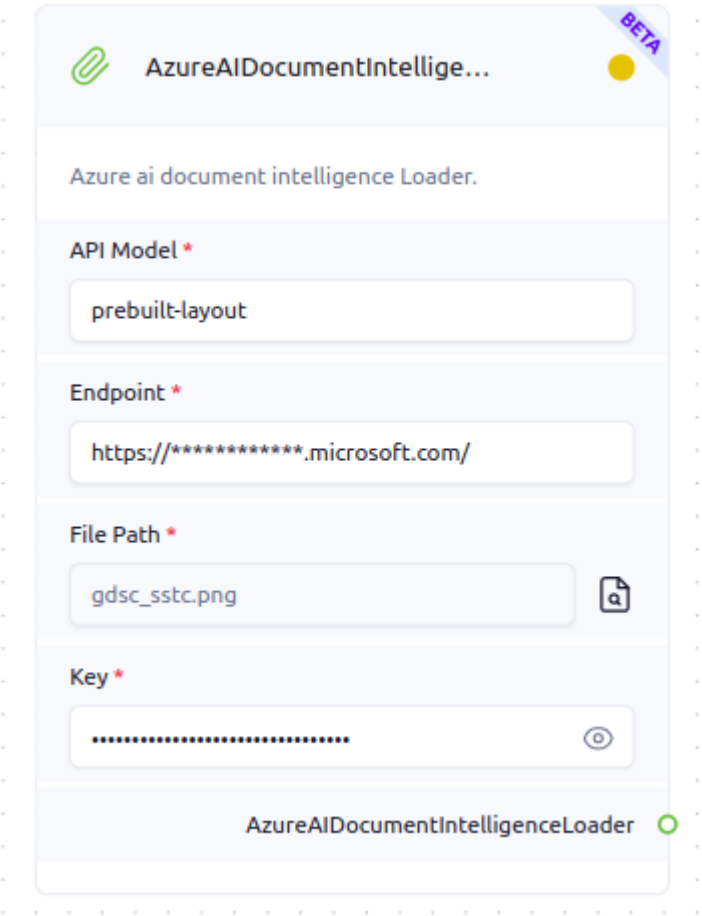
AzureAI Document Intelligence Loader

Azure AI Document Intelligence (formerly known as Azure Form Recognizer) is a machine-learning-based service that extracts text (including handwriting), tables or key-value-pairs from scanned documents or images. It allows you to turn your documents into usable data and shift your focus to acting on information rather than compiling it.

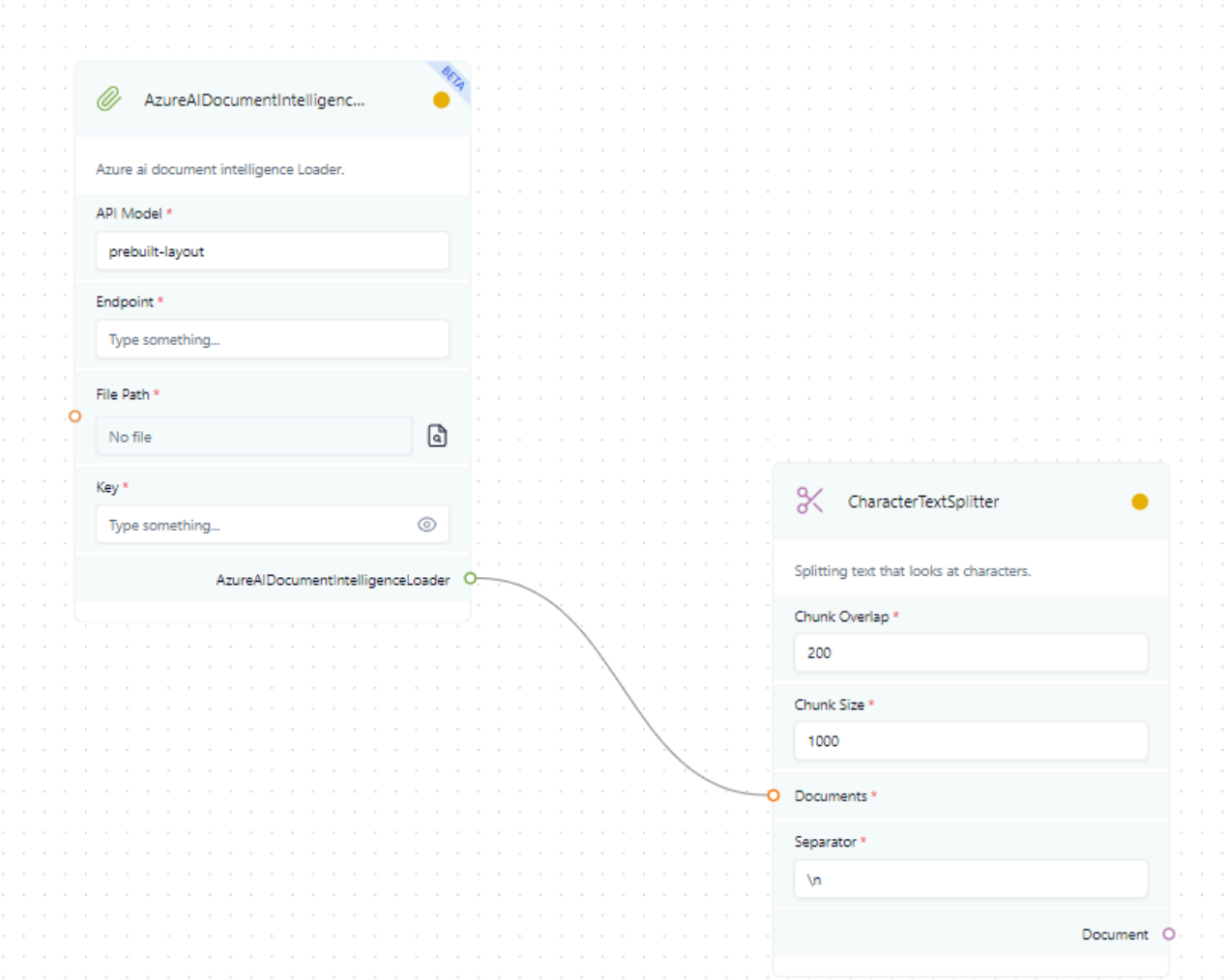
Parameters:

- **Endpoint**: Azure Cloud endpoint to integrate OCR
- **API Key**: API key from Azure AI
- **File Path**: Upload your input file

Example Usage:



These loaders can be connected with other components to create flows, for instance the AzureAIDocumentIntelligence Loader here can be connected to TextSplitters for the flow



Dropbox Loader

Dropbox is a cloud storage service provider that enables easy file sharing and access to team data from your computer, mobile device, or any web browser in a highly secure manner. DropboxLoader can load data from a list of Dropbox file paths or a single Dropbox folder path. Both paths should be relative to the root directory of the Dropbox account linked to the access token.

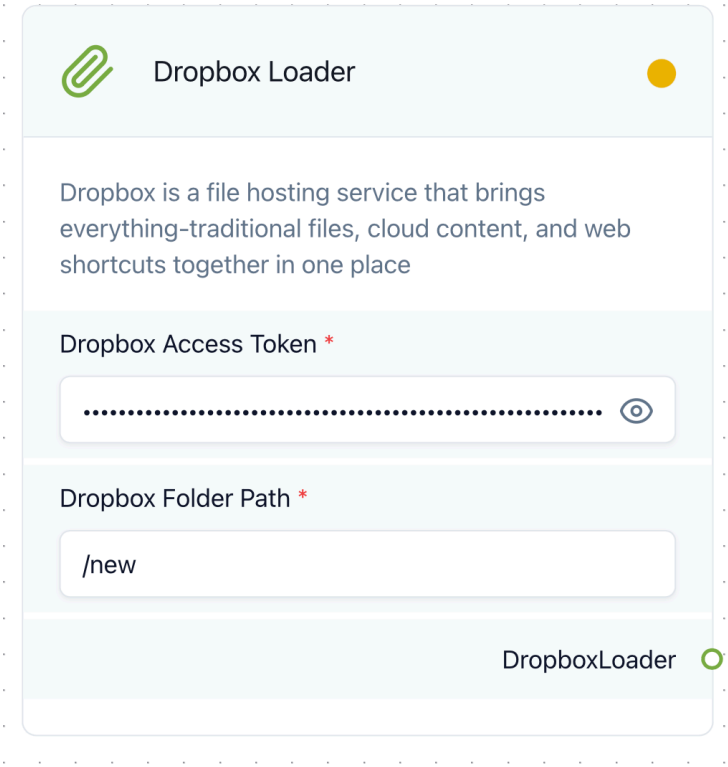
Steps to follow:

- Create your developer app on Dropbox.
- Once the app is built, navigate to permissions and select `files.metadata.read` and `files.content.read`. Then, click `Submit`.
- From the app dashboard, click on `Generate Access Token` and copy your code.

Parameters:

- Dropbox Access Token: The developer access token to access files from Dropbox
- Dropbox Folder Path: The folder name where the data is stored in Dropbox.

Example Usage



As per the above loader, the folder path refers to the folders stored in your Dropbox. The format to follow is '/' followed by your folder name. Please note that when you open the app, you will also find a key and secret; do not copy those. Instead, remember to generate an access token.

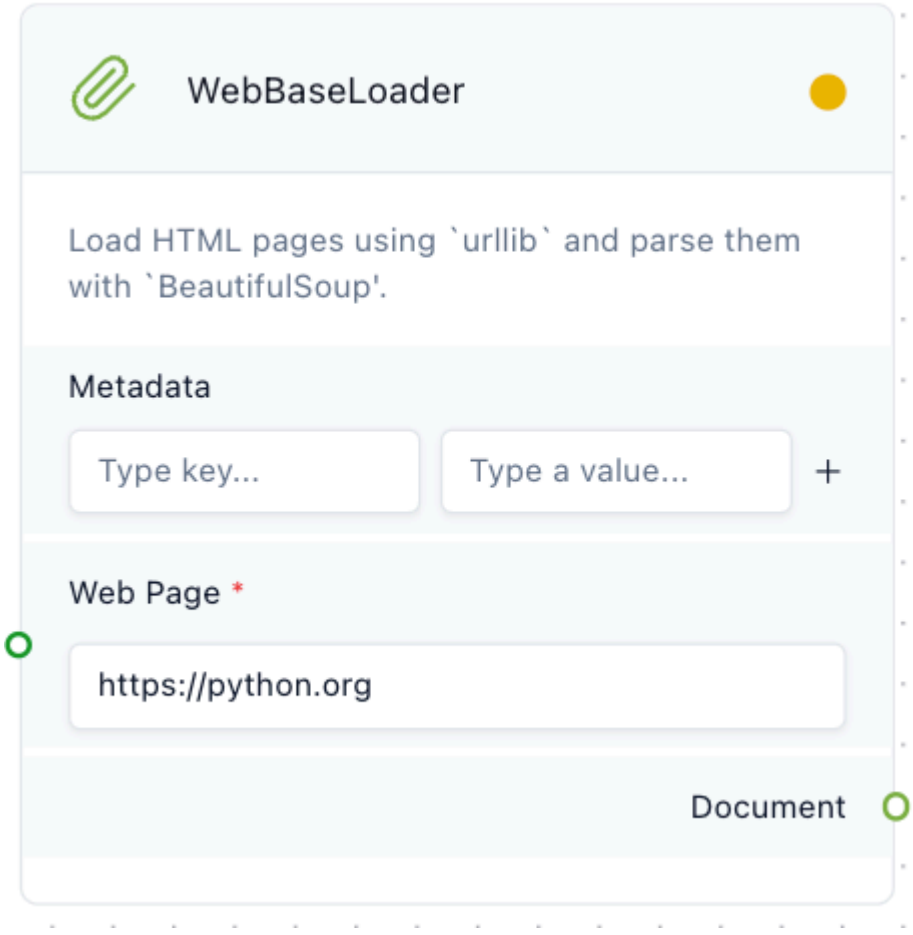
WebBaseLoader

This document loader outlines the usage of WebBaseLoader to extract all text from HTML web pages and format it into a document structure for downstream processing. In this way, this loaders are used to load web resources.

Parameters:

- **Web Page** : Enter the Webpage URL as your input
- **Metadata :** Metadata is used to provide the source and tag for the given input.

Example Usage:



CSVLoader

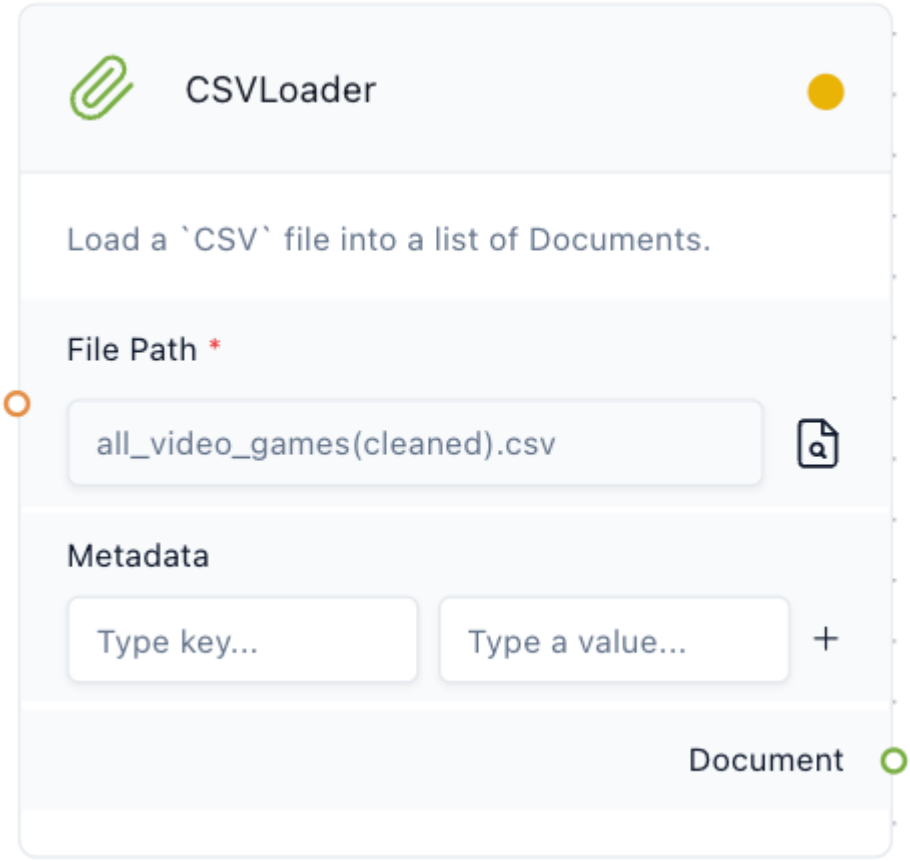
This document loader we can utilize the CSV Loader to process the delimited text, separating values and organizing them into a document format for further downstream applications.

Parameters:

- **File Path :** Upload the *CSV file as your input.

- **Metadata :** Metadata is used to provide the source and tag for the given input.

Example Usage:



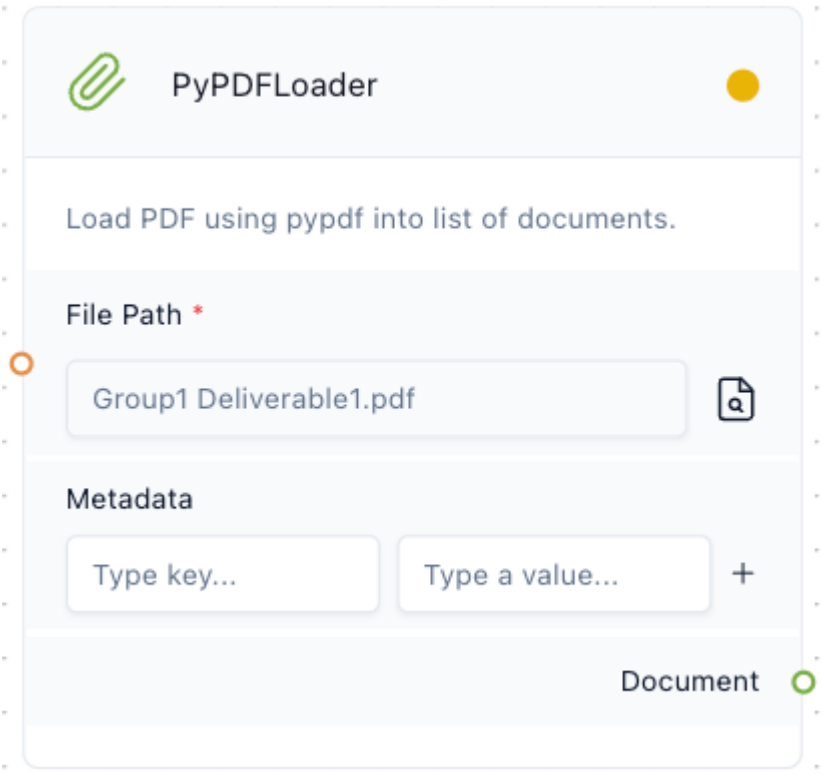
PyPDFLoader

This document loader, loads PDF using pypdf into an array of documents, where each document contains the page content and metadata with page number.

Parameters:

- **File Path :** Upload the *PDF file as your input.
- **Metadata :** Metadata is used to provide the source and tag for the given input.

Example Usage:



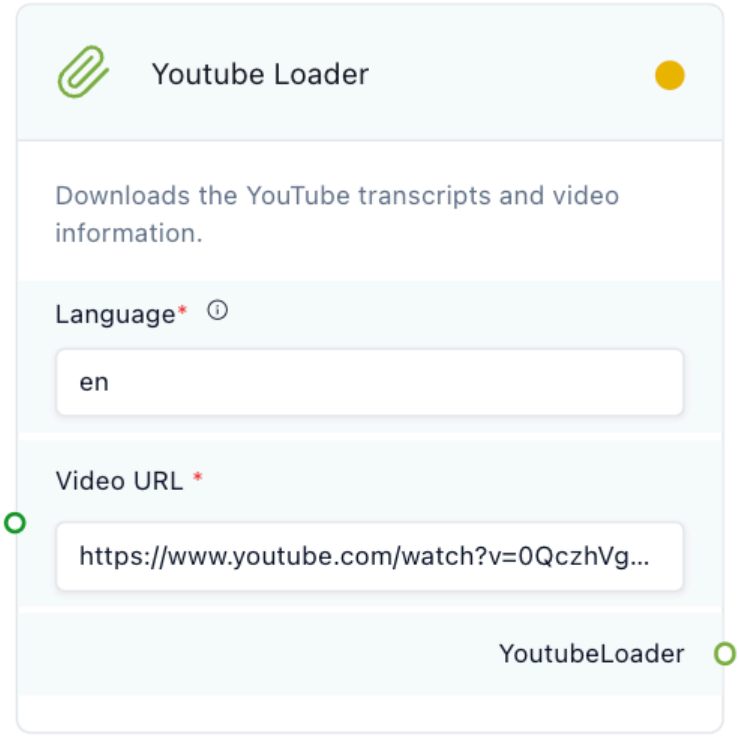
YouTubeLoader

YouTubeLoader downloads the YouTube transcripts and video information.

Parameters:

- **Video URL :** YouTube URL
- **Language :** Language code to extract transcript. please check subtitles/cc to know available transcripts. By default: en (English)

Example Usage:



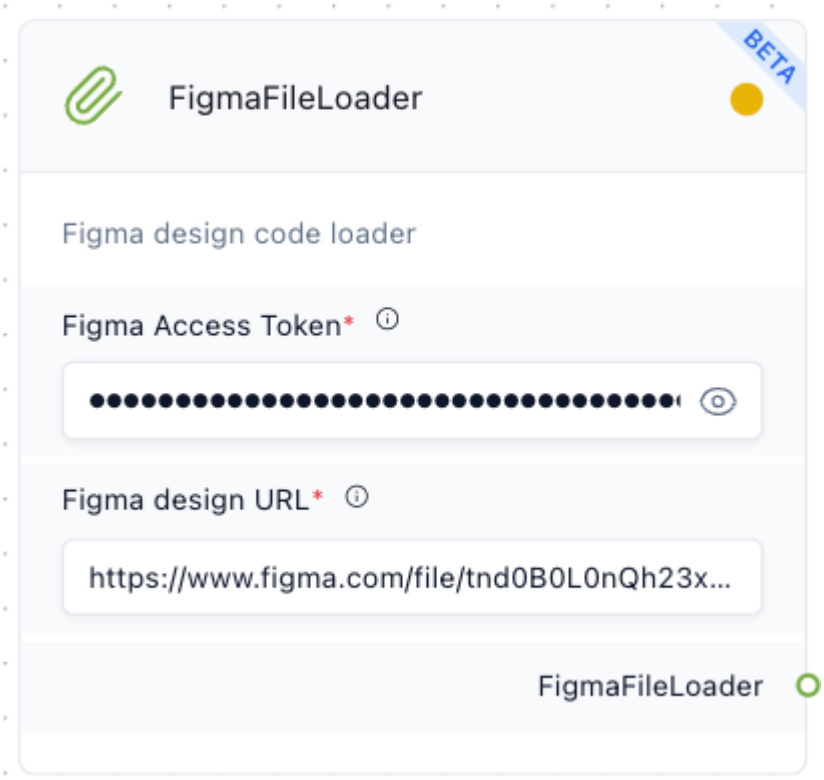
FigmaFileLoader

FigmaFileLoader enables loading Figma design files in a structured format that enables using the visual design data for things like automatic code generation.

Parameters:

- Figma Access Token: The access token you generate in Figma settings.
- Figma design URL: The URL of your figma design that you see in search bar.

Example Usage:



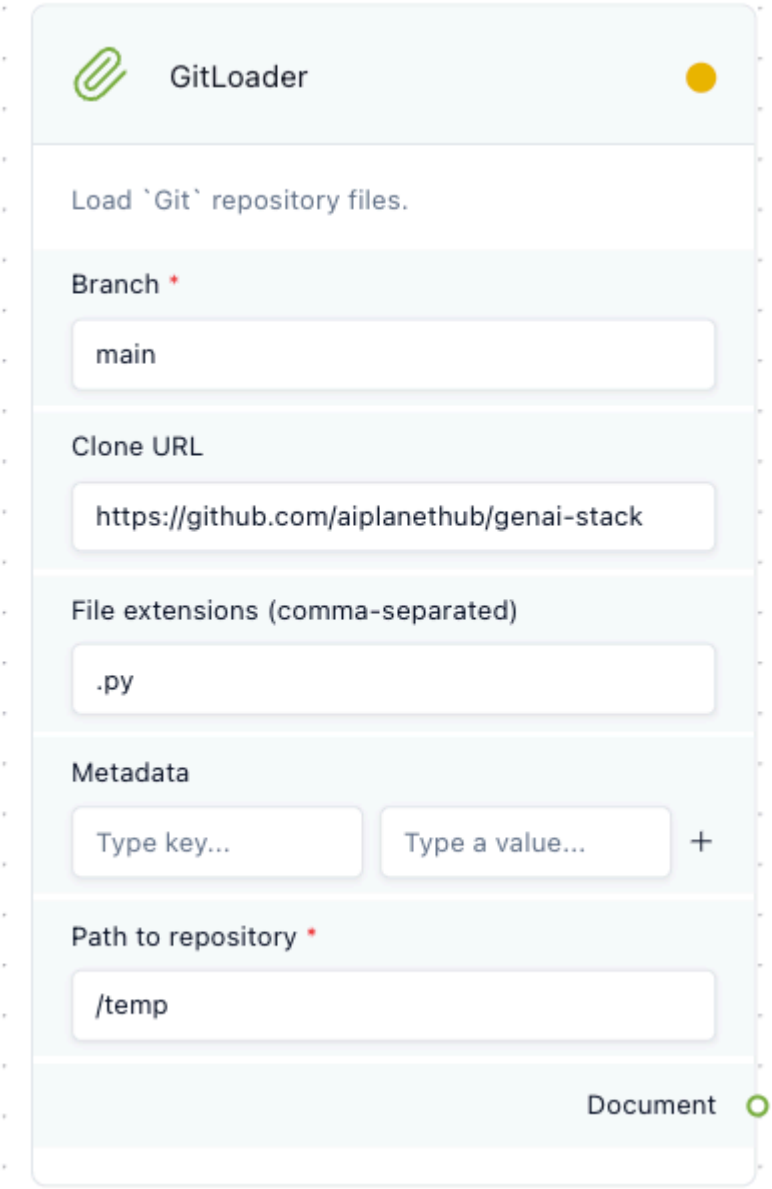
GitLoader

Gitloader is used to load files from a Git repository by cloning a repository from a URL

Parameters:

- Branch: The branch you need to clone.
- Clone URL: The URL of the Git repository.
- File extension: The file type you need to clone (.py, .txt, etc)
- Metadata: Metadata is used to provide the source and tag for the given input.

Example Usage:



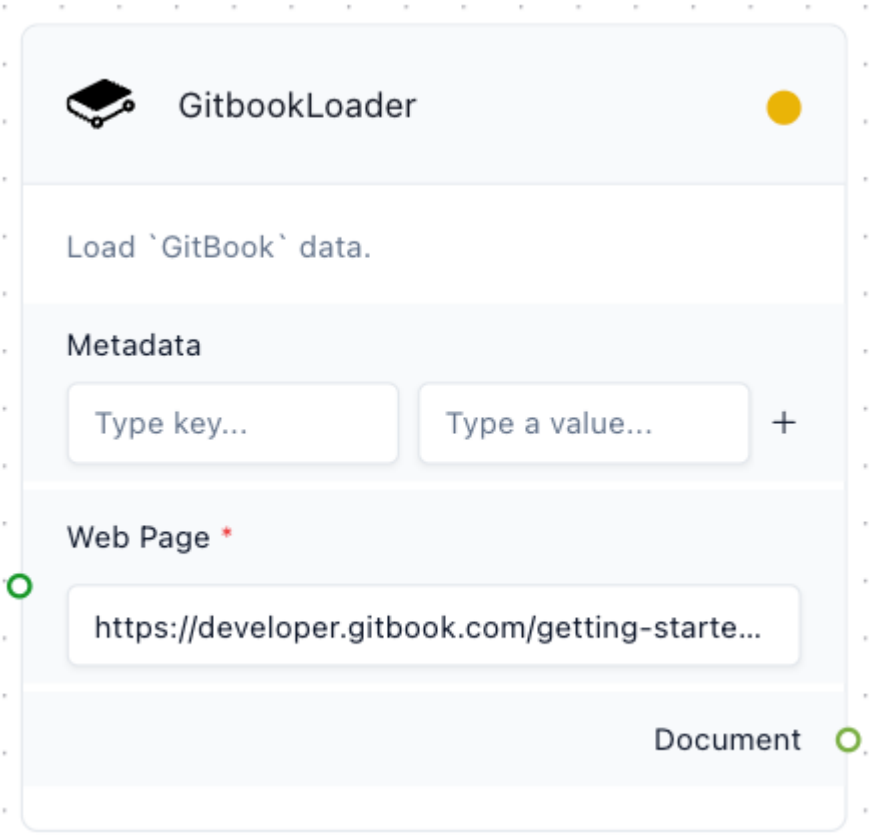
GitbookLoader

The Gitbook loader is used to load GitBook data, using the URL of the web page.

Parameters:

- Web Page: The URL of the web page of the Git book
- Metadata: Metadata is used to provide the source and tag for the given input.

Example Usage:



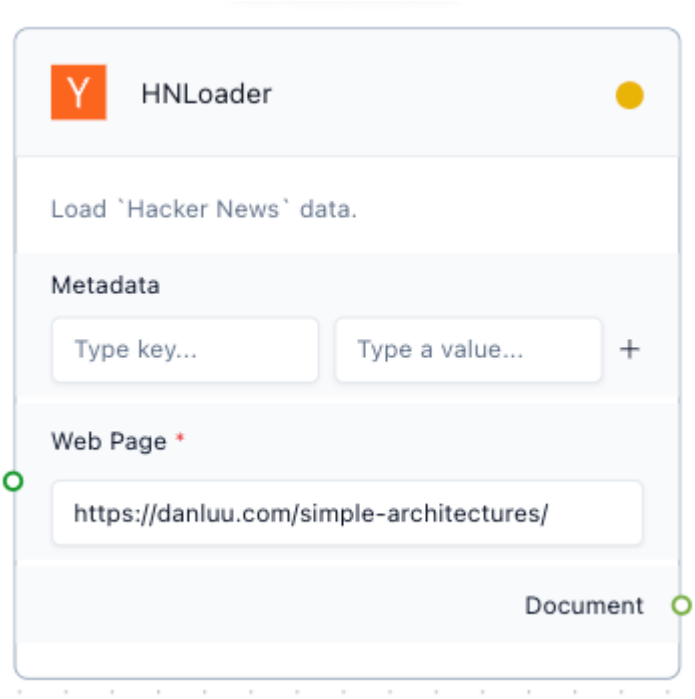
HNLoader

The HNLoader is used to load data from Hacker News, either from the main page results or the comments page. It can fetch all URLs concurrently with rate limiting and scrape data from a webpage, returning it in BeautifulSoup format.

Parameters:

- Web Page: The URL of the web page of Hacker News.
- Metadata: Metadata is used to provide the source and tag for the given input.

Example Usage:



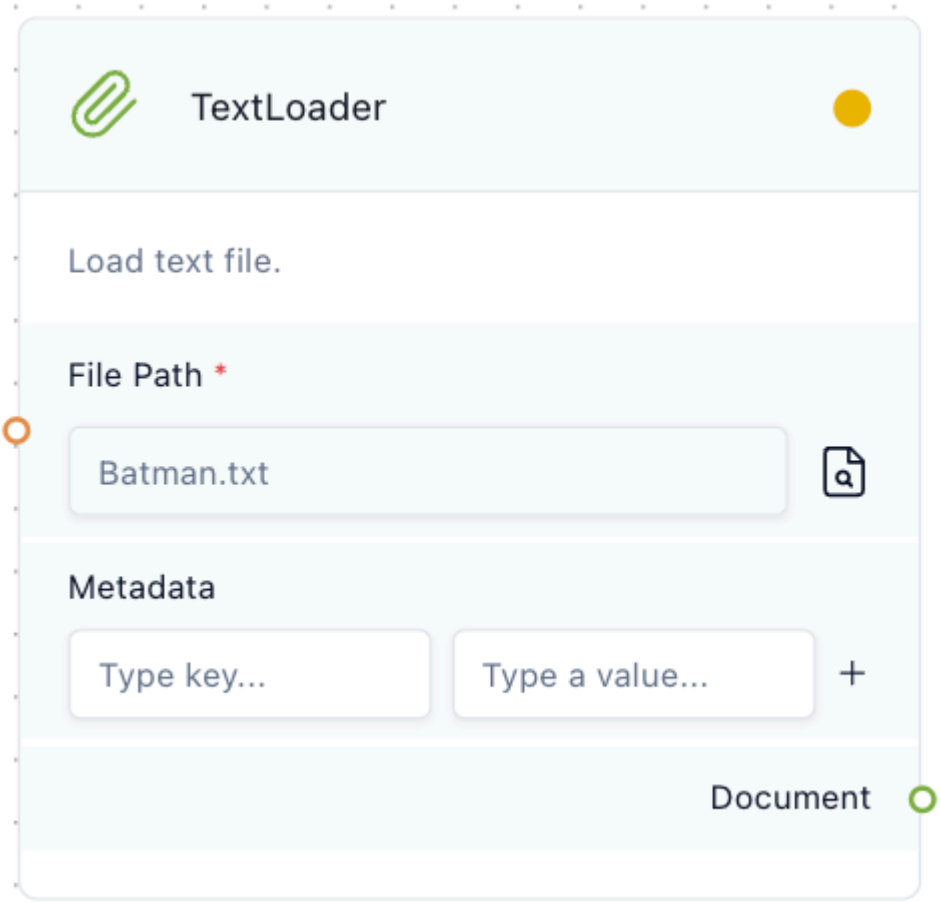
TextLoader

The TextLoader is to load a simple .txt file.

Parameters:

- File Path: The path to .txt file to load.
- Metadata: Metadata is used to provide the source and tag for the given input.

Example Usage:



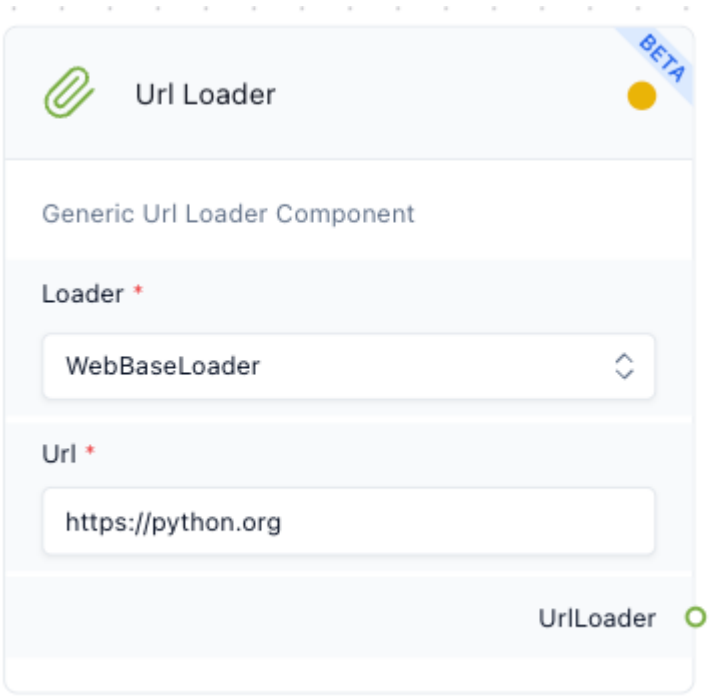
Url Loader

The URL loader is used to load HTML documents using URL, you can select a specific loader from drop down list. By default, it's 'WebBaseLoader', which works for any URL.

Parameters:

- URL: The URL of the web page you want to load.

Example Usage:



AirbyteJSONLoaders

The AirbyteJSONLoader is used to load local Airbyte JSON files. It initializes with a file path and can load data into Document objects. This loader is specifically designed to handle data integration from Airbyte.

Parameters:

- File Path: the file path to Airbyte JSON file.
- Metadata: Metadata is used to provide the source and tag for the given input.

