Langri-La

> NOTE: This package is under rebuilding including significant changes on the interface.

Langrila is an open-source third-party python package allows you to develop type-safe multi-agent in an easy way. This package is just personal project.

Motivation

Widely used existing agent framework is all awesome, but I feel like:

- Often highly abstracted, making their behavior unclear at times and it causes some limitations as well.
- Code readability has some issues derived from many decorators and special operator when we are developing agent using that framework.
- Handling an agent's state or context is a little annoying. Typically it requires special arguments in functions and the state is updated hidden by tool. This might causes confusion on treaceability of the state. Ideally, functions implemented in regular development should be usable as-is with an agent, and state scope is explained by dependencies of agents, I think.
- The arguments, specifications, and available models of providers are frequently updated. To follow that is a little cumbersome.

To address these issues, I breathed new life into langrila, which has the following features:

- Deliberately avoiding over-abstracting the module.:
  - Wrapper classes of the provider sdk are minimally designed, avoiding unnecessary unification of arguments and eliminating processing dependent on model names. Langrila gives you high-level interface to use LLM while almost all the provider original parameters can be accepted as it is.
- No need for special argument to inject context into tools:
  - There is no requirement to set instances of special classes as arguments when injecting context into tools.
- Support for sub-agents as tools:
  - In addition to passing tools to agents, sub-agents can be passed directly to the agent. Sub-agents are dynamically converted into tools internally and controlled by the parent agent, enabling easy construction of multi-agent systems. This feature brings to you intuitive coding and readability of the multi-agent.
- Unified message-response model independent of provider APIs:
  - Langrila defines a standardized message-response model that allows multi-agent systems to be built across different providers.
- Serializable conversation history:
  - The standardized message-response model can be serialized to JSON and easily stored not only in memory but also in formats like JSON, Pickle, Azure Cosmos DB, and AWS S3.
- Type-safe structured output:
  - Inspired by PydanticAI. Args are validated when a tool is invoked by the pydantic schema validator.
- Multi-modal I/O:
  - Whatever the provider supports like image/video/pdf/audio/uri input, image/audio generation, embed texts and so on.
    - I will be rolling out support progressively.
- Others:
  - Automatic retry when error raised.
  - Customizable internal prompts.
  - Usage gathering for all sub-agents.
  - All we have to do to support new provider is to implement a single class in many cases.

Prerequisites

If necessary, set environment variables to use OpenAI API, Azure OpenAI Service, Gemini API, and Claude API; if using VertexAI or Amazon Bedrock, check each platform's user guide and authenticate in advance VertexAI and Amazon Bedrock.

Supported llm providers

- OpenAI
- Azure OpenAI
- Gemini on Google AI Studio
- Gemini on VertexAI
- Claude on Anthropic

- Claude on Amazon Bedrock
- Claude on VertexAI (not tested)

Breaking changes

Significant breaking changes will be introduced to become agent framework. It's more like a rebuild than just an update. Please be careful to update from the previous version.

Basic usage

Coming soon.

Dependencies

must

```
python = ">=3.10,<3.13"
matplotlib = "^3.8.0"
plotly = "^5.17.0"
numpy = "^1.26.1"
pandas = "^2.1.1"
scipy = "^1.11.3"
scikit-learn = "^1.3.2"
pydantic = "^2.10.0"
griffe = "^1.5.1"
loguru = "^0.7.3"
```

as needed

Langrila has various extra installation options. See the following installation section and pyproject.toml.

Installation

See extra dependencies section in pyproject.toml for more detail installation options.

For user

pip

```
# For OpenAI
pip install langrila[openai]

# For Gemini
pip install langrila[gemini]

# For Claude
pip install langrila[claude]

# For multiple providers
pip install langrila[openai,gemini,claude]

# With dependencies to handle specific data. Here is an example using gemini
pip install langrila[gemini,audio,video,pdf]

# With dependencies for specific platform. Here is an example using gemini on VertexAI
pip install langrila[gemini,vertexai]

# With dependencies for specific vectorDB. Here is an example using Qdrant
pip install langrila[openai,qdrant]
```

poetry

```
# For OpenAI
poetry add langrila --extras openai

# For Gemini
poetry add langrila --extras gemini

# For Claude
poetry add langrila --extras claude

# For multiple providers
poetry add langrila --extras "openai gemini claude"

# With dependencies to handle specific data. Here is an example using gemini
poetry add langrila --extras "gemini audio video pdf"

# With dependencies for specific platform. Here is an example using gemini on VertexAI
```

```
poetry add langrila --extras "gemini vertexai"

# With dependencies for specific vectorDB. Here is an example using Qdrant
poetry add langrila --extras "openai qdrant"
```

For developer

clone

```
git clone git@github.com:taikinman/langrila.git
```

pip

```
cd langrila

pip install -e .{extra packages}
```

poetry

```
# For OpenAI
poetry add --editable /path/to/langrila/ --extras "{extra packages}"
```

Multi-agent example

In langrila, we can build orchestrator-typed multi-agent, not graph-based multi-agent. The orchestrator routes the execution of tools to individual agents, aggregates the results, and outputs the final answer.

