



LangServe



LangServe

release v0.0.41

downloads/month 66k

open issues 47

 LangChain Community

We will be releasing a hosted version of LangServe for one-click deployments of LangChain applications. [Sign up here](#) to get on the waitlist.

Overview

LangServe helps developers deploy LangChain **runnables** and **chains** as a REST API.

This library is integrated with **FastAPI** and uses **pydantic** for data validation.

In addition, it provides a client that can be used to call into runnables deployed on a server. A javascript client is available in **LangChainJS**.

Features

- Input and Output schemas automatically inferred from your LangChain object, and enforced on every API call, with rich error messages
- API docs page with JSONSchema and Swagger (insert example link)
- Efficient `/invoke/`, `/batch/` and `/stream/` endpoints with support for many concurrent requests on a single server
- `/stream_log/` endpoint for streaming all (or some) intermediate steps from your chain/agent
- **new** as of 0.0.40, supports `astream_events` to make it easier to stream without needing to parse the output of `stream_log`.
- Playground page at `/playground/` with streaming output and intermediate steps
- Built-in (optional) tracing to [LangSmith](#), just add your API key (see [Instructions](#))
- All built with battle-tested open-source Python libraries like FastAPI, Pydantic, uvloop and asyncio.
- Use the client SDK to call a LangServe server as if it was a Runnable running locally (or call the HTTP API directly)
- [LangServe Hub](#)

Limitations

- Client callbacks are not yet supported for events that originate on the server
- OpenAPI docs will not be generated when using Pydantic V2. Fast API does not support [mixing pydantic v1 and v2 namespaces](#). See section below for more details.

Hosted LangServe

We will be releasing a hosted version of LangServe for one-click deployments of LangChain applications. [Sign up here](#) to get on the waitlist.

Security

- Vulnerability in Versions 0.0.13 - 0.0.15 -- playground endpoint allows accessing arbitrary files on server.
[Resolved in 0.0.16.](#)

Installation

For both client and server:

```
pip install "langserve[all]"
```

or `pip install "langserve[client]"` for client code, and
`pip install "langserve[server]"` for server code.

LangChain CLI

Use the `LangChain` CLI to bootstrap a `LangServe` project quickly.

To use the langchain CLI make sure that you have a recent version of `langchain-cli` installed. You can install it with
`pip install -U langchain-cli`.

```
langchain app new ../path/to/directory
```

Examples

Get your LangServe instance started quickly with [LangChain Templates](#).

For more examples, see the templates [index](#) or the [examples](#) directory.

Description	Links
LLMs Minimal example that reserves OpenAI and Anthropic chat models. Uses async, supports batching and streaming.	server , client
Retriever Simple server that exposes a retriever as a runnable.	server , client
Conversational Retriever A Conversational Retriever exposed via LangServe	server , client
Agent without conversation history based on OpenAI tools	server , client
Agent with conversation history based on OpenAI tools	server , client
RunnableWithMessageHistory to implement chat persisted on backend, keyed off a <code>session_id</code> supplied by client.	server , client
RunnableWithMessageHistory to implement chat persisted on backend, keyed off a <code>conversation_id</code> supplied by client, and	server , client

Description	Links
<code>user_id</code> (see Auth for implementing <code>user_id</code> properly).	
Configurable Runnable to create a retriever that supports run time configuration of the index name.	server , client
Configurable Runnable that shows configurable fields and configurable alternatives.	server , client
APIHandler Shows how to use <code>APIHandler</code> instead of <code>add_routes</code> . This provides more flexibility for developers to define endpoints. Works well with all FastAPI patterns, but takes a bit more effort.	server
LCEL Example Example that uses LCEL to manipulate a dictionary input.	server , client
Auth with <code>add_routes</code> : Simple authentication that can be applied across all endpoints associated with app. (Not useful on its own for implementing per user logic.)	server

Description	Links
Auth with <code>add_routes</code> : Simple authentication mechanism based on path dependencies. (No useful on its own for implementing per user logic.)	server
Auth with <code>add_routes</code> : Implement per user logic and auth for endpoints that use per request config modifier. (Note: At the moment, does not integrate with OpenAPI docs.)	server , client
Auth with <code>APIHandler</code> : Implement per user logic and auth that shows how to search only within user owned documents.	server , client
Widgets Different widgets that can be used with playground (file upload and chat)	server
Widgets File upload widget used for LangServe playground.	server , client