Example Usage:

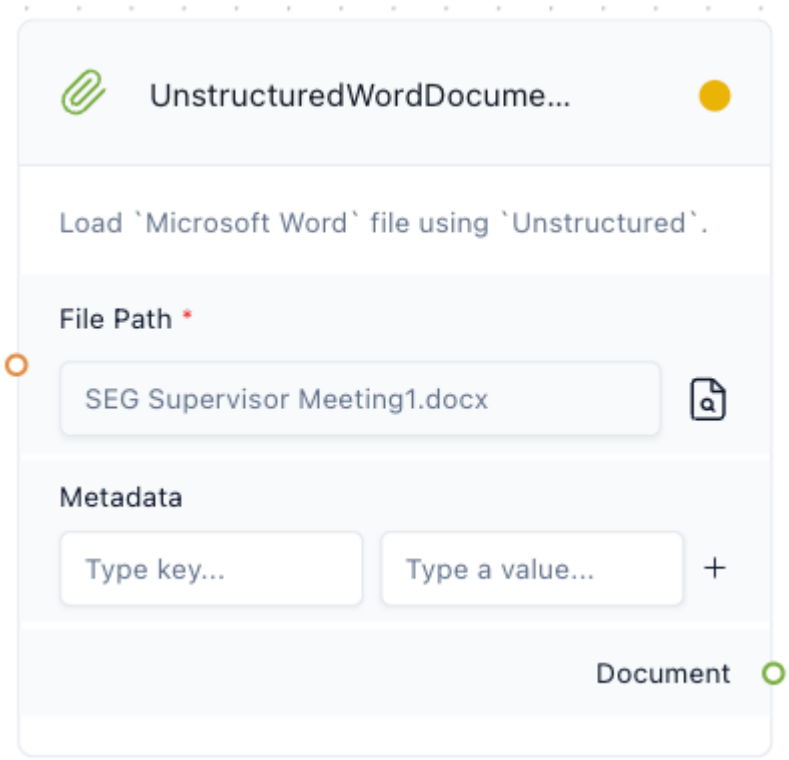
Unstructured Loaders

In GenAI Stack, Unstructured currently supports loading of text files, powerpoints, html, pdfs, images, and more. Unstructured detects the file type and extracts the same types of elements.

Parameters:

- **File Path:** Based on the given unstructured, Upload the appropriate file as your input.
- **Metadata:** Metadata is used to provide the source and tag for the given input.

Example Usage:



Prompts

A prompt serves as the instruction for inputting text to Large Language Models (LLMs), prompting them to generate factual responses. Writing effective prompts is crucial as it helps mitigate hallucinations to a certain extent. Several prompt techniques exist, including Few-shot prompting, Zero-shot prompting, and Chain of Thoughts. In this analogy, the prompt acts as the parent, guiding the LLM, which acts as the child, to produce accurate responses.

Note: Keep in mind that every Large Language Model has a maximum content length limit, determining the extent to which it can store tokens. Writing longer prompts consumes more of this context length, reducing the space available for generating responses.

System Message Prompt Template

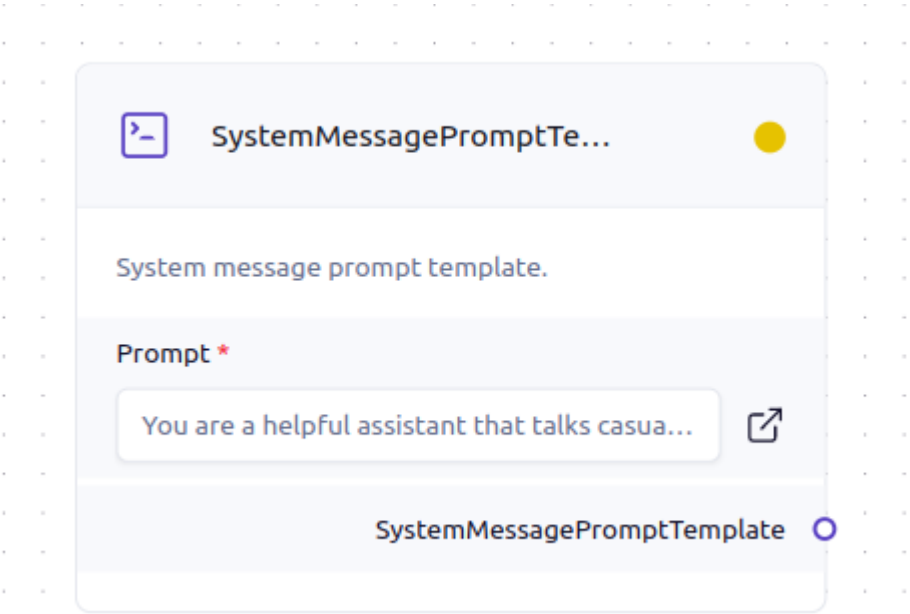
System messages are instructions for the overall task that you are prompting the Large Languages models for. These messages guide the model on the specific goal or objective it needs to achieve within the context of the task at hand.

Parameters

- prompt: This template is provided by you to enter instructions for the System prompt

Example

```
You are a helpful assistant that talks casually about life in general.
You are a good listener and you can talk about anything.
```



System Message Prompt template doesn't require any input component, further its prompt template can be passed to Sequential Chain or Chat Prompt Template.

Human Message Prompt Template

The Human message prompt provides a specific example relevant to the broad task outlined in the system message. It serves as a focused input to guide the model's response generation process, aiding in achieving the desired outcome efficiently.

Parameters

- prompt: This template accepts user query along with rules it should follow to generate the response.

Example Usage

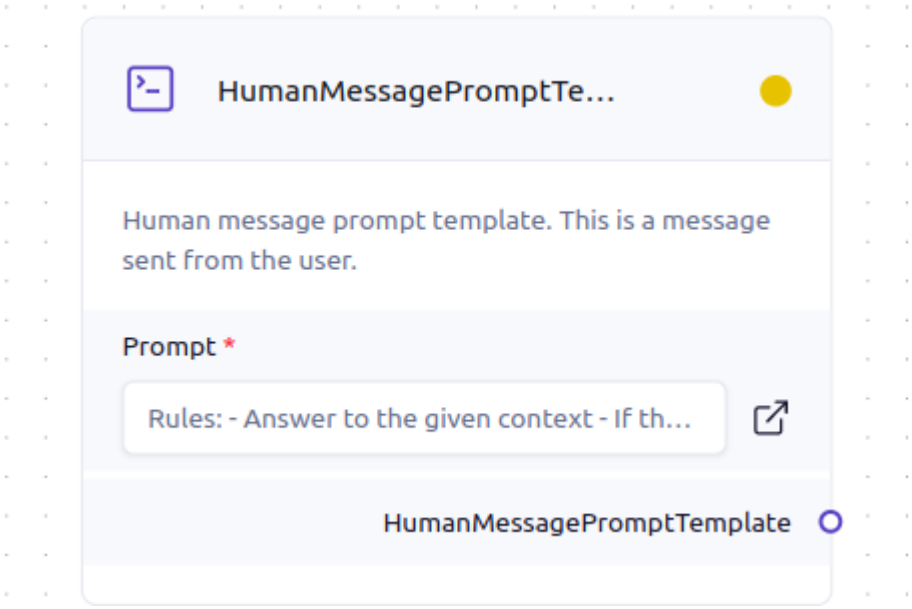
For the Human message prompt template, there may be occasions where you need to incorporate `user input`. To integrate user input or variables, enclose the variable within curly brackets `{}`.

Without Input Variable

One can define the rules that a LLM needs to adapt to generate the response.

Since the prompt does not include any input variables, there is no provision for connecting input to the Human Message Prompt template. The output component connection remains the same as that used for the System Message Prompt.

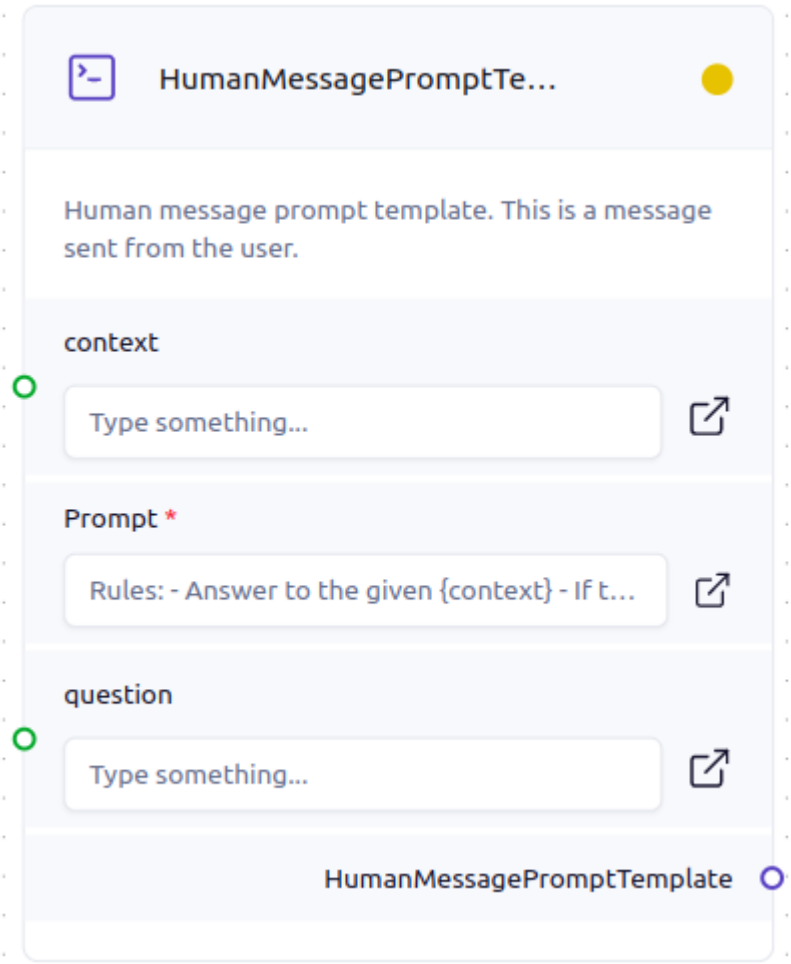
```
Rules:
- Answer to the given context
- If the user question is not in context, say "I don't know"
```



With Input Variable

Let's use the same prompt as before but with input variables.

```
Rules:
- Answer to the given {context}
- If the user {question} is not in context, say "I don't know"
```



Now, within the Human message, two new entries have been added. These can be connected to either a loader or a text splitter as input for the input variable. However, if you choose not to connect the loader or text splitter to the input variables, it will not result in a stack error. Nonetheless, when utilizing a chat interface, you will need to fill in these variables manually.

Chat Prompt Template

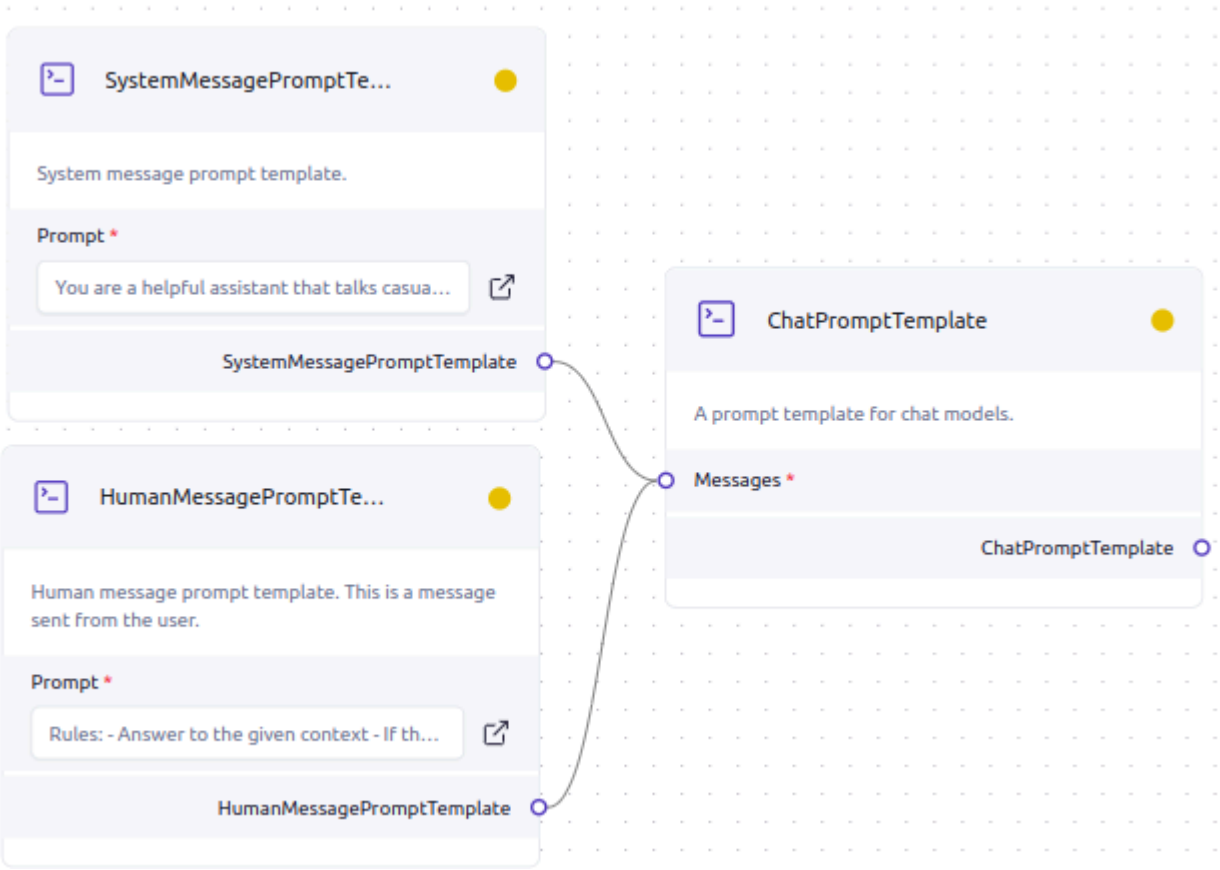
The Prompt Template is a hybrid template that combines elements of both the System Message Prompt and the Human Message Prompt. It is passed directly input to the chain, facilitating seamless passage into the Large Language Models for generating responses.

Parameters

- Messages: This is the input component in ChatPromptTemplate, where you can connect Human Message and System Message prompts.

Example Usage

The Chat Prompt Template requires a message as input, and it can be connected to both Human Message and System Message as inputs. The chain then serves as the output for the Chat Prompt Template.



Prompt Template

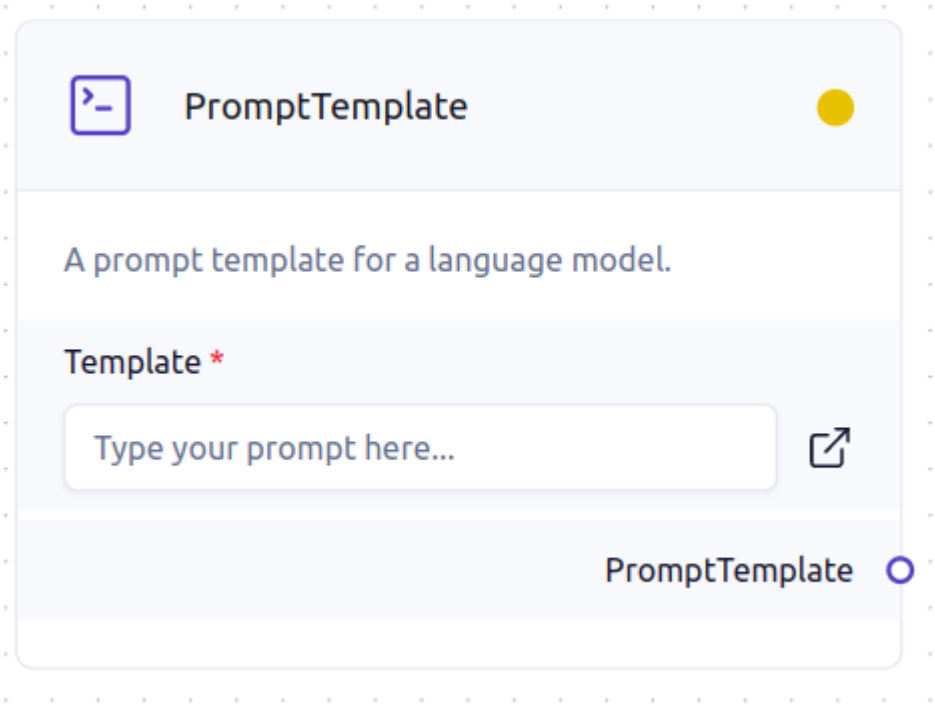
Prompt Template is a component where you can integrate different prompt techniques such as Few Shot, Zero Shot, Chain of thoughts and so on. A prompt template can be constructed from either a set of examples, or from an Example Selector object.

Parameters

- Template: A user entry to implement different prompt techniques.

Example Usage

This prompt component is very straightforward; you can enter any instructions that you need to provide to the LLM. To receive a response, this component should be connected to either an LLM Chain or a RetrievalQAPrompt.



Chat Message Prompt Template

The Chat Message Prompt Template resembles the Prompt Template but offers the flexibility to designate whether the entered prompt is a system message or a human message.

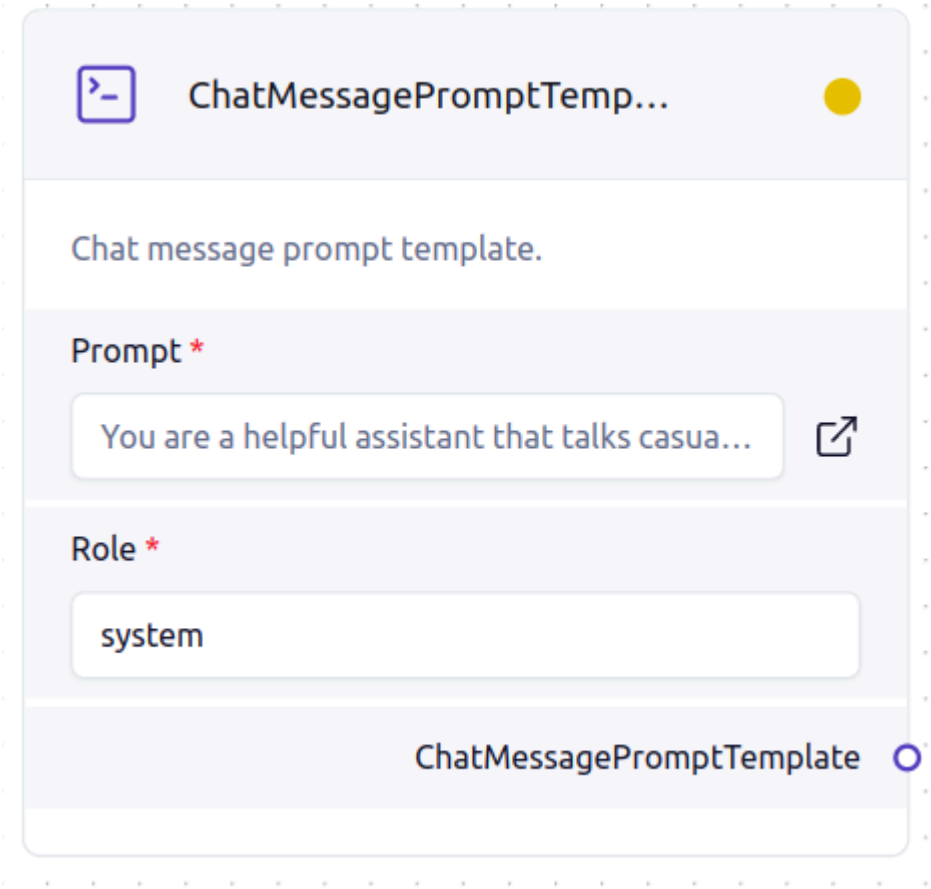
Parameters

- prompt: Input either a system prompt or a human prompt according to your requirements.
- role: Assign the role as `system` for a System prompt and `user` for a Human prompt based on the entered prompt.

Example Usage

The Chat Message Prompt Template determines the role assigned as the input and links the component to the Chat Prompt Template as the output.

Think of the Chat Message Prompt Template as an alternative for both the System and Human message prompt templates.



Text Splitters

In GenAI Stack, after loading documents, users frequently need to tailor them for optimal application use. This often involves breaking down lengthy texts into smaller chunks that align with model context windows. GenAI Stack simplifies this process with built-in document transformers, offering easy-to-use functionalities for splitting, combining, filtering, and manipulating documents according to application needs.

```
NOTE: All the Text splitters below have the same input and output compone
```

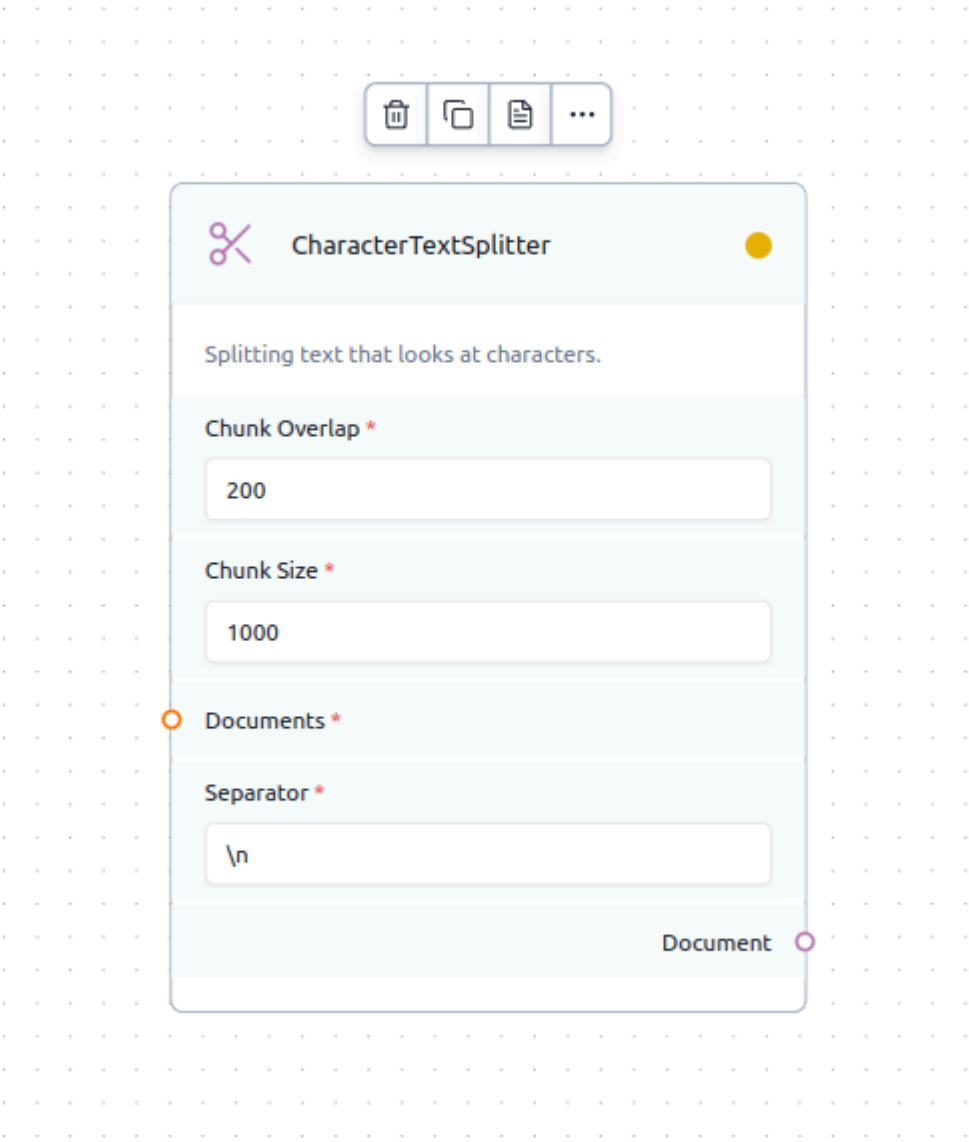
CharacterTextSplitter

Splits text based on a user defined character. One of the simpler methods.

Parameters

- **Documents:** Input documents to split.
- **chunk_size:** Determines the maximum number of characters in each chunk when splitting a text. It specifies the size or length of each chunk.
- **chunk_overlap:** Determines the number of characters that overlap between consecutive chunks when splitting text. It specifies how much of the previous chunk should be included in the next chunk.
- **separator:** Specifies the character that will be used to split the text into chunks.

Example usage



The available input components for Text Splitters include Document objects (the output of loaders), other Text Splitters, Chains- PromptRunner and SequentialLLMChain or certain Utilities.

RecursiveCharacterTextSplitter

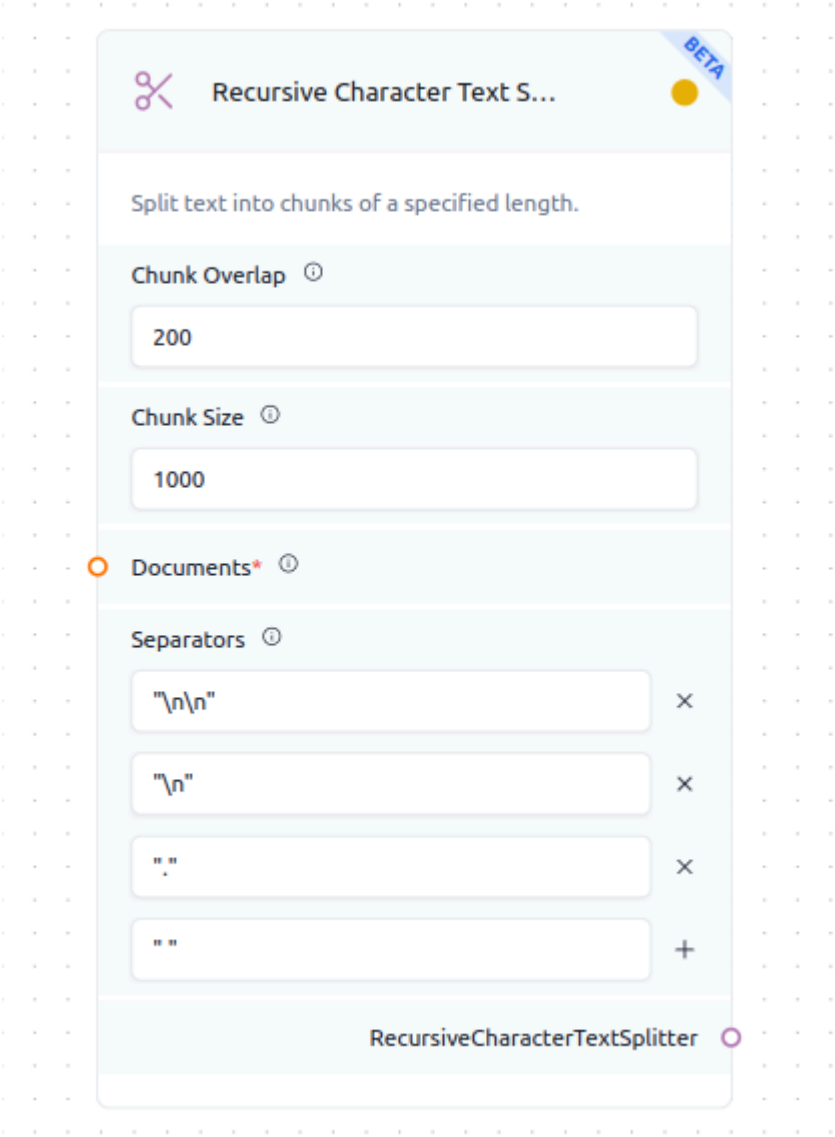
Text is recursively divided with the goal of maintaining the proximity of related content. How this workis is that, the first separator chunks data, and this chunked data is then recursively split using the subsequent separators.

Parameters

- **Documents:** Input documents to split.

- **chunk_size:** Determines the maximum number of characters in each chunk when splitting a text. It specifies the size or length of each chunk.
- **chunk_overlap:** Determines the number of characters that overlap between consecutive chunks when splitting text. It specifies how much of the previous chunk should be included in the next chunk.
- **separator:** Specifies the character(s) that will be used to split the text into chunks.

Example usage



The recursive text splitter uses the list of Seperators one after another, first the document is chunked according to the top most seperator (paragraphs), then the next seperator(newlines), and then sentences and words, and finally characters.

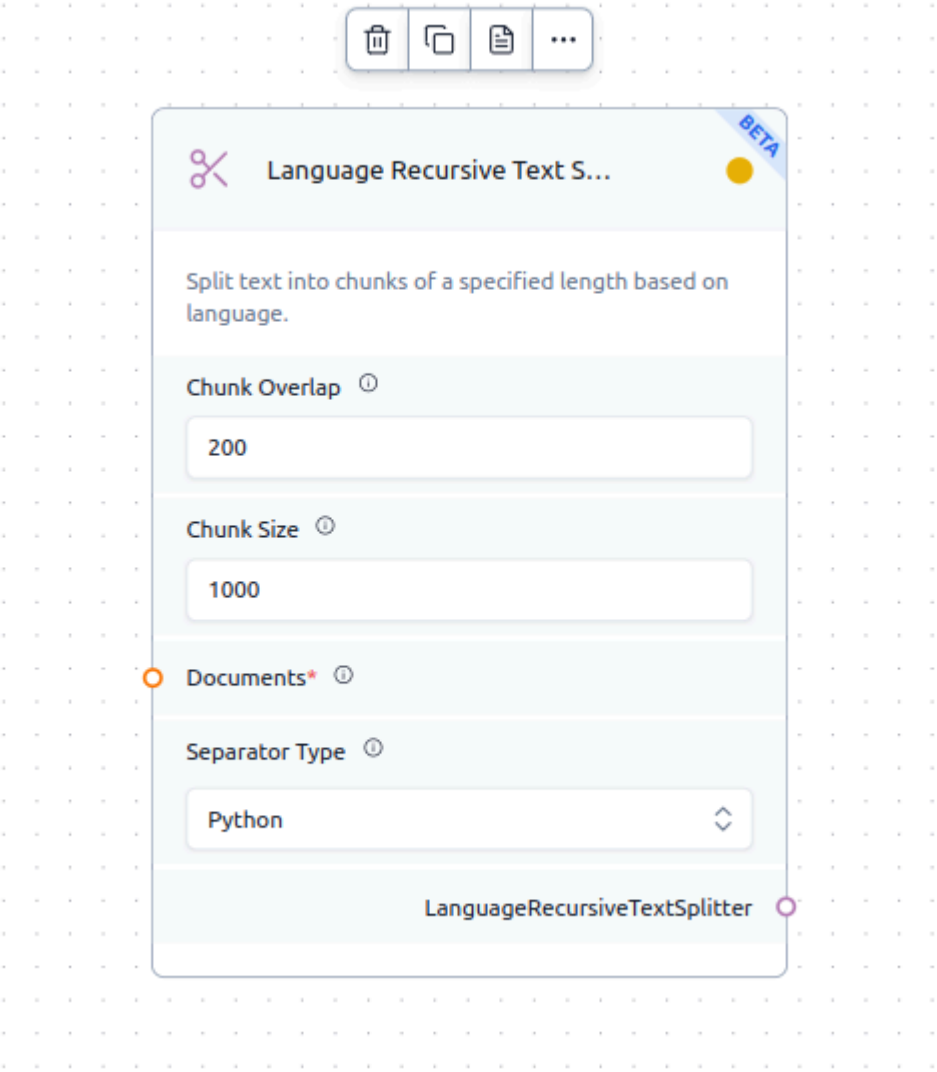
LanguageRecursiveTextSplitter

The LanguageRecursiveTextSplitter is a text splitter that splits the text into smaller chunks based on the (programming) language of the text.

Parameters

- **Documents:** Input documents to split.
- **chunk_size:** Determines the maximum number of characters in each chunk when splitting a text. It specifies the size or length of each chunk.
- **separator_type:** The parameter allows the user to split the code with multiple language support. It supports various languages such as Ruby, Python, Solidity, Java, and more. Defaults to `Python`.

Example use case:



The Language Recursive Text Splitter, takes Document objects and splits them according to separators that can be chosen in the dropdown menu according to your required programming language (Ex: "\ndef", "   \ndef", for Python).

Embeddings

Embedding is a technique in machine learning where words, phrases, or entire documents are represented as vectors in a high-dimensional space. This numerical representation captures semantic relationships, enabling algorithms to better understand and process the underlying meaning of the textual data.

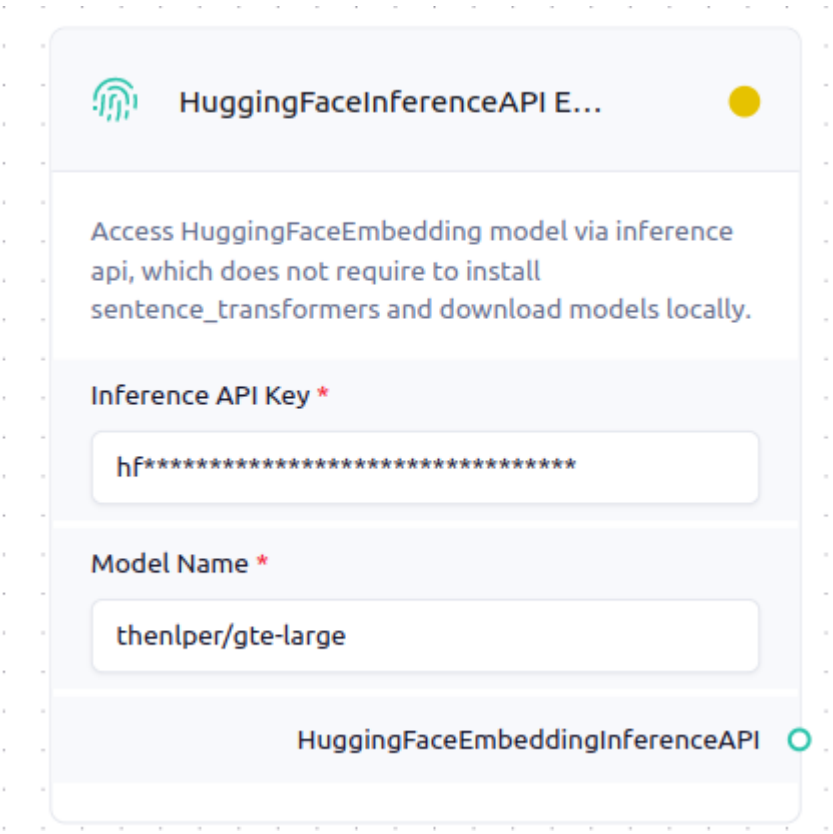
HuggingFace Inference API Embeddings

HuggingFace provides a range of Open Source embedding models. In GenAI Stack we can implement embeddings via the Hugging Face Inference API, which does not require us to install `sentence_transformers` and download models locally.

Parameters

- Inference API key: HuggingFace Access Token to run the embedding model on Inference API.
- Model name: One can select the model name from Massive Text Embedding Benchmark (MTEB) Leaderboard ↗

Example Usage



To utilize HuggingFace Embeddings, you must initially register an account at huggingface.co ↗ and obtain your Access Token. With HuggingFace Inference API Embeddings, no input is needed; after entering your Access Token and model name, you can seamlessly connect this component to the `Vector Store` for further use.

