



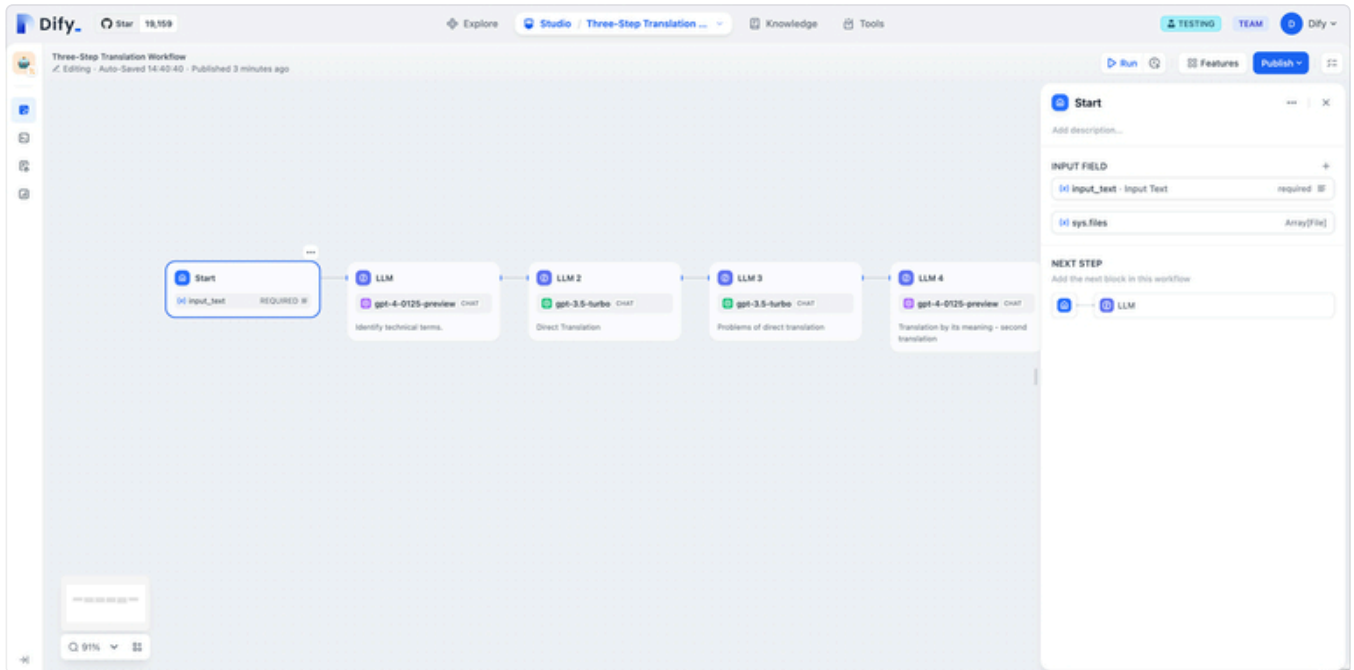
# Nodes

<a href="#">Start</a> >	<a href="#">End</a> >
<a href="#">Answer</a> >	<a href="#">LLM</a> >
<a href="#">Knowledge Retrieval</a> >	<a href="#">Question Classifier</a> >
<a href="#">IF/ELSE</a> >	<a href="#">Code</a> >
<a href="#">Template</a> >	<a href="#">Variable Assigner</a> >
<a href="#">HTTP Request</a> >	<a href="#">Tools</a> >
<a href="#">Previous Key Concept</a>	
<a href="#">Next Start</a>	



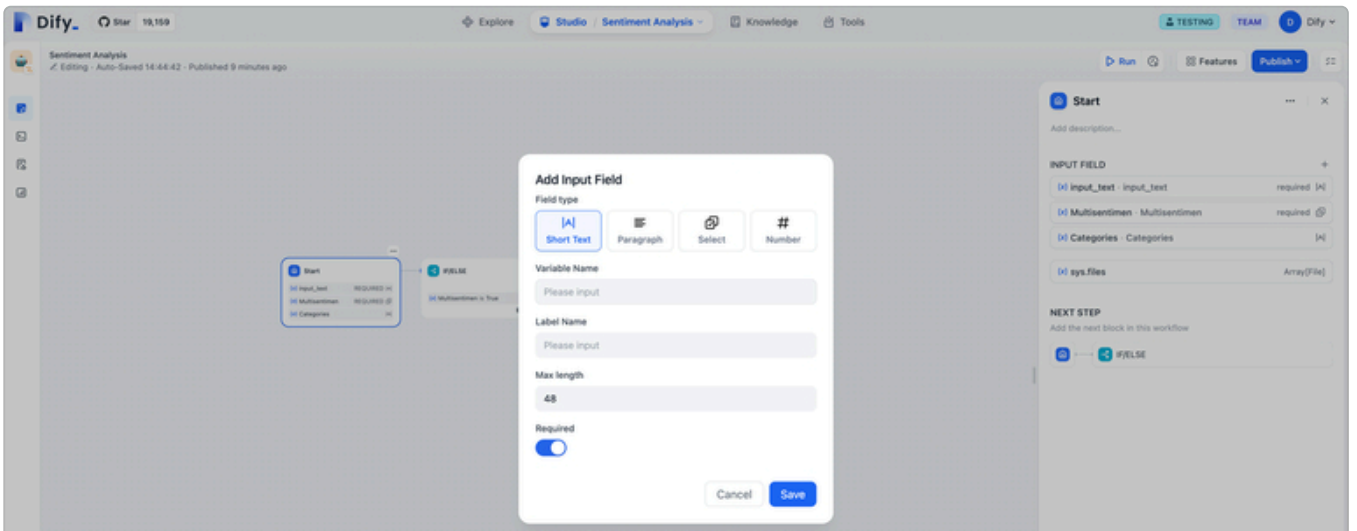
# Start

Defining initial parameters for a workflow process initiation allows for customization at the start node, where you input variables to kick-start the workflow. Every workflow necessitates a start node, acting as the entry point and foundation for the workflow's execution path.

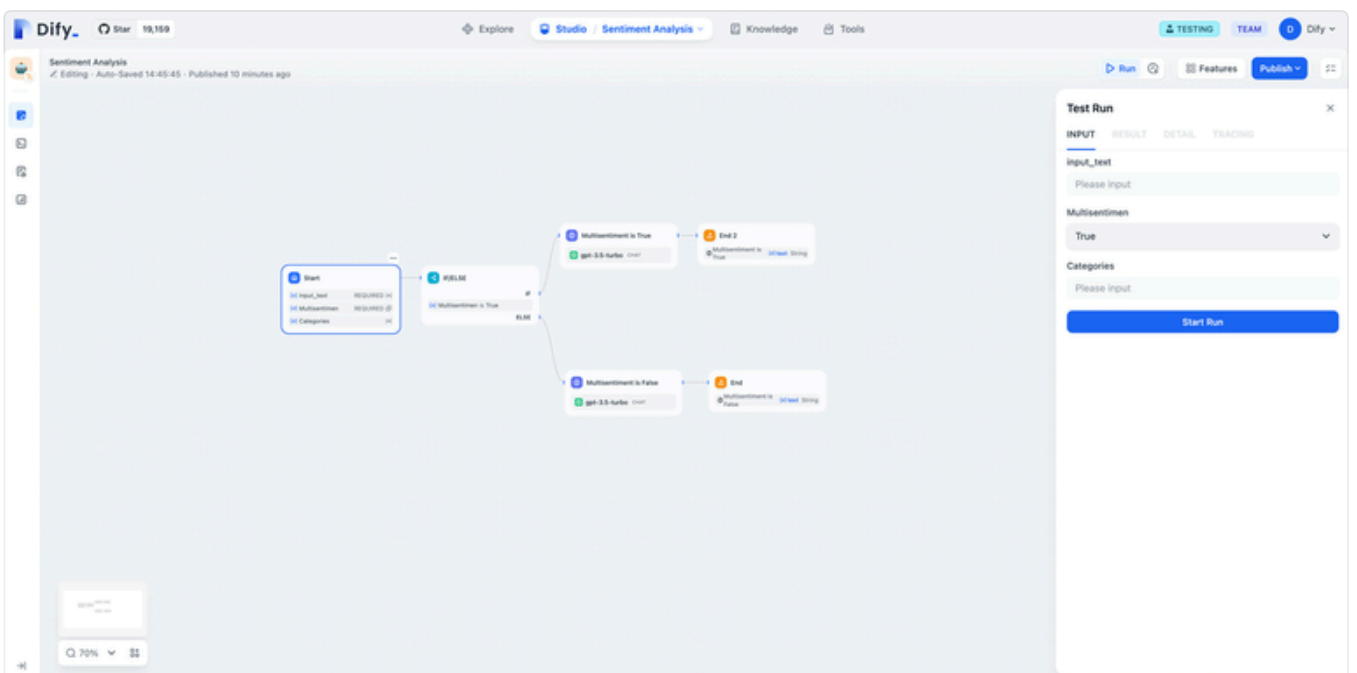


Within the "Start" node, you can define input variables of four types:

- **Text:** For short, simple text inputs like names, identifiers, or any other concise data.
- **Paragraph:** Supports longer text entries, suitable for descriptions, detailed queries, or any extensive textual data.
- **Dropdown Options:** Allows the selection from a predefined list of options, enabling users to choose from a set of predetermined values.
- **Number:** For numeric inputs, whether integers or decimals, to be used in calculations, quantities, identifiers, etc.