Sample Application

Server

Here's a server that deploys an OpenAI chat model, an Anthropic chat model, and a chain that uses the Anthropic model to tell a joke about a topic.

```
#!/usr/bin/env python
from fastapi import FastAPI
from langchain.prompts import
ChatPromptTemplate
from langchain.chat_models import
ChatAnthropic, ChatOpenAI
from langserve import add_routes

app = FastAPI(
    title="LangChain Server",
    version="1.0",
    description="A simple api server using
Langchain's Runnable interfaces",
)

add_routes(
    app,
    ChatOpenAI(),
    path="/openai",
)

add_routes(
    app,
    ChatAnthropic(),
```



```
    path="/anthropic",
)


model = ChatAnthropic()
prompt =
ChatPromptTemplate.from_template("tell me a
joke about {topic}")
add_routes(
    app,
    prompt | model,
    path="/joke",
)

if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="localhost",
port=8000)
```

Docs

If you've deployed the server above, you can view the generated OpenAPI docs using:

 If using pydantic v2, docs will not be generated for *invoke*, *batch*, *stream*, *stream_log*. See [Pydantic](#) section below for more details.

```
curl localhost:8000/docs
```

make sure to **add** the `/docs` suffix.

⚠️ Index page `/` is not defined by **design**, so `curl localhost:8000` or visiting the URL will return a 404. If you want content at `/` define an endpoint `@app.get("/")`.

Client

Python SDK

```
from langchain.schema import SystemMessage,
HumanMessage
from langchain.prompts import ChatPromptTemplate
from langchain.schema.runnable import RunnableMap
from langserve import RemoteRunnable

openai =
RemoteRunnable("http://localhost:8000/openai/")
anthropic =
RemoteRunnable("http://localhost:8000/anthropic")
joke_chain =
RemoteRunnable("http://localhost:8000/joke/")

joke_chain.invoke({"topic": "parrots"})
```

```
# or async
await joke_chain.ainvoke({"topic": "parrots"})

prompt = [
    SystemMessage(content='Act like either a cat
or a parrot.'),
    HumanMessage(content='Hello!')
]

# Supports astream
async for msg in anthropic.astream(prompt):
    print(msg, end="", flush=True)

prompt = ChatPromptTemplate.from_messages(
    [("system", "Tell me a long story about
{topic}")]
)

# Can define custom chains
chain = prompt | RunnableMap({
    "openai": openai,
    "anthropic": anthropic,
})

chain.batch([{"topic": "parrots"}, {"topic":
"cats"}])
```

In TypeScript (requires LangChain.js version 0.0.166 or later):

```
import {RemoteRunnable} from
"langchain/runnables/remote";

const chain = new RemoteRunnable({
  url: `http://localhost:8000/joke/`,
});
const result = await chain.invoke({
  topic: "cats",
});
```

Python using `requests`:

```
import requests

response = requests.post(
    "http://localhost:8000/joke/invoke",
    json={'input': {'topic': 'cats'}}
)
response.json()
```

You can also use `curl`:

```
curl --location --request POST
'http://localhost:8000/joke/invoke' \
  --header 'Content-Type: application/json' \
  --data-raw '{
```

```
"input": {  
    "topic": "cats"  
}  
'
```

Endpoints

The following code:

```
...  
add_routes(  
    app,  
    runnable,  
    path="/my_runnable",  
)
```

adds of these endpoints to the server:

