📚Universal LLM Wrapper Documentation

📌Overview

This document outlines the architecture and implementation of a **Universal LLM Wrapper System**, designed to interact with multiple AI model providers (OpenAI, Anthropic, Groq, DeepSeek, Fireworks, Together, Ollama, etc.) through a **centralized, extensible, and modular interface**.

The core goal is to:

- **Abstract provider-specific logic** from the application
- **Standardize request and response handling**
- **Simplify switching between models**
- **Support retry, timeout, and tool-use control**
- **Provide a universal CLI to test models**
- Enable integration with LangChain

📦Folder Structure

```
llm_wrapper/
├── providers_config.py         # Central configuration for all providers
├── llm_client_factory.py       # Provider dispatcher (returns correct client)
├── llm_cli.py                  # CLI tool for model testing
├── llm_clients/
│   ├── __init__.py
│   ├── base.py                 # BaseLLMClient (abstract class)
│   ├── openai_client.py
│   ├── anthropic_client.py
│   ├── groq_client.py
│   ├── deepseek_client.py
│   └── ...
```

Core Concepts

🔗Why Use a Wrapper?

- Model APIs change, wrappers hide the volatility.
- Easily add/remove providers (like plugins).
- Central config (no need to touch app code).
- CLI/debug tools and UIs become simpler.

- Retry/timeout/config is unified.
- Abstracts LangChain or non-LangChain usage.

🧰providers_config.py

Central registry for all LLMs, their models, capabilities, timeouts, retry logic, etc.

🛠️Structure:

```
PROVIDERS = {
    "openai": {
        "name": "OpenAI",
        "base_url": "https://api.openai.com/v1",
        "models": {
            "gpt-4o": {
                "capabilities": ["chat", "stream", "json_mode",
"function_calling"],
                "default": True
            },
            "gpt-4o-mini": {
                "capabilities": ["chat", "stream", "json_mode"]
            }
        },
        "default_model": "gpt-4o-mini",
        "api_key_env": "OPENAI_API_KEY",
        "timeout": 30,
        "max_retries": 3
    },
    ...
}
```

🔗Capabilities Supported:

Capability	Meaning
`chat`	Supports chat completion APIs
`stream`	Supports streaming output
`json_mode`	Supports structured JSON mode generation
`function_calling`	Native function call support (OpenAI etc.)
`tool_use`	Advanced tool use support (Claude, Gemini)
`code`	Optimized for code (DeepSeek Coder, Codestral, etc.)

🔐 API Key Handling

Each provider entry contains an `api_key_env` key which specifies the environment variable used to inject the key dynamically. This avoids hardcoding secrets in the codebase.

📑 llm_client_factory.py

This is the **dispatcher** that returns the correct client based on provider name.

```
def get_llm_client(provider_name, model=None, config=None):
    if provider_name == "openai":
        return OpenAIClient(model, config)
    elif provider_name == "anthropic":
        return AnthropicClient(model, config)
    ...
    else:
        raise ValueError(f"Unknown provider: {provider_name}")
```

🧱 llm_clients/base.py

Defines a **standard interface** for all providers.

```
class BaseLLMClient:
    def __init__(self, model, config):
        self.model = model
        self.config = config

    def chat(self, messages, temperature=0.7):
        raise NotImplementedError
```

Retry and Timeout

Use `tenacity` for retry logic:

```
from tenacity import retry, stop_after_attempt, wait_fixed

@retry(stop=stop_after_attempt(3), wait=wait_fixed(2))
```

```
def chat(...):
    ...
```

The retry count and timeout duration should come from `config['max_retries']` and `config['timeout']`.

🔦Unified Response Format

Every `.chat()` method must return this schema:

```
{
    "provider": "openai",
    "model": "gpt-4o",
    "message": "string response here",
    "usage": {
        "prompt_tokens": 45,
        "completion_tokens": 130,
        "total_tokens": 175
    },
    "raw": {...}  # Optional raw API response
}
```

This makes it easy to log, display in UI, debug, or do evals across providers.

🔗LangChain Compatibility

You can optionally power each client using LangChain wrappers internally:

```
from langchain.chat_models import ChatOpenAI

class OpenAIClient(BaseLLMClient):
    def chat(self, messages, temperature=0.7):
        llm = ChatOpenAI(...)
        response = llm.invoke(messages)
        return {
            "provider": "openai",
            "model": self.model,
            "message": response.content,
            "raw": response
        }
```

If the model is not available in LangChain (like Ollama or DeepSeek), you can just call HTTP directly.

🔧CLI Tool: llm_cli.py

A test utility to call any provider/model from the command line.

💻Usage:

```
$ python llm_cli.py --provider openai --model gpt-4o --prompt "Tell me a joke"
```

💡Code:

```
import argparse
from providers_config import PROVIDERS
from llm_client_factory import get_llm_client

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--provider", required=True)
    parser.add_argument("--model")
    parser.add_argument("--prompt", required=True)
    args = parser.parse_args()

    config = PROVIDERS[args.provider]
    model = args.model or config["default_model"]

    client = get_llm_client(args.provider, model, config)
    response = client.chat([{"role": "user", "content": args.prompt}])
    print(response["message"])

if __name__ == "__main__":
    main()
```

🧪Future-Proofing Ideas

Feature	Status
Eval multiple models	🟡Planned
Auto fallback	🟡Planned
Streaming via WebSocket	🟡Optional

Feature	Status
LangGraph Integration	🟡Optional
Model pricing estimation	🟡Optional
Token cost calculator	🟡Optional
Usage caching	🟡Optional
Unified telemetry/logging	🟡Optional

🔗Conclusion

This wrapper gives you a powerful and flexible base to:

- Standardize model usage
- Scale to new providers quickly
- Create internal tooling (CLI, evals, dashboards)
- Swap LangChain in or out
- Keep all model logic in one place

It's modular, robust, and dev-friendly. Use this as your LLM DevOps foundation ✔️