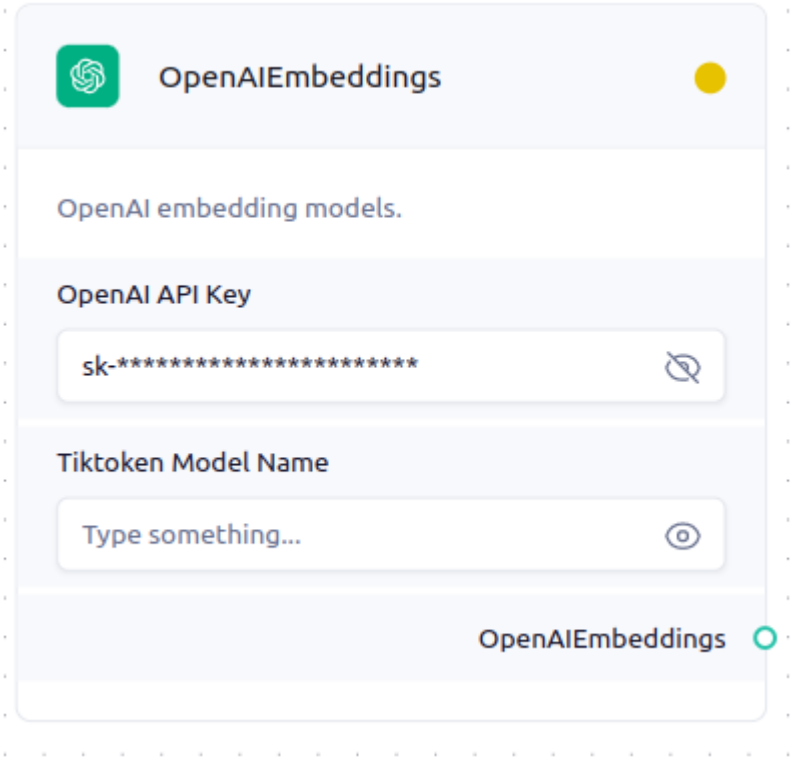
OpenAI Embedding

OpenAI Embedding is a closed source embedding model.

Parameters:

- api_key: OpenAI Embeddings requires OpenAI api key.
- tiktoken model name: OpenAI provides various embedding models, by default it is text-embedding-ada-002.

Example Usage



OpenAI embedding component only needs OpenAI API key that you can get from https://platform.openai.com/. TikToken model is to keep count of token, it can be None. The output for this component is `Vector Store`.

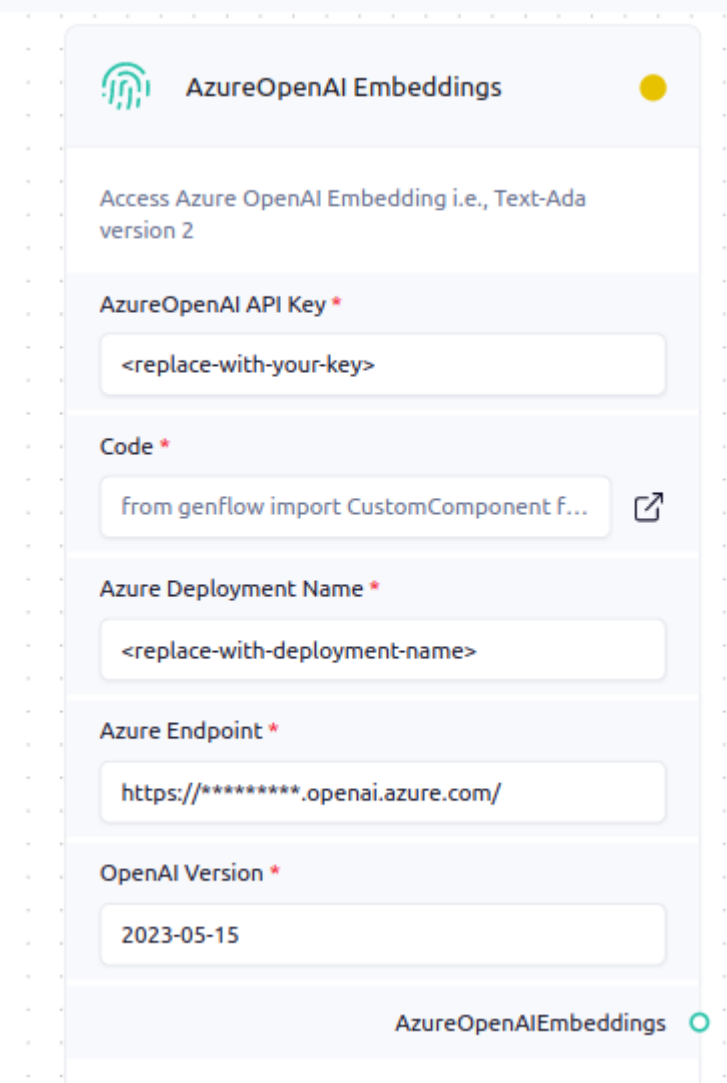
Azure OpenAI Embeddings

Azure provides Azure AI Studio services that supports OpenAI services such as GPT3, GPT4 and Embedding models. Dimension attributes have a minimum of 2 and a maximum of 2048 dimensions per vector field.

Parameters

- Azure OpenAI API key: API key that is created on AzureOpenAI Studio service.
- Azure Deployment Name: Create a model deployment on Azure using text-ada version 2 embedding model
- Azure Endpoint: Endpoint URL that is created on AzureOpenAI Studio service.
- OpenAI Version: API Version property depends on the method you are calling in the API. This is mainly the datetime.

Example Usage



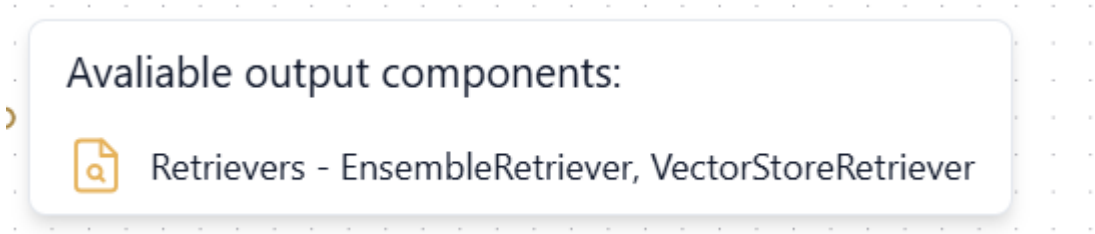
To utilize AzureOpenAI, start by creating a resource group on `Azure AI Studio`. Once the resource is set up, you can easily retrieve your Endpoint URL and API key, which you can then copy and paste for integration. Within the same dashboard, navigate to model deployments and select `Text-Ada` embedding to obtain your deployment name. Now your component is ready to connect to the `Vector Store`.

Vector Store

Vector databases make it possible to draw comparisons via semantic search, identify relationships, and understand the context within the vector Embeddings. One additional advantage of vector stores is that we can use them as a retriever, where, based on a user query, it returns the relevant information.

A few of the popular Vector Store or Vector Databases are: Chroma, Weaviate, Qdrant, Milvus, FAISS, ElasticSearch and Pinecone. GenAI Stack currently supports Reddis, and Weaviate.

These Vector Stores can further be connected to retrievers.



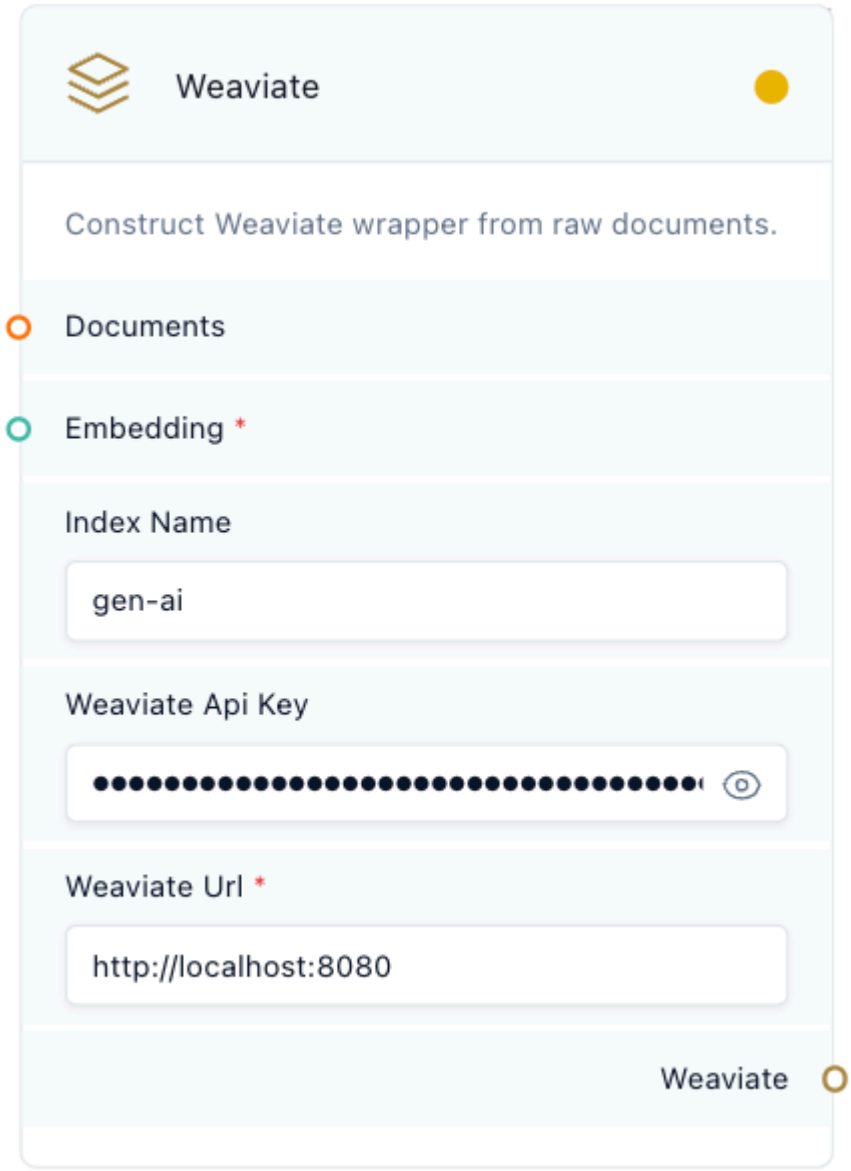
Weaviate

Weaviate is an open-source vector search engine that stores both objects and vectors. It also supports cloud clients to store vector embeddings.

Parameters

- Documents: The chunked or loaded document component.
- Embeddings: Embedding component to convert documents into vectors.
- Index Name: A unique identifier to save the index
- Weaviate API Key: Weaviate API key for the cluster created on Weaviate cloud
- Weaviate URL: Cluster URL on Weaviate cloud

Example usage:



Redis

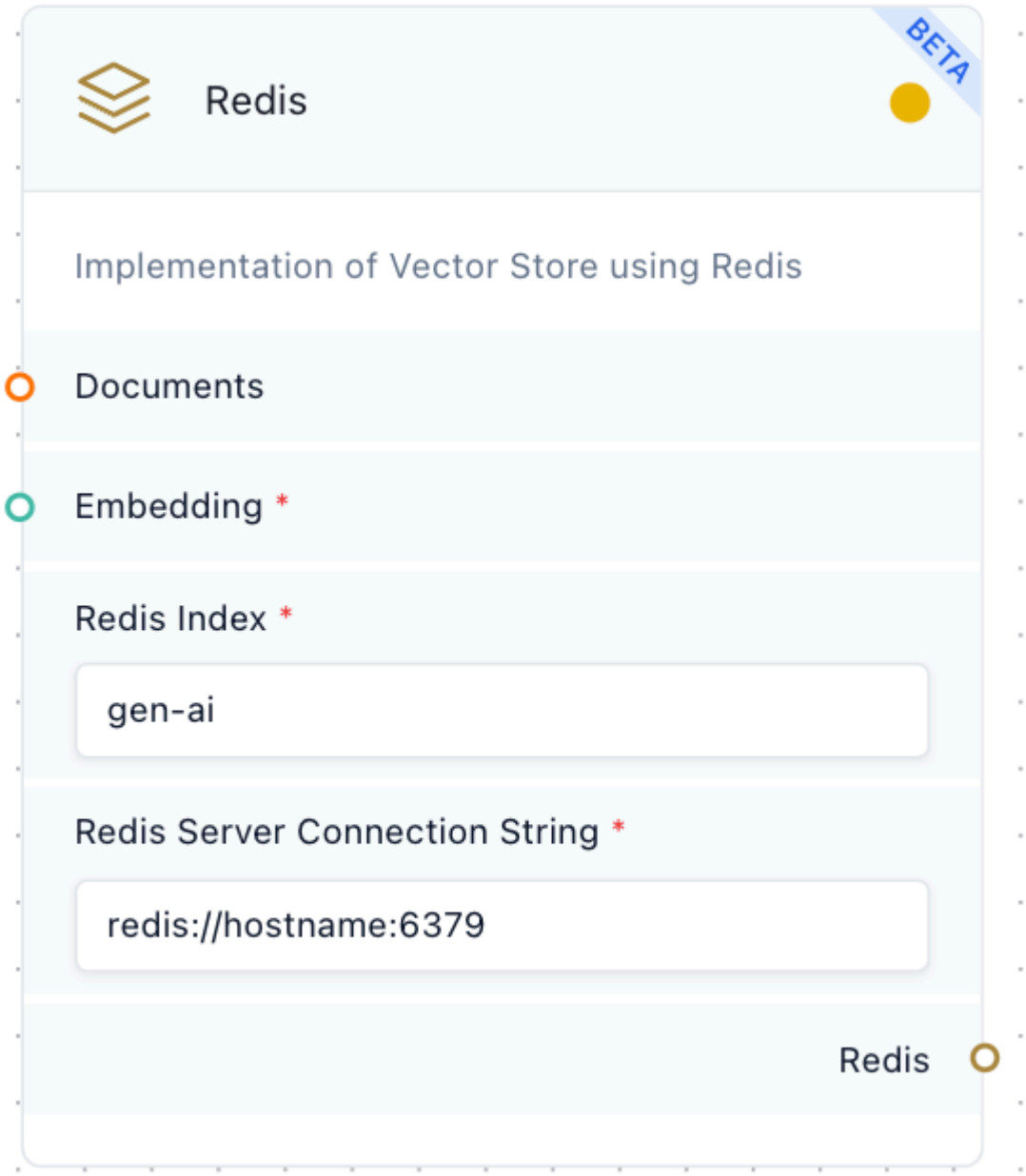
Redis is an open-source, in-memory vector store. It offers low-latency reads and writes, making it suitable for use cases that require a cache.

Parameters

- Documents: The chunked or loaded document component.
- Embeddings: Embedding component to convert documents into vectors.
- Redis Index: A unique identifier for the index created in Redis

- Redis server connection string: The server connection string is the address and credentials for connecting to the Redis server

Example usage:



Retrievers

A retriever is an interface that returns documents given an unstructured query. It is more general than a vector store. A retriever does not need to be able to store documents, only to return (or retrieve) them. Vector stores can be used as the backbone of a retriever, but there are other types of retrievers as well. Retrievers accept a string query as input and return a list of Documents as output. retrievers help you search and retrieve information from your indexed documents.

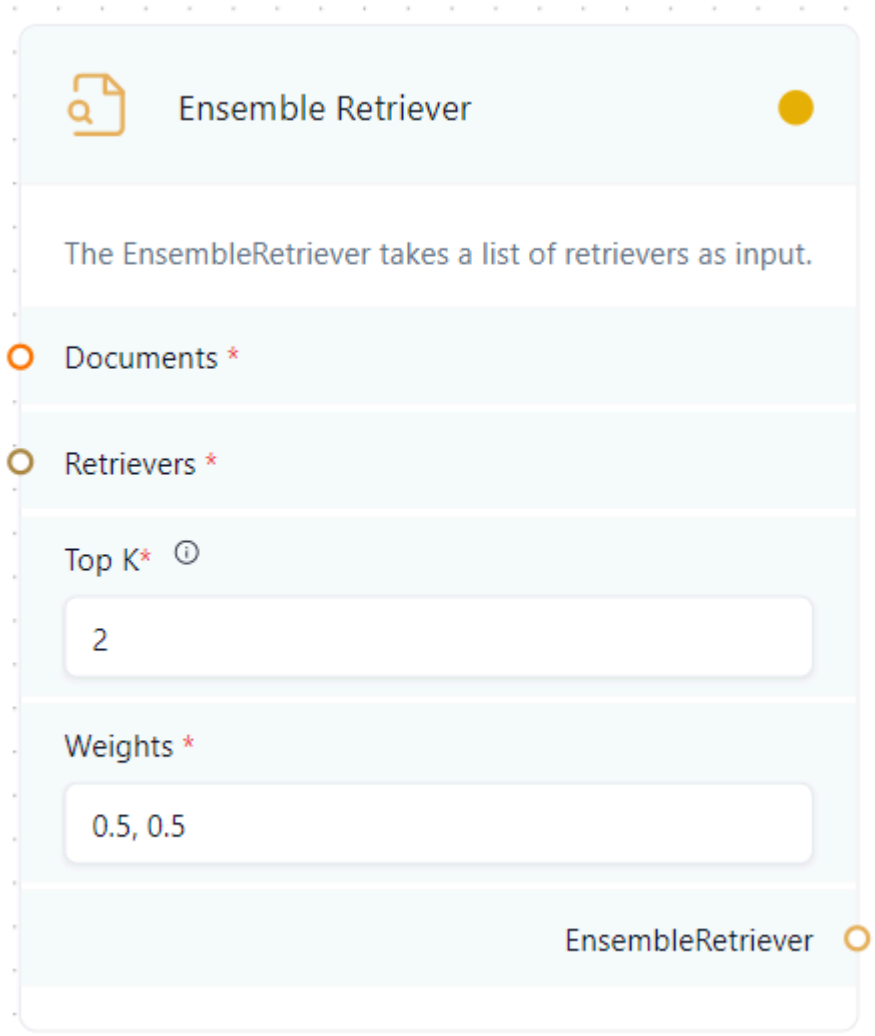
Ensemble Retriever

The EnsembleRetriever improves retrieval performance by combining outcomes from multiple retrievers using the Reciprocal Rank Fusion algorithm. Typically, it combines a sparse retriever like BM25 with a dense retriever such as embedding similarity, capitalizing on their complementary strengths to create a hybrid search system. This integration enhances retrieval accuracy compared to using individual algorithms alone.

Params

- **retrievers** – A list of retrievers to ensemble.
- **weights** – A list of weights corresponding to the retrievers. Defaults to equal weighting for all retrievers.
- **c** – A constant added to the rank, controlling the balance between the importance of high-ranked items and the consideration given to lower-ranked items.

Example



To use the Ensemble Retriever Component, the parameters **Top K**(the number of results to return) and **Weights** are to be provided by the User as input. The component is then to be connected to the Documents(Chains, Loaders, Utilities) and Retriever(Vector Stores) Components.

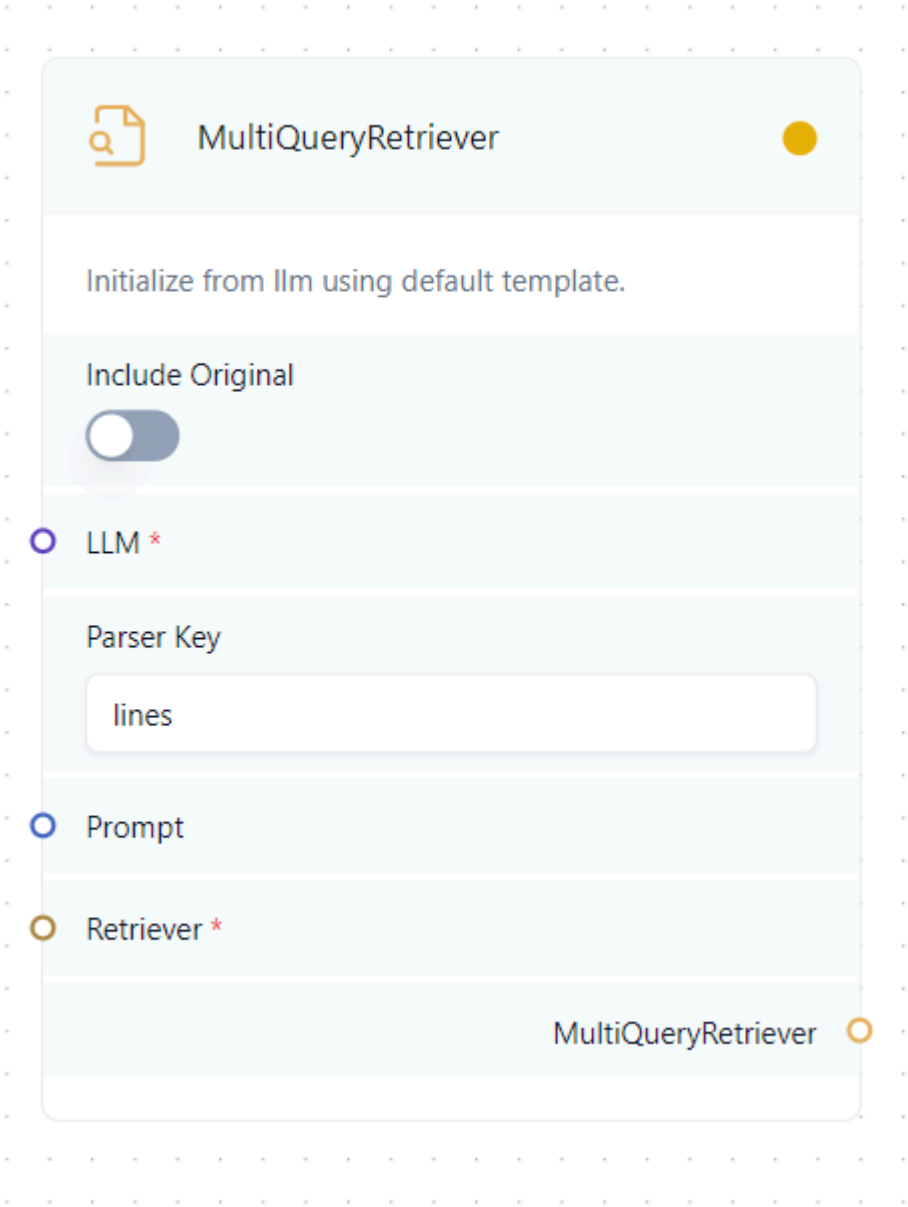
MultiQuery Retriever

Automating prompt tuning, the MultiQuery Retriever employs an LLM to generate diverse queries from various perspectives for a user input query. Retrieving relevant documents for each query, it then combines the unique union of all results to obtain a broader set of potentially relevant documents. By offering multiple viewpoints on the same inquiry, the MultiQuery Retriever mitigates the challenges of distance-based retrieval, potentially enriching the retrieved results.

Params

- question – the query provided by the user
- llm- the LLM used for query generation

Example



To use the MultiQuery Retriever, the component is to be to connected to an LLM, Prompt and Retriever Component

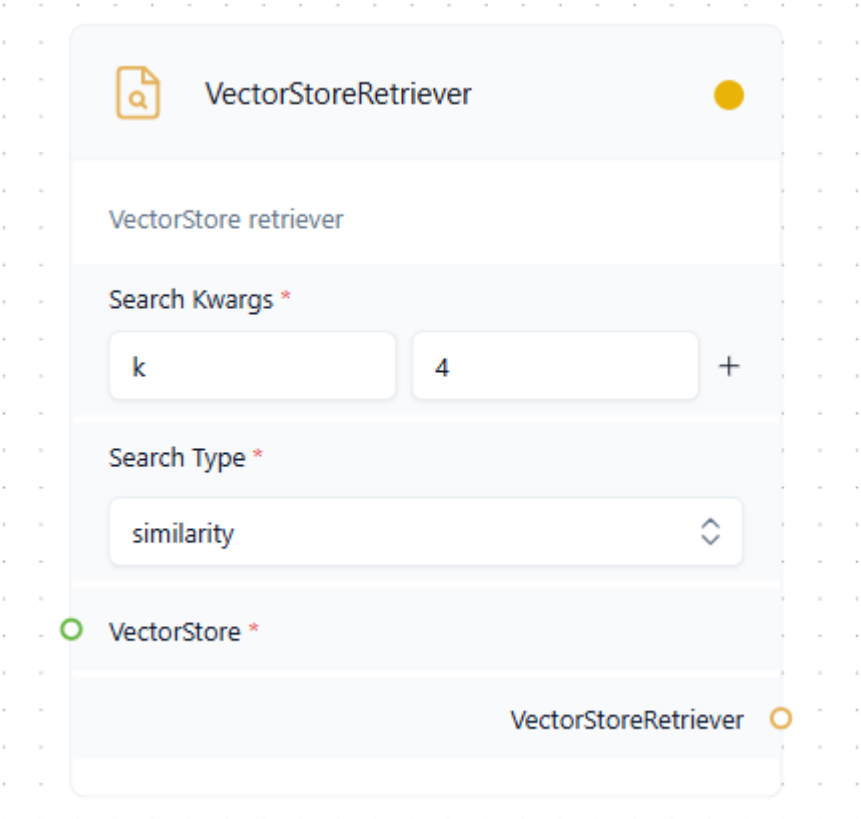
VectorStore Retriever

The VectorStoreRetriever is a retriever that utilizes vector similarity to retrieve documents based on an unstructured query. This retriever is typically backed by a vector store, where documents are embedded as vectors in high-dimensional space. Using similarity search, it finds documents most closely aligned with the input query, making it suitable for semantic search applications. This approach is particularly effective for queries where context or meaning matters, as it retrieves documents based on semantic similarity rather than exact keyword matching.

Params

- **Search Type** – The type of similarity measure to use (e.g., similarity, distance).
- **Search Kwargs (k)** – The number of top results to retrieve, where `k` represents the number of closest matches to the query.

Example:



Multi Modals

Multi Modals

Multi-modal is a specialized custom component that utilizes advanced models, such as diffusion models or speech generators, to create images or voice outputs from a simple text prompt. Currently, we have two Multi-Modals available: OpenAITextToImage and OpenAITextToSpeech.

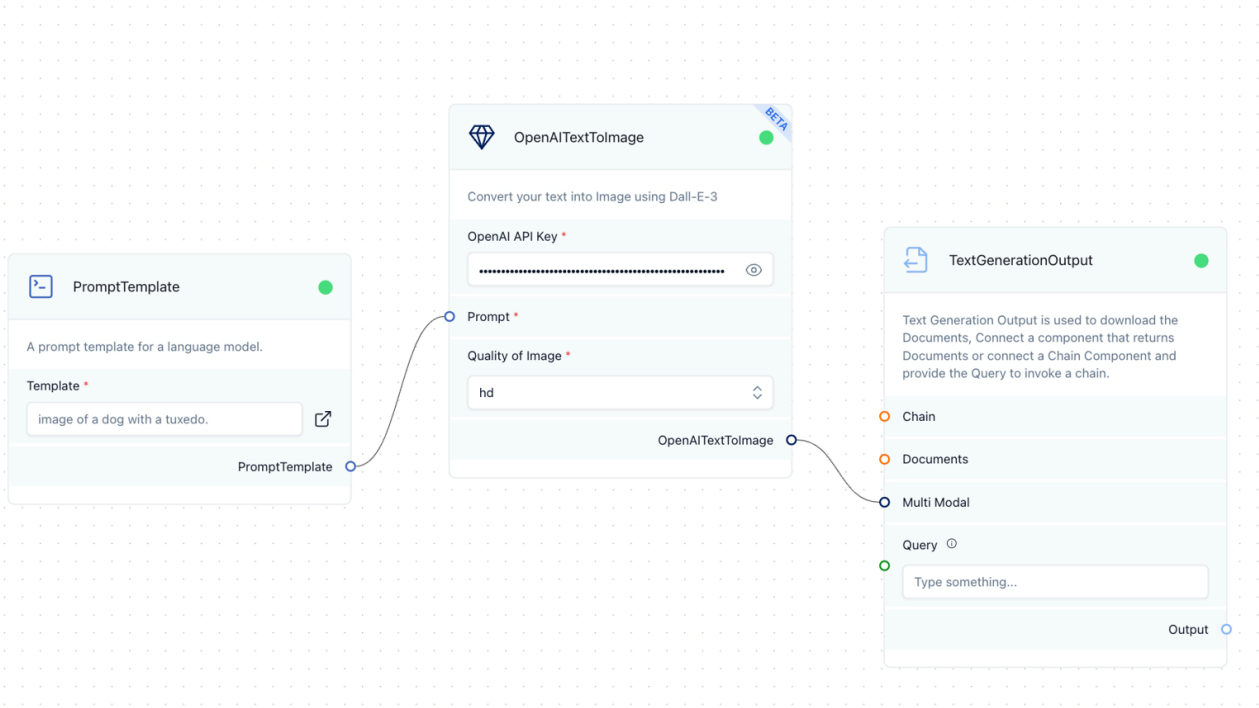
OpenAITextToImage

This component uses the `Dall-e-3` model from OpenAI behind the hood. Users need to input specific parameters to generate the desired output.

Parameters

- **OpenAI API Key:** Key used to authenticate and access the OpenAI API.
- **Prompt:** Prompt template or ChatPrompt Prompt, that contains the prompt to be instructed for the component.
- **Quality of the Image:** This refers to the visual quality of the image, which can be either standard or High Definition (HD).

Example Usage



using the OpenAITextToImage to generate an image

The OpenAITextToImage component requires an `OpenAI API Key` that you can get from https://platform.openai.com/↗. The Prompt can be specified using a simple `PromptTemplate`. The `Quality of Image` can be set either to HD or standard. This component returns an `Ouput`.

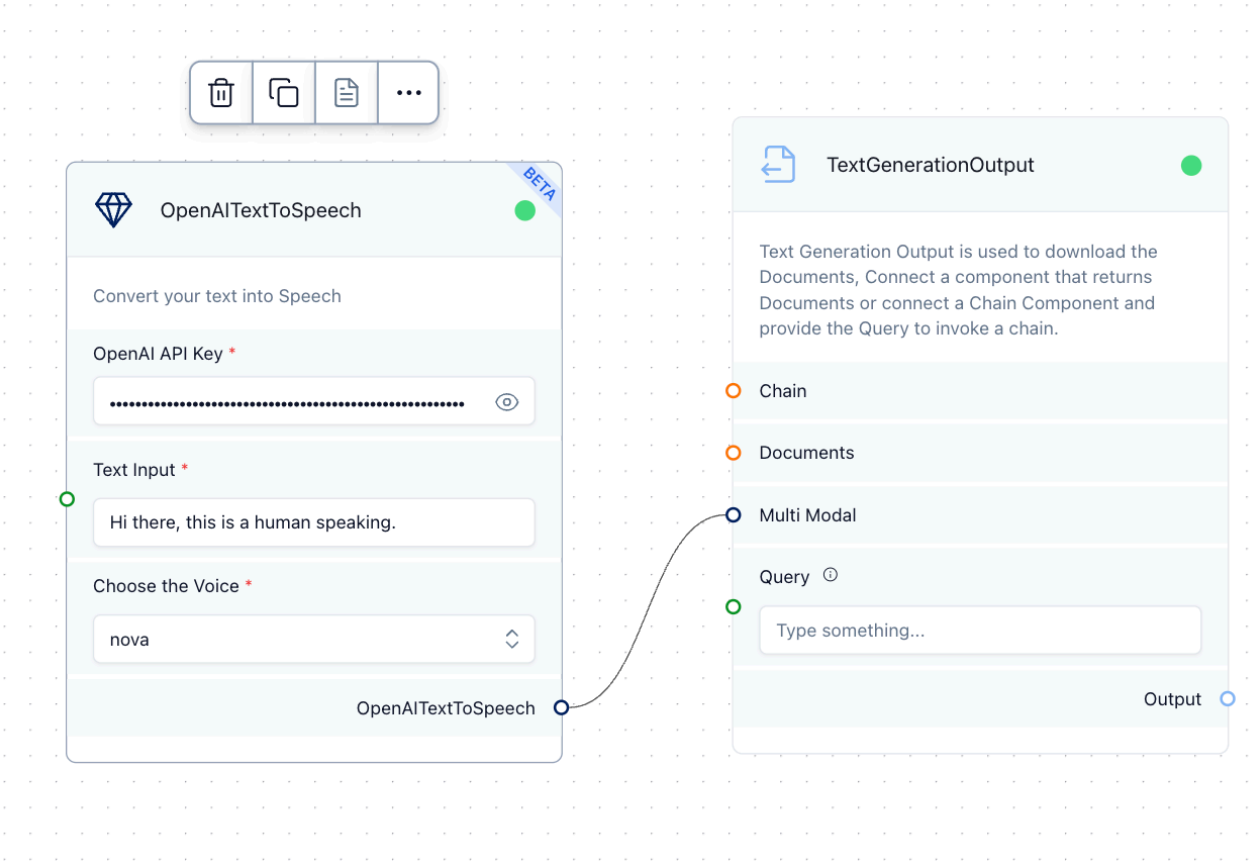
OpenAITextToSpeech

This component uses the `tts-1` model from OpenAI in the background. Users can generate a voice speech with a specified vocal tone, using this component.

Parameters

- **OpenAI API Key:** Key used to authenticate and access the OpenAI API.
- **Text Input:** The simple text prompt, which will be converted to speech.
- **Choose a Voice:** This option lets us choose the type of vocal tone for generating the speech.

Example Usage



using the OpenAITextToSpeech to generate a voice speech

The OpenAITextToSpeech component requires an `OpenAI API Key` that you can get from https://platform.openai.com/ ↗. The `text input` is the field that gets converted to the speech. The `Choose the Voice` option allows you to adjust the type of vocal tone in the output speech. This component returns an `Ouput`.

Agents

Agents utilize language models as reasoning engines to dynamically select actions based on context and goals, unlike hardcoded chains. By analyzing the current state and anticipating outcomes, agents make informed decisions on action sequences, offering flexibility and efficiency.