Once the configuration is completed, the workflow's execution will prompt for the values of the variables defined in the start node. This step ensures that the workflow has all the necessary information to proceed with its designated processes.



**Tip:** In Chatflow, the start node provides system-built variables: `sys.query` and `sys.files`. `sys.query` is utilized for user question input in conversational applications, enabling the system to process and respond to user queries. `sys.files` is used for file uploads within the conversation, such as uploading an image to understand its content. This requires the integration of image understanding models or tools designed for processing image inputs, allowing the workflow to interpret and act upon the uploaded files effectively.

Next  
End

Last updated 2 months ago



# End

Defining the Final Output Content of a Workflow Process. Every workflow needs at least one "End" node to output the final result after full execution.

The "End" node serves as the termination point of the process, beyond which no further nodes can be added. In workflow applications, execution results are only output when the process reaches the "End" node. If the process involves conditional branching, multiple "End" nodes must be defined.

Single-Path Execution Example:

The screenshot displays the Dify AI interface for a workflow titled "Three-Step Translation Workflow". The workflow consists of a single path of nodes: Start (Input: input\_text), LLM 1 (Model: gpt-4o-mini-preview, Task: Identify technical terms), LLM 2 (Model: gpt-3.5-turbo, Task: Direct Translation), LLM 3 (Model: gpt-3.5-turbo, Task: Problems of direct translation), LLM 4 (Model: gpt-4o-mini-preview, Task: Translation by its meaning - second translation), and End (Model: LLM 4 Output, Task: String). A "Test Run#1" panel on the right shows the execution details, including the input text and the output text. The output text is: "second\_translation": "大型语言模型 (LLM) 正在被越来越多地用于处理那些复杂和多步骤的任务, 比如那些需要推理和与外部环境交互的任务。为了促进这类应用的发展, 我们提出了一个新的模式。"

STATUS	ELAPSED TIME	TOTAL TOKENS
SUCCESS	17.068s	1961 Tokens

INPUT

```
1 {
2   "input_text": "LLMs have increasingly been employed to solve complex, multi-step tasks, e.g., tasks that require a sequence of complex reasoning and interacting with external environments and tools. To facilitate the development of such applications, we introduce StateFlow, a new paradigm that conceptualizes complex
```

OUTPUT

```
1 {
2   "second_translation": "大型语言模型 (LLM) 正在被越来越多地用于处理那些复杂和多步骤的任务, 比如那些需要推理和与外部环境交互的任务。为了促进这类应用的发展, 我们提出了一个新的模式。"
```

Executor: Dify  
Start Time: 2024-04-08 02:47:59  
Elapsed Time: 17.068s  
Total Tokens: 1961 Tokens  
Run Steps: 6

Multi-Path Execution Example:

[Dify](#) Star 19,159 Explore Studio / Sentiment Analysis Knowledge Tools TESTING TEAM Dify

Sentiment Analysis ▶ In Run Mode - Test Run#2 Go back to editor Run Features Publish

```

    graph LR
      Start[Start] --> IfElse[IF/ELSE]
      IfElse -- "Multisentimen is True" --> GPT1[gpt-3.5-turbo]
      GPT1 --> End2[End 2]
      IfElse -- "Multisentimen is False" --> GPT2[gpt-3.5-turbo]
      GPT2 --> End[End]
  
```

**Test Run#1** INPUT RESULT DETAIL TRACING

STATUS: SUCCESS ELAPSED TIME: 1.243s TOTAL TOKENS: 257 Tokens

**INPUT**

```

1 {
2   "input_text": "I don't like it",
3   "Multisentimen": "True",
4   "Categories": "",
5   "sys_files": []
6 }
  
```

**OUTPUT**

```

1 {
2   "text": "[\n  \"score\": -0.8,\n  \"sentiment\": \"Negative\"\n]"
3 }
  
```

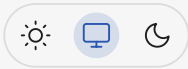
**METADATA**

Status	SUCCESS
Executor	Dify
Start Time	2024-04-09 02:46:49
Elapsed Time	1.243s
Total Tokens	257 Tokens
Run Steps	4

[Previous](#)  
[Start](#)

[Next](#)  
[Answer](#)

Last updated 2 months ago



# Answer

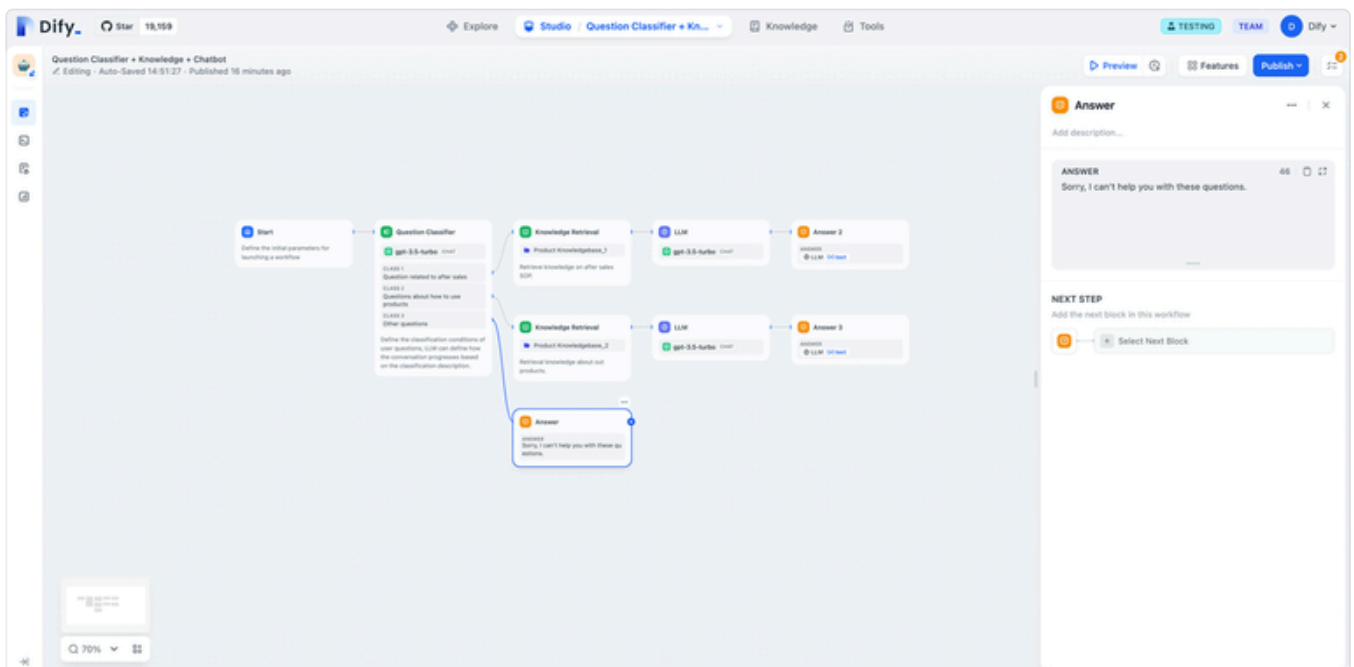
## Answer

Defining Reply Content in a Chatflow Process. In a text editor, you have the flexibility to determine the reply format. This includes crafting a fixed block of text, utilizing output variables from preceding steps as the reply content, or merging custom text with variables for the response.

Answer node can be seamlessly integrated at any point to dynamically deliver content into the dialogue responses. This setup supports a live-editing configuration mode, allowing for both text and image content to be arranged together. The configurations include:

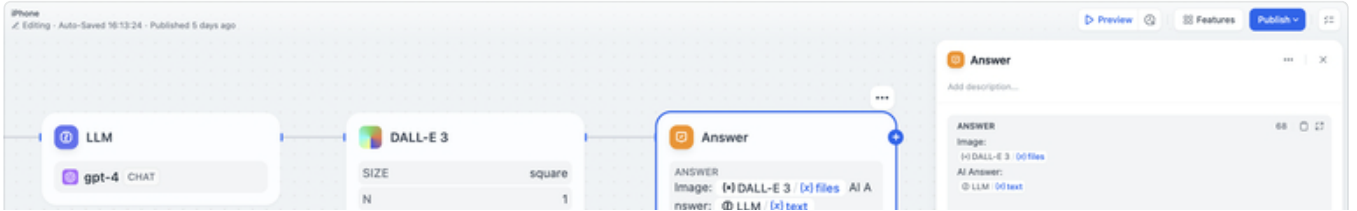
1. Outputting the reply content from a Language Model (LLM) node.
2. Outputting generated images.
3. Outputting plain text.


Example 1: Output plain text.




Example 2: Output image and LLM reply.





 Workflow Process >




Using an iPhone involves several steps:

1. Power On: Press and hold the side button until the Apple logo appears.
2. Set Up: Follow the instructions on the screen to set up your iPhone. This includes selecting a language, setting up Wi-Fi, enabling location services, setting up a passcode, and signing in with your Apple ID.
3. Home Screen: Once setup is complete, you'll be taken to the home screen. Here, you can access all your apps by swiping left or right.

Remember, the exact steps might vary slightly depending on which iPhone model you have.

CITATIONS

 upload\_files\_8882d60f-188b-48...