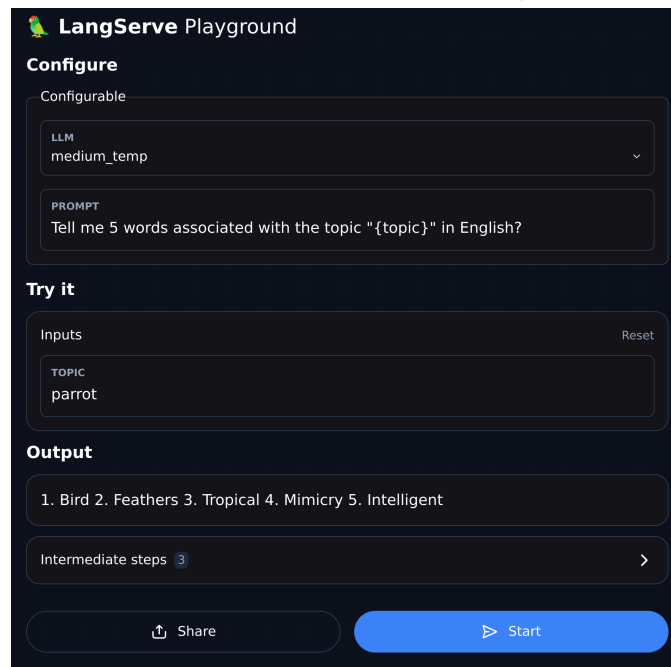
- `POST /my_runnable/invoke` - invoke the runnable on a single input
- `POST /my_runnable/batch` - invoke the runnable on a batch of inputs
- `POST /my_runnable/stream` - invoke on a single input and stream the output

- `POST /my_runnable/stream_log` - invoke on a single input and stream the output, including output of intermediate steps as it's generated
- `POST /my_runnable/astream_events` - invoke on a single input and stream events as they are generated, including from intermediate steps.
- `GET /my_runnable/input_schema` - json schema for input to the runnable
- `GET /my_runnable/output_schema` - json schema for output of the runnable
- `GET /my_runnable/config_schema` - json schema for config of the runnable

These endpoints match the [LangChain Expression Language interface](#) -- please reference this documentation for more details.

Playground

You can find a playground page for your runnable at `/my_runnable/playground/`. This exposes a simple UI to [configure](#) and invoke your runnable with streaming output and intermediate steps.



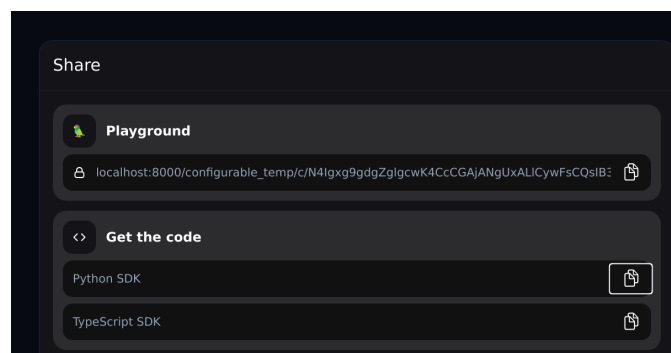
The image shows the LangServe Playground interface. At the top, it says 'LangServe Playground'. Below that is a 'Configure' section with a 'Configurable' dropdown set to 'LLM medium_temp' and a 'PROMPT' text area containing 'Tell me 5 words associated with the topic "{topic}" in English?'. The 'Try it' section has an 'Inputs' text area with 'TOPIC parrot' and a 'Reset' button. The 'Output' section shows the result: '1. Bird 2. Feathers 3. Tropical 4. Mimicry 5. Intelligent'. Below the output is a button for 'Intermediate steps' with a right arrow. At the bottom are 'Share' and 'Start' buttons.

Widgets

The playground supports **widgets** and can be used to test your runnable with different inputs. See the **widgets** section below for more details.

Sharing

In addition, for configurable runnables, the playground will allow you to configure the runnable and share a link with the configuration:



The image shows a 'Share' dialog box. It has a 'Playground' section with a link to 'localhost:8000/configurable_temp/c/N4l9xg9gdgZglgcwK4CcCGAJANGUxALI/CywFsCQsiB:'. Below that is a 'Get the code' section with options for 'Python SDK' and 'TypeScript SDK', each with a copy icon.

Legacy Chains

LangServe works with both Runnables (constructed via [LangChain Expression Language](#)) and legacy chains (inheriting from `Chain`). However, some of the input schemas for legacy chains may be incomplete/incorrect, leading to errors. This can be fixed by updating the `input_schema` property of those chains in LangChain. If you encounter any errors, please open an issue on THIS repo, and we will work to address it.