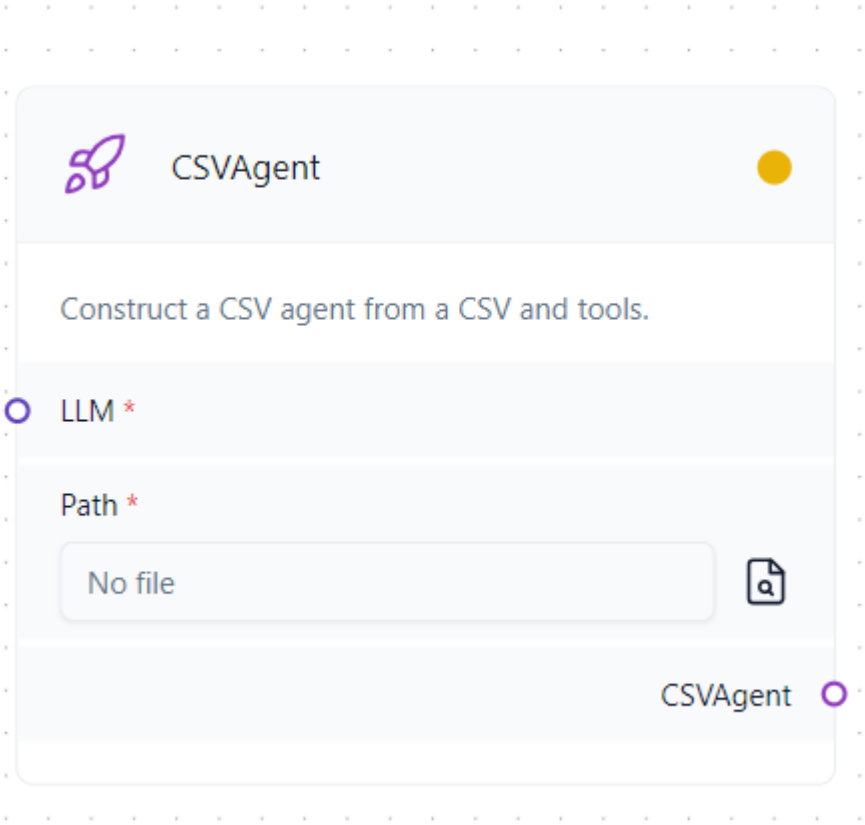
CSVAgent

The CSVAgent function in LangChain is used to create a CSV agent by loading data into a pandas DataFrame and using a pandas agent.

Params

- **llm** – Language model to use for the agent.
- **path** – A string path, file-like object or a list of string paths/file-like objects that can be read in as pandas DataFrames with pd.read_csv().

Examples



To use the CSVAgent Component, the user is to provide the necessary CSV file through the path input. The component is also to be connected to an LLM component.

Large Language Models

An LLM, or Large Language Model, is a fundamental element of GenAI Stack as the generator RAG model. This offers a standardized interface to seamlessly engage with various LLMs from providers like OpenAI, Anthropic, Cohere, and HuggingFace.

Note : GenAI Stack does not host its own LLMs but rather provides a universal interface, enabling interaction with diverse LLMs across the platform, particularly within chains and agents. The LLM class in GenAI Stack serves as a standardized interface for multiple LLM providers, ensuring consistency in handling input strings and generating corresponding text outputs.

```
All LLM integrations are individual components that do not require any in
Chains and Agents serve as the ideal components to connect with the LLM.
```

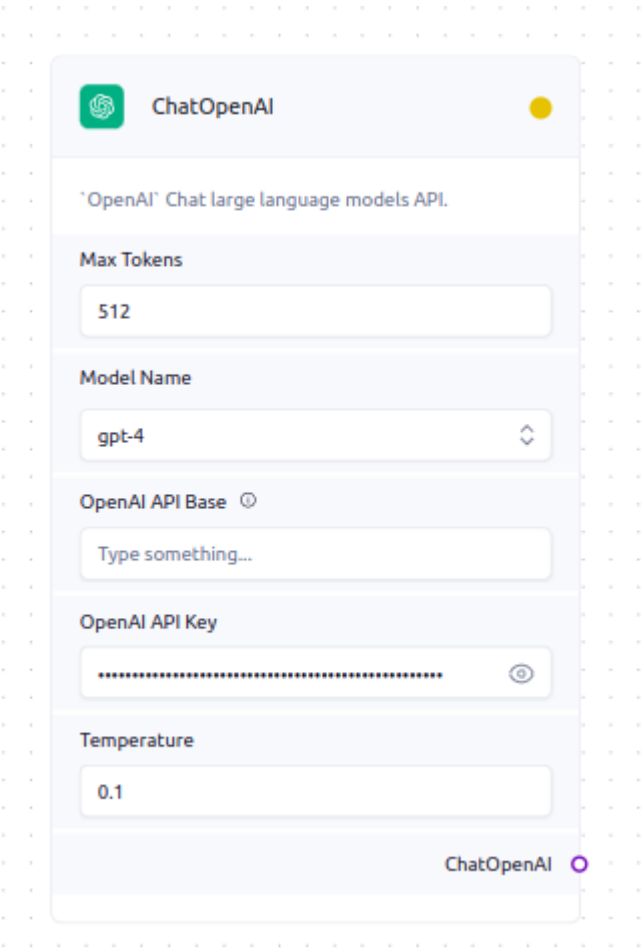
ChatOpenAI

ChatOpenAI is a chat model provided by OpenAI which is trained on instructions dataset in a large corpus.

Parameters

- **OpenAI API Key** : Key used to authenticate and access the OpenAI API.
- **Max Tokens :** The output sequence length response from the model.
- **Model Name :** Defines the OpenAI chat model to be used in eg: GPT3 and GPT4 series.
- **OpenAI API Base :** Used to specify the base URL for the OpenAI API. It is typically set to the API endpoint provided by the OpenAI service.
- **Temperature** : It can be used to control the randomness or creativity in responses.

Example Usage



ChatOpenAI supports both GPT3.5 and GPT4 models. You can access it after paying for the credits at platform.openai.com.

AzureChatOpenAI

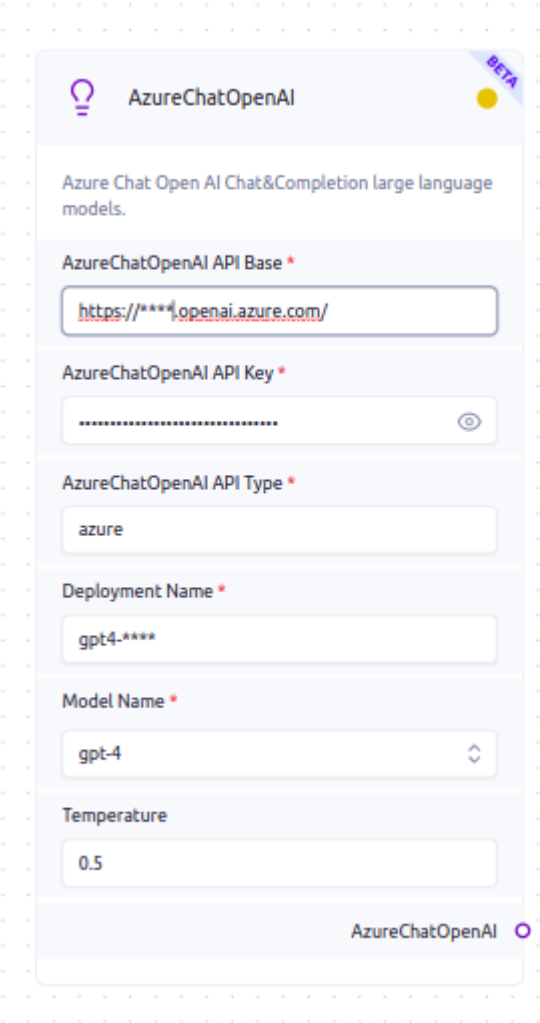
Azure OpenAI Service provides REST API access to OpenAI's powerful language models including the GPT-4, GPT-3.5-Turbo, and Embeddings model series.

Parameters

- **AzureChatOpenAI API Base :** Azure Cloud endpoint base URL
- **AzureChatOpenAI API Key:** Azure api key for AzureChatOpenAI service
- **AzureChatOpenAI API Type :** by default enter azure
- **Deployment Name :** Enter the the deployment name that is created on Model deployments on Azure

- **Model Name :** AzureChatOpenAI enables the access to GPT4 models.
- **API Version :** API Version property depends on the method you are calling in the API. This is mainly the datetime.
- **Max Tokens :** The maximum sequence length for the model response.
- **Temperature :** It can be used to control the randomness or creativity in responses.

Example Usage



You can obtain your Azure OpenAI credentials from Azure AI Studio services. The instructions are identical to those mentioned for Azure Embeddings.

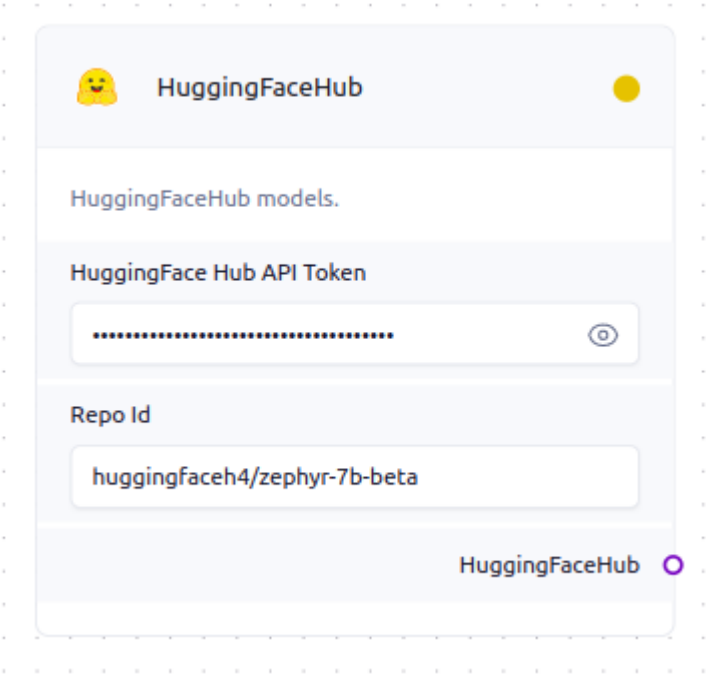
HuggingFaceHub

The Hugging Face Hub is a platform with over 350k models, 75k datasets, and 150k demo apps (Spaces), all open source and publicly available, in an online platform where people can easily collaborate and build ML together.

Parameters

- **Repo Id :** Model name from the HuggingFace Hub – defaults to gpt2.
- **HuggingFacehub API Token :** HuggingFace Access Token to run the model on Inference API.
- **Model Kwargs** : Few parameters that need to be configured to get better results from the model. Such max_new_tokens.

Example Usage



To utilize HuggingFace Hub, you must initially register an account at huggingface.co ↗ and obtain your Access Token. With HuggingFace Hub you need not have to load the model. Utilizing 7B parameter models with HuggingFace Hub is free of charge.

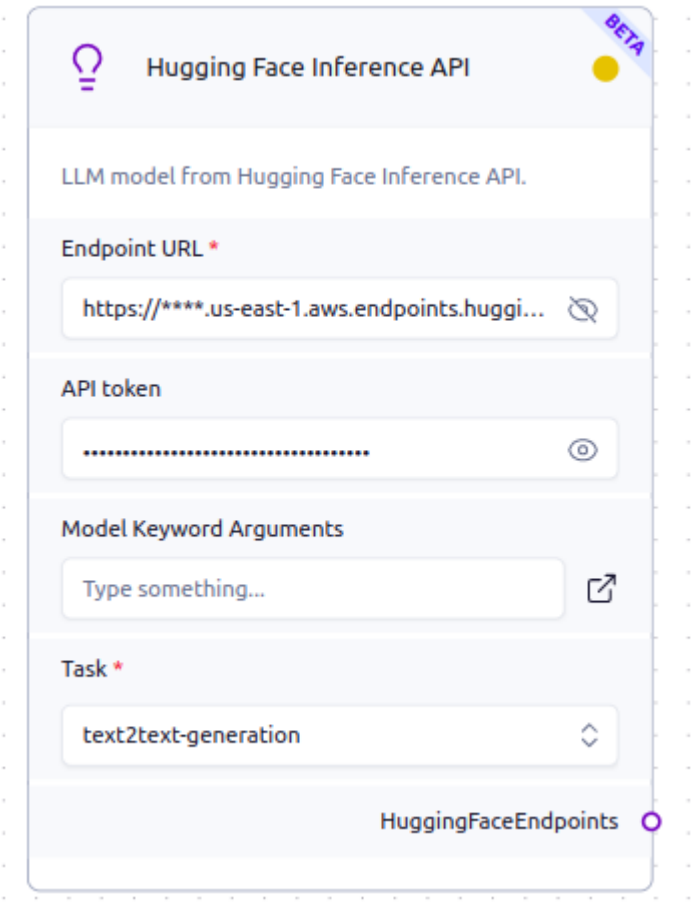
HuggingFace Inference API

The HuggingFace Inference API relies on the Inference Endpoint provided by HuggingFace.

Parameters

- Endpoint URL: Deployed GPU instances cloud URL from HuggingFace Inference endpoint.
- API Token: HuggingFace Access Token to run the inference.
- Model Keyword Arguments: This is model_kwargs that contains max_new_tokens, top_p, top_k, temperature and other arguments that is used in Transformers pipeline.
- Task: Currently inference endpoint support text2text-generation", "text-generation", "summarization". Source: https://github.com/langchain-ai/langchain/blob/370becdfc2dea35eab6b56244872001116d24f0b/langchain/llms/huggingface_endpoint.py ↗

Example Usage



It's important to note that while the large language models deployed on HuggingFace GPU instances are based on open source models, they are not free to use; access requires payment. HuggingFace Endpoint supports GCP, Azure and AWS cloud services.

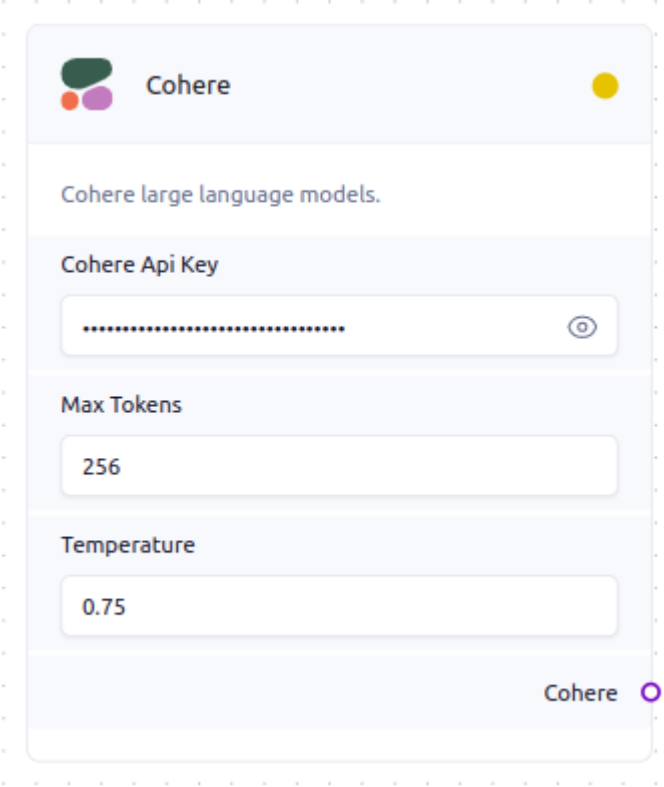
Cohere

Cohere, a Canadian startup, offers natural language processing models aimed at enhancing human-machine interactions for businesses. It is renowned for its expertise in semantic search and embedding models.

Parameters

- **Cohere API Key**: You need to create new API from Cohere dashboard.
- **Max Tokens**: The output sequence length response from the model.
- **Temperature**: It can be used to control the randomness or creativity in responses.

Example Usage



Cohere is also a close source Large Language model and requires API key to do the inference. It provides free usage upto certain token limits.

OpenAI

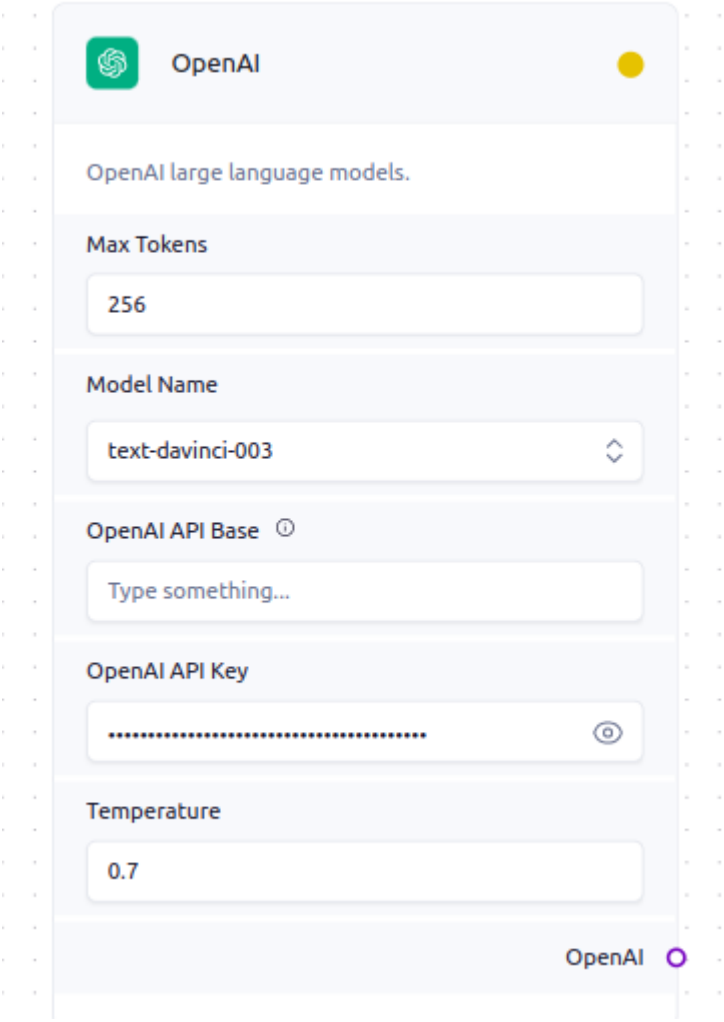
OpenAI is a Large Language Model based on GPT 3.5 model. ChatOpenAI is instruct based, whereas OpenAI model is generic LLM. OpenAI only supports GPT 3 based models and doesn't support GPT-4 model as of now.

Parameters

- **Max Tokens :** The output sequence length response from the model.
- **OpenAI API Key** : Key used to authenticate and access the OpenAI API.
- **Model Name :** Defines the OpenAI chat model to be used in eg: GPT3 and GPT4 series.
- **OpenAI API Base :** Used to specify the base URL for the OpenAI API. It is typically set to the API endpoint provided by the OpenAI service.

- **Temperature** : It can be used to control the randomness or creativity in responses.

Example Usage



The arguments are same as ChatOpenAI, but OpenAI LLM doesn't support GPT4.

Vertex AI

Vertex AI empowers ML to take their project to deployment. Here in this context Vertex AI provides Model Garden and Generative Studio that supports base foundational model such as PaLM LLM.

Parameters

- **Credentials**: Once you create a Google Cloud Platform(GCP) account, download the creds json file and upload the file as your credentials.
- **Model Name**: Vertex AI contains PaLM as LLM. You need pick a base foundation model provided under PaLM such as `text-bison` (default), or `text-unicorn`.
- **Location**: Region on which the API calls needs to take place, us-central1 by default.
- **Max Retries**: Request iteration to the server to make API call.
- **Metadata**: To define the source, can be None.
- **Project**: Add your GCP project to make API call.
- **Temperature**: It can be used to control the randomness or creativity in responses.
- **Streaming** (bool): If enabled, Vertex AI streams the output response.

Amazon Bedrock

Amazon Bedrock is a comprehensive managed service providing access to a variety of high-performance foundation models (FMs) sourced from prominent AI companies such as AI21 Labs, Anthropic, Cohere, Meta, Stability AI, and Amazon. These models are accessible through a unified API, accompanied by a wide range of functionalities essential for constructing generative AI applications, all while ensuring security, privacy, and responsible AI practices.

Parameters

- **Credentials Profile Name**: Fetch your serverless profile name from AWS, e.g., bedrock-admin
- **Model Id**: Amazon Bedrock contains model support from A121 Labs, Anthropic, Cohere and Meta, using the dropdown option, you can pick a releavant model of your choice.

Memories

Memory component ensures that all conversations are stored in a buffer, enabling the ChatBot to leverage chat history for more contextually relevant responses and facilitating the ability to ask follow-up questions.

```
Note: Most of these components have the same input and output components
```

ConversationBufferWindowMemory

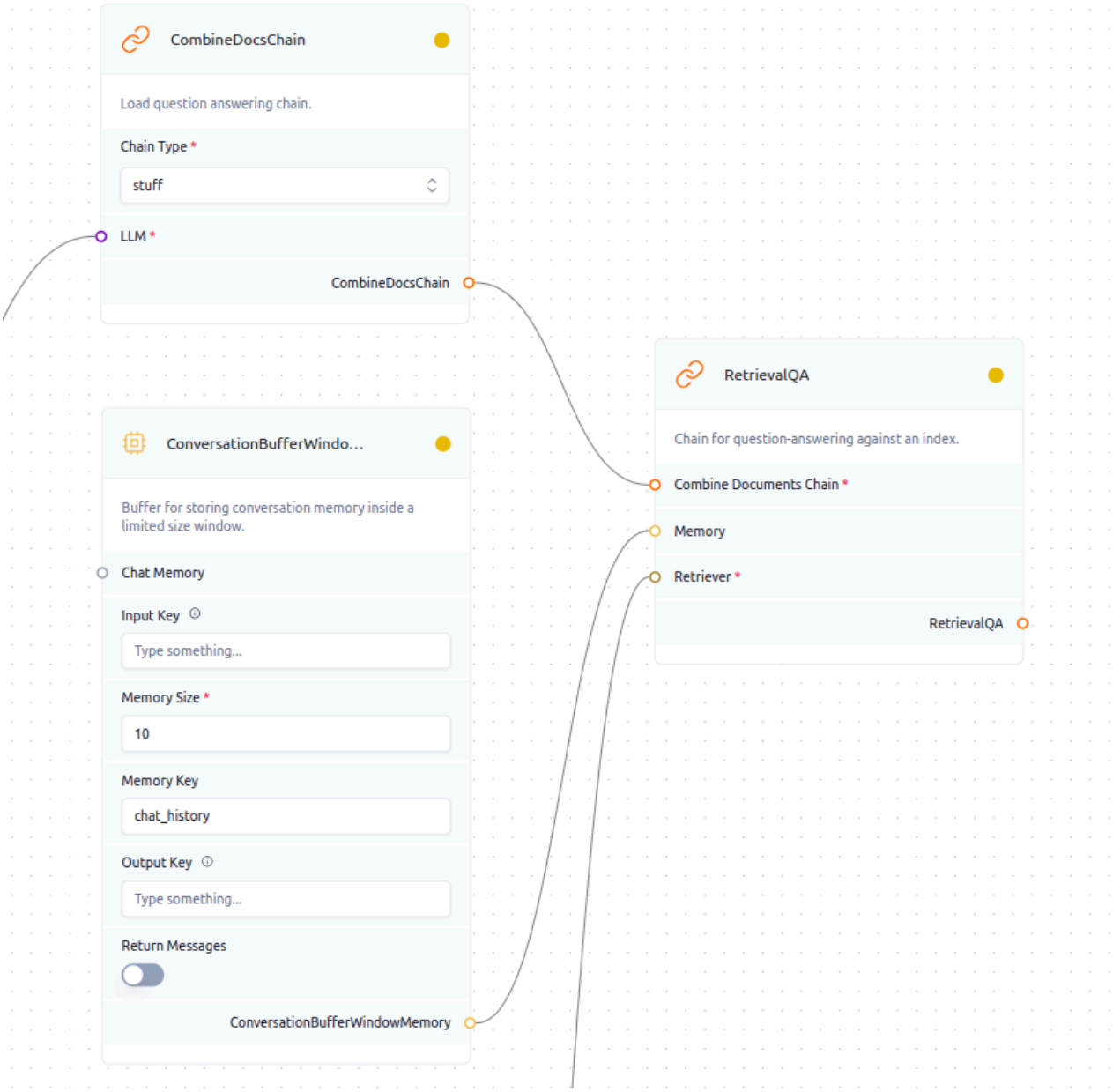
The ConversationBufferWindowMemory maintains a record of all the exchanges that occur during a conversation, but it only retains the last K interactions.

For example: Set a low k=5, to only keep the last 5 interactions in memory.

Parameters

- **input_key**: Input key is the human query that is defined in the prompt. This key provide the input user query to the chain.
- **memory Size (k)**: Window size to retain the past interactions
- **memory_key**: Memory key is where all the input querys as stored i.e., chat_history.
- **output_key**: If you have multiple input variables and if you want to return one key to another sequential that time you define output_key.
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



Chat memory is not needed for Conversation Buffer Window memory. This component stores a fixed size rolling window (of length = memory size) of previous chat messages between the system and user to be passed to the LLM. Keep in mind the context length of the LLM when specifying the Memory size.

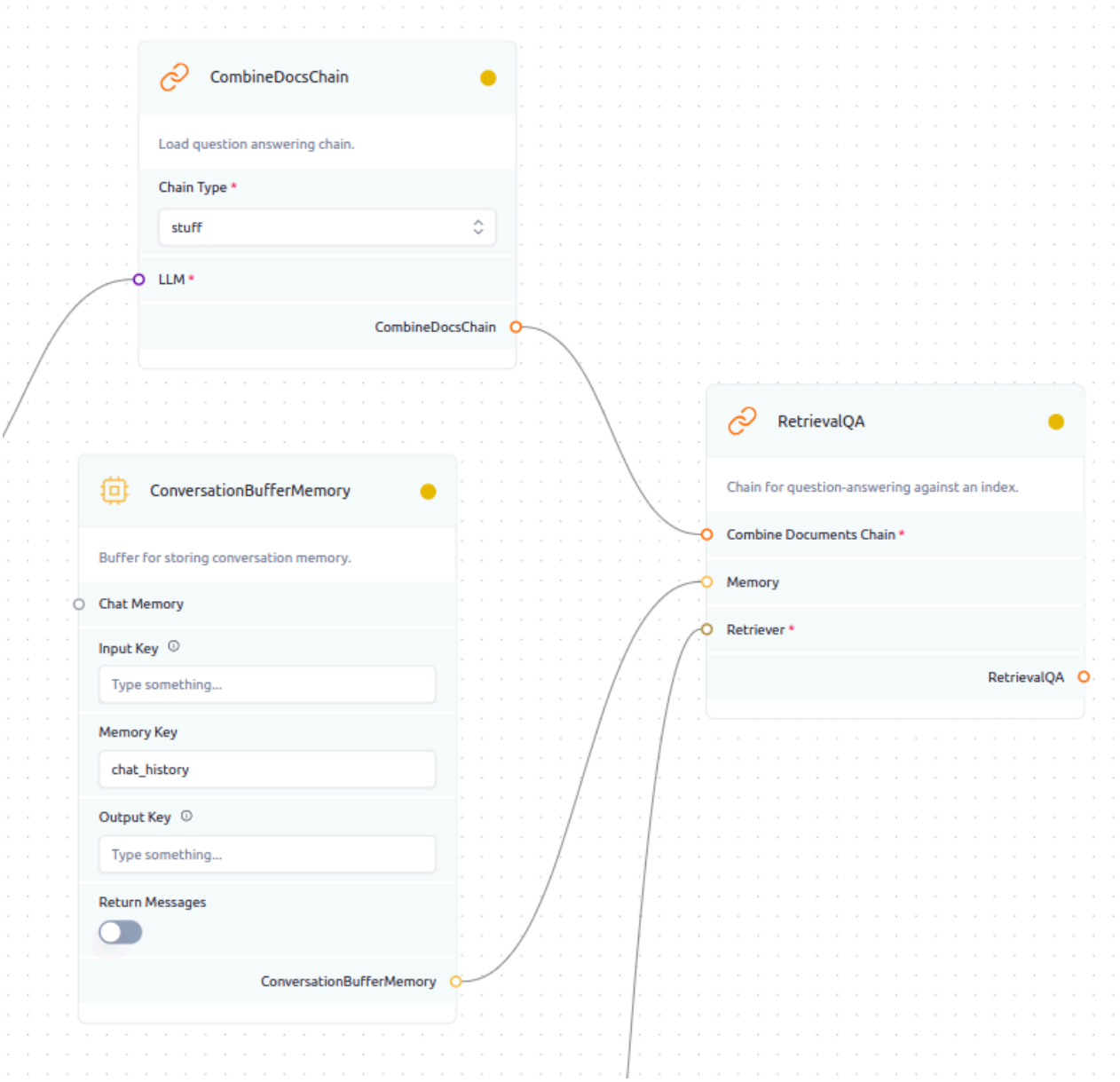
ConversationBufferMemory

This memory facilitates the storage of messages and subsequently retrieves them into a variable.

Parameters

- **input_key**: Input key is the human query that is defined in the prompt. This key provide the input user query to the chain.
- **memory_key**: Memory key is where all the input querys as stored i.e., chat_history.
- **output_key**: If you have multiple input variables and if you want to return one key to another sequential that time you define output_key.
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



This component works similar to the previous one, but it stores the entire chat history instead of limited number of chat messages. However on long conversations this isn't a good option.

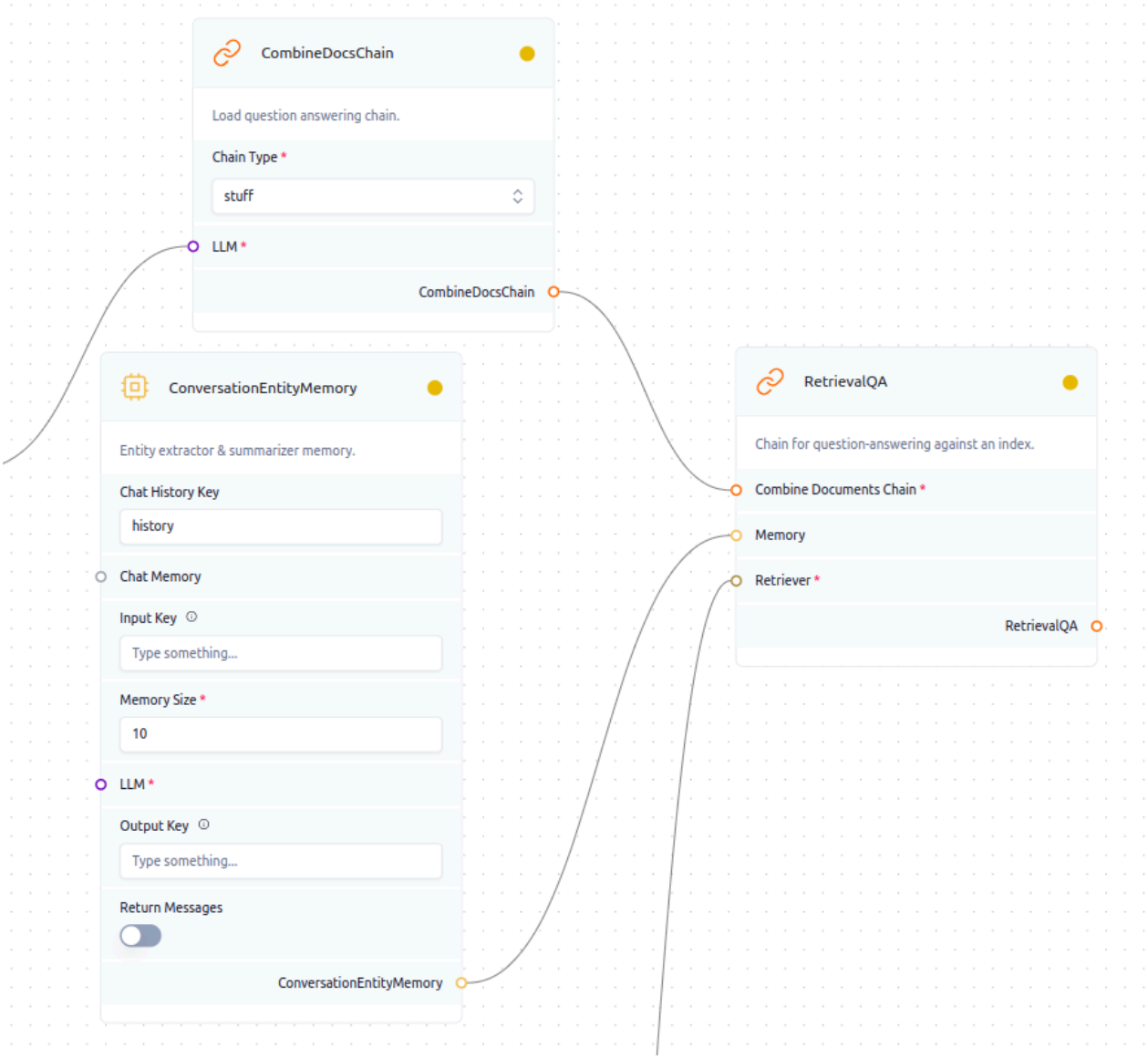
ConversationEntityMemory

The Entity Memory stores factual information pertaining to particular entities discussed within a conversation. It utilises a LLM (Large Language model) to extract details about these entities and gradually enhances its understanding of them over time

Parameters

- **LLM**: Add a large language model to extract entities.
- **chat_history_key**: Unique id for individual entity.
- **input_key**: Input key is the human query that is defined in the prompt. This key provide the input user query to the chain.
- **memory Size (k)**: Window size to retain the past interactions
- **output_key**: If you have multiple input variables and if you want to return one key to another sequential that time you define output_key.
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



ConversationEntityMemory extracts named entities from the chat history using an NLP model and generates summaries about those entities and their context. This is stored to be provided to the LLM instead of the entire chat history.

