ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Thực tập ngoài trường - CO3335

Báo cáo

AUTOFILL SERVICE

Giảng viên hướng dẫn: Trương Quỳnh Chi

Sinh viên thực hiện: 2111762 - Phạm Võ Quang Minh

TP. Hồ Chí Minh, 12/2024

Mục lục

Danh mục hình ảnh

Danh mục bảng biểu

1 Giới thiệu doanh nghiệp

Công ty TNHH Apollogix chuyên cung cấp các giải pháp phần mềm quản lý chuyên dụng trong lĩnh vực giao thông vận tải Thông tin công ty:

- Tên công ty: Công ty TNHH Apollogix
- Tên quốc tế: APOLLOGIX COMPANY LIMITED
- Tên viết tắt: APOLLOGIX CO LTD
- Trụ sở chính: 39 Đường B4, phường An Lợi Đông, Quận 2, thành phố Thủ Đức.
- Văn phòng làm việc: 39 Đường B4, phường An Lợi Đông, Quận 2, thành phố Thủ Đức.
- Điện thoại: 0796513044 (Ms. Huyền).
- Email: contact@apollogix.com.
- Weblink: apollogix.com.
- Logo nhận diện:

Hình 1: Logo doanh nghiệp Apollogix

2 Nội dung thực tập

2.1 Giới thiệu dề tài

Trong hệ thống TMS có một chức năng gọi là Autofill, chức năng này có input đầu vào là file pdf booking, output mong đợi là đọc từ file pdf này ra cấu trúc json của dữ liệu Transport Job đang có để sau đó sẽ hiển thị data này trên giao diện form Transport Job cho end user xem trước khi lưu lại mà không cần phải nhập. Ngoài ra dịch vụ có thể được sử dụng để tạo đơn hàng có trạng thái chờ xác nhận từ email của khách hàng hoặc trên customer portal sau này.

File PDF Booking này khách hàng có được từ bên các hãng tàu hoặc các dịch vụ bên thứ 3 cung cấp, có chứa các thông tin về lịch cảng tàu, thông tin container cần vận chuyển, v.v

Sinh viên thực hiện: Phạm Võ Quang Minh

Nhân sự hỗ trợ kỹ thuật: Anh Thi

Nhân sự hỗ trợ nghiệp vụ: Anh Phúc, Chị Nguyệt, và các nhân sự bên vận hành

2.2 Phân tích chi tiết đề tài

Đề tài có thể được chia ra làm hai bước chính

2.2.1 Biến dữ liệu chưa có cấu trúc về dữ liệu có cấu trúc

Phát triển một hàm có khả năng biến dữ liệu từ định dạng dữ liệu không cấu trúc (chuỗi/hình ảnh/pdf) sang dạng json (dữ liệu có cấu trúc).

Đối với bước này, ta có thể biến yêu cầu đề bài thành hai hướng giải quyết bài toán khác nhau:

- Named Entity Recognition (NER) - tạm dịch Nhận dạng Thực thể để xác định các token trên dữ liệu đầu vào thuộc miền dữ liệu nào trong dạng json mà công ty yêu cầu.
- Text Extraction - tạm dịch: trích xuất văn bản. Bằng cách viết prompt thông minh cho mô hình ngôn ngữ, ta có thể truyền vào định nghĩa các miền giá trị cần trích xuất trong văn bản cho mô hình có khả năng hiểu ngôn ngữ loài người và trả về kết quả dữ liệu có cấu trúc như mong đợi.

2.2.2 Dịch vụ API

Phát triển dịch vụ (API service) để nhận đầu vào dữ liệu không cấu trúc từ nơi gọi (hệ thống TMS) và trả về dữ liệu có cấu trúc.

2.3 Phân tích các hướng tiếp cận

2.3.1 Biến dữ liệu chưa có cấu trúc về dữ liệu có cấu trúc

Đề xuất hai phương pháp có thể sử dụng:

1. Huấn luyện mô hình:
   a. Xây dựng mô hình: Đưa bài toán về bài toán Nhận dạng Thực thể **NER** (Named Entity Recognition) và sử dụng mô hình học máy kinh điển để xử lý bài toán
   b. Fine-tuning (transfer learning): Huấn luyện mô hình ngôn ngữ để mô hình luôn sinh ra được câu trả lời có cấu trúc từ đầu vào
2. Sử dụng mô hình đã có sẵn (**RAG**): Sử dụng mô hình ngôn ngữ học sâu (deep learning model) như GPT-4o để trích xuất thông tin. Trong đó, người dùng yêu cầu mô hình ngôn ngữ trả về dạng chuỗi (string) theo format của của 1 dạng dữ liệu có cấu trúc như json.
   a. Sử dụng mô hình ngôn ngữ giao tiếp (chat model): chuyển dạng pdf/hình ảnh/... về dạng chuỗi bằng công nghệ OCR.
   b. Sử dụng mô hình đa thể thức (multi modal model): truyền thẳng dạng pdf/hình ảnh/... mà không cần có bước OCR.