Previous  
End

Next  
LLM

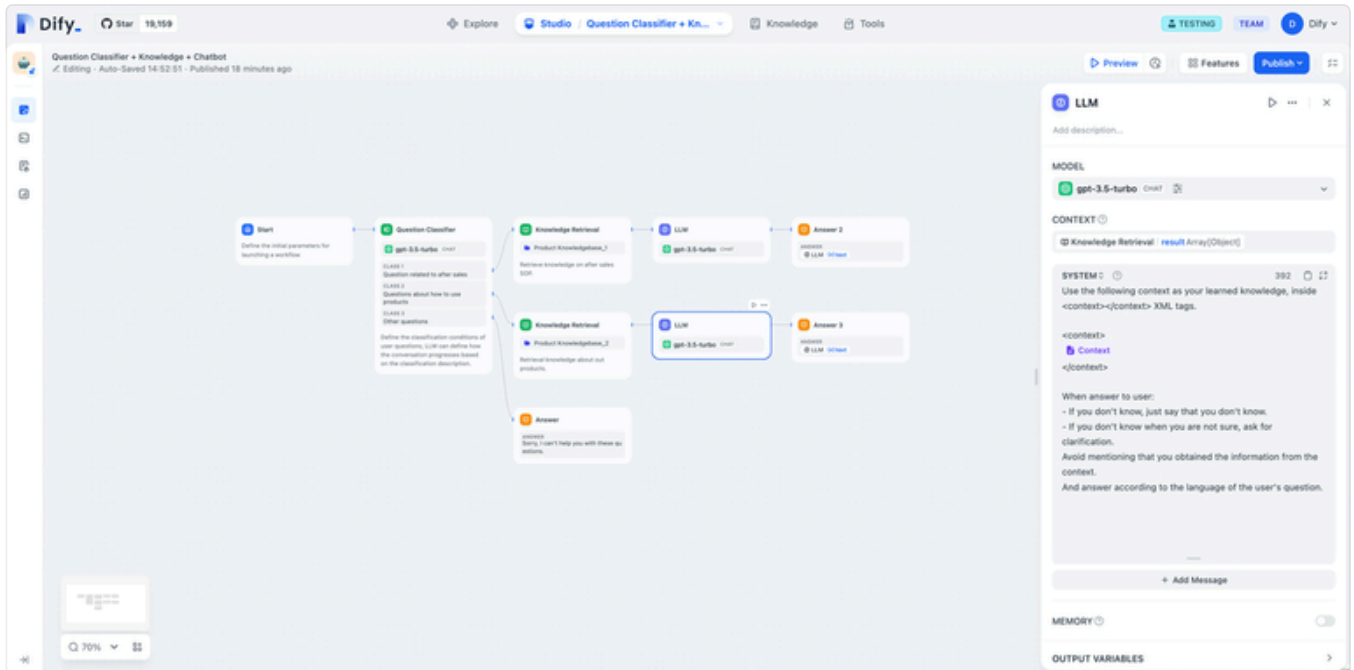
Last updated 2 months ago





# LLM

Invoking a Large Language Model for Question Answering or Natural Language Processing. Within an LLM node, you can select an appropriate model, compose prompts, set the context referenced in the prompts, configure memory settings, and adjust the memory window size.

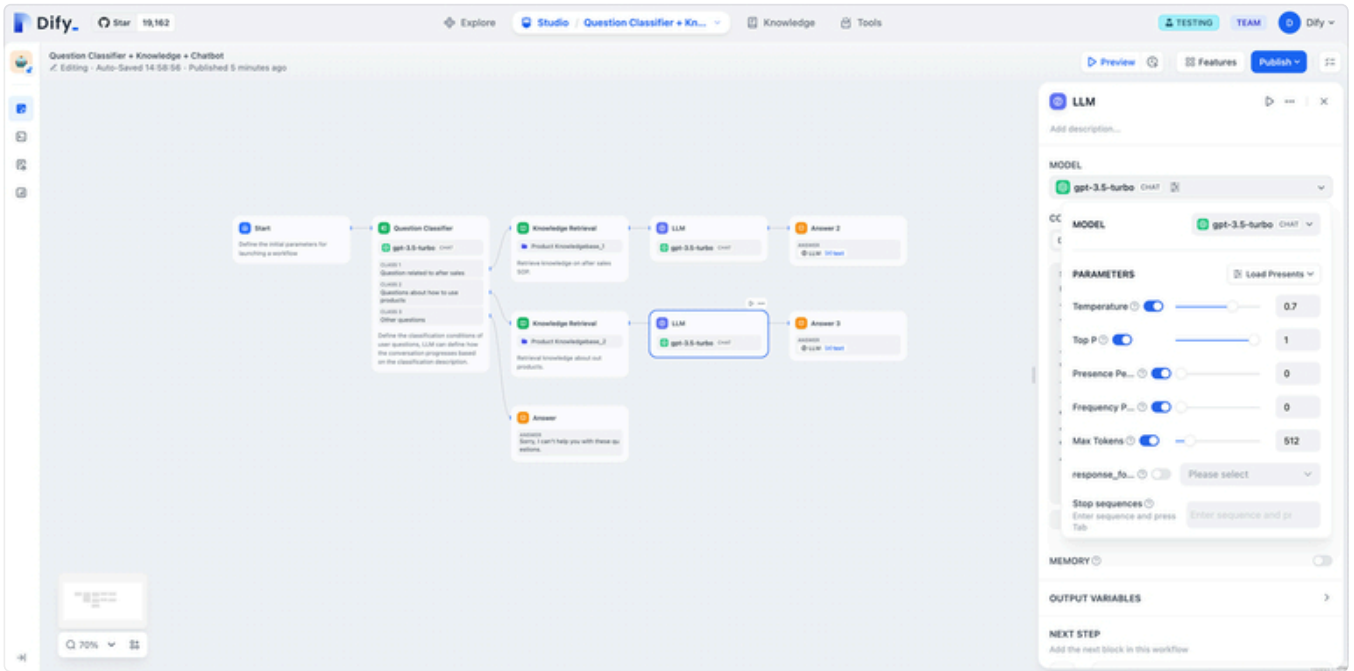


Configuring an LLM node primarily involves two steps:

1. Selecting a model
2. Composing system prompts

## Model Configuration

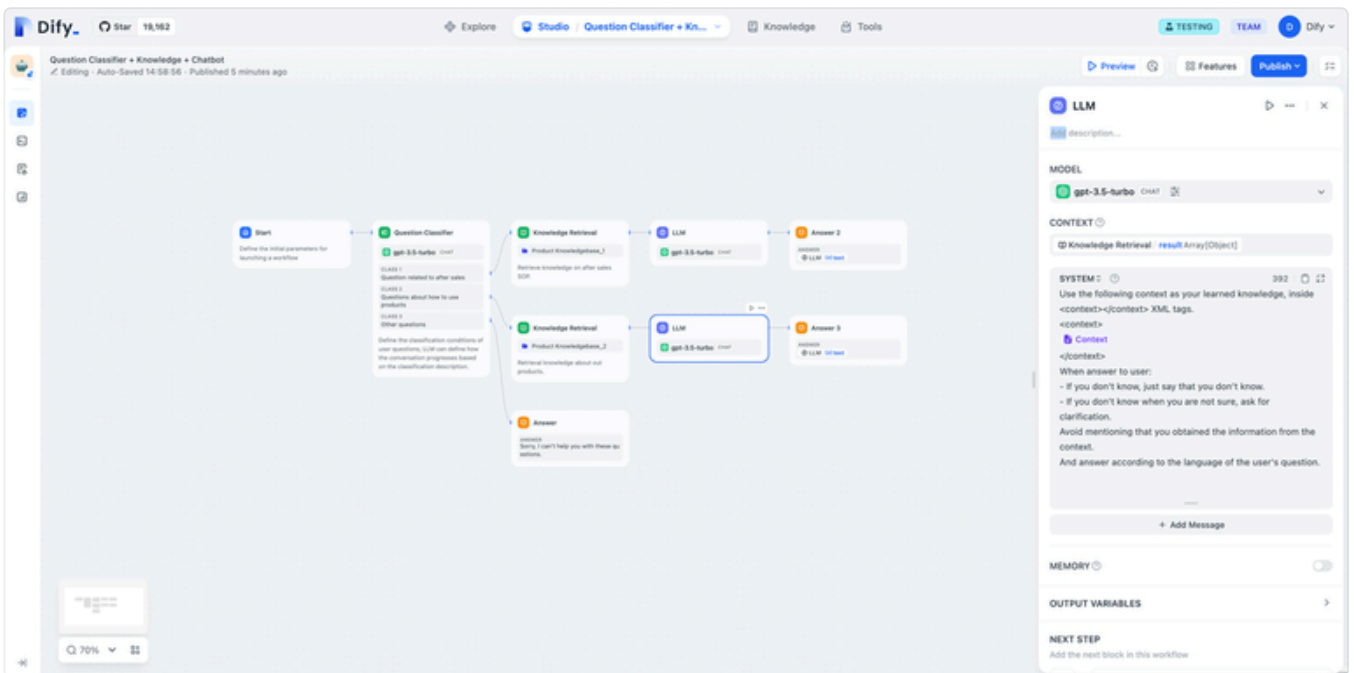
Before selecting a model suitable for your task, you must complete the model configuration in "System Settings—Model Provider". The specific configuration method can be referenced in the [model configuration instructions](#). After selecting a model, you can configure its parameters.