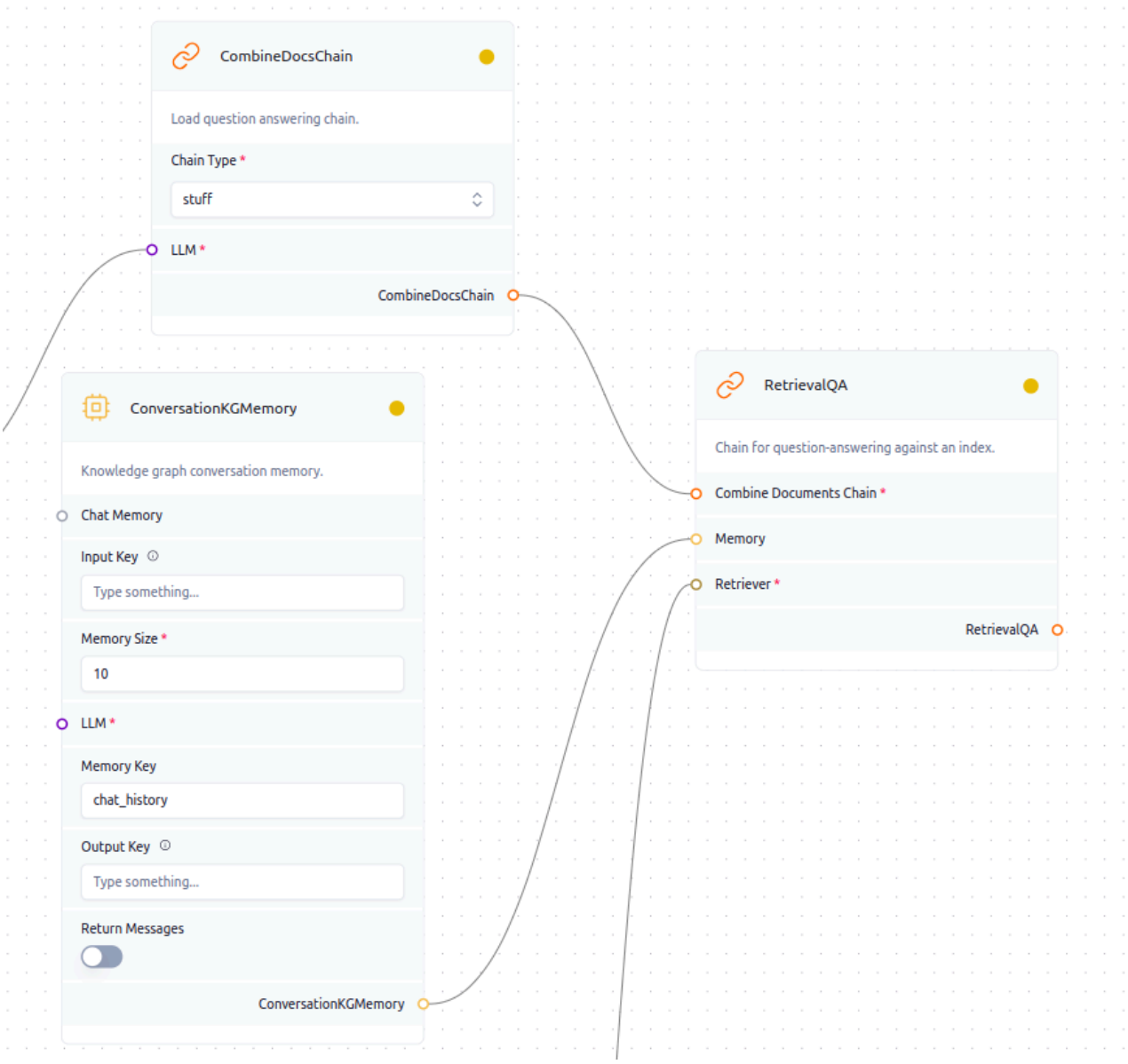
ConversationKGMemory

This type of memory uses an NLP model to extract subject-predicate-object style knowledge triples from the chat messages. This is then stored in a knowledge graph to recreate memory.

Parameters

- **input_key**: The variable to be used as Chat Input when more than one variable is available.
- **memory Size (k)**: Window size to retain the past interactions
- **LLM**: Add a large language model to extract entities.
- **memory_key**: Memory key is where all the input queries are stored i.e., chat_history.
- **output_key**: If you have multiple input variables and if you want to return one key to another sequential that time you define output_key. The variable to be used as Chat Output (e.g. answer in a ConversationalRetrievalChain)
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



The knowledge triples stored in an integrated knowledge graph are used in future conversations by querying the knowledge graph to retrieve relevant facts about mentioned entities to provide as additional context to the LLM.

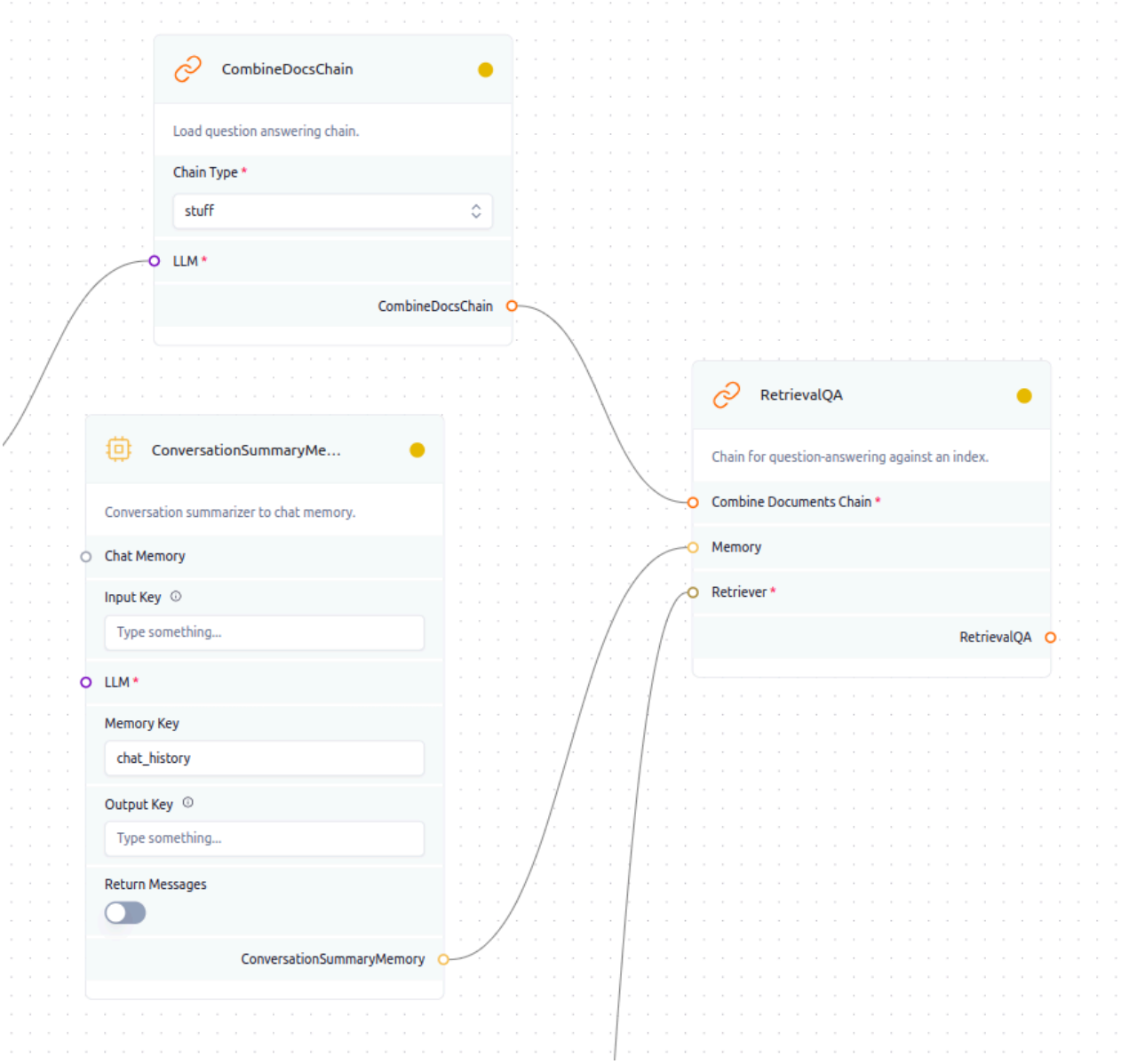
ConversationSummaryMemory

Conversation summary memory summarises the conversation as it happens and stores the current summary in memory. This memory can then be used to inject the summary of the conversation so far into a prompt/chain. This memory is most useful for longer conversations, where keeping the past message history in the prompt verbatim would take up too many tokens.

Parameters

- **input_key**: The variable to be used as Chat Input when more than one variable is available.
- **LLM**: Add a large language model to extract entities.
- **memory_key**: Memory key is where all the input queries are stored i.e., chat_history.
- **output_key**: If you have multiple input variables and if you want to return one key to another sequential that time you define output_key. The variable to be used as Chat Output (e.g. answer in a ConversationalRetrievalChain)
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



This component helps reduce prompt length for long conversations compared to showing all messages by generating summaries of the chat history periodically, condensing the content to be passed to the LLM.

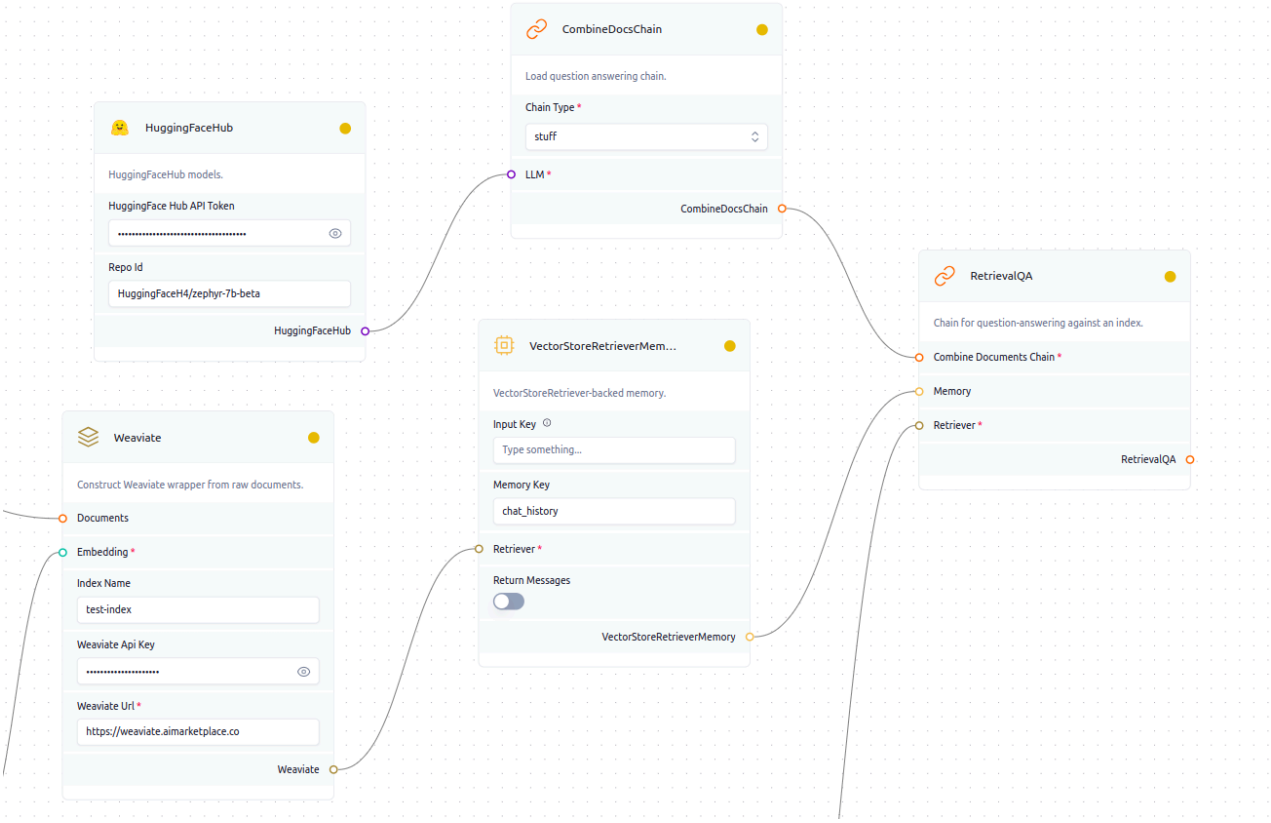
VectorstoreRetrieverMemory

VectorStoreRetrieverMemory stores previous interactions in a vector database, where each snippet is represented as a point in high-dimensional space. When you ask a question, it searches this space for the snippets closest to your meaning, regardless of their initial order in the conversation.

Parameters

- **input_key**: The variable to be used as Chat Input when more than one variable is available.
- **memory_key**: Memory key is where all the input queries are stored i.e., chat_history.
- **retriever:** The vectorstore which is to be used as retriever.
- **return_messages (boolean)**: Default it is False. If set True or enabled, then the history will be returned as list of message.

Example usage:



In addition to the inputs specified for the previous components, this memory requires a vector store retriever (currently only supports weaviate). This component stores memories/conversation snippets in a vector store and queries the most relevant documents when required.

Chains

Chains in GenAI Stack are cohesive assemblies of interconnected and easily reusable components. They encapsulate a sequence of calls to various components such as models, document retrievers, other chains, etc., offering a streamlined and user-friendly interface to navigate through this sequence.

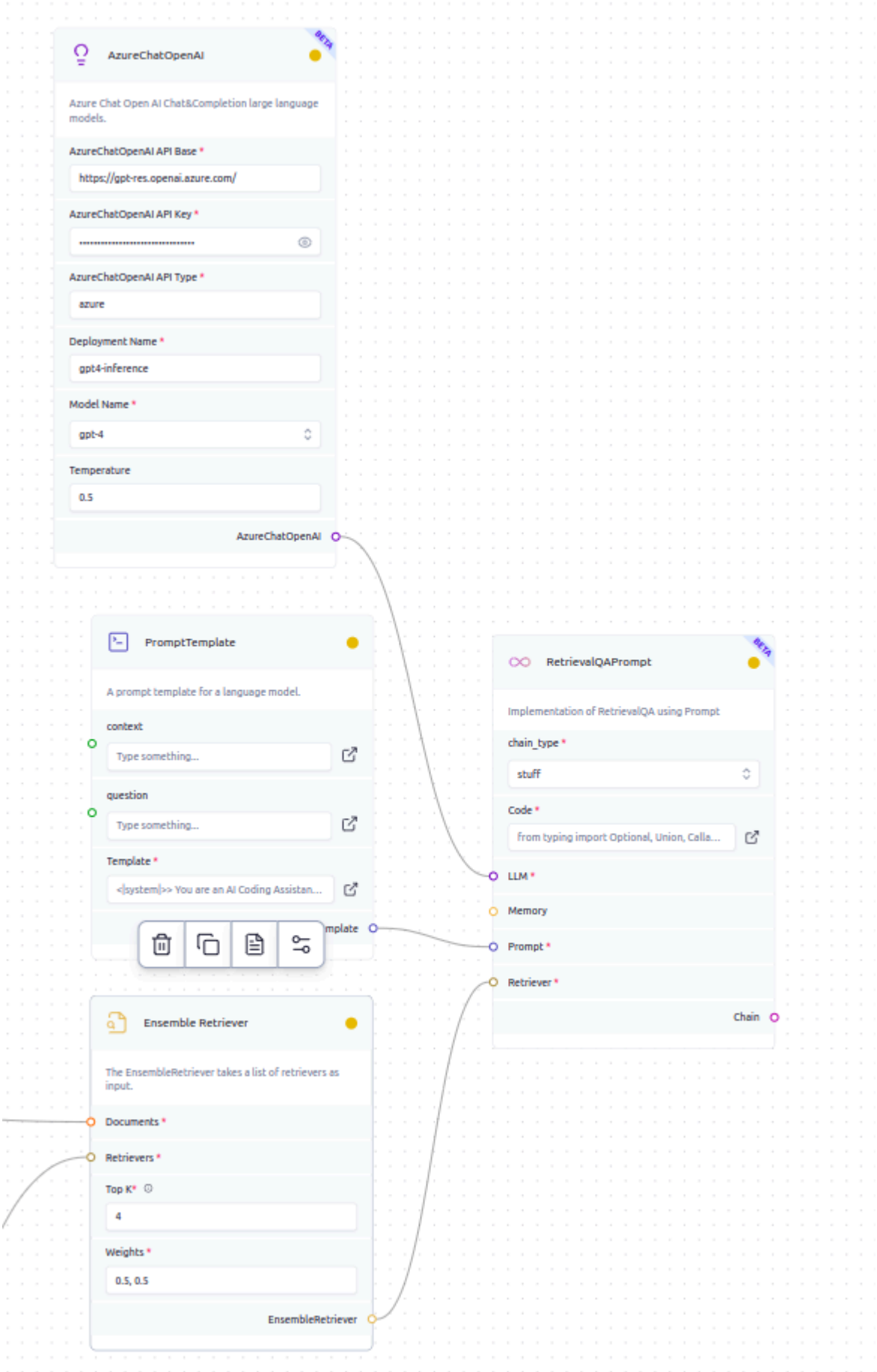
RetreivalQAPrompt

The Retrieval QA prompt is a chain component that allows the user to combine their own prompt with the retriever and generator models.

Parameters

- **chain_type:** Different chain types with distinct combination strategies. Methods:
  - stuff: Inserts a list of small documents into a prompt, suitable for applications with a few small documents.
  - map_reduce: Applies an LLM chain to each document individually and combines the outputs into a single result.
  - map_rerank: Runs an initial prompt on each document, scoring responses and returning the highest-scoring one.
  - refine: Constructs a response iteratively by updating answers for each document using an LLM chain in the GenAI Stack.
- **LLM:** Large Language Model integration used in the GenAI Stack.
- **Memory[Optional]:** To store the chat response in history.
- **Prompt:** Prompt template or ChatPrompt Prompt, that contains the prompt to be instructured for the chain.
- **Retriever:** The retriever used to fetch relevant documents.

Example usage:


AzureChatOpenAI
BETA
Azure Chat Open AI Chat&Completion large language models.
AzureChatOpenAI API Base *
https://gpt-res.openai.azure.com/
AzureChatOpenAI API Key *
AzureChatOpenAI API Type *
azure
Deployment Name *
gpt4-inference
Model Name *
gpt-4
Temperature
0.5
AzureChatOpenAI
PromptTemplate
A prompt template for a language model.
context
Type something...
question
Type something...
Template *
<|system|>> You are an AI Coding Assistan...
mplate
RetrievalQAPrompt
BETA
Implementation of RetrievalQA using Prompt
chain_type *
stuff
Code *
from typing import Optional, Union, Calla...
LLM *
Memory
Prompt *
Retriever *
Chain
Ensemble Retriever
The EnsembleRetriever takes a list of retrievers as input.
Documents *
Retrievers *
Top K*
4
Weights *
0.5, 0.5
EnsembleRetriever

Retrieval QA with prompt requires an LLM, an optional memory component, and it uses a Prompt to specify what the LLM should do/how it should behave in response to the user question by using the information in the context provided by a retriever component where we can also pass a vectorstore, (question and context will be the variables in your prompt).

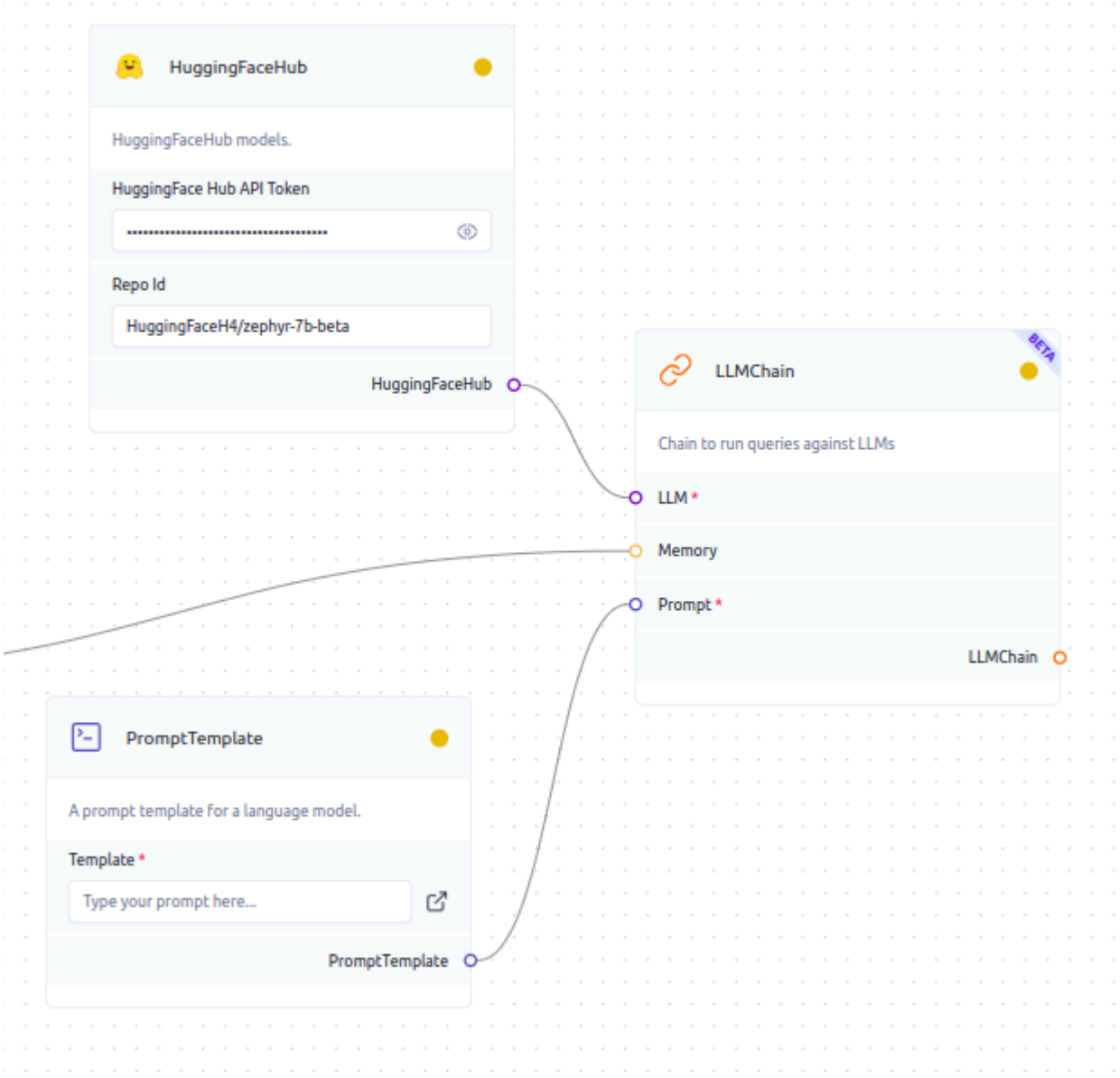
LLM Chain

A chain component that utilizes a large language model (LLM) to generate responses based on a given prompt.

Parameters

- **LLM:** Large Language Model integration used in the GenAI Stack.
- **Memory[Optional]:** To store the chat response in history.
- **Prompt:** Prompt template or ChatPrompt Prompt, that contains the prompt to be instructured for the chain.

Example usage:



The LLM chain takes ChatPromptTemplate or PromptTemplate, along with LLM and Memory to work as a simple chat bot without any external knowledge source, relying on the LLM provided.

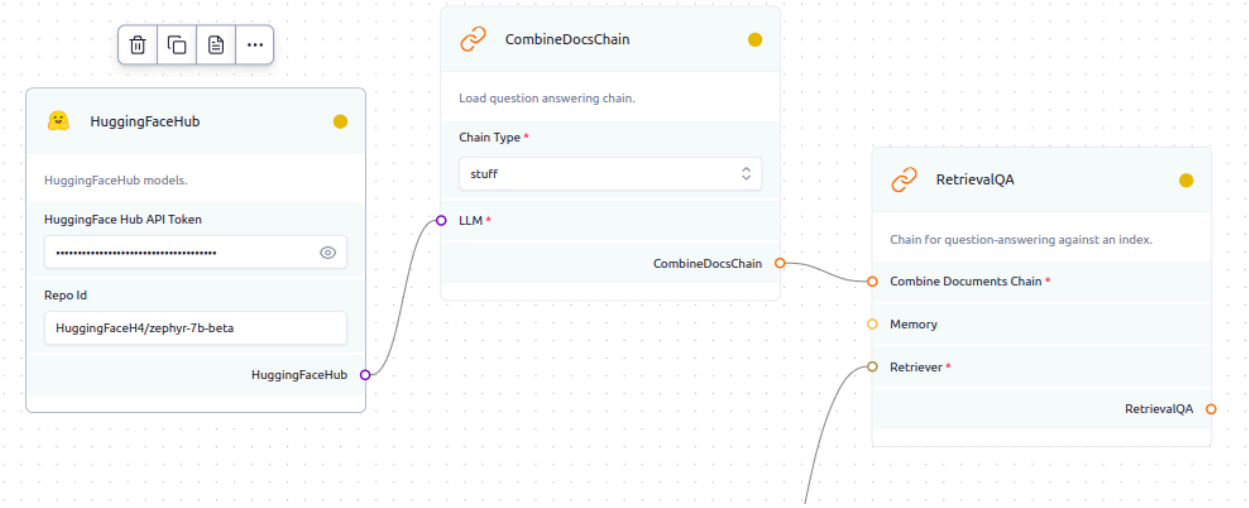
CombineDocsChain

A chain component that takes in a list of documents and combines them into a single prompt for the LLM to process.

Params:

- **LLM:** Large Language Model integration used in the GenAI Stack.
- **chain_type:** Different chain types with distinct combination strategies. Methods:
  - stuff: Inserts a list of small documents into a prompt, suitable for applications with a few small documents.
  - map_reduce: Applies an LLM chain to each document individually and combines the outputs into a single result.
  - map_rerank: Runs an initial prompt on each document, scoring responses and returning the highest-scoring one.
  - refine: Constructs a response iteratively by updating answers for each document using an LLM chain in the GenAI Stack.

Example Usage:



RetrievalQA Chain

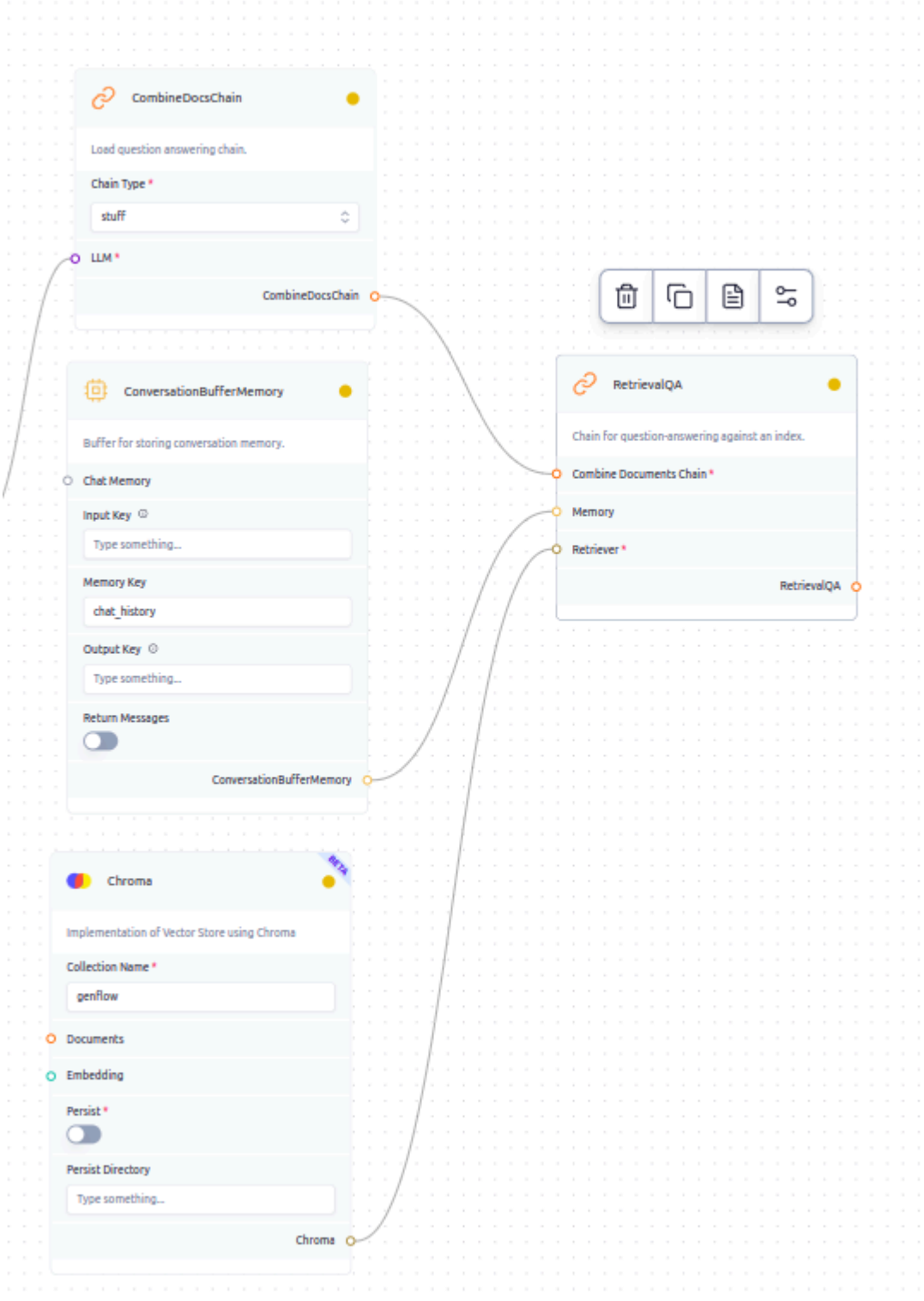
The RetrievalQA chain is a chain component that integrates a retriever to fetch relevant documents and an LLM to generate answers for a given question based on the retrieved documents.

Parameters

- **CombineDocsChain:** Chain to use to combine the documents with RetrievalQA chain type.
- **Memory[Optional]:** To store the chat response in history.

- **Retriever:** The retriever used to fetch relevant documents.

Example usage



The RetrievalQA chain requires a CombineDocChain which will have the LLM, and a Retriever to be passed optionally along with Memory. This acts as the final component that allows using the attached components to answer user queries.

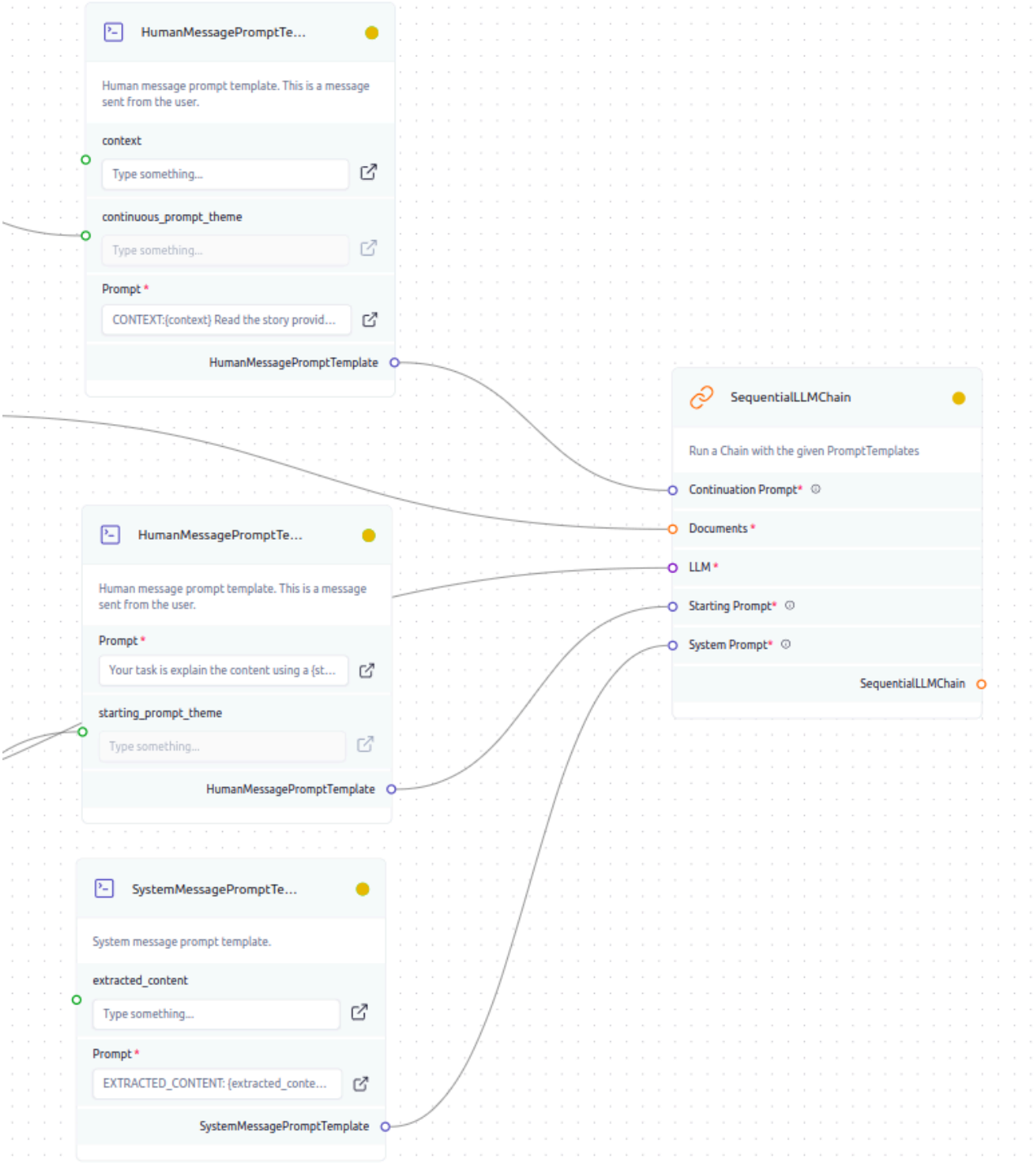
SequentialLLMChain

SequentialLLMChain includes both extracted documents from a source and user-defined prompts. By feeding these inputs to the language model in a specific order, the tool builds upon prior context and guides the model towards generating coherent and relevant text.

Parameters

- **LLM :** A Language model to be used for text generation.
- **Starting Prompt :** This prompt template serves as the initial prompt to start the text generation process. It is used to set the tone, provide context, or ask for specific input to kickstart the generation.
- **Continuation Prompt :** This prompt template used for subsequent prompts in the text generation process. Once the initial prompt is provided, the continuation prompt guides the generation of subsequent text, maintaining coherence and relevance.
- **System Prompt :** This prompt template is used to inject system-level information or instructions into the text generation process. Offers instructions or guidance to the text generation process based on system-level considerations.
- **Documents :** Documents to be used in text generation.

Example usage:



This component is mostly used for Text Generation use cases. The Starting, Continuation prompts take HumanMessagePromptTemplate as input and System prompt takes the SystemMessagePromptTemplate as input. The system message prompt contains the context that is to be passed to the LLM along with a guidance instruction. The starting prompt then is used to start the generation, and the continuation prompt helps refine this content to the required format.