Deployment

Deploy to AWS

You can deploy to AWS using the [AWS Copilot CLI](#)

```
copilot init --app [application-name] --name  
[service-name] --type 'Load Balanced Web  
Service' --dockerfile './Dockerfile' --deploy
```

Click [here](#) to learn more.

Deploy to Azure

You can deploy to Azure using Azure Container Apps (Serverless):

```
az containerapp up --name [container-app-name] --source . --resource-group [resource-group-name] --environment [environment-name] --ingress external --target-port 8001 --env-vars=OPENAI_API_KEY=your_key
```

You can find more info [here](#)

Deploy to GCP

You can deploy to GCP Cloud Run using the following command:

```
gcloud run deploy [your-service-name] --source . --port 8001 --allow-unauthenticated --region us-central1 --set-env-vars=OPENAI_API_KEY=your_key
```

Community Contributed

Deploy to Railway

[Example Railway Repo](#)



Pydantic

LangServe provides support for Pydantic 2 with some limitations.

1. OpenAPI docs will not be generated for `invoke/batch/stream/stream_log` when using Pydantic V2. Fast API does not support [mixing pydantic v1 and v2 namespaces].
2. LangChain uses the v1 namespace in Pydantic v2. Please read the [following guidelines to ensure compatibility with LangChain](#)

Except for these limitations, we expect the API endpoints, the playground and any other features to work as expected.

Advanced

Handling Authentication

If you need to add authentication to your server, please read Fast API's documentation about [dependencies](#) and [security](#).

The below examples show how to wire up authentication logic LangServe endpoints using FastAPI primitives.

You are responsible for providing the actual authentication logic, the users table etc.

If you're not sure what you're doing, you could try using an existing solution [Auth0](#).

Using `add_routes`

If you're using `add_routes`, see examples [here](#).

Description	Links
Auth with <code>add_routes</code> : Simple authentication that can be applied across all endpoints associated with app. (Not useful on its own for implementing per user logic.)	server
Auth with <code>add_routes</code> : Simple authentication mechanism based on path dependencies. (No useful on its own for implementing per user logic.)	server
Auth with <code>add_routes</code> : Implement per user logic and auth for endpoints that use per request config modifier. (Note: At the moment, does not integrate with OpenAPI docs.)	server , client

Alternatively, you can use FastAPI's [middleware](#).

Using global dependencies and path dependencies has the advantage that auth will be properly supported in the OpenAPI docs page, but these are not sufficient for implement per user logic (e.g., making an application that can search only within user owned documents).

If you need to implement per user logic, you can use the `per_req_config_modifier` or `APIHandler` (below) to implement this logic.

Per User

If you need authorization or logic that is user dependent, specify `per_req_config_modifier` when using `add_routes`. Use a callable receives the raw `Request` object and can extract relevant information from it for authentication and authorization purposes.

Using APIHandler

If you feel comfortable with FastAPI and python, you can use LangServe's [APIHandler](#).

Description	Links
Auth with <code>APIHandler</code> : Implement per user logic and auth that shows how to search only within user owned documents.	server, client
APIHandler Shows how to use <code>APIHandler</code> instead of <code>add_routes</code> . This provides more flexibility for developers to define endpoints. Works well with all FastAPI patterns, but takes a bit more effort.	server, client

It's a bit more work, but gives you complete control over the endpoint definitions, so you can do whatever custom logic you need for auth.

Files

LLM applications often deal with files. There are different architectures that can be made to implement file processing; at a high level:

1. The file may be uploaded to the server via a dedicated endpoint and processed using a separate endpoint
2. The file may be uploaded by either value (bytes of file) or reference (e.g., s3 url to file content)

3. The processing endpoint may be blocking or non-blocking
4. If significant processing is required, the processing may be offloaded to a dedicated process pool

You should determine what is the appropriate architecture for your application.

Currently, to upload files by value to a runnable, use base64 encoding for the file (`multipart/form-data` is not supported yet).

Here's an **example** that shows how to use base64 encoding to send a file to a remote runnable.

Remember, you can always upload files by reference (e.g., s3 url) or upload them as multipart/form-data to a dedicated endpoint.

Custom Input and Output Types

Input and Output types are defined on all runnables.

You can access them via the `input_schema` and `output_schema` properties.

`LangServe` uses these types for validation and documentation.