## Write Prompts

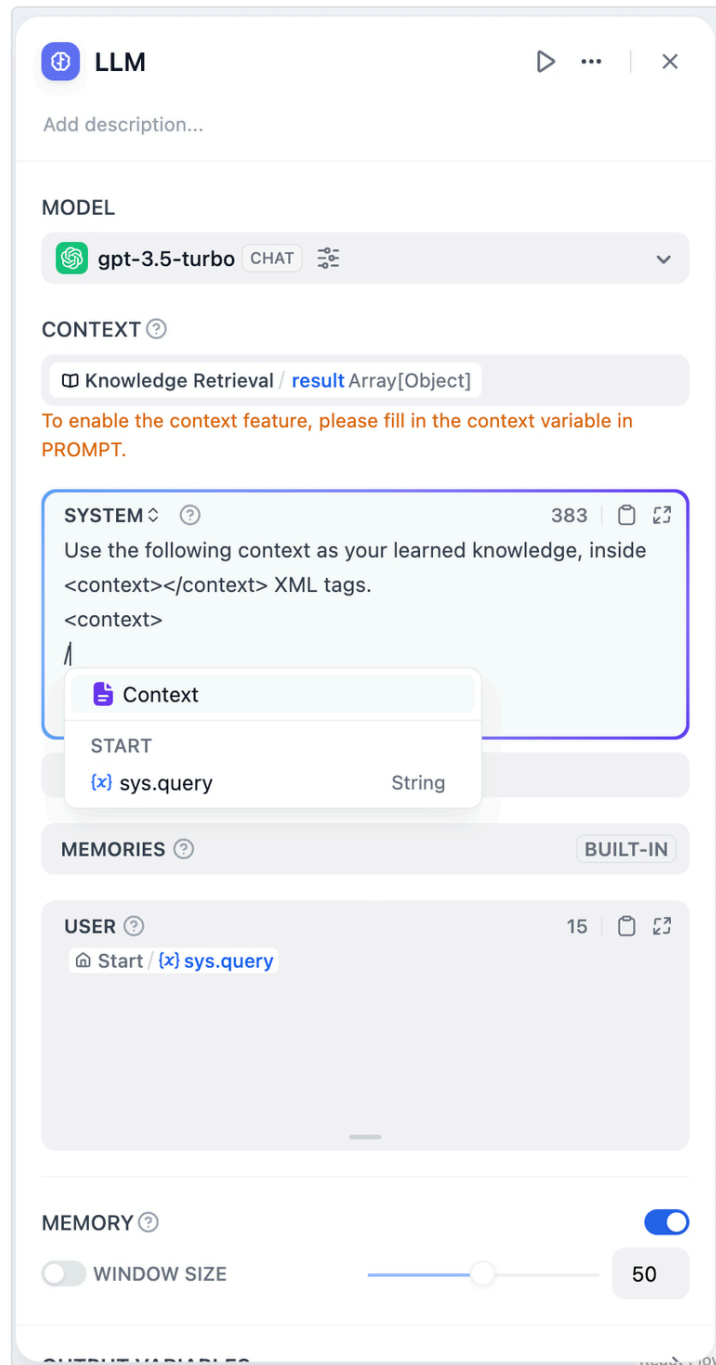
Within an LLM node, you can customize the model input prompts. If you choose a conversational model, you can customize the content of system prompts, user messages, and assistant messages.

For instance, in a knowledge base Q&A scenario, after linking the "Result" variable from the knowledge base retrieval node in "Context", inserting the "Context" special variable in the prompts will use the text retrieved from the knowledge base as the context background information for the model input.

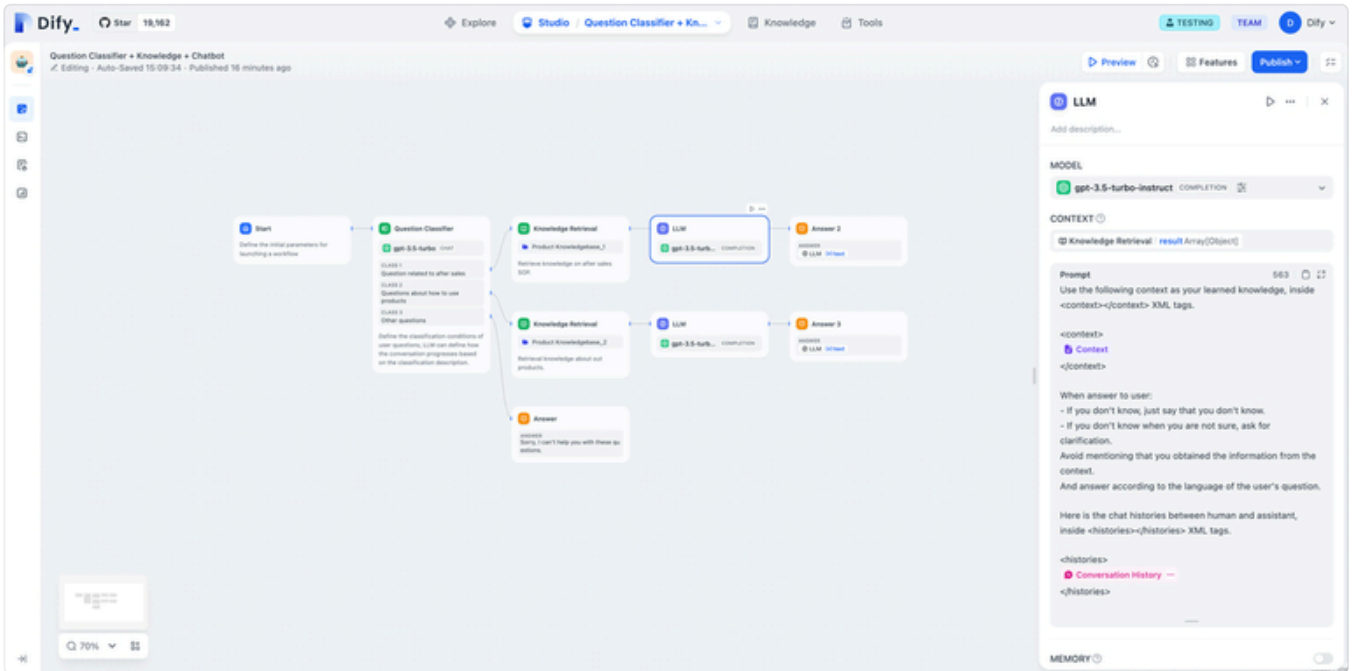


In the prompt editor, you can bring up the variable insertion menu by typing "/" or "{" to insert special variable blocks or variables from preceding flow nodes into the prompts as context

content.



If you opt for a completion model, the system provides preset prompt templates for conversational applications. You can customize the content of the prompts and insert special variable blocks like "Conversation History" and "Context" at appropriate positions by typing "/" or "{", enabling richer conversational functionalities.



## Memory Toggle Settings

In conversational applications (Chatflow), the LLM node defaults to enabling system memory settings. In multi-turn dialogues, the system stores historical dialogue messages and passes them into the model. In workflow applications (Workflow), system memory is turned off by default, and no memory setting options are provided.

## Memory Window Settings

If the memory window setting is off, the system dynamically passes historical dialogue messages according to the model's context window. With the memory window setting on, you can configure the number of historical dialogue messages to pass based on your needs.

## Dialogue Role Name Settings

Due to differences in model training phases, different models adhere to role name commands to varying degrees, such as Human/Assistant, Human/AI, 人类/助手, etc. To adapt to the prompt response effects of multiple models, the system allows setting dialogue role names, modifying the role prefix in conversation history.