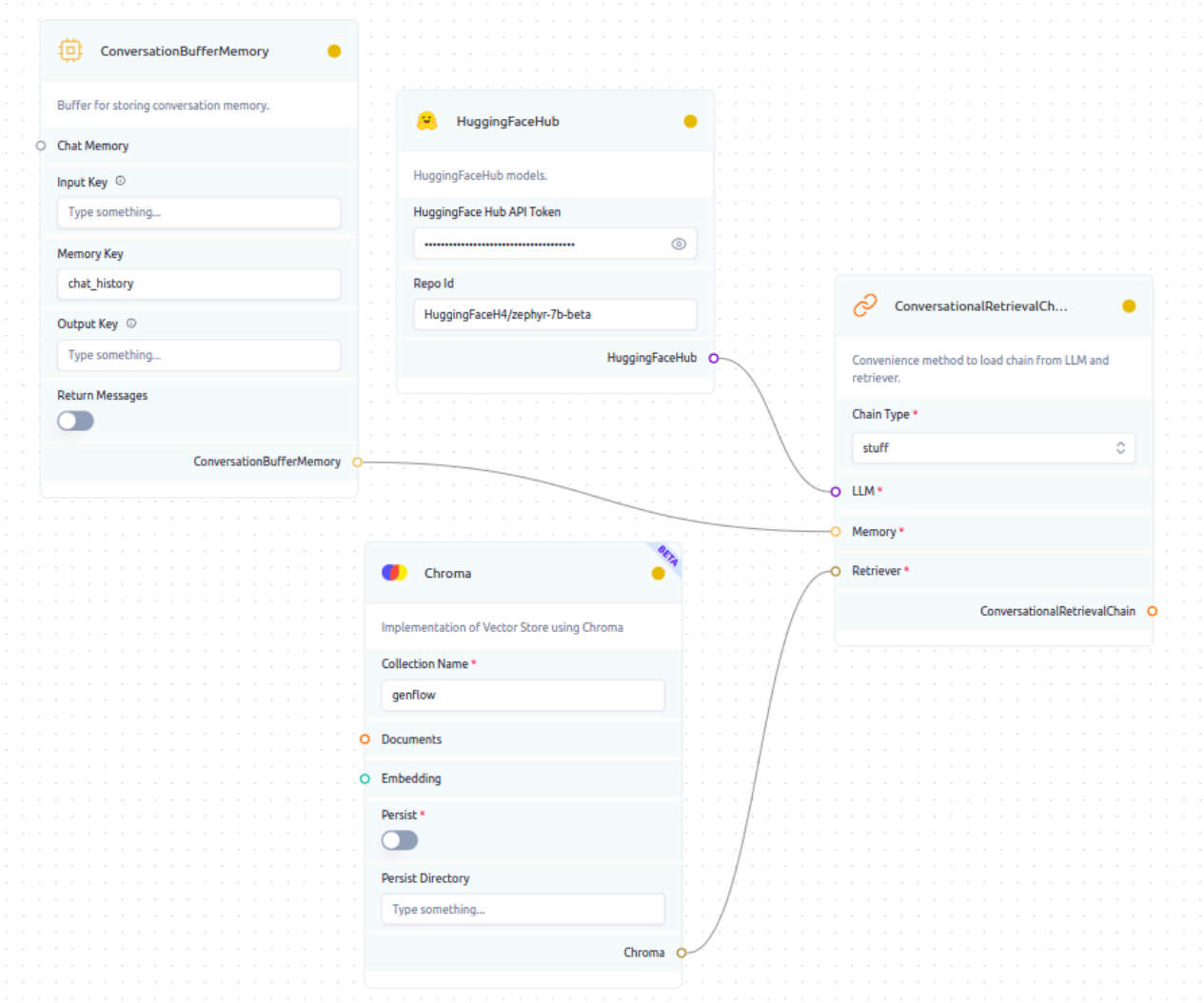
ConversationalRetrievalChain

Chain for having a conversation based on retrieved documents. This chain takes in chat history (a list of messages) and new questions, and then returns an answer to that question.

Parameters

- **chain_type :** The chain type to use to decide the combination strategy used for integrating the LLM and retriever components.. Defaults to "stuff"
- **LLM :** A Language model to be used for text generation.
- **Memory[Optional]:** To store the chat response in history.
- **Retriever:** The retriever used to fetch relevant documents.

Example usage:



This serves as a convenient method to allow an LLM to be given access to a Retriever and Memory to serve as chat history for a application that has access to external Knowledge source.

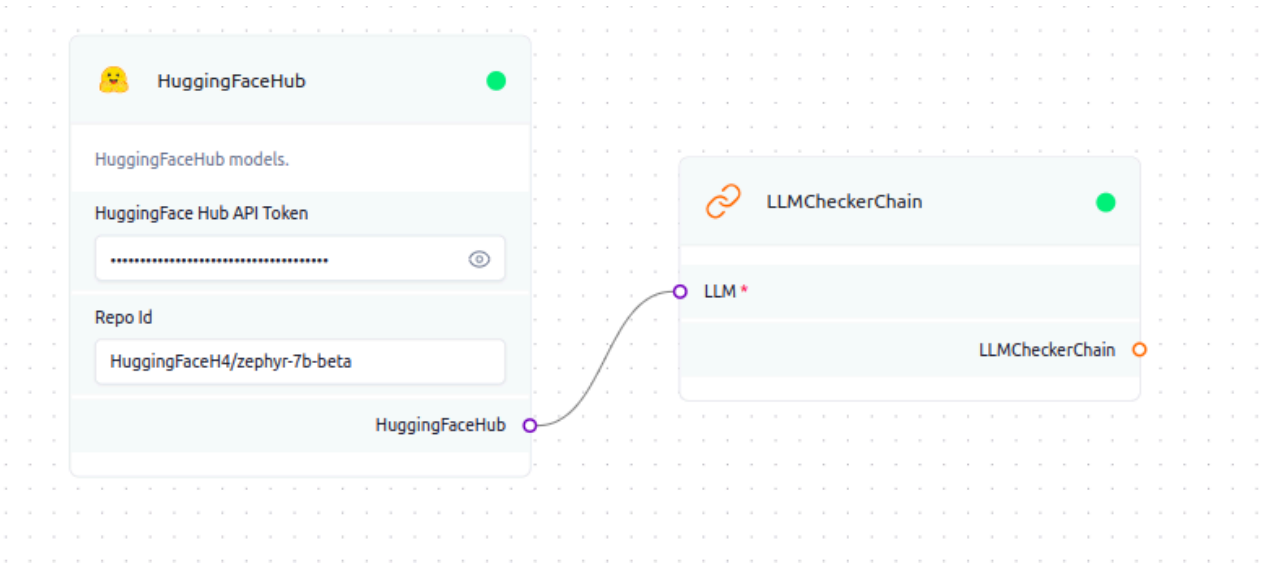
LLMCheckerChain

Chain for question-answering with self-verification.

Parameters

- **LLM :** A Language model to be used for QA and verification.

Example Usage:



This component only takes an LLM as the input, and internally performs a chain of self-verififcation before giving the output.

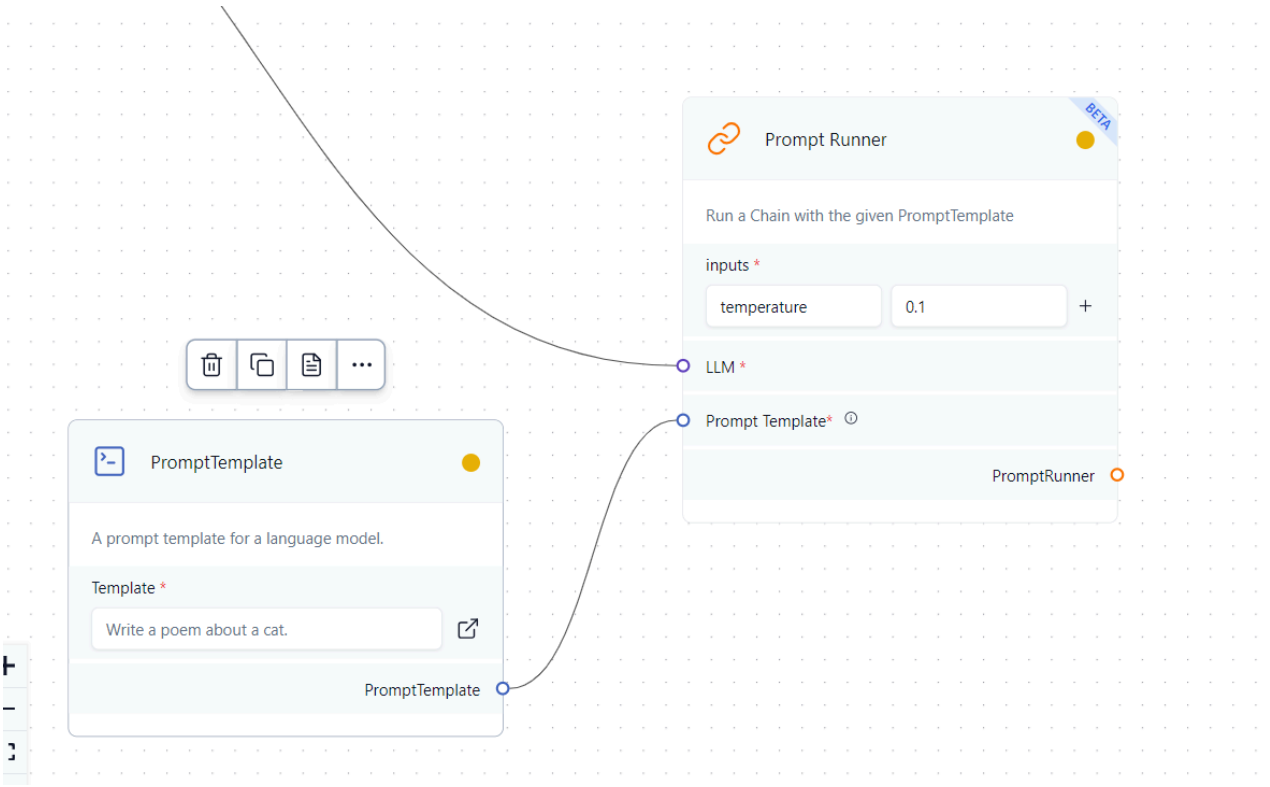
PromptRunner

The Prompt Runner chain is a chain component that executes a prompt with a specified language model (LLM) and returns the generated output. It is responsible for running a prompt through an LLM and managing the input and output processing.

Parameters

- **inputs:** Key-value pairs of variables to be used in the prompt
- **LLM :** A Language model to be used for QA and verification.
- **Prompt Template:** PromptTemplate object containing the prompt to be passed to the LLM.

Example usage:



Prompt Runner chain runs a chain with the given Prompt Template. Additional parameters like temperature can be specified as inputs to the chain.

Output Parsers

Output parsers are used to transform the output of a language model into a more suitable format, particularly when generating structured data.

Response Schema

ResponseSchema is used to define the structure of the response returned by an output parser. You can use ResponseSchema to create custom schemas for your output parser, which can help validate and parse the output more effectively.

Parameters:

- Name: The name of the schema.
- Description: A brief description of the schema, including its purpose and any relevant details. This field helps others understand the schema's intended use.
- Type: The type of the schema. For example, "object", "array", "string", "number", etc. This field indicates the top-level type of the schema.

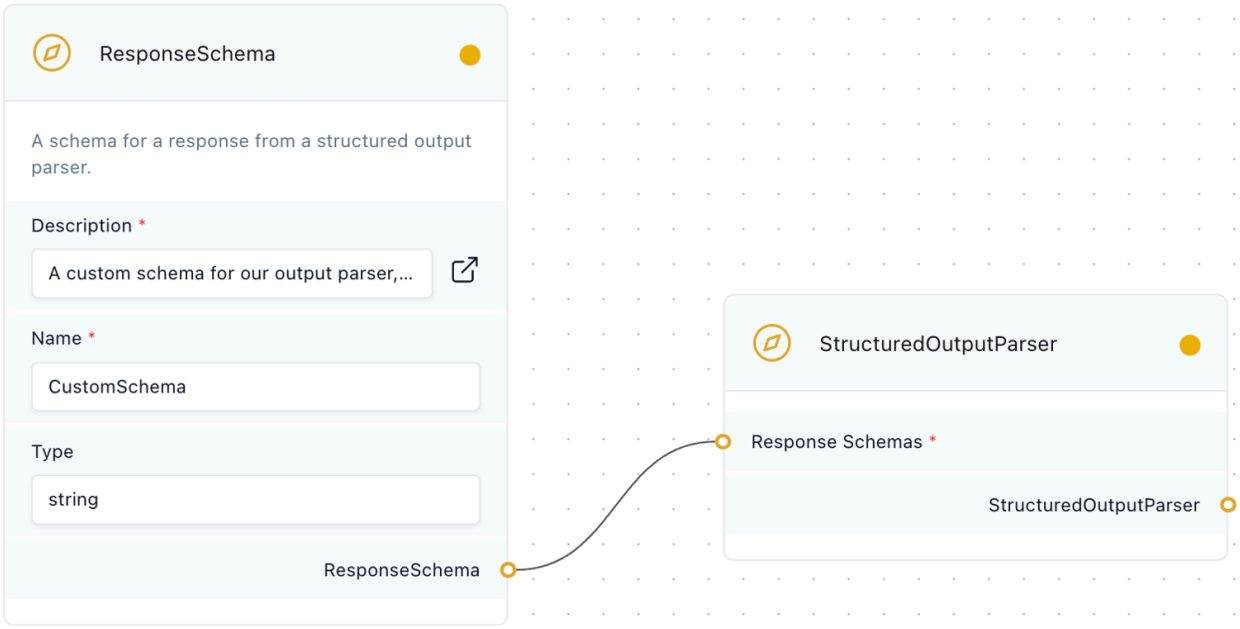
Structured Output Parsers

StructuredOutputParser in Langchain is an output parser used to transform the raw output from a language model (LLM) into a more structured format. The primary responsibility of a structured output parser is to provide methods for getting format instructions and parsing the output from an LLM into a structured format. When using a StructuredOutputParser, you need to specify the response_schemas parameter, which should contain one or more ResponseSchema instances. The response_schemas parameter is used to define the expected structure of the output, and it helps the parser validate and transform the raw output from a language model into a more structured format.

Parameters

- Response Schema

Example usage:



Customization

Writing Custom Components in GenAI Stack

Introduction

This documentation provides guidelines and examples to help you write custom components for the GenAI stack. Custom components allow you to extend the functionality of your application by integrating various external services and custom logic. This guide covers the following topics:

1. How to write a custom component.
2. Understanding the typing system.
3. Available types.
4. Example of a real custom component.

Writing a Custom Component

A custom component in the GenAI stack is a class that inherits from `CustomComponent` and defines its configuration and behavior through specific methods. The key methods are `build_config` and `build`.

Structure of a Custom Component

Here is a basic structure of a custom component:

```
from genflow.interface.custom.custom_component import CustomComponent
from langchain.schema import Document
from typing import List

class ExampleComponent(CustomComponent):
    display_name: str = "ExampleComponent"
    description: str = "This is an example custom component."
    version: str = "1.0"

    def build_config(self):
        return {
            "param1": {
                "display_name": "Parameter 1",
                "required": True,
                "input_types": ["Input"],
            },
            "param2": {
                "display_name": "Parameter 2",
                "required": False,
                "input_types": ["Input"],
            },
            "code": {"show": False},
        }

    def build(self, param1: str, param2: str) -> List[Document]:
        # Custom logic to build and return documents
        documents = [Document(page_content=f"Param1: {param1}, Param2: {p
        return documents
```

Explanation

- `display_name`: The name of the component as it will appear in the UI.
- `description`: A brief description of the component's functionality.
- `version`: The version of the component.
- `build_config`: This method defines the configuration parameters for the component.
- `build`: This method contains the custom logic to process the inputs and produce the output.

Understanding the Typing System

The GenAI stack uses specific types to ensure type safety and clarity in component development. Input and Output types MUST follow this typing system in order for you to connect the components. Any mismatch in types will lead to the components being unconnectable. Commonly used types include:

- **str:** String data type for textual information.
- **Document:** Represents a document object containing content (e.g., text) and optional metadata (e.g., source URL). This serves as the primary data transfer medium between components.
- **VectorStore:** Refers to a storage mechanism for vectors, which are numerical representations of text data used in retrieval tasks.
- **BaseRetriever:** Represents a retriever component that retrieves relevant documents from a vector store based on a query.
- **Embeddings:** Denotes embeddings, a technique for converting text into vectors, facilitating vector-based retrieval and analysis.
- **PromptTemplate:** A pre-defined template for prompting large language models (LLMs) with specific instructions or formats.
- **Chain:** Represents a sequence of connected components that process data in a pipeline fashion.
- **BaseChatMemory:** Represents a base class for components that manage chat state and context within a conversational setting.
- **BaseLLM:** Represents a base class for large language models (LLMs) used for tasks like text generation and translation.
- **BaseLoader:** Represents a base class for components that load data from various sources (e.g., files, databases).
- **BaseMemory:** Represents a base class for components that store and retrieve information relevant to a particular task or session.
- **BaseOutputParser:** Represents a base class for components that process and structure the output generated by other components.
- **TextSplitter:** Represents a component that splits text into smaller chunks for further processing.

Maintaining Type Consistency

When developing custom components, it's essential to:

- **Declare Expected Input Types:** In your `build` method, specify the expected input types for each configuration parameter using appropriate typing annotations.
- **Define Return Types:** Clearly indicate the output type(s) returned by your component's `build` method. This ensures other components can correctly handle the generated data.
- **Adhere to Established Types:** Utilize the built-in types provided by the GenAI Stack to maintain consistency and avoid potential compatibility issues.

Example: GitHub File Loader Component

This component loads a single file from a GitHub repository.

```
typing import List
from genflow.interface.custom.custom_component import CustomComponent
from langchain.schema import Document
import requests
import re
from genflow.utils.util import build_loader_repr_from_documents

class GithubFileLoaderComponent(CustomComponent):
    display_name: str = "GitHub File Loader"
    description: str = "Loads a single file from a GitHub repository."
    documentation: str = "https://docs.aiplanet.com/components/document-l
    version: str = "1.0"
    beta = False

    def build_config(self):
        return {
            "github_file_url": {
                "display_name": "GitHub File URL",
                "info": "The URL of the file in the GitHub repository",
                "required": True,
                "input_types": ["Input"]
            },
            "code": {"show": True},
        }

    def build(self, github_file_url: str) -> List[Document]:
        # Extract repository information from the URL
        match = re.match(r"https://github.com/(.+)/(.+)/blob/(.+)/(.+)",
        if match:
            repo_owner, repo_name, branch, file_path = match.groups()
            raw_url = f"https://raw.githubusercontent.com/{repo_owner}/{r

            # Send a GET request to the raw URL
            response = requests.get(raw_url)

            # Check if the request was successful
            if response.status_code == 200:
                # Get the file content
                file_content = response.text
                docs = [Document(page_content=file_content, metadata={"fi
                self.repr_value = build_loader_repr_from_documents(docs)
                return docs
            else:
                raise Exception(f"Failed to fetch file: {response.status_
        else:
            raise ValueError("Invalid GitHub file URL format")
```

Explanation

- The `GithubFileLoaderComponent` class inherits from `CustomComponent`.
- `build_config` method defines the configuration parameter `github_file_url`.
  - `github_file_url`: The URL of the file in the GitHub repository. It is required and accepts input as a string.
- `build` method:
  - Extracts the repository information (owner, repository name, branch, and file path) from the provided GitHub URL using regular expressions.
  - Constructs the raw URL to fetch the file content directly from GitHub.
  - Sends a GET request to the raw URL and checks if the request is successful.
  - If successful, retrieves the file content and creates a `Document` object with the content and metadata.
  - Returns a list of `Document` objects containing the file content

Build your own custom component

An advantage of utilizing GenAI Stack is the ability to customize the application's components within the stack. Each component offers a range of hyperparameters, and users are required to provide values for these parameters, allowing for flexible configurations.

Import modules for the required components

```
from typing import List
from genflow import CustomComponent
from langchain.document_loaders import YoutubeLoader
from langchain.schema import Document
```

Define parameters used for required components

To integrate a YouTube document loader from LangChain into GenAI Stack, define a class and inherit the custom component. Each component necessitates a display name, description, and documentation URL that should be appropriately reflected on the user interface (UI).

Inside the class, create a `build_config()` method to specify the parameters utilized by the component. In the context of integrating the YouTube loader, user input for the URL and the desired transcription language is essential. For these parameters, construct a Python dictionary and assign the display_name, is_list, required, and value as keys that will be updated on the UI. For instance, consider the URL as an example component.

- **display_name** (str): The component's name, e.g., Video URL
- **is_list** (bool): Indicates whether the required parameter is a list input
- **required** (bool): Specifies whether the input is mandatory
- **value** (str): The default value, which can be left empty or assigned a default value

These parameters will help define and communicate the necessary information for the YouTube loader within the UI. Once all the essential parameters are defined, utilize the `build(**kwargs)` method to instantiate the custom component with the specified configurations.

```
class YoutubeLoaderComponent(CustomComponent):
    display_name: str = "Youtube Loader"
    description: str = "Downloads the YouTube transcripts and video infor
    documentation: str = (
        "https://python.langchain.com/docs/integrations/providers/youtube
    )
    beta = False

    def build_config(self):
        return {
            "youtube_url": {
                "display_name": "Video URL",
                "is_list": False,
                "required": True,
                "value": "",
            },
            "language": {
                "display_name": "Language",
                "is_list": False,
                "required": True,
                "value": "en",
                "info": "language code to extract transcript. please chec
            },
     }

    def build(self, youtube_url: str, language: str) -> List[Document]:
        loader_instance = YoutubeLoader.from_youtube_url(
            youtube_url=youtube_url, add_video_info=False, language=langu
        )
        return loader_instance.load()
```

Usecases

Simple QA using Open Source Large Language Models

In this use, we will build a simple Question and Answering assistant just like ChatGPT but with help of Open Source Language Models.

Step 1: Define your prompt

Every Large Language Model requires a well-crafted prompt template to guide its actions effectively. For the creation of a Q&A chatbot, a Prompt Template is essential to provide clear instructions to the Large Language Model on its tasks. This component is invaluable as it ensures that the model understands the desired task and generates accurate responses based on the provided instructions.

Click on Prompt Template, and edit the template with your prompt:

Know more about Prompts: https://docs.aiplanet.com/components/prompts ↗

Step 2: Define your LLM - Open Source LLM

To enhance the intelligence of the responses, we require a Large Language Model capable of understanding instructions and generating corresponding responses. In this context, we utilize an Open Source Large Language Model due to its accessibility and cost-effectiveness. By leveraging the HuggingFace Hub Large Language Models, users can input their Access token and model name.

Notably, this approach offers the advantage of accessing 7B Large Language Models without the need for manual loading.

Know more about LLM: https://docs.aiplanet.com/components/large-language-models ↗

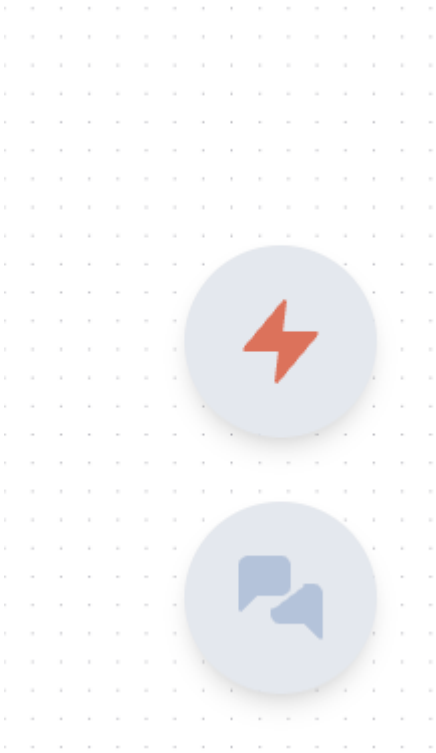
Step 3: Build Chain

To link the prompt with the Large Language Model (LLM), we use a tool called LLMChain. Just connect your Prompt Template and LLM directly to the LLM Chain, and you can optionally include a memory component if needed. The LLM Chain helps the LLM learn from your prompt, making it better at giving sensible responses.

flow

Know more about Chains: https://docs.aiplanet.com/components/chains ↗

Once the chain is complete click on the build icon in the bottom right corner of the page.



icons

Step 4: Testing

Once the build is complete you can click on the chat icon to test the flow.

chat interface

If you're able to get the response then the flow is running successfully & now you can deploy the flow to share it with others.

Check out how to use Chat Interface: https://docs.aiplanet.com/quickstart/chat-interface-genai-stack-chat ↗

Multilingual Indic Language Translation

In this use case, we aim to establish a language translation system from English to six prominent Indian languages: Hindi, Kannada, Telugu, Tamil, Punjabi, and Gujarati.

Let's build the stack

Step 1- Define your prompt

The initial component involves writing a prompt where the user input query and the desired language translation are specified. This component is invaluable as it ensures that the model understands the desired task on what language it need to generate accurate responses.

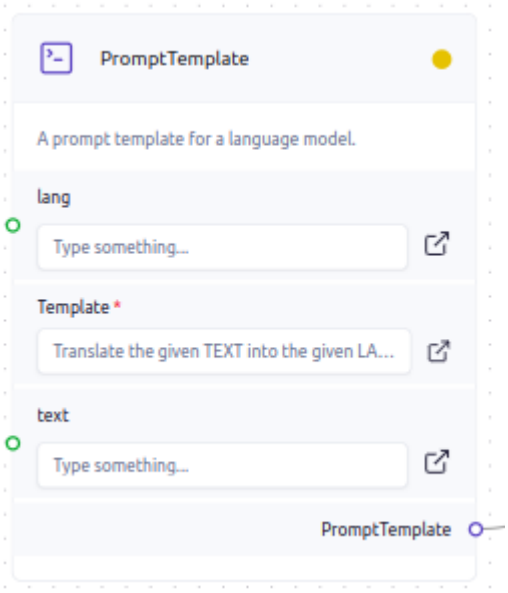
Prompt template, edit the template :

```
Translate the given TEXT into the given LANGUAGE. You are the only Indic

TEXT: {text}
LANGUAGE: {lang}
```

As you look in the stack, we get lang and text as input variables that need to be entered by the user.

Know more about Prompts: https://docs.aiplanet.com/components/prompts ↗

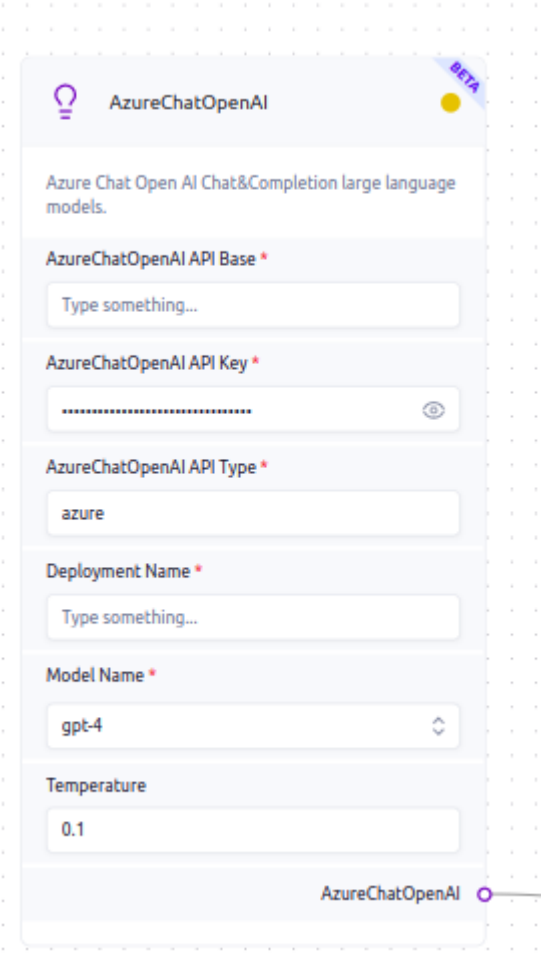


Step 2- Define your LLM

Based on our requirements, we must determine the most suitable Large Language Model for our needs. Given that our use case involves developing a Multilingual Indic bot that must comprehend the language aligned with the prompt instructions, GPT models excel in this aspect. Hence, we have opted for the Azure ChatOpenAI model for this particular use case.

Note: Several Multilingual fine-tuned models are available on the HuggingFace Hub, providing an alternative option for utilizing open source Large Language Models in this use case.

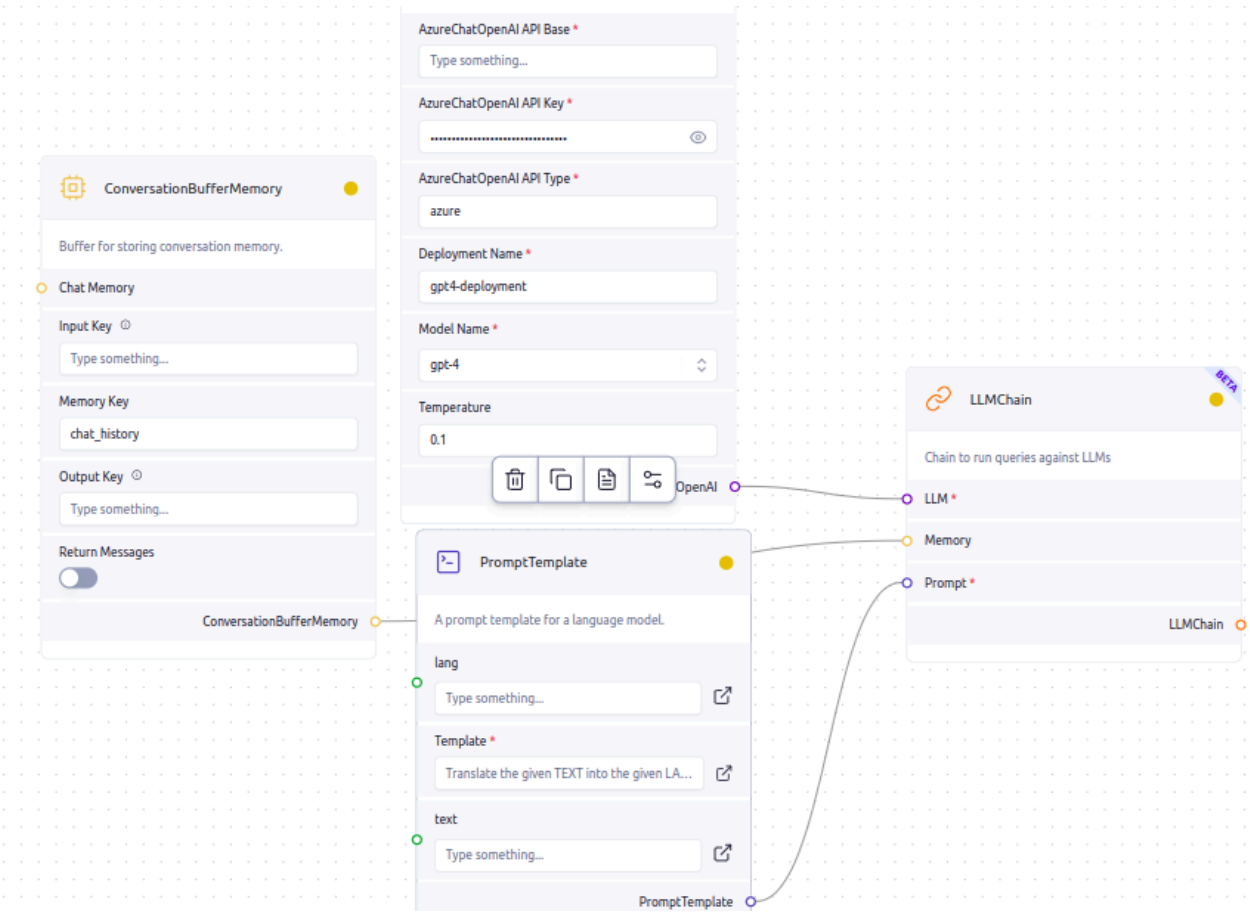
Know more about LLM: https://docs.aiplanet.com/components/large-language-models ↗



Step 3- Build Chain with Memory

In order to connect the prompt with LLM, we need a chain, this is where we use LLMChain. Just connect your Prompt Template, LLM directly to the LLM Chain along with the memory component being optional. This LLM Chain ensures your prompt is added as in-context learning to the LLM model so that it can return reasonable response.

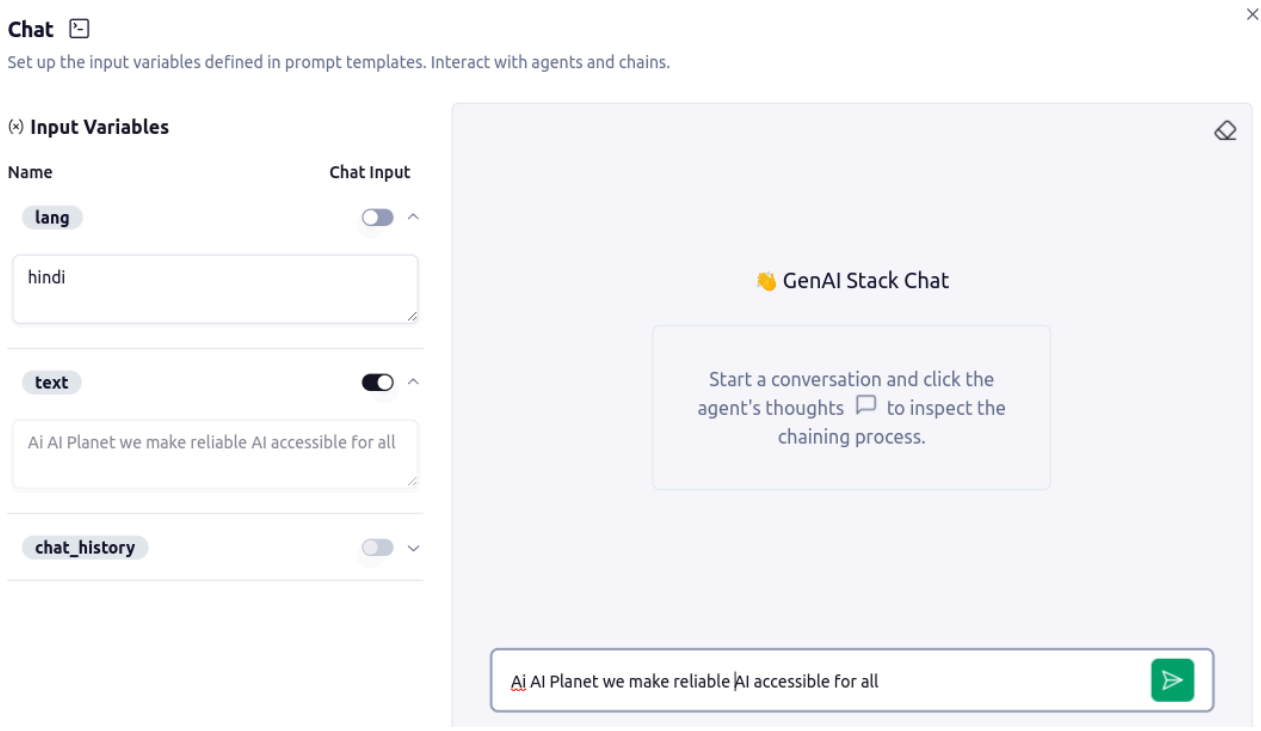
Know more about Chains: https://docs.aiplanet.com/components/chains ↗ and Memory: https://docs.aiplanet.com/components/memories ↗



Step 4- How to use Chat Interface for multiple prompts

Given that our prompt template comprises two input variables, it is crucial to specify the user input correctly. The Chat Interface defaults to selecting the first input variable as the user query, but this may not always be accurate. To address this, enter the language value under the "lang" input variable and utilize the text entry box to input your query.