If you want to override the default inferred types, you can use the `with_types` method.

Here's a toy example to illustrate the idea:

```
from typing import Any

from fastapi import FastAPI
from langchain.schema.runnable import RunnableLambda

app = FastAPI()

def func(x: Any) -> int:
    """Mistyped function that should accept
    an int but accepts anything."""
    return x + 1

runnable = RunnableLambda(func).with_types(
    input_schema=int,
)

add_routes(app, runnable)
```

Custom User Types

Inherit from `CustomUserType` if you want the data to de-serialize into a pydantic model rather than the equivalent dict representation.

At the moment, this type only works *server* side and is used to specify desired *decoding* behavior. If inheriting from this type the server will keep the decoded type as a pydantic model instead of converting it into a dict.

```
from fastapi import FastAPI
from langchain.schema.runnable import
RunnableLambda

from langserve import add_routes
from langserve.schema import CustomUserType

app = FastAPI()

class Foo(CustomUserType):
    bar: int

def func(foo: Foo) -> int:
    """Sample function that expects a Foo
    type which is a pydantic model"""
    assert isinstance(foo, Foo)
    return foo.bar
```



```
# Note that the input and output type are
# automatically inferred!
# You do not need to specify them.
# runnable = RunnableLambda(func).with_types(
# <-- Not needed in this case
#     input_schema=Foo,
#     output_schema=int,
#
add_routes(app, RunnableLambda(func),
path="/foo")
```

Playground Widgets

The playground allows you to define custom widgets for your runnable from the backend.

Here are a few examples:

Description	Links
Widgets Different widgets that can be used with playground (file upload and chat)	server , client
Widgets File upload widget used for LangServe playground.	server , client

Schema

- A widget is specified at the field level and shipped as part of the JSON schema of the input type
- A widget must contain a key called `type` with the value being one of a well known list of widgets
- Other widget keys will be associated with values that describe paths in a JSON object

```
type JsonPath = number | string | (number | string)[];
type NameSpacedPath = { title: string; path: JsonPath }; // Using title to mimick json schema, but can use namespace
type OneOfPath = { oneOf: JsonPath[] };

type Widget = {
  type: string // Some well known type (e.g., base64file, chat etc.)
  [key: string]: JsonPath | NameSpacedPath | OneOfPath;
};
```

Available Widgets

There are only two widgets that the user can specify manually right now:

1. File Upload Widget

2. Chat History Widget

See below more information about these widgets.

All other widgets on the playground UI are created and managed automatically by the UI based on the config schema of the Runnable. When you create Configurable Runnables, the playground should create appropriate widgets for you to control the behavior.

File Upload Widget

Allows creation of a file upload input in the UI playground for files that are uploaded as base64 encoded strings. Here's the full [example](#).

Snippet:

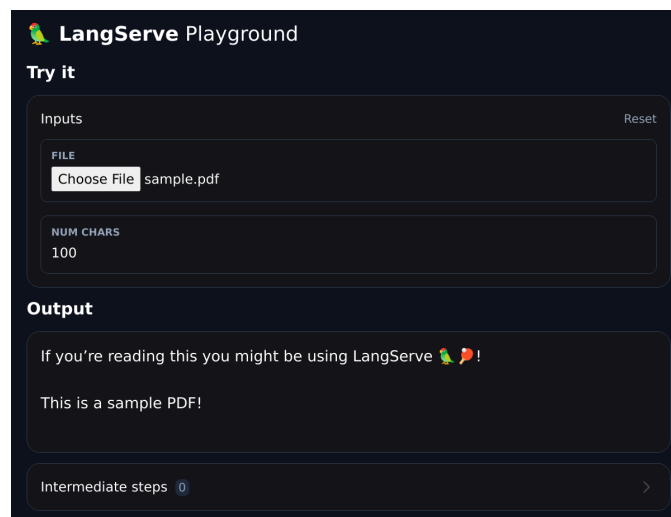
```
try:
    from pydantic.v1 import Field
except ImportError:
    from pydantic import Field

from langserve import CustomUserType

# ATTENTION: Inherit from CustomUserType
# instead of BaseModel otherwise
# the server will decode it into a
dict instead of a pydantic model.
```

```
class FileProcessingRequest(CustomUserType):  
    """Request including a base64 encoded  
    file."""  
  
    # The extra field is used to specify a  
    widget for the playground UI.  
    file: str = Field(..., extra={"widget":  
{"type": "base64file"}})  
    num_chars: int = 100
```