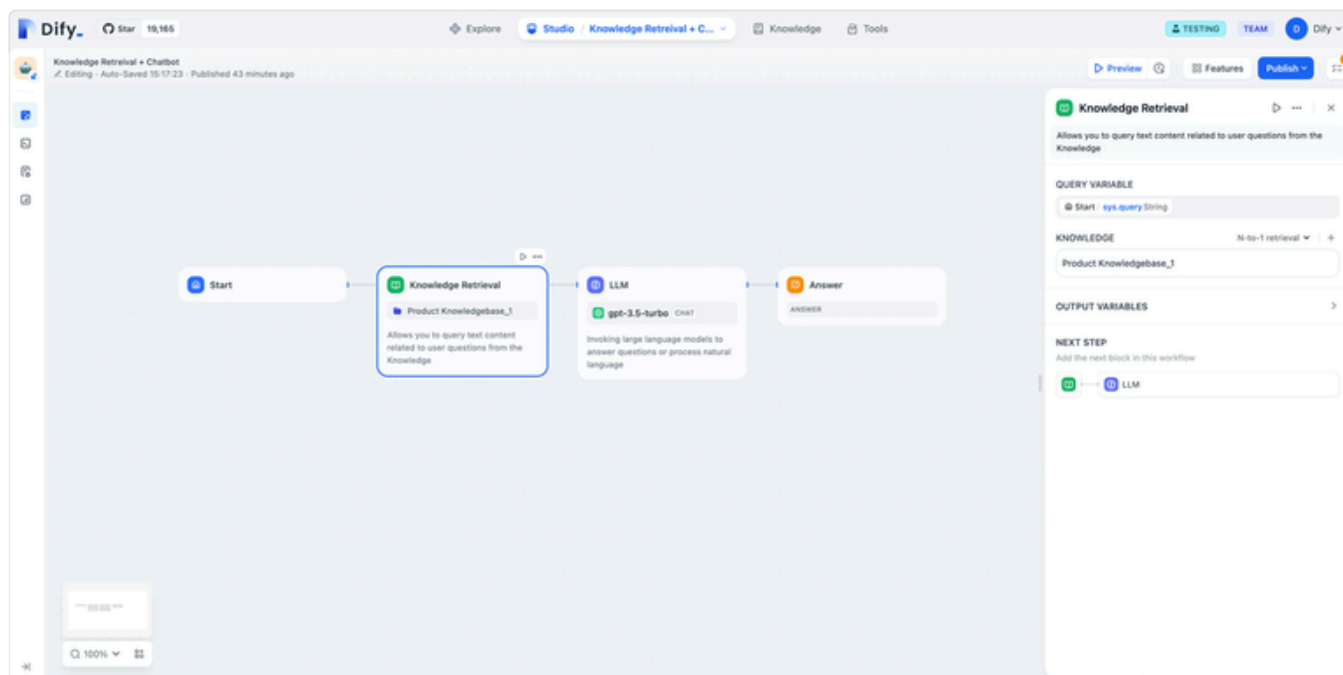
Previous Answer

Next Knowledge Retrieval



# Knowledge Retrieval

The Knowledge Base Retrieval Node is designed to query text content related to user questions from the Dify Knowledge Base, which can then be used as context for subsequent answers by the Large Language Model (LLM).



Configuring the Knowledge Base Retrieval Node involves three main steps:

1. **Selecting the Query Variable**
2. **Choosing the Knowledge Base for Query**
3. **Configuring the Retrieval Strategy**

## Selecting the Query Variable

In knowledge base retrieval scenarios, the query variable typically represents the user's input question. In the "Start" node of conversational applications, the system pre-sets "sys.query" as the user input variable. This variable can be used to query the knowledge base for text segments most closely related to the user's question.

## Choosing the Knowledge Base for Query

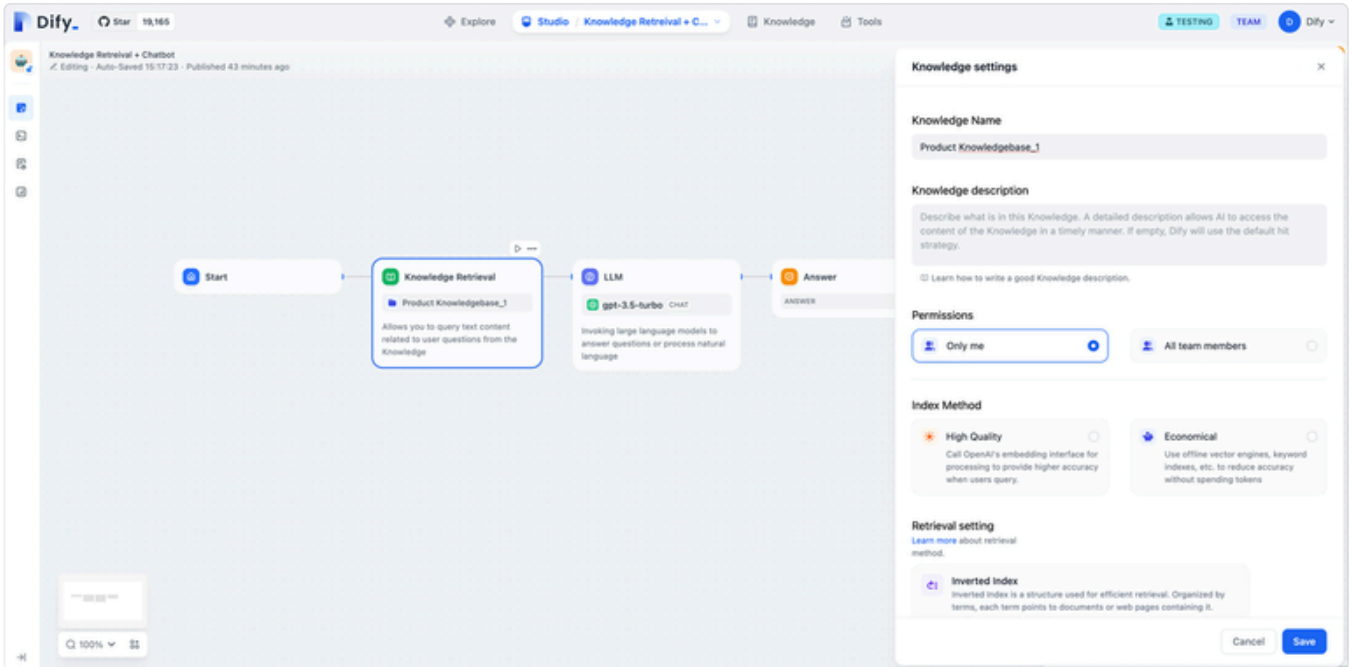
Within the knowledge base retrieval node, you can add an existing knowledge base from Dify. For instructions on creating a knowledge base within Dify, please refer to the knowledge



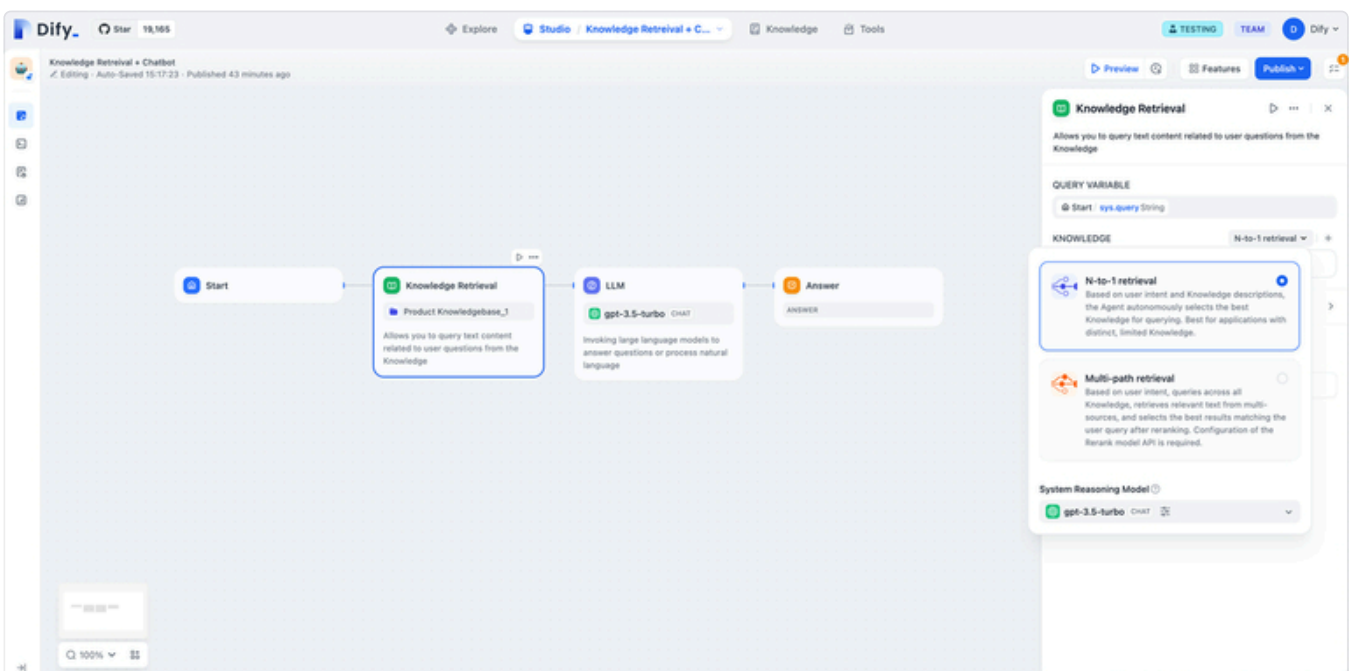
base [help documentation](#).

## Configuring the Retrieval Strategy

It's possible to modify the indexing strategy and retrieval mode for an individual knowledge base within the node. For a detailed explanation of these settings, refer to the knowledge base [help documentation](#).



executed through function calling, requiring the selection of a system reasoning model. In the multi-way recall mode, a Rerank model needs to be configured for result re-ranking. For a detailed explanation of these two recall strategies, refer to the retrieval mode explanation in the [help documentation](#).



Previous  
LLM

Next  
Question Classifier

Last updated 2 months ago



# Question Classifier

Question Classifier node defines the categorization conditions for user queries, enabling the LLM to dictate the progression of the dialogue based on these categorizations. As illustrated in a typical customer service robot scenario, the question classifier can serve as a preliminary step to knowledge base retrieval, identifying user intent. Classifying user intent before retrieval can significantly enhance the recall efficiency of the knowledge base.

The screenshot displays the Dify AI Studio interface for configuring a 'Question Classifier' node. The main workspace shows a workflow diagram with the following components:

- Start Node:** 'Define the initial parameters for launching a workflow'.
- Question Classifier Node:** Configured with 'gpt-3.5-turbo' as the model. It defines three classification classes:
  - Class 1:** 'Question related to after sales' (31 questions).
  - Class 2:** 'Questions about how to use products' (35 questions).
  - Class 3:** 'Other questions' (15 questions).
- Knowledge Retrieval Nodes:** Two nodes, each linked to a specific class from the classifier.
- LLM Nodes:** Two nodes, each linked to a Knowledge Retrieval node, using 'gpt-3.5-turbo' as the model.
- Answer Node:** The final output of the workflow.