Check out how to use Chat Interface: https://docs.aiplanet.com/quickstart/chat-interface-genai-stack-chat ↗



NOTICE: In the above image, we write the language name under the lang input variable and then enable text as the chat input. Please make sure that we enter the language name and disable the chat input for lang.

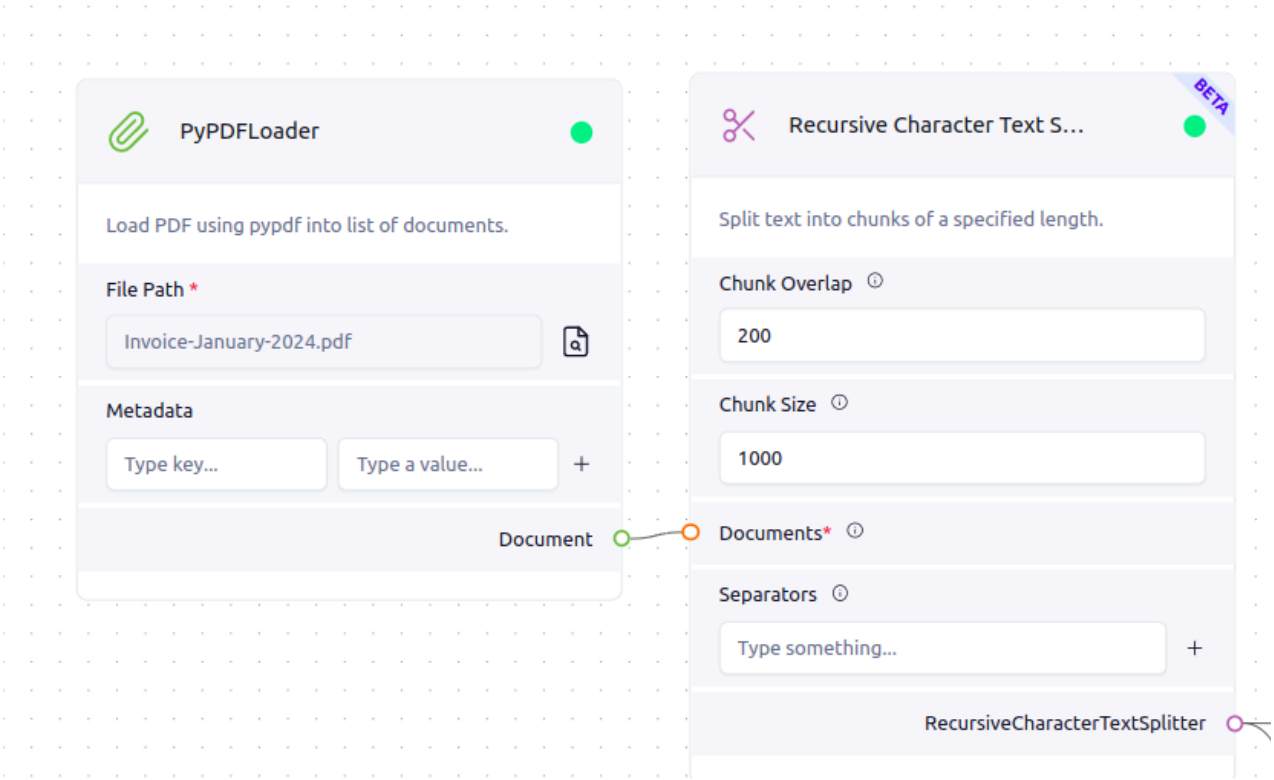
Document Search and Chat

In this use case, we will implement a Chat with PDF. The process involves searching for a pertinent document within the collection based on the user query and subsequently utilizing the Chat Interface

Step 1- Load the Data and Create chunks

The initial step involves selecting the suitable loader depending on your specific use case. This can include options like PDF, Document, URL, YouTube, subtitles, etc., depending on your requirements. After loading the content, we divide the document into smaller segments. This segmentation is necessary because it enables us to provide users with relevant information from the appropriate chunk when they query any prompt.

Check all the loader list: https://docs.aiplanet.com/components/document-loaders ↗ and more information on text splitters: https://docs.aiplanet.com/components/text-splitters ↗

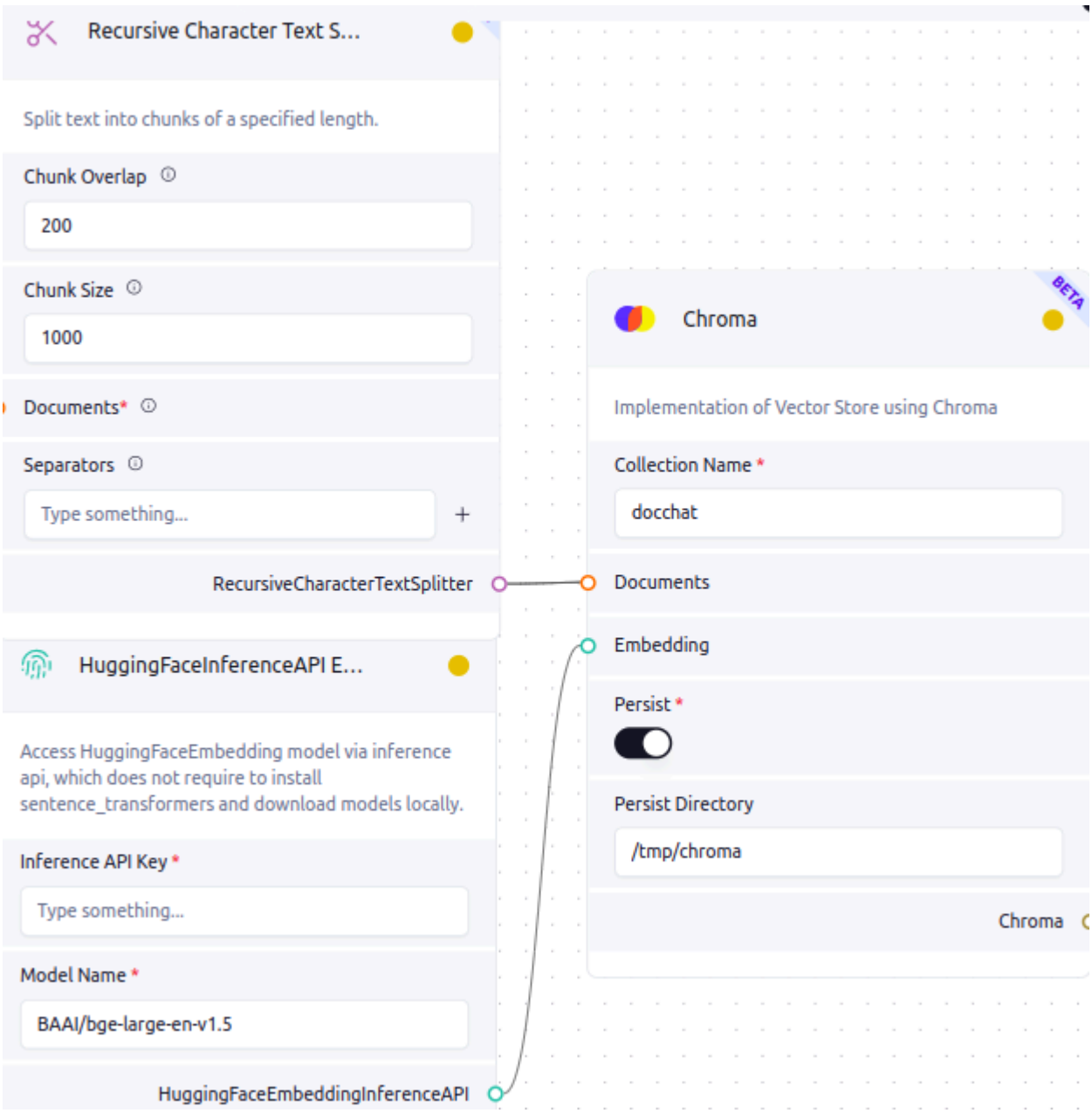


Step 2- Index your document

To index a document, it's essential to generate an embedding for each chunk and store it in a vector database. When utilizing a drag to any vector store, ensure the usage of a unique collection name. Enabling the persist option as true eliminates the necessity to recreate indexes for existing content.

We use vector storage to store the embeddings of the document and retrieve them based on search techniques for the given prompt. The Vector Store accommodates various search types, with similarity search being utilized in this instance.

Check the Vector Store documentation:
https://docs.aiplanet.com/components/vector-store ↗

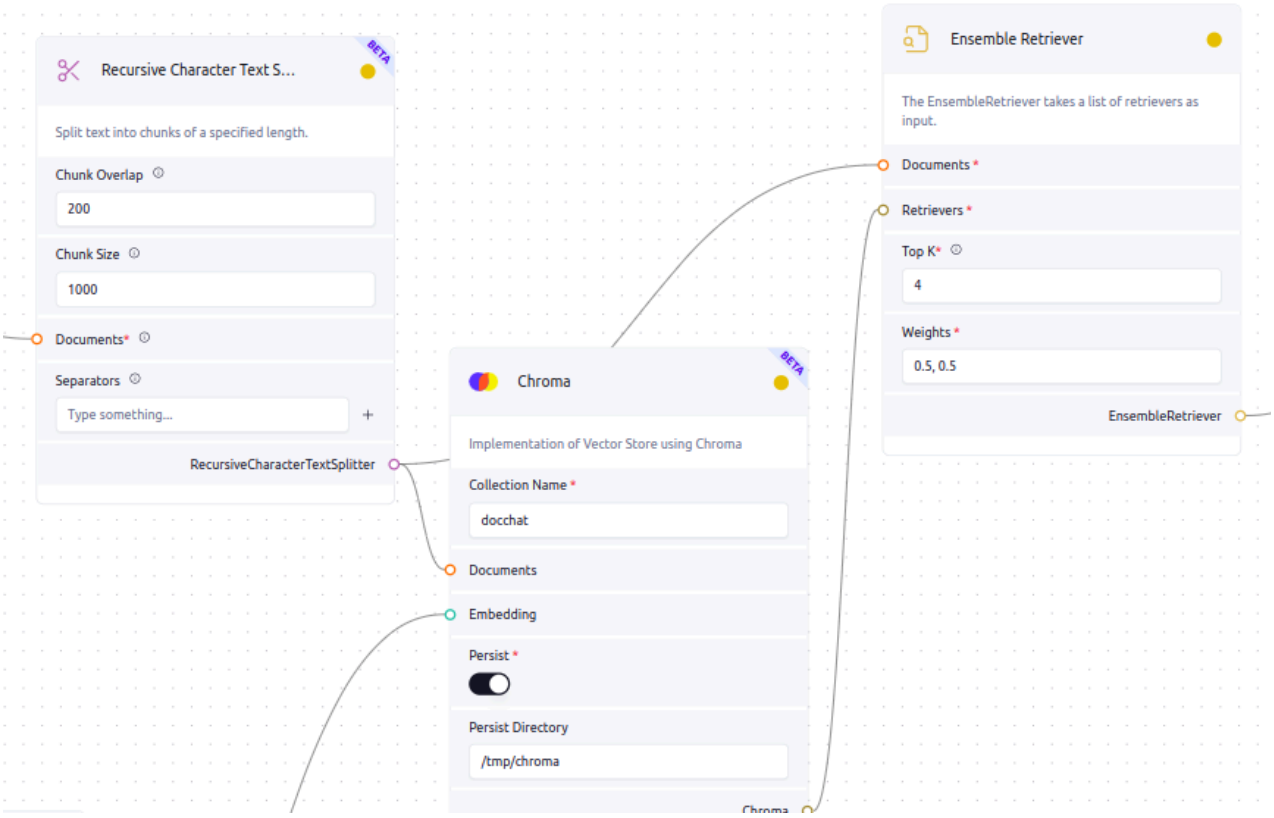


Step 3- Hybrid Search using Ensemble Retriever- Define your Retriever

To implement an advanced RAG concept, we use a Hybrid Search approach incorporating our custom component, the Ensemble Retriever. This retriever seamlessly combines both the Keyword Retriever and the Vector Store Semantic Retriever, representing a critical step. The Ensemble Retriever plays a pivotal role in retrieving context relevant to the user query.

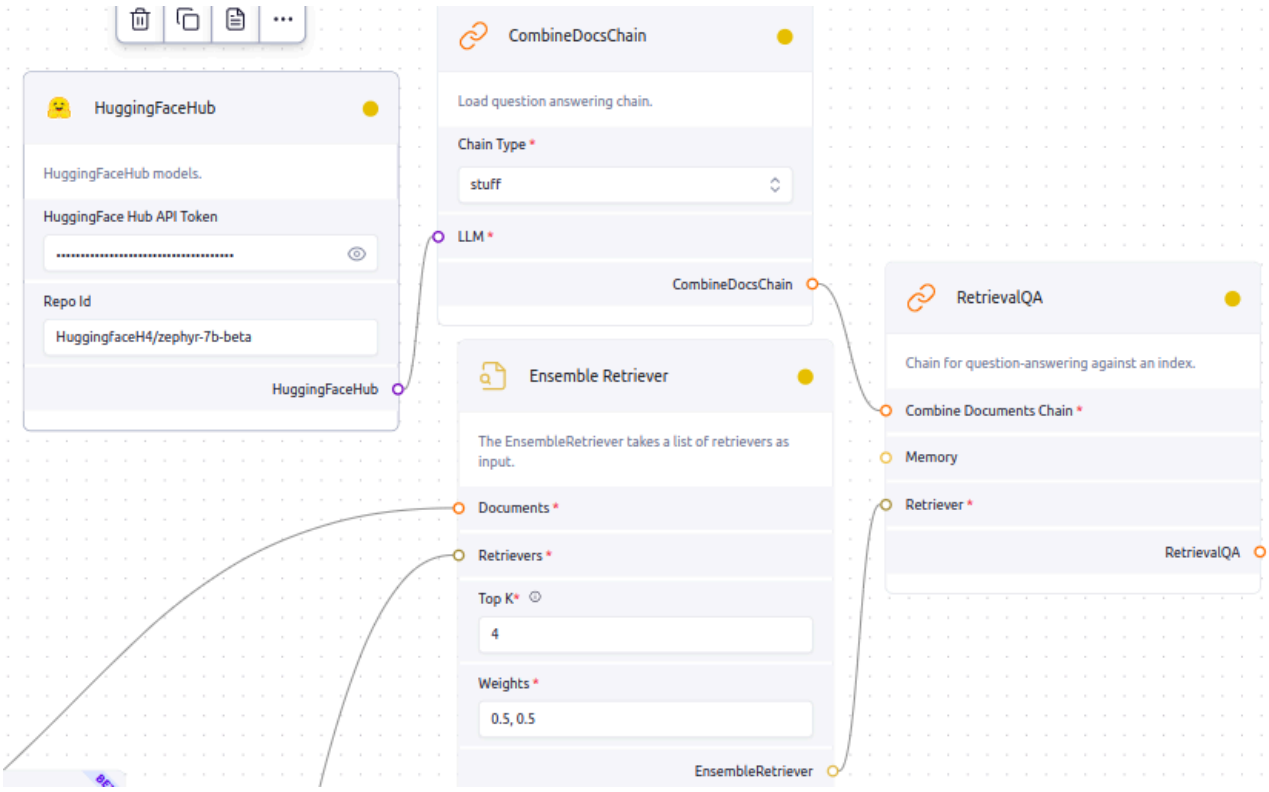
The Ensemble Retriever necessitates the document to conduct internal keyword searches. Additionally, it incorporates a Vector Store Retriever to enable the connection of both components for Hybrid Search functionality.

We have provided detailed documentation outlining the necessity of Hybrid Search: https://docs.aiplanet.com/terminologies/hybrid-search-ensemble-retriever ↗



Step 4- Generator Model - Chain your LLM with Retriever

The Retrieval-Augmented Generation (RAG) pipeline involves two main parts: the Retriever and the Generator. The Retriever finds useful information from the index based on the user's query. Then, this relevant information, along with the prompt, is given to the Large Language Model (LLM), which acts as the generator in the RAG pipeline.



That's it. You can now use Chat Interface to chat with your document.

Chat with Multiple Documents

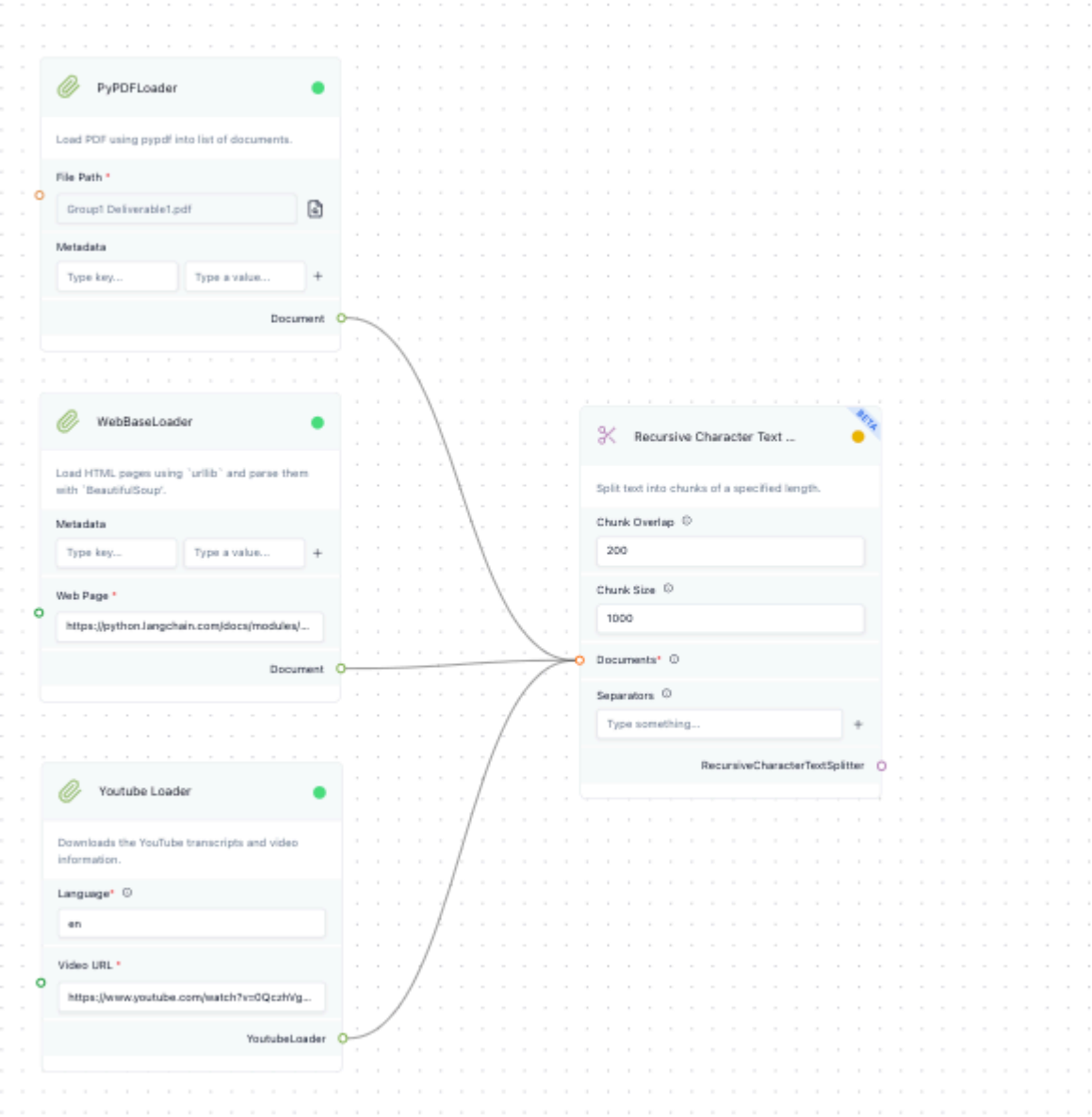
In this use case, we will implement a Chat with multiple documents. We will have different kinds of documents including Youtube video, PDF and Web URL.

Let's build the stack. Create a new project and select Chat stack.

Step-1: Load the Data from different sources and chunk them

The first step is to pick the different loaders needed for your use case. You can also combine PDF, Document, URL, YouTube, subtitles, song lyrics together. Once we load the content, we split the document into smaller chunks. Make sure, to provide meta data when you are working with multiple data loaders.

After loading the content, we divide the document into smaller segments. This segmentation is necessary because it enables us to provide users with relevant information from the appropriate chunk when they query any prompt.



Check all the loader list: https://docs.aiplanet.com/components/document-loaders ↗ and more information on text splitters: https://docs.aiplanet.com/components/text-splitters ↗

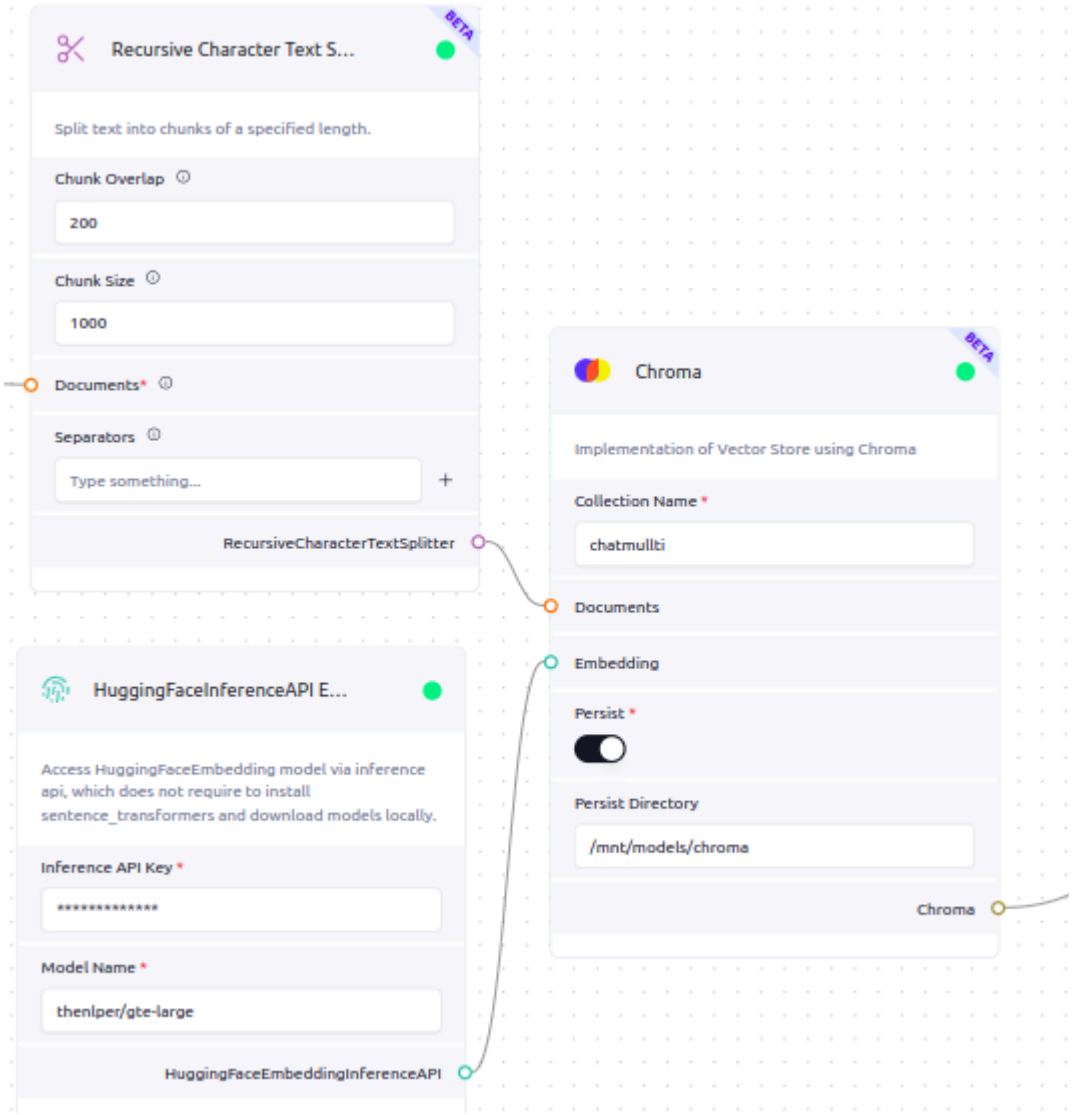
Step 2: Index your document

To index a document, it's crucial to generate an embedding for each section and save it in a vector database. Turning on the persist option avoids the need to recreate indexes for existing content.

We use vector storage to store the embeddings of the document and retrieve them based on search techniques for the given prompt. The Vector Store accommodates

various search types, with similarity search being utilized in this instance.

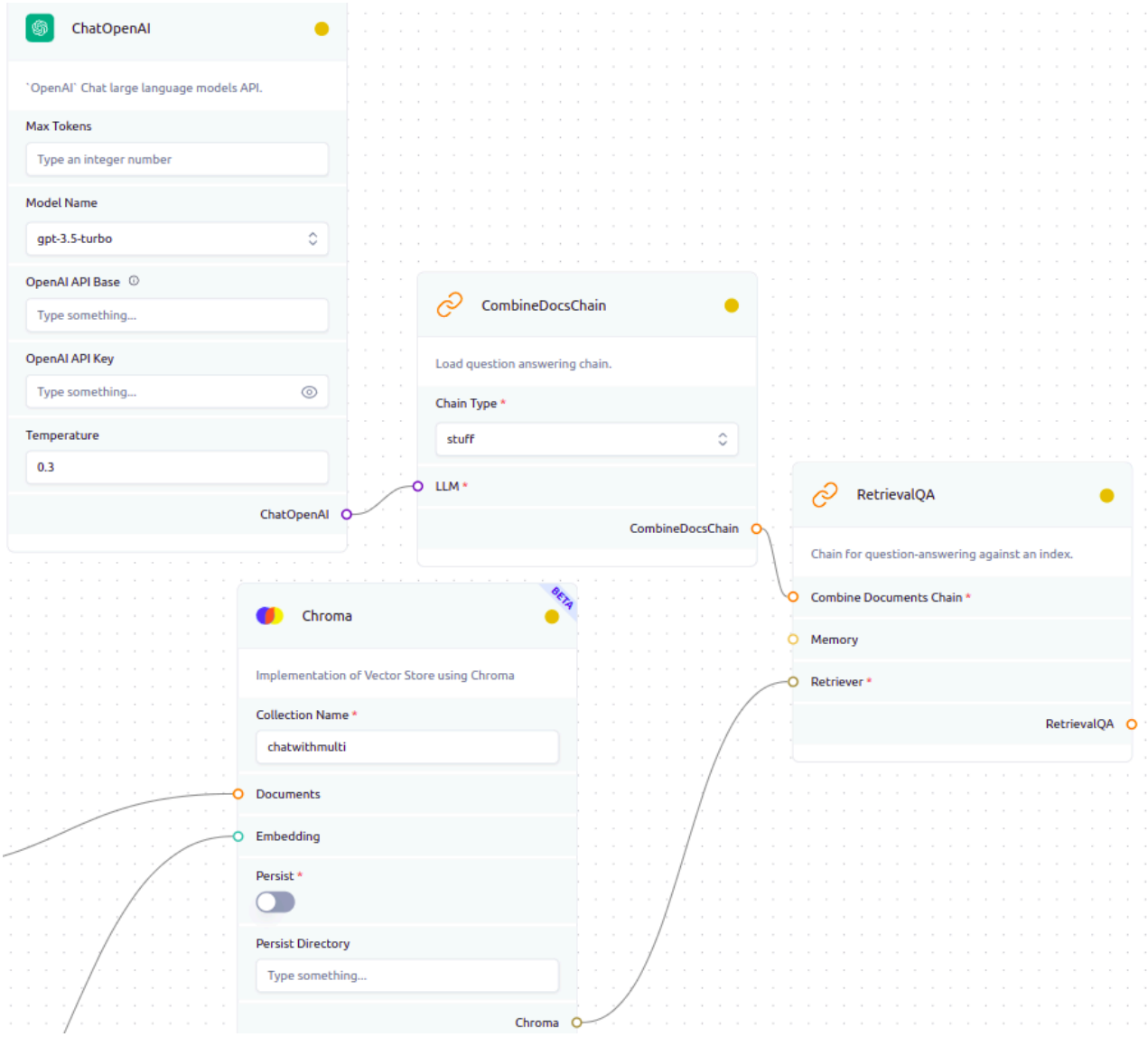
Since we are handling large chunks of data, it is advisable to enable persistence in the directory. This ensures that we do not recreate the index if it has already been created. We can use the directory `/mnt/models/chroma` for persistence.



Step-3: Generator and Retriever

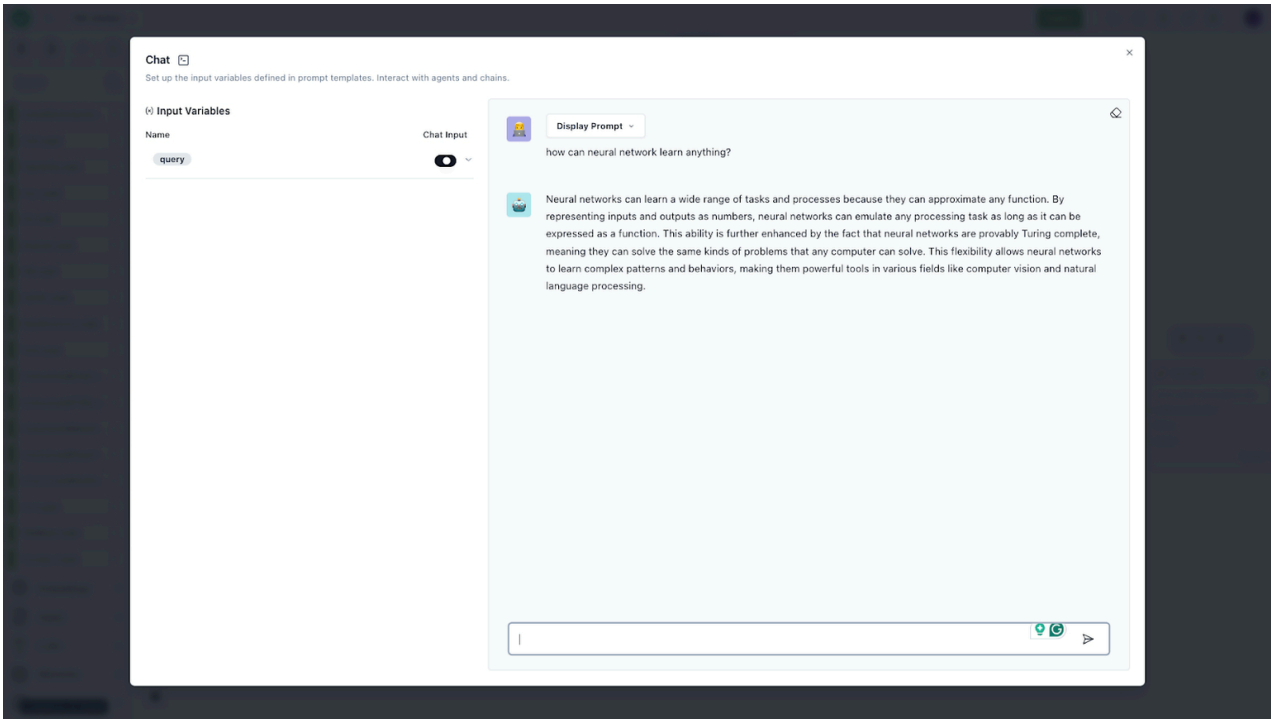
The Retrieval-Augmented Generation (RAG) pipeline involves two main parts: the Retriever and the Generator. The Retriever finds useful information from the index based on the user's query. Then, this relevant information, along with the prompt, is given to the Large Language Model (LLM), which acts as the generator in the RAG pipeline.

Learn what is RAG and how it works: [https://docs.aiplanet.com/terminologies/rag-retrieval-augmented-generation ↗](https://docs.aiplanet.com/terminologies/rag-retrieval-augmented-generation)



Step-4: Chat with your multiple document

Build the stack(⚡), Upon a successful build, navigate to the chat icon to commence interaction.


Chat
Set up the input variables defined in prompt templates. Interact with agents and chains.
Input Variables
Name
Chat Input
query
Display Prompt
how can neural network learn anything?
Neural networks can learn a wide range of tasks and processes because they can approximate any function. By representing inputs and outputs as numbers, neural networks can emulate any processing task as long as it can be expressed as a function. This ability is further enhanced by the fact that neural networks are provably Turing complete, meaning they can solve the same kinds of problems that any computer can solve. This flexibility allows neural networks to learn complex patterns and behaviors, making them powerful tools in various fields like computer vision and natural language processing.

Terminologies

RAG - Retrieval Augmented Generation

What is RAG?

Retrieval Augmented Generation (RAG) is a natural language processing (NLP) technique that combines two fundamental tasks in NLP: information retrieval and text generation. It aims to enhance the generation process by incorporating information from external sources through retrieval. The goal of RAG is to produce more accurate and contextually relevant responses in text generation tasks.

In traditional text generation models like GPT-3, the model generates text based on patterns learned from a large corpus of data, but it may not always have access to specific, up-to-date, or contextually relevant information. Retrieval Augmented Generation addresses this limitation by introducing an information retrieval component.

How RAG works?

`Retrieval`: The model performs a retrieval step to gather relevant information from external sources. These sources could include a database, a knowledge base, a set of documents, or even search engine results. The retrieval process aims to find snippets or passages of text that contain information related to the given input or prompt.

`Augmentation`: The retrieved information is then combined with the original input or prompt, enriching the context available to the model for generating the output. By incorporating external knowledge, the model can produce more informed and accurate responses.

`Generation`: Finally, the model generates the response, taking into account the retrieved information and the original input. The presence of this additional context helps the model produce more contextually appropriate and relevant outputs.

RAG can be beneficial in various NLP tasks, such as `Question-Answering`, `Dialogue generation`, `Summarization`, and more. By incorporating external knowledge, RAG models have the potential to provide more accurate and informative responses compared to traditional generation models that rely solely on the data they were trained on.