2.3.1.a Huấn luyện mô hình NER hoặc fine-tune mô hình có sẵn:

Để hiện thực hóa giải pháp NER, cần thực hiện các bước sau:

1. Thu thập dữ liệu: Thu thập tập dữ liệu có chứa các văn bản đã được gắn nhãn thực thể. Việc gắn nhãn có thể thực hiện thủ công hoặc bằng các công cụ tự động hóa. Đảm bảo dữ liệu đa dạng và đại diện tốt cho bài toán.
2. Tiền xử lý dữ liệu: Làm sạch dữ liệu, loại bỏ các ký tự không cần thiết. Chuẩn hóa văn bản (ví dụ: chuyển chữ hoa thành chữ thường, xóa khoảng trắng thừa). Tách văn bản thành câu hoặc token (từ hoặc cụm từ).
3. Trích xuất đặc trưng: Sử dụng các kỹ thuật như gán nhãn từ loại (POS tagging), tạo vector từ biểu diễn (word embeddings), và khai thác ngữ cảnh xung quanh từ. Chọn các đặc trưng phù hợp với mô hình NER cụ thể.
4. Huấn luyện mô hình: Sử dụng mô hình học máy (như CRF, SVM) hoặc học sâu (như BiLSTM-CRF, Transformer-based models như BERT) để huấn luyện trên tập dữ liệu gắn nhãn. Điều chỉnh các siêu tham số (hyperparameters) để tối ưu hóa hiệu suất.
5. Đánh giá mô hình: Đánh giá chất lượng mô hình qua các chỉ số như Precision, Recall, và F1 Score. Phân tích các trường hợp mô hình nhận dạng sai để cải thiện.
6. Tinh chỉnh mô hình: Tăng cường dữ liệu huấn luyện hoặc sử dụng kỹ thuật như tăng cường dữ liệu (data augmentation). Áp dụng các phương pháp như học chuyển giao (transfer learning) để cải thiện độ chính xác.
7. Suy luận (Inference): Triển khai mô hình để xử lý các văn bản mới, gắn nhãn các thực thể trong văn bản.
8. Hậu xử lý: Liên kết thực thể với cơ sở tri thức hoặc các nguồn dữ liệu khác để làm phong phú thêm thông tin. Loại bỏ các thực thể không phù hợp hoặc hợp nhất các thực thể trùng lặp.

Ưu điểm:

- Khả năng tự động hóa cao, tiết kiệm thời gian so với các phương pháp thủ công.
- Có thể xử lý dữ liệu văn bản khổng lồ một cách hiệu quả.
- Kết hợp được nhiều loại đặc trưng để cải thiện độ chính xác.

Nhược điểm:

- Yêu cầu tập dữ liệu được gắn nhãn chất lượng cao, chi phí gắn nhãn lớn.

- Hiệu quả của mô hình phụ thuộc mạnh vào chất lượng dữ liệu và đặc trưng.
- Mô hình có thể gặp khó khăn khi xử lý ngôn ngữ không chính thức hoặc lỗi chính tả (tốn kém việc tiền xử lý sửa lỗi chính tả).

Hình 2: Minh họa công tác dãn nhãn dữ liệu

Tham khảo:

https://www.ibm.com/topics/named-entity-recognition https://medium.com/data-science-in-your-pocket/named-entity-recognition-ner-using-conditional-random-fields-in-nlp-3660df22e95c

Phân tích độ phù hợp của cách tiếp cận này với bài toán:
- Bộ dữ liệu cung cấp cho công việc huấn luyện quá ít ỏi (khoảng 40 mẫu).
- Việc dán nhãn phải làm thủ công, đòi hỏi người có chuyên môn hoặc quen với việc nhập mẫu (ví dụ: người bên nhánh vận hành của công ty). Nghĩa là chi phí dãn nhãn (thời gian, nhân lực) là quá lớn đối với 1 thực tập sinh trong 1 kỳ thực tập.

Vì vậy phương pháp này là không khả thi.

Tương tự, ta không cần phải đi sâu vào các bước thực hiện fine-tuning, vì nó cũng sẽ yêu cầu một lượng lớn dữ liệu huấn luyện và không khả thi trong bối cảnh hiện tại.

2.3.1.b Học tăng cường (RAG)

Vì những nhược điểm nêu ra ở hướng tiếp cận trên. Tôi đã lựa chọn phương pháp học tăng cường.

Học tăng cường là gì?

Là phương pháp dựa trên mô hình đã được huấn luyện sẵn (các mô hình ngôn ngữ lớn như GPT-4o, Llama3, Mistral,...).

Phần lớn các mô hình ngôn ngữ lớn chỉ có chức năng nhận vào một đầu vào là một chuỗi nhập của người dùng và trả về một chuỗi kết quả. Năng lực của mô hình ngôn ngữ lớn như vậy là còn quá hạn hẹp.

Một ví dụ cho học tăng cường là ChatGPT hay nhiều chat bot khác, đầu vào và kết quả của lần gọi trước sẽ được nối với đầu vào mới và tạo thành 1 chuỗi, ta sẽ tạo được một mô hình có khả năng trò chuyện với người dùng.

	Người dùng nhập	Đầu vào	Kết quả
1	Xin chào	Human: Xin chào	Xin chào, tôi có thể giúp gì được cho bạn?
2	Tôi muốn đặt câu hỏi về học tăng cường.	Human: Xin chào AI: Xin chào, tôi có thể giúp gì được cho bạn? Human: Tôi muốn đặt câu hỏi về học tăng cường.	Học tăng cường là phương pháp dựa trên mô hình đã được huấn luyện sẵn...