The right sidebar provides detailed configuration for the 'Question Classifier' node, including input variables (e.g., 'sys.query String'), model selection ('gpt-3.5-turbo'), and the class definitions.

1. **Selecting the Input Variable**
2. **Configuring the Inference Model**
3. **Writing the Classification Method**

**Selecting the Input Variable** In conversational customer scenarios, you can use the user input variable from the "Start Node" (sys.query) as the input for the question classifier. In automated/batch processing scenarios, customer feedback or email content can be utilized as input variables.

**Configuring the Inference Model** The question classifier relies on the natural language processing capabilities of the LLM to categorize text. You will need to configure an inference model for the classifier. Before configuring this model, you might need to complete the model setup in "System Settings - Model Provider". The specific configuration method can be found

in the [model configuration instructions](#). After selecting a suitable model, you can configure its parameters.

**Writing Classification Conditions** You can manually add multiple classifications by composing keywords or descriptive sentences that fit each classification. Based on the descriptions of these conditions, the question classifier can route the dialogue to the appropriate process path according to the semantics of the user's input.

Previous  
Knowledge Retrieval

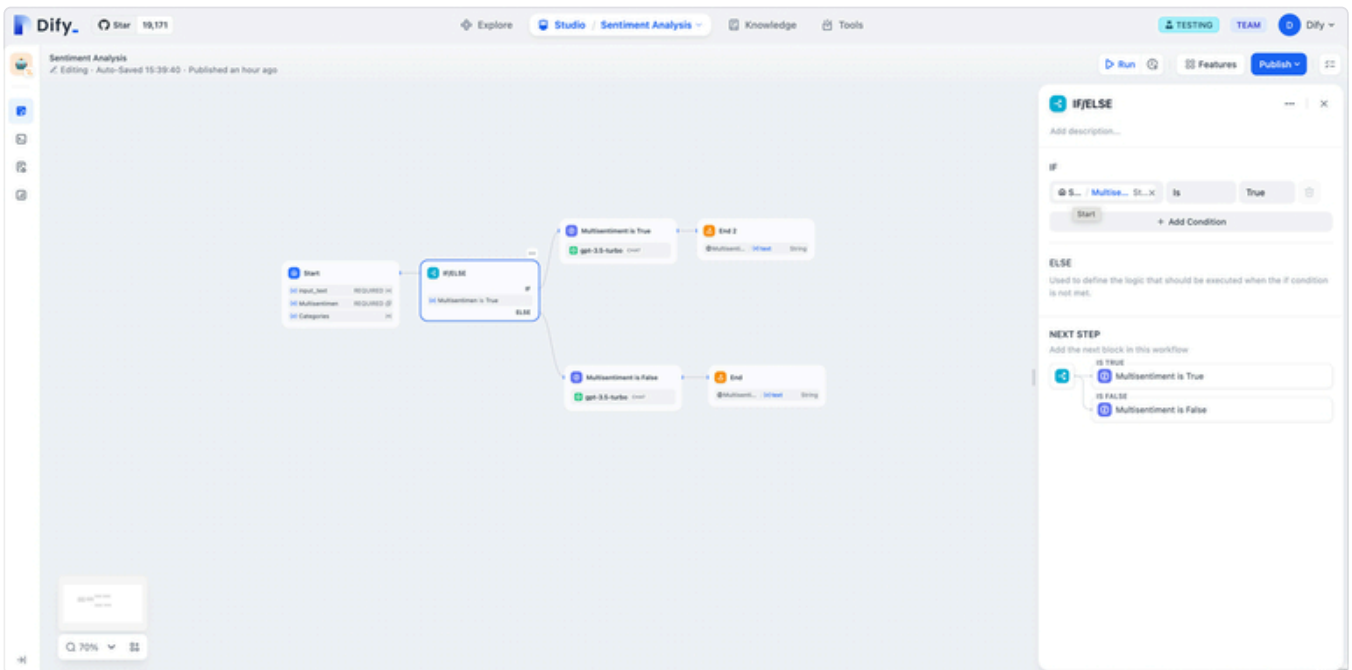
Next  
IF/ELSE

Last updated 2 months ago





The IF/ELSE Node allows you to split a workflow into two branches based on if/else conditions. In this node, you can set one or more IF conditions. When the IF condition(s) are met, the workflow proceeds to the next step under the "IS TRUE" branch. If the IF condition(s) are not met, the workflow triggers the next step under the "IS FALSE" branch.



[Previous Question Classifier](#)

[Next Code](#)

Last updated 2 months ago



# Code

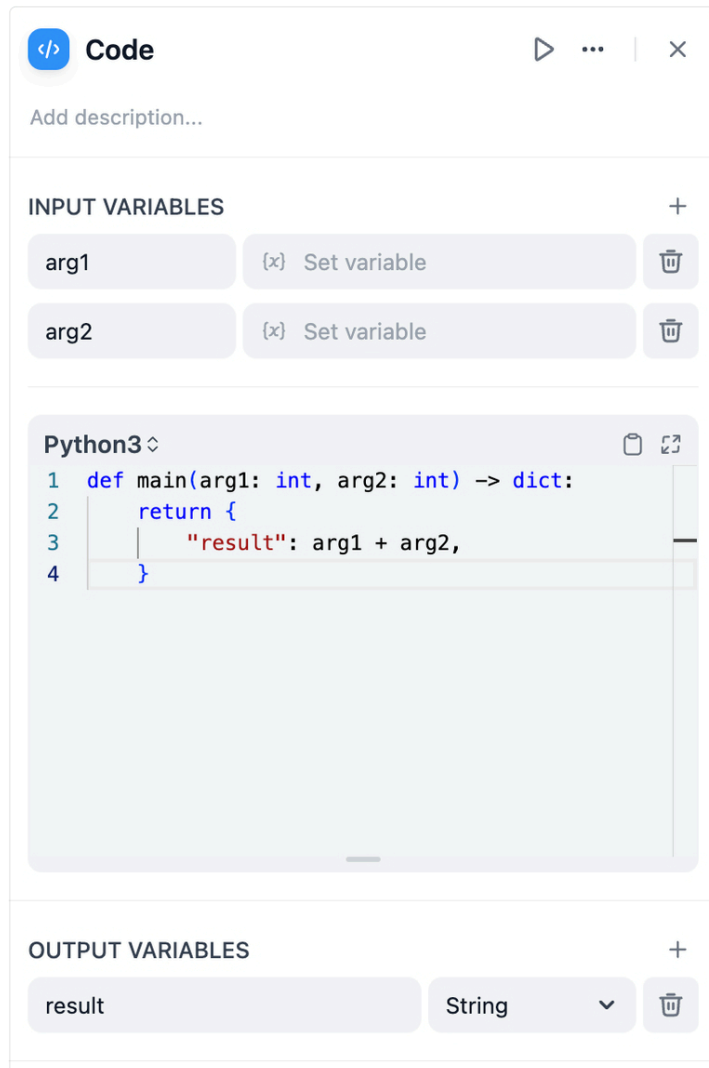
## Navigation

- [Introduction](#)
- [Use Cases](#)
- [Local Deployment](#)
- [Security Policy](#)

## Introduction

The code node supports the execution of Python / NodeJS code to perform data transformations within workflows. It simplifies your workflows, suitable for Arithmetic, JSON transform, text processing, and more scenarios.

This node significantly enhances developers' flexibility, allowing them to embed custom Python or Javascript scripts in their workflows and manipulate variables in ways that preset nodes cannot achieve. Through configuration options, you can specify the required input and output variables and write the corresponding execution code:



## Configuration

If you need to use variables from other nodes within the code node, you need to define the variable names in `input variables` and reference these variables, see [Variable Reference](#) for details.

## Use Cases

With the code node, you can perform the following common operations:

### Structured Data Processing

In workflows, it's often necessary to deal with unstructured data processing, such as parsing, extracting, and transforming JSON strings. A typical example is data processing in the HTTP node, where data might be nested within multiple layers of JSON objects, and we need to



extract certain fields. The code node can help you accomplish these tasks. Here's a simple example that extracts the `data.name` field from a JSON string returned by an HTTP node:

```
def main(http_response: str) -> str:
    import json
    data = json.loads(http_response)
    return {
        # do not forget to declare 'result' in the output variables
        'result': data['data']['name']
    }
```

When complex mathematical calculations are needed in workflows, the code node can also be used. For example, to calculate a complex mathematical formula or perform some statistical analysis on the data. Here is a simple example that calculates the variance of a list:

```
def main(x: list) -> float:
    return {
        # do not forget to declare 'result' in the output variables
        'result': sum([(i - sum(x) / len(x)) ** 2 for i in x]) / len(x)
    }
```

## Data Concatenation

Sometimes, you may need to concatenate multiple data sources, such as multiple knowledge retrievals, data searches, API calls, etc. The code node can help you integrate these data sources. Here's a simple example that merges data from two knowledge bases:

```
def main(knowledge1: list, knowledge2: list) -> list:
    return {
        # do not forget to declare 'result' in the output variables
        'result': knowledge1 + knowledge2
    }
```

## Local Deployment

If you are a user deploying locally, you need to start a sandbox service, which ensures that malicious code is not executed. Also, launching this service requires Docker, and you can find specific information about the Sandbox service [here](#). You can also directly start the service using docker-compose

```
docker-compose -f docker-compose.middleware.yaml up -d
```

# Security Policy

The execution environment is sandboxed for both Python and Javascript, meaning that certain functionalities that require extensive system resources or pose security risks are not available. This includes, but is not limited to, direct file system access, network calls, and operating system-level commands.