Checkout full article on our Medium Page: https://medium.aiplanet.com/retrieval-augmented-generation-using-qdrant-huggingface-embeddings-and-langchain-and-evaluate-the-3c7e3b1e4976 ↗

Hybrid Search - Ensemble Retriever

Hybrid Vector Search

Hybrid Search is basically a combination of keyword style search and a vector style search. It has the advantage of doing keyword search as well as the advantage of doing a semantic lookup that we get from embeddings and a vector search.

Keyword Search :

In GenAI Stack we internally use `BM25 Algorithm`. It generates a sparse vector. BM25 (Best Match 25) is an information retrieval algorithm used to rank and score the relevance of documents to a particular search query. It's an extension of the `TF-IDF (Term Frequency-Inverse Document Frequency)` approach.

Key points about BM25:

1. **Term Frequency (TF):** Measures the frequency of a term in a document.
2. **Inverse Document Frequency (IDF):** Measures the importance of a term based on its frequency across the entire document collection.
3. **BM25 Weighting:** Combines TF and IDF to calculate the relevance score of a document for a given query.
4. **Query Terms:** BM25 considers the occurrence of query terms in the document and adjusts their scores accordingly.
5. **Parameter Tuning:** BM25 involves tuning parameters (k1, b) to optimize the ranking performance based on the dataset.

Semantic Search:

Semantic search is a search method that aims to improve the accuracy and relevance of search results by understanding the context and meaning behind a search query. Unlike traditional keyword-based search, which primarily relies on

matching keywords, semantic search tries to comprehend the intent and context of the user's query and the content of the documents being searched.

Semantic search strives to mimic human understanding of language and context, ultimately delivering search results that align better with the user's information needs.

In GenAI Stack, we use Vector Store retriever for Semantic Search.

Ensemble Retriever

The `EnsembleRetriever` takes a list of retrievers as input and ensemble the results of their `get_relevant_documents()` methods and rerank the results based on the Reciprocal Rank Fusion algorithm.

By leveraging the strengths of different algorithms, the `EnsembleRetriever` can achieve better performance than any single algorithm.

The most common pattern is to combine a sparse retriever (like BM25) with a dense retriever (like embedding similarity), because their strengths are complementary. It is also known as **"Hybrid search".**

REST APIs

GenAI Stack REST APIs

How to Use Our REST APIs

GenAI Stack APIs ↗ power developers to integrate the stacks that they built & deployed in GenAI Stack into their platform.

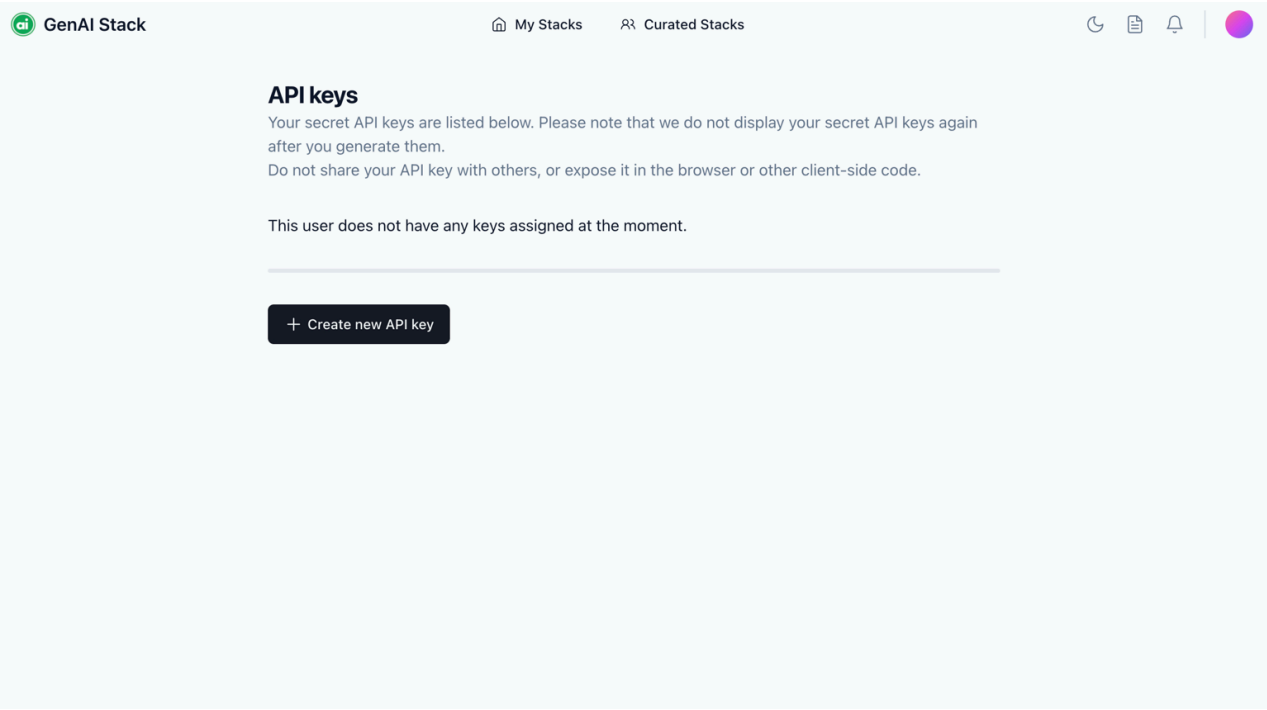
Working with GenAI Stack APIs

GenAI Stack APIs support bearer authentication. Where you have to send the api key in the **Authorization** header when you're making api requests.

```
Authorization: Bearer <api-key>
```

How to get API keys?

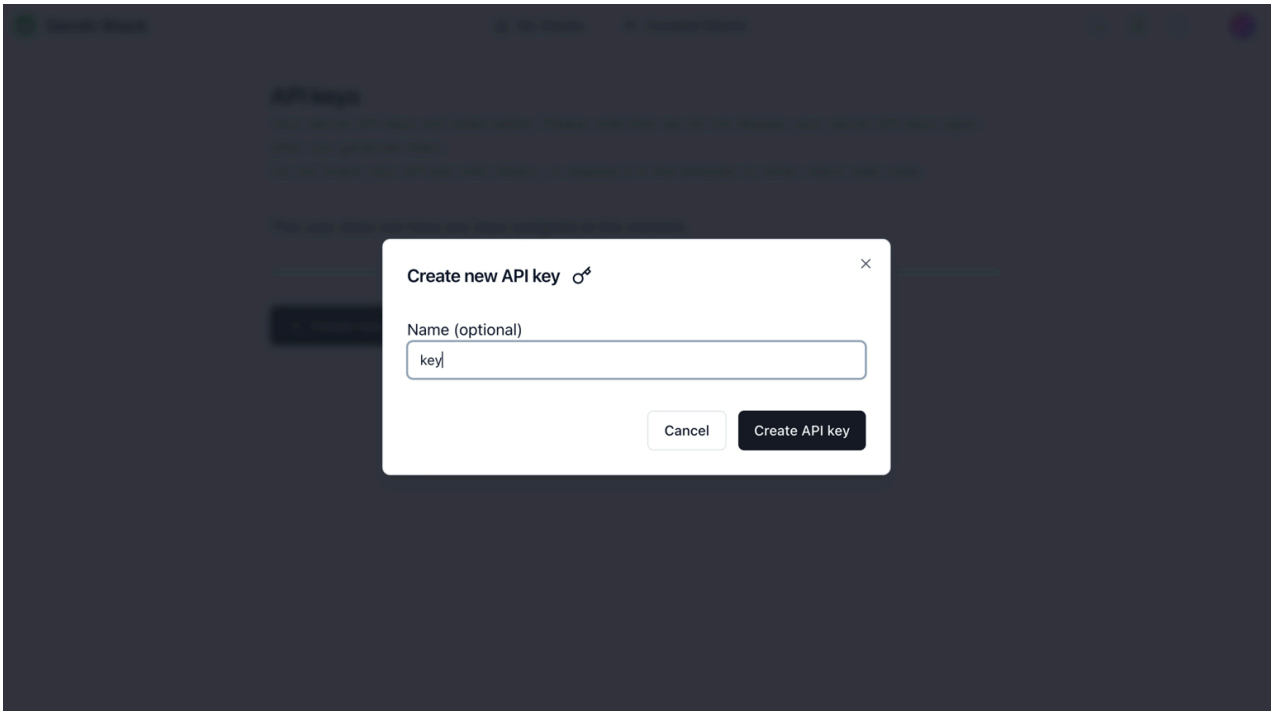
Step 1: Go to the API Keys page (https://app.aiplanet.com/account/api-keys ↗)



API Keys

Step 2: Click on the `Create new API key` & add the name.

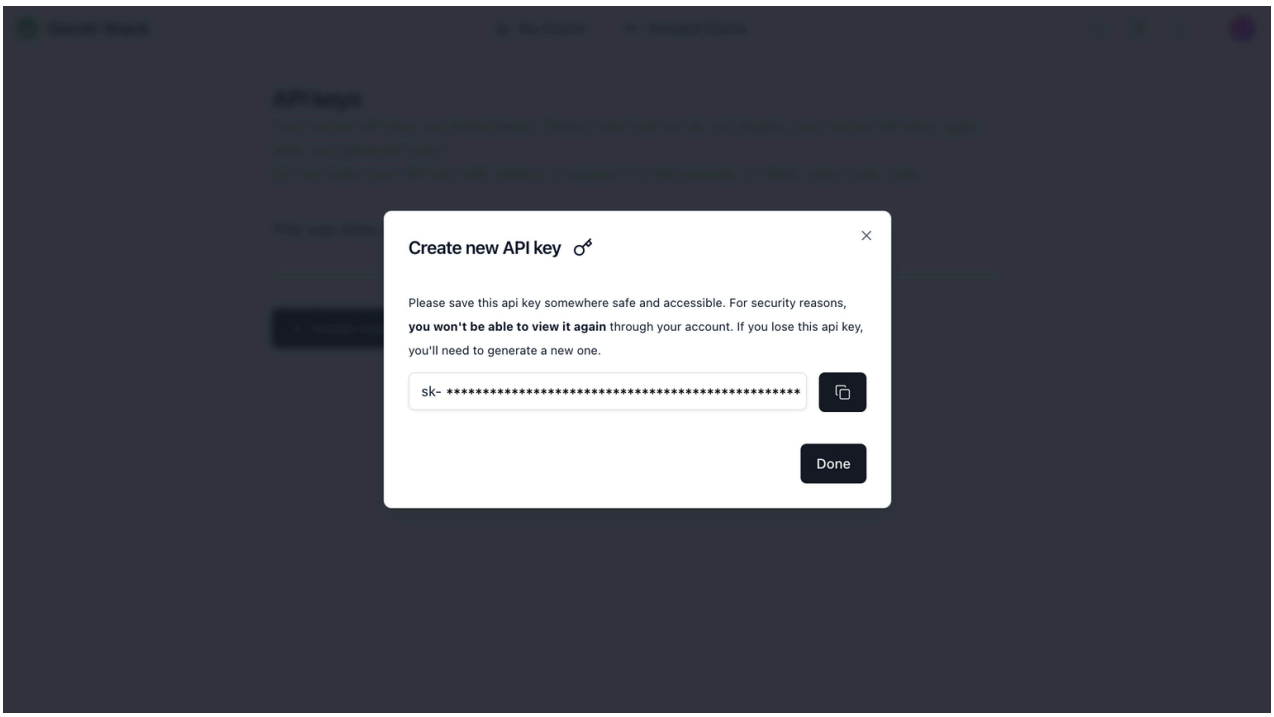
Click on the `Create API key` to generate the key.



API Key Modal

Step 3: Save the API Key.

Once the key is generated please save the api key somewhere. Once the api key modal is closed you will not be able to view the api key for security reasons.



API Key generated

Chat API Reference

Following are the steps to use `Test it` feature.

- Replace the {{url}} with the deployed url.
- Get the api key ↗ from the GenAI Stack for authentication.
- Now you're ready to test the APIs.

Health

`GET` `/health`

Responses

> 200 Successful Response

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/health'
```

Test it ▶

200

No Content

Successful Response

Get Stack App

`GET` `/api/v1/apps`

Retrieve the stack app instance.

This endpoint fetches the stack app, including its associated chat sessions or text generations based on the interface type selected, using the provided token/api key.

Parameters:

- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns a detailed view of the stack app including chat sessions and text generations.
- `404 Not Found`: If no stack app is found for the given criteria.

Exceptions:

- Raises a `404 HTTPException` if no stack app is found.

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/apps'
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "data": {},
  "form_keys_data": {},
  "stack_type": "chat",
  "chat_sessions": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "title": "text",
      "data": [
        {
          "is_bot": true,
          "message": "text",
          "chatKey": "text",
          "type": "text",
          "intermediate_steps": "text",
          "files": []
        }
      ]
    }
  ],
  "text_generations": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "inputs": {},
      "outputs": {},
      "logs": "text",
      "meta_data": {},
      "status": "pending"
    }
  ]
}
```


Successful Response

Create Chat Session

POST `/api/v1/{stack_app_id}/chat-sessions`

Create a new chat session associated with a specific stack app.

Parameters:

- `stack_app_id`: UUID - Path parameter to specify the stack app ID the chat session will be associated with.
- `title`: Optional[str] - A custom title for the chat session. If not provided, a default title "New Chat" is used.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns the created chat session's details.
- `401 Unauthorized`: If user authentication fails.
- `429 Too Many Requests`: If the request is throttled based on rate limiting.

Exceptions:

- Raises `401 HTTPException` for unauthorized access.
- Raises `429 HTTPException` for exceeding rate limit, with retry information.
- Raises `404 HTTPException` if stack app doesn't exist

Path parameters

`stack_app_id` `string · uuid` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Body

`title` any of optional

Show child attributes

Show child attributes

Responses

200 Successful Response

422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions' \
  --header 'Content-Type: application/json' \
  --data '{
    "title": "text"
  }'
```

Test it ▶

200 422

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "data": [
    {
      "is_bot": true,
      "message": "text",
      "chatKey": "text",
      "type": "text",
      "intermediate_steps": "text",
      "files": []
    }
  ]
}
```

Successful Response

Get Chat Session

`GET` `/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}`

Retrieve details of a specific chat session by its ID and associated stack app ID.

Parameters:

- `stack_app_id`: UUID - Path parameter to specify the stack app ID.
- `chat_session_id`: str - Path parameter to specify the chat session ID.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Successfully retrieves and returns the details of the chat session.
- `400 Bad Request`: If the `chat_session_id` is not a valid UUID.
- `401 Unauthorized`: If user authentication fails.
- `404 Not Found`: If no chat session is found matching the criteria.

Exceptions:

- Raises `400 HTTPException` if `chat_session_id` is invalid.
- Raises `401 HTTPException` for unauthorized access.
- Raises `404 HTTPException` if the chat session is not found.
- Raises `404 HTTPException` if stack app doesn't exist

Path parameters

`stack_app_id` `string · uuid` required

`chat_session_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` string optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}'
```

Test it ▶

200 422

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "data": [
    {
      "is_bot": true,
      "message": "text",
      "chatKey": "text",
      "type": "text",
      "intermediate_steps": "text",
      "files": []
    }
  ],
  "data_pipelines": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "input_key": "text",
      "meta_data": {},
      "created_at": "2025-04-05T15:27:15.266Z",
      "files": [
        {
          "id": "123e4567-e89b-12d3-a456-426614174000",
          "file": "text",
          "url": "text",
          "text": "text",
          "created_at": "2025-04-05T15:27:15.266Z",
          "data_pipeline_status": "queued",
          "meta_data": {}
        }
      ]
    }
  ],
  "can_chat": true
}
```


Successful Response

Chat

POST `/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}`

Initiates or continues a chat session by sending a payload to the specified chat session.

This endpoint allows users to interact with a chat session by sending a payload. The payload will be processed according to the chat session's logic, and a response will be streamed back. This endpoint requires user authentication and is subject to rate limiting to ensure fair usage.

Parameters:

- `stack_app_id`: str - The unique identifier of the stack app associated with the chat session. It is a path parameter.
- `chat_session_id`: str - The unique identifier of the chat session to which the message is being sent. It is a path parameter.
- `payload`: dict - A JSON object containing the chatKey & inputs. The values of chatKey & inputs will be dynamic check the form_keys_data object in stack app data in which chatKey is key given in input_keys & inputs is object containing all the required inputs. When there are multiple input_keys chatKey can be the key which you want to make it primary.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Successful Response:

- Returns a streaming response, which streams the response from the chat session back to the client. The response is in the `text/event-stream` format.

Errors:

- HTTP 401 Unauthorized: Returned if the user cannot be authenticated or authorized, indicating either missing, invalid, or expired credentials.
- HTTP 429 Too Many Requests: Returned if the request is throttled based on rate limiting. The response includes a `Retry-After` header indicating how long the client should wait before retrying.
- HTTP 500 Internal Server Error: Returned if an unexpected error occurs during the processing of the chat message.
- Raises `404 HTTPException` if stack app doesn't exist

Notes:

- The endpoint is designed for real-time chat interactions and uses Server-Sent Events (SSE) to stream responses back to the client.
- Clients must be prepared to handle streaming responses and display them to the user in real-time.
- The rate limiting mechanism is in place to ensure that all users have a smooth experience.

Path parameters

`stack_app_id` `string · uuid` required

`chat_session_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Body

`chatKey` `string` required

`inputs` `object` required

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}' \
  --header 'Content-Type: application/json' \
  --data '{
    "chatKey": "text",
    "inputs": {}
  }'
```

Test it ▶

200 422

```
{
  "is_bot": true,
  "message": "text",
  "chatKey": "text",
  "type": "text",
  "intermediate_steps": "text",
  "files": []
}
```

Successful Response

Update Chat Session

`PATCH` `/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}`

Update details of an existing chat session, such as its title.

Parameters:

- `stack_app_id`: UUID - Path parameter for the associated stack app.
- `chat_session_id`: UUID - Path parameter for the chat session to update.
- `update_request`: ChatSessionUpdate - The request body containing the update details.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns the updated chat session details.
- `401 Unauthorized`: If user authentication fails.
- `404 Not Found`: If no chat session is found matching the criteria.

Exceptions:

- Raises `401 HTTPException` for unauthorized access.
- Raises `404 HTTPException` if the chat session not found
- Raises `404 HTTPException` if stack app doesn't exist

Path parameters

`stack_app_id` `string · uuid` required

`chat_session_id` `string · uuid` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Body

`title` `any of` optional

Show child attributes

Show child attributes

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request PATCH \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}' \
  --header 'Content-Type: application/json' \
  --data '{
    "title": "text"
  }'
```

Test it ▶

200 422

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "data": [
    {
      "is_bot": true,
      "message": "text",
      "chatKey": "text",
      "type": "text",
      "intermediate_steps": "text",
      "files": []
    }
  ],
  "data_pipelines": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "input_key": "text",
      "meta_data": {},
      "created_at": "2025-04-05T15:27:15.266Z",
      "files": [
        {
          "id": "123e4567-e89b-12d3-a456-426614174000",
          "file": "text",
          "url": "text",
          "text": "text",
          "created_at": "2025-04-05T15:27:15.266Z",
          "data_pipeline_status": "queued",
          "meta_data": {}
        }
      ]
    }
  ],
  "can_chat": true
}
```


Successful Response

Data Pipeline List

GET `/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/uploads`

This endpoint lists all files, URLs, or text snippets associated with a chat session, allowing for a comprehensive view of the resources related to a conversation.

Parameters:

- `stack_app_id`: str - The unique identifier of the stack app associated with the chat session.
- `chat_session_id`: str - The unique identifier of the chat session for which to list uploads.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Successfully returns a list of uploads for the chat session.
- `401 Unauthorized`: User authentication failed.
- Raises `404 HTTPException` if stack app doesn't exist

This endpoint provides an overview of all resources uploaded or linked in the context of a specific chat session.

Path parameters

`stack_app_id` `string · uuid` required

`chat_session_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/up
```

Test it ▶

200 422


```
[
  {
    "id": "123e4567-e89b-12d3-a456-426614174000",
    "file": "text",
    "url": "text",
    "text": "text",
    "chat_session_id": "123e4567-e89b-12d3-a456-426614174000",
    "created_at": "2025-04-05T15:27:15.266Z",
    "meta_data": {},
    "data_pipeline_status": "queued",
    "data_pipeline": {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "input_key": "text",
      "meta_data": {},
      "created_at": "2025-04-05T15:27:15.266Z"
    }
  }
]
```


Successful Response

Data Pipeline

POST `/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/uploads`

Upload a file, or provide a URL or text to be associated with a chat session. This endpoint supports file uploads for further processing and inclusion in the chat session's context or data pipelines.

Parameters:

- `stack_app_id`: str - The unique identifier for the stack app associated with this chat session.
- `chat_session_id`: str - The unique identifier for the chat session where the file or text is to be uploaded.
- `input_key`: str - A key identifying the type of input or the specific pipeline this upload is associated with.
- `url`: str (optional) - A URL pointing to a resource to be associated with the chat session.
- `file`: UploadFile (optional) - The file to be uploaded. Files should be less than 50MB in size.
- `text`: str (optional) - Text content to be associated with the chat session.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns the details of the uploaded file or the provided URL/text including any metadata.
- `401 Unauthorized`: User authentication failed.
- `400 Bad Request`: Data pipeline for the provided input_key not found or the file size exceeds the limit.
- `429 Too Many Requests`: The request is throttled based on rate limiting.
- Raises `404 HTTPException` if stack app doesn't exist

Usage:

- To upload a file, include it in the request's form-data.
- To associate a URL or text with the chat session, provide them in the form-data without a file.

This endpoint is subject rate limiting to ensure smooth experience for all the users.

Path parameters

`stack_app_id` `string · uuid` required

`chat_session_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Body

`input_key` `string` required

`url` `any of` optional

Show child attributes

Show child attributes

`file` `any of` optional

Show child attributes

Show child attributes

`text` `any of` optional

Show child attributes

Show child attributes

Responses

`200` Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/up
  --header 'Content-Type: multipart/form-data' \
  --form 'input_key=text' \
  --form 'url=text' \
  --form 'file=binary' \
  --form 'text=text'
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "file": "text",
  "url": "text",
  "text": "text",
  "chat_session_id": "123e4567-e89b-12d3-a456-426614174000",
  "created_at": "2025-04-05T15:27:15.266Z",
  "meta_data": {},
  "data_pipeline_status": "queued",
  "data_pipeline": {
    "id": "123e4567-e89b-12d3-a456-426614174000",
    "input_key": "text",
    "meta_data": {},
    "created_at": "2025-04-05T15:27:15.266Z"
  }
}
```


Successful Response

Get Data Pipeline

GET /api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/uploads/{uploa

Retrieve the status and details of a specific file upload, URL, or text associated with a chat session. This allows for detailed tracking and management of resources within a chat context.

Parameters:

- `stack_app_id`: str - The unique identifier of the stack app associated with the chat session.
- `upload_id`: str - The unique identifier of the upload whose details are being retrieved.
- `chat_session_id`: str - The unique identifier of the chat session associated with the upload.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Successfully returns the details of the specified upload.
- `401 Unauthorized`: User authentication failed.
- Raises `404 HTTPException` if stack app doesn't exist

Path parameters

`stack_app_id` `string · uuid` required

`upload_id` `string` required

`chat_session_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/{stack_app_id}/chat-sessions/{chat_session_id}/up
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "file": "text",
  "url": "text",
  "text": "text",
  "chat_session_id": "123e4567-e89b-12d3-a456-426614174000",
  "created_at": "2025-04-05T15:27:15.266Z",
  "meta_data": {},
  "data_pipeline_status": "queued",
  "data_pipeline": {
    "id": "123e4567-e89b-12d3-a456-426614174000",
    "input_key": "text",
    "meta_data": {},
    "created_at": "2025-04-05T15:27:15.266Z"
  }
}
```


Successful Response

Text Generation API Reference

Following are the steps to use `Test it` feature.

- Replace the {{url}} with the deployed url.
- Get the api key ↗ from the GenAI Stack for authentication.
- Now you're ready to test the APIs.

Health

GET `{{url}}/health`

Responses

> 200 Successful Response

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/health'
```

Test it ▶

200

No Content

Successful Response

Get Stack App

GET `{{url}}/api/v1/apps`

Retrieve the stack app instance.

This endpoint fetches the stack app, including its associated chat sessions or text generations based on the interface type selected, using the provided token/api key.

Parameters:

- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns a detailed view of the stack app including chat sessions and text generations.
- `404 Not Found`: If no stack app is found for the given criteria.

Exceptions:

- Raises a `404 HTTPException` if no stack app is found.

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/apps'
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "data": {},
  "form_keys_data": {},
  "stack_type": "chat",
  "chat_sessions": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "title": "text",
      "data": [
        {
          "is_bot": true,
          "message": "text",
          "chatKey": "text",
          "type": "text",
          "intermediate_steps": "text",
          "files": []
        }
      ]
    }
  ],
  "text_generations": [
    {
      "id": "123e4567-e89b-12d3-a456-426614174000",
      "title": "text",
      "inputs": {},
      "outputs": {},
      "logs": "text",
      "meta_data": {},
      "status": "pending"
    }
  ]
}
```


Successful Response

List Text Generations

GET {{url}}/api/v1/generations

Retrieves a list of text generations created by the authenticated user, ordered by creation date in descending order.

Parameters:

- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns a list of `TextGenerationRead` objects, each representing a text generation instance.
- `401 Unauthorized`: User authentication failed.

Exceptions:

- Raises `401 HTTPException` for unauthorized access.

This endpoint allows users to view all their text generation activities within the system, providing a comprehensive history of generated texts.

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/generations'
```

Test it ▶

200 422


```
[
  {
    "id": "123e4567-e89b-12d3-a456-426614174000",
    "title": "text",
    "inputs": {},
    "outputs": {},
    "logs": "text",
    "meta_data": {},
    "status": "pending"
  }
]
```


Successful Response

Create Text Generation

POST `{{url}}/api/v1/generations`

Creates a new text generation request based on the input provided and associates it with the authenticated user.

Parameters:

- `request`: Request - The request object, allowing access to the request body.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Request Body: The body should contain form data representing the inputs for text generation. form data should be a list of dict(s).

- `input_key`: str - A key for identifying a input or the specific pipeline this upload is associated with.
- `input_value`: str | UploadFile - A resource to be associated with the input_key for Text Generation session.

Responses:

- `200 OK`: Successfully creates a new text generation instance and returns its details as a `TextGenerationRead` object.
- `401 Unauthorized`: User authentication failed.

Exceptions:

- Raises `401 HTTPException` for unauthorized access.

Upon successful creation, this endpoint asynchronously triggers the text generation process and returns the created instance's details without waiting for the generation to complete.

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request POST \
  --url '{{url}}/api/v1/generations'
```

Test it ▶

200 422

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "inputs": {},
  "outputs": {},
  "logs": "text",
  "meta_data": {},
  "status": "pending"
}
```


Successful Response

Read Text Generation

`GET` `{{url}}/api/v1/generations/{text_gen_id}`

Retrieves the details of a specific text generation instance by its ID, for the authenticated user.

Parameters:

- `text_gen_id`: str - The unique identifier of the text generation instance to retrieve.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns the details of the specified text generation instance as a `TextGenerationRead` object.
- `400 Bad Request`: If the `text_gen_id` is not a valid UUID.
- `401 Unauthorized`: User authentication failed.
- `404 Not Found`: If no text generation instance is found matching the `text_gen_id` for the authenticated user.

This endpoint allows users to access detailed information about a specific text generation activity, including its inputs and any generated outputs.

Path parameters

`text_gen_id` string required

Query parameters

`token` string optional

Header parameters

`authorization` string optional

Responses

> 200 Successful Response

> 422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --url '{{url}}/api/v1/generations/{text_gen_id}'
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "inputs": {},
  "outputs": {},
  "logs": "text",
  "meta_data": {},
  "status": "pending"
}
```


Successful Response

Update Text Generation

`PATCH` `{{url}}/api/v1/generations/{text_gen_id}`

Updates the title of a text generation session.

Parameters:

- TextGenerationUpdate - The request body containing the title to be updated.
- `text_gen_id`: str - The unique identifier of the text generation instance to retrieve.
- `authorization`: str - Provide your API key as a Bearer token in the Authorization header to authenticate requests.

Responses:

- `200 OK`: Returns the details of the specified text generation instance as a `TextGenerationRead` object after updating it.
- `400 Bad Request`: If the `text_gen_id` is not a valid UUID.
- `401 Unauthorized`: User authentication failed.
- `404 Not Found`: If no text generation instance is found matching the `text_gen_id` for the authenticated user.

This endpoint allows users to update the title of text generation session.

Path parameters

`text_gen_id` `string` required

Query parameters

`token` `string` optional

Header parameters

`authorization` `string` optional

Body

`title` any of optional

Show child attributes

Show child attributes

Responses

200 Successful Response

422 Validation Error

cURL JavaScript Python HTTP

```
curl -L \
  --request PATCH \
  --url '{{url}}/api/v1/generations/{text_gen_id}' \
  --header 'Content-Type: application/json' \
  --data '{
    "title": "text"
  }'
```

Test it ▶

200 422


```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "title": "text",
  "inputs": {},
  "outputs": {},
  "logs": "text",
  "meta_data": {},
  "status": "pending"
}
```


Successful Response

Rate Limiting and Sleep Mode

Rate limiting

By default stack `create chat session` & `chat` API has the following rate limit.

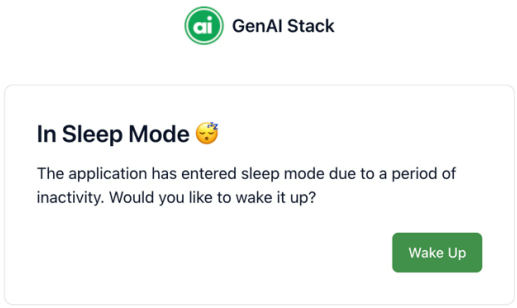
- An application can handle 40 requests per minute
- A user can make 4 requests per minute

These limits are added to make sure the application works smoothly for all users.

Sleep Mode

To use the resources efficiently GenAI Stack apps automatically go into sleep mode if it's not used for the last 60 mins.

If the app is in sleep mode you will see the following UI:



sleep mode

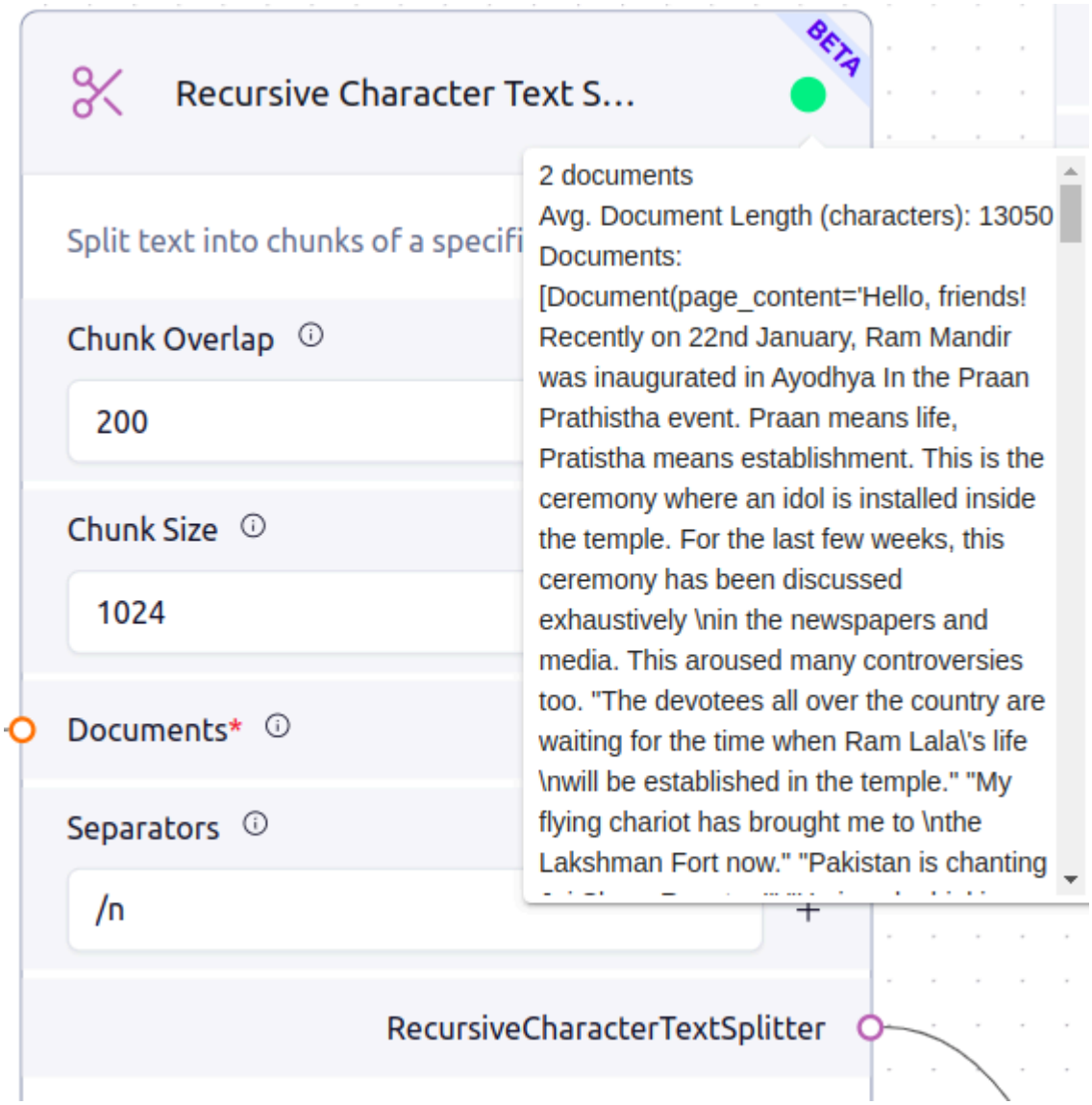
Click on the W**ake Up** button to activate the app.

Note: If you want to use the GenAI Stack API contact us at `tech@aiplanet.com` to increase the rate limit & to remove the sleep mode.

Troubleshooting

How to verify what is loaded and chunked from the loader?

Once you build the stack you can hover the green circular icon. As attached in the screenshot below.



Acknowledgements

Special Mentions

This work would not have been possible without the incredible support from various open source and other open integrations. Our special thanks to the following open-source tools for their inspiration.

- Langchain
- Langflow
- ChromaDB
- Hugging Face

We open-sourced our GenAI Stack ↗ dev tool several months ago and are working towards releasing a more stable open-source version of our studio and agents, stay tuned for updates. Our heartfelt gratitude goes to all the team members at AI Planet for putting this together.