Here is a fragment of the example code with dummy tools.

```python
from langrila import Agent, InMemoryConversationMemory
from langrila.anthropic import AnthropicClient
from langrila.google import GoogleClient
from langrila.openai import OpenAIClient
from enum import Enum
from pydantic import BaseModel, Field


##################################
# Client modules
##################################

# For OpenAI
openai_client = OpenAIClient(api_key_env_name="OPENAI_API_KEY")

# For Gemini on Google AI Studio
google_client = GoogleClient(api_key_env_name="GEMINI_API_KEY")

# For Claude of Anthropic
anthropic_client = AnthropicClient(api_key_env_name="ANTHROPIC_API_KEY")


##################################
# Tool definition
##################################

def power_disco_ball(power: bool) -> bool:
    """
    Powers the spinning dissco ball.

    Parameters
    ----------
    power : bool
        Whether to power the disco ball or not.

    Returns
    ----------
    bool
        Whether the disco ball is spinning or not.
    """
    return f"Disco ball is {'spinning!' if power else 'stopped.'}"

...

##################################
# Definition of response schema
##################################

class DiscoBallSchema(BaseModel):
    power: bool = Field(..., description="Whether to power the disco ball.")
    spinning: bool = Field(..., description="Whether the disco ball is spinning.")


class MusicGenre(str, Enum):
    rock = "rock"
```

```
    pop = "pop"
    jazz = "jazz"
    classical = "classical"
    hip_hop = "hip-hop"


class MusicSchema(BaseModel):
    genre: MusicGenre = Field(
        ...,
        description="The genre of music to play.",
    )
    bpm: int = Field(
        ...,
        description="The BPM of the music.",
        ge=60,
        le=180,
    )
    volume: float = Field(
        ...,
        description="The volume level to set the music to.",
        ge=0,
        le=1,
    )


class LightsSchema(BaseModel):
    brightness: float = Field(
        ...,
        description="The brightness level to set the lights to.",
        ge=0,
        le=1,
    )


class ResponseSchema(BaseModel):
    disco_ball: DiscoBallSchema = Field(None, description="The disco ball settings.")
    music: MusicSchema = Field(None, description="The music settings.")
    lights: LightsSchema = Field(None, description="The lights settings.")


##################################
# Orchestration
##################################

lights_agent = Agent(
    client=openai_client,
    model="gpt-4o-mini-2024-07-18",
    temperature=0.0,
    tools=[dim_lights, brighten_lights, turn_light_on],
)

disco_ball_agent = Agent(
    client=openai_client,
    model="gpt-4o-mini-2024-07-18",
    temperature=0.0,
    tools=[power_disco_ball, stop_disco_ball],
    max_tokens=500,
)

music_power_agent = Agent(
    client=openai_client,
    model="gpt-4o-mini-2024-07-18",
    temperature=0.0,
    tools=[start_music],
)

music_control_agent = Agent(
    client=openai_client,
    model="gpt-4o-mini-2024-07-18",
    temperature=0.0,
    tools=[change_music, adjust_volume, change_bpm],
)

# Orchestrator as a sub-agent
music_agent_orchestrator = Agent(
    client=anthropic_client,
    model="claude-3-5-sonnet-20240620",
    temperature=0.0,
    subagents=[music_power_agent, music_control_agent],
    max_tokens=500,
)

# Master orchestrator
master_orchestrator = Agent(
    client=google_client,
    model="gemini-2.0-flash-exp",
    temperature=0.0,
    subagents=[lights_agent, disco_ball_agent, music_agent_orchestrator],
    response_schema_as_tool=ResponseSchema,
    conversation_memory=InMemoryConversationMemory(),
)

##################################
# Invoke agent
```

```
##################################

prompt = "Turn this place into a party mood."

# synchronous generation
response = master_orchestrator.generate_text(prompt=prompt)

# asynchronous generation
# response = await master_orchestrator.generate_text_async(prompt=prompt)

##################################
# Result
##################################

ResponseSchema.model_validate_json(response.contents[0].text)


# >>> ResponseSchema(disco_ball=DiscoBallSchema(power=True, spinning=True), music=MusicSchema(genre=<MusicGenre.pop: 'pop'>, bpr

##################################
# Usage
##################################

list(response.usage.items())

# >>> [('music_power_agent',
# >>>   Usage(model_name='gpt-4o-mini-2024-07-18', prompt_tokens=123, output_tokens=23, raw=None)),
# >>>  ('music_agent_orchestrator',
# >>>   Usage(model_name='claude-3-5-sonnet-20240620', prompt_tokens=2510, output_tokens=368, raw=None)),
# >>>  ('music_control_agent',
# >>>   Usage(model_name='gpt-4o-mini-2024-07-18', prompt_tokens=541, output_tokens=83, raw=None)),
# >>>  ('lights_agent',
# >>>   Usage(model_name='gpt-4o-mini-2024-07-18', prompt_tokens=345, output_tokens=60, raw=None)),
# >>>  ('root',
# >>>   Usage(model_name='gemini-2.0-flash-exp', prompt_tokens=3273, output_tokens=82, raw=None)),
# >>>  ('disco_ball_agent',
# >>>   Usage(model_name='gpt-4o-mini-2024-07-18', prompt_tokens=211, output_tokens=31, raw=None))]
```

Roadmap

- [ ] Error handling more
- [ ] Preparing example notebooks
- [ ] Linting and refactor
- [ ] Supporting Huggingface
- [ ] Aim integration