[Previous](#)  
[IF/ELSE](#)

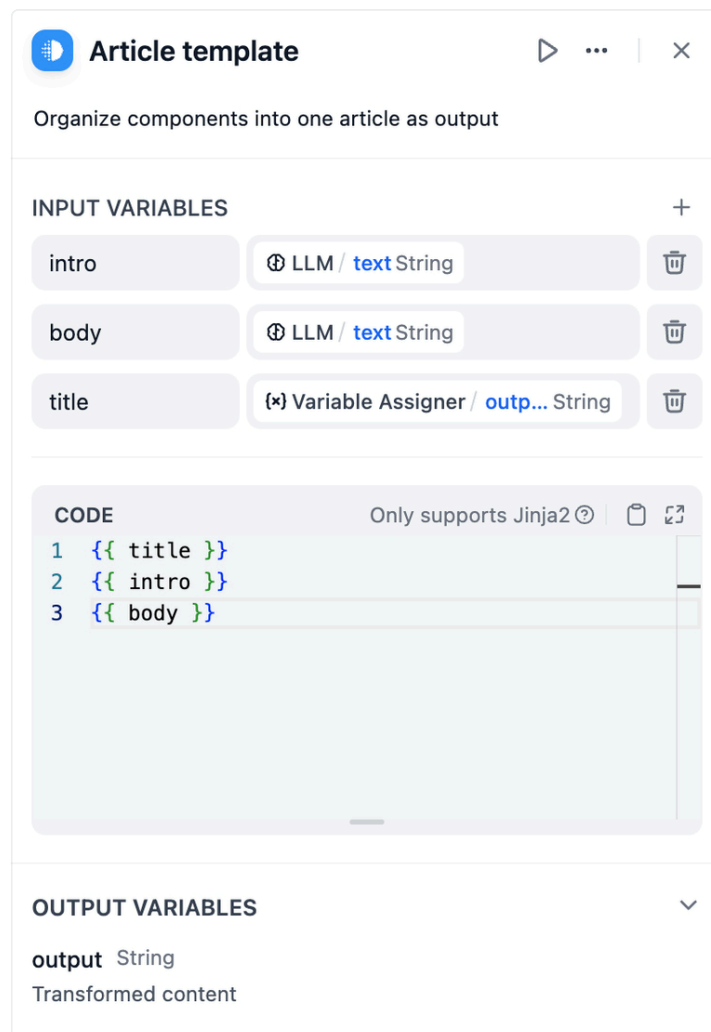
[Next](#)  
[Template](#)

Last updated 2 months ago



# Template

Template lets you dynamically format and combine variables from previous nodes into a single text-based output using Jinja2, a powerful templating syntax for Python. It's useful for combining data from multiple sources into a specific structure required by subsequent nodes. The simple example below shows how to assemble an article by piecing together various previous outputs:



**Article template**

Organize components into one article as output

**INPUT VARIABLES**

- intro (LLM / text String)
- body (LLM / text String)
- title ((\*) Variable Assigner / outp... String)

**CODE** (Only supports Jinja2)

```
1 {{ title }}
2 {{ intro }}
3 {{ body }}
```

**OUTPUT VARIABLES**

- output String  
Transformed content

Beyond naive use cases, you can create more complex templates as per Jinja's [documentation](#) for a variety of tasks. Here's one template that structures retrieved chunks and their relevant metadata from a knowledge retrieval node into a formatted markdown:

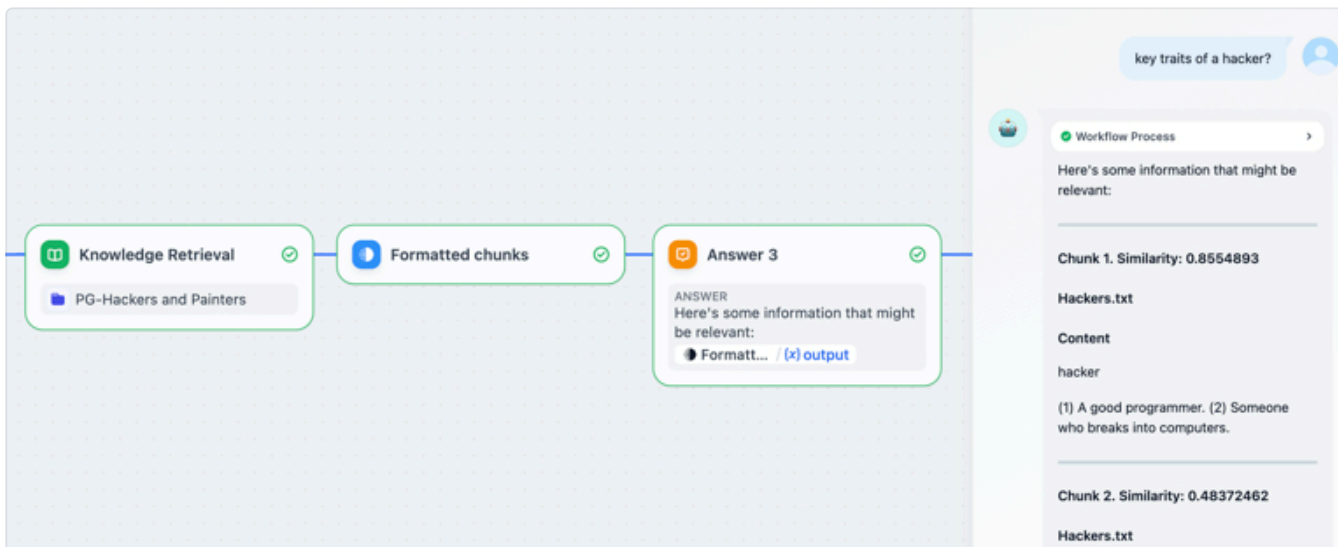
```
### Similarity: {{ item.metadata.score | default('N/A') }}

#### {{ item.title }}

##### Content
{{ item.content | replace('\n', '\n\n') }}

---

{% endfor %}
```



This template node can then be used within a Chatflow to return intermediate outputs to the end user, before a LLM response is initiated.

The `Answer` node in a Chatflow is non-terminal. It can be inserted anywhere to output responses at multiple points within the flow.

Previous Code

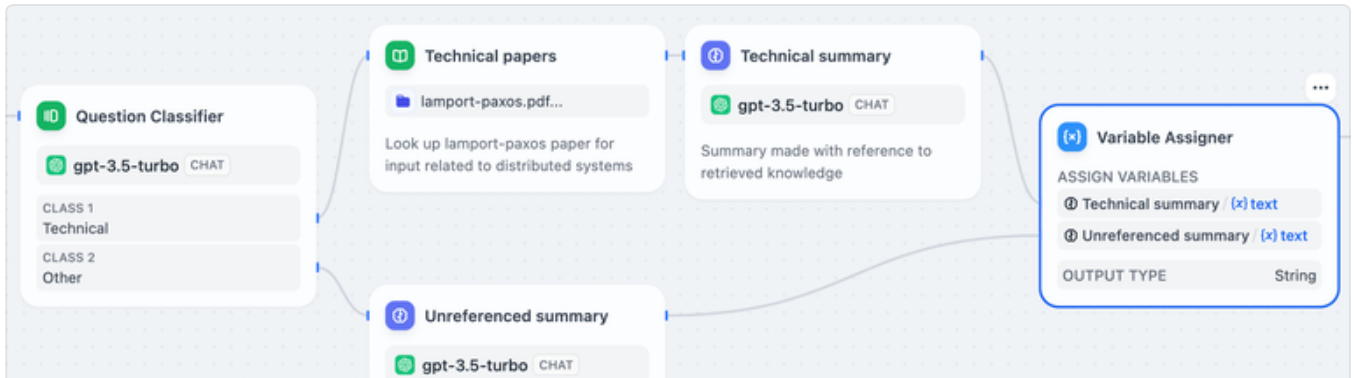
Next Variable Assigner

Last updated 2 months ago

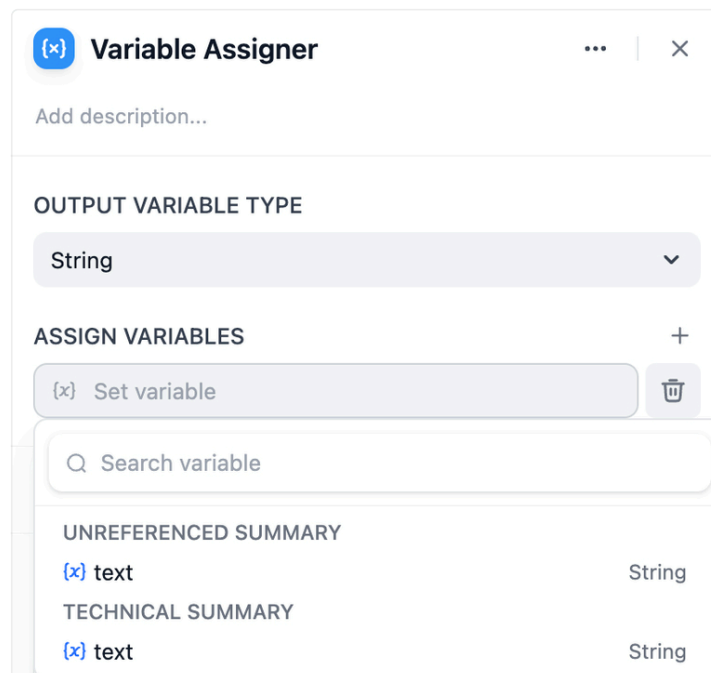


# Variable Assigner

The Variable Assigner node serves as a hub for collecting branch outputs within the workflow, ensuring that regardless of which branch is taken, the output can be referenced by a single variable. The output can subsequently be manipulated by nodes downstream.