Example widget:



The screenshot shows the LangServe Playground interface. At the top, it says "LangServe Playground". Below that is a "Try it" section. Under "Inputs", there are two fields: "FILE" with a "Choose File" button and "sample.pdf" text, and "NUM CHARS" with a value of "100". A "Reset" button is in the top right of the inputs section. Below the inputs is an "Output" section. It contains two lines of text: "If you're reading this you might be using LangServe 🦜🔗!" and "This is a sample PDF!". At the bottom of the output section is a bar for "Intermediate steps" with a count of "0" and a right arrow.

Chat Widget

Look at [widget example](#).

To define a chat widget, make sure that you pass "type": "chat".

- "input" is JSONPath to the field in the *Request* that has the new input message.
- "output" is JSONPath to the field in the *Response* that has new output message(s).
- Don't specify these fields if the entire input or output should be used as they are (e.g., if the output is a list of chat messages.)

Here's a snippet:

```
class ChatHistory(CustomUserType):
    chat_history: List[Tuple[str, str]] = Field(
        ...,
        examples=[["human input", "ai
response"]],
        extra={"widget": {"type": "chat",
"input": "question", "output": "answer"}},
    )
    question: str

def _format_to_messages(input: ChatHistory) ->
List[BaseMessage]:
    """Format the input to a list of messages."""
    history = input.chat_history
    user_input = input.question

    messages = []
```

```
for human, ai in history:

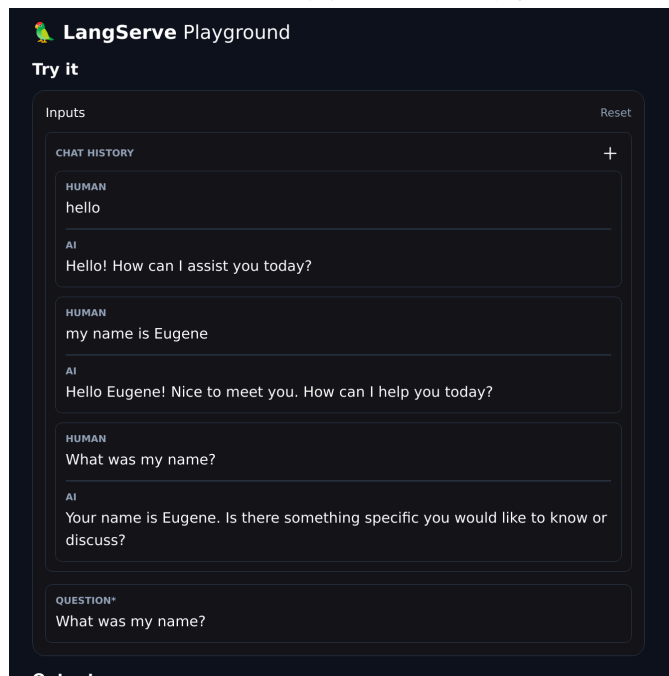
messages.append(HumanMessage(content=human))
    messages.append(AIMessage(content=ai))

messages.append(HumanMessage(content=user_input)
return messages

model = ChatOpenAI()
chat_model = RunnableParallel({"answer":
(RunnableLambda(_format_to_messages) | model)})
add_routes(
    app,

chat_model.with_types(input_type=ChatHistory),
    config_keys=["configurable"],
    path="/chat",
)
```

Example widget:



Enabling / Disabling Endpoints (LangServe >=0.0.33)

You can enable / disable which endpoints are exposed. Use `enabled_endpoints` if you want to make sure to never get a new endpoint when upgrading langserve to a newer version.

Enable: The code below will only enable `invoke`, `batch` and the corresponding `config_hash` endpoint variants.

```
add_routes(app, chain, enabled_endpoints=[
    "invoke", "batch", "config_hashes"])
```

Disable: The code below will disable the playground for the chain

```
add_routes(app, chain, disabled_endpoints=  
["playground"])
```