Object, and Array. Given the specified output type, you may add input variables from the dropdown list of variables to the node. The list of variables is derived from previous branch outputs and autofiltered based on the specified type.



Variable Assigner gives a single `output` variable of the specified type for downstream use.

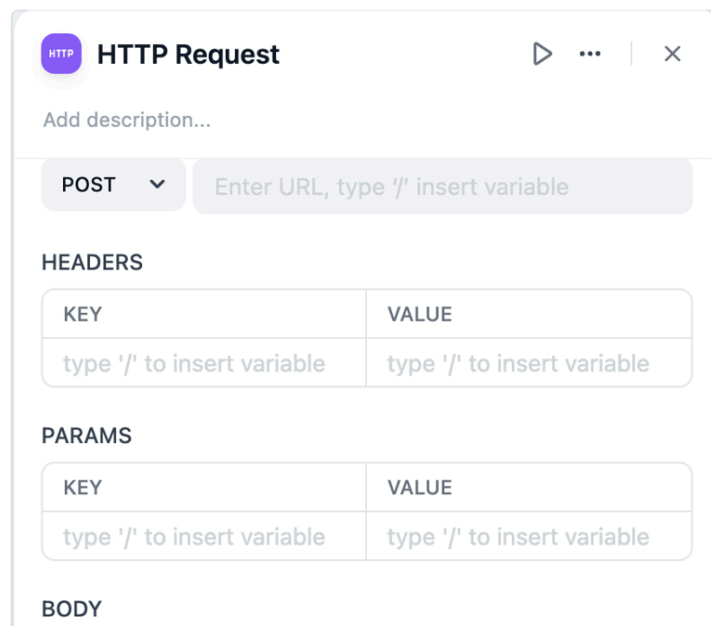
Next  
HTTP Request

Last updated 2 months ago

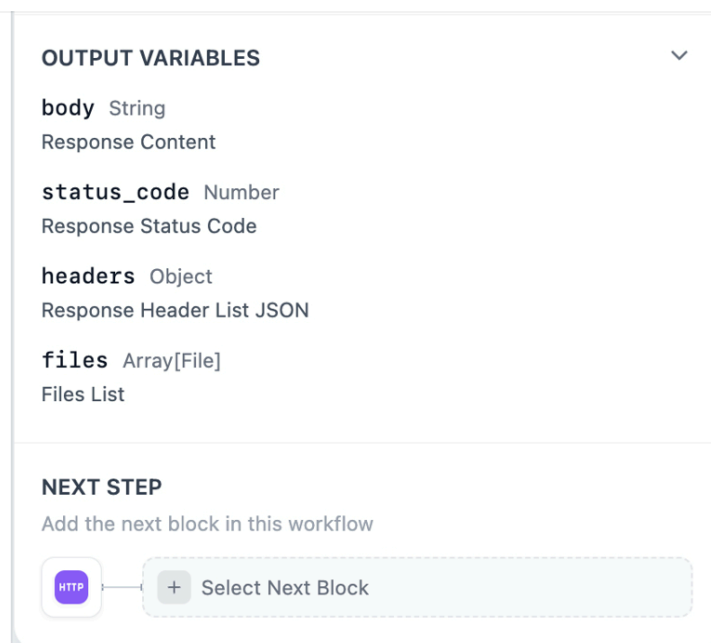


# HTTP Request

HTTP Request node lets you craft and dispatch HTTP requests to specified endpoints, enabling a wide range of integrations and data exchanges with external services. The node supports all common HTTP request methods, and lets you fully customize over the URL, headers, query parameters, body content, and authorization details of the request.



The screenshot shows the configuration interface for the HTTP Request node. At the top, there's a title "HTTP Request" with a play button, a menu icon, and a close button. Below the title is a text input field for "Add description...". The main configuration area is divided into sections: "METHOD" (set to "POST"), "URL" (with a placeholder "Enter URL, type '/' insert variable"), "HEADERS", "PARAMS", and "BODY". The "HEADERS" and "PARAMS" sections are represented as tables with "KEY" and "VALUE" columns, each containing a placeholder "type '/' to insert variable".



The screenshot shows the "OUTPUT VARIABLES" and "NEXT STEP" sections of the HTTP Request node configuration. The "OUTPUT VARIABLES" section lists the following variables: "body" (String, Response Content), "status\_code" (Number, Response Status Code), "headers" (Object, Response Header List JSON), and "files" (Array[File], Files List). The "NEXT STEP" section is titled "Add the next block in this workflow" and contains a button labeled "+ Select Next Block" with an "HTTP" icon to its left.

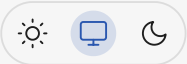
A really handy feature with HTTP request is the ability to dynamically construct the request by inserting variables in different fields. For instance, in a customer support scenario,

variables such as username or customer ID can be used to personalize automated responses sent via a POST request, or retrieve individual-specific information related to the customer. The HTTP request returns `body`, `status_code`, `headers`, and `files` as outputs. If the response includes files of [MIME](#) types (currently limited to images), the node automatically saves these as `files` for downstream use.

[Previous](#)  
[Variable Assigner](#)

[Next](#)  
[Tools](#)

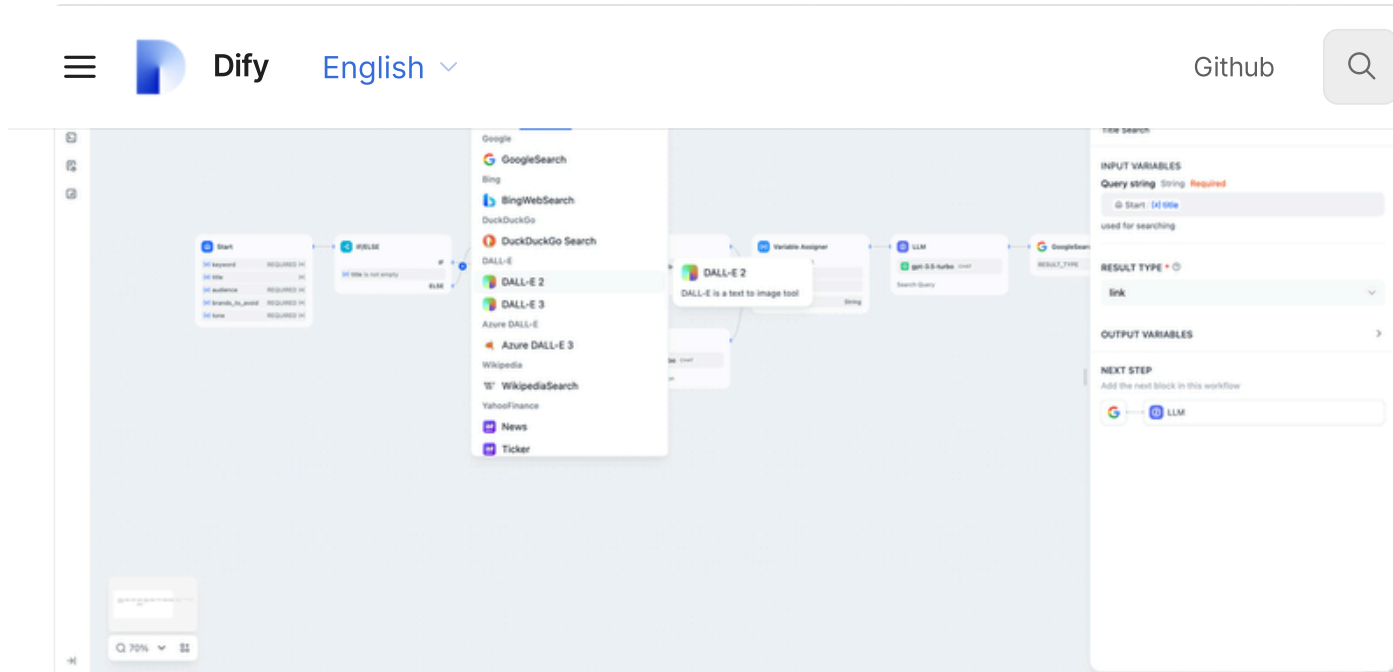
Last updated 2 months ago





# Tools

Within a workflow, Dify provides both built-in and customizable tools. Before utilizing these tools, you need to "authorize" them. If the built-in tools do not meet your requirements, you can create custom tools within "Dify—Tools".



Configuring a tool node generally involves two steps:

1. **Authorizing the Tool/Creating Custom Tools**
2. **Configuring Tool Inputs and Parameters**

For guidance on creating custom tools and configuring them, please refer to the [tool configuration instructions](#).

Previous  
HTTP Request

Next  
Preview&Run

Last updated 2 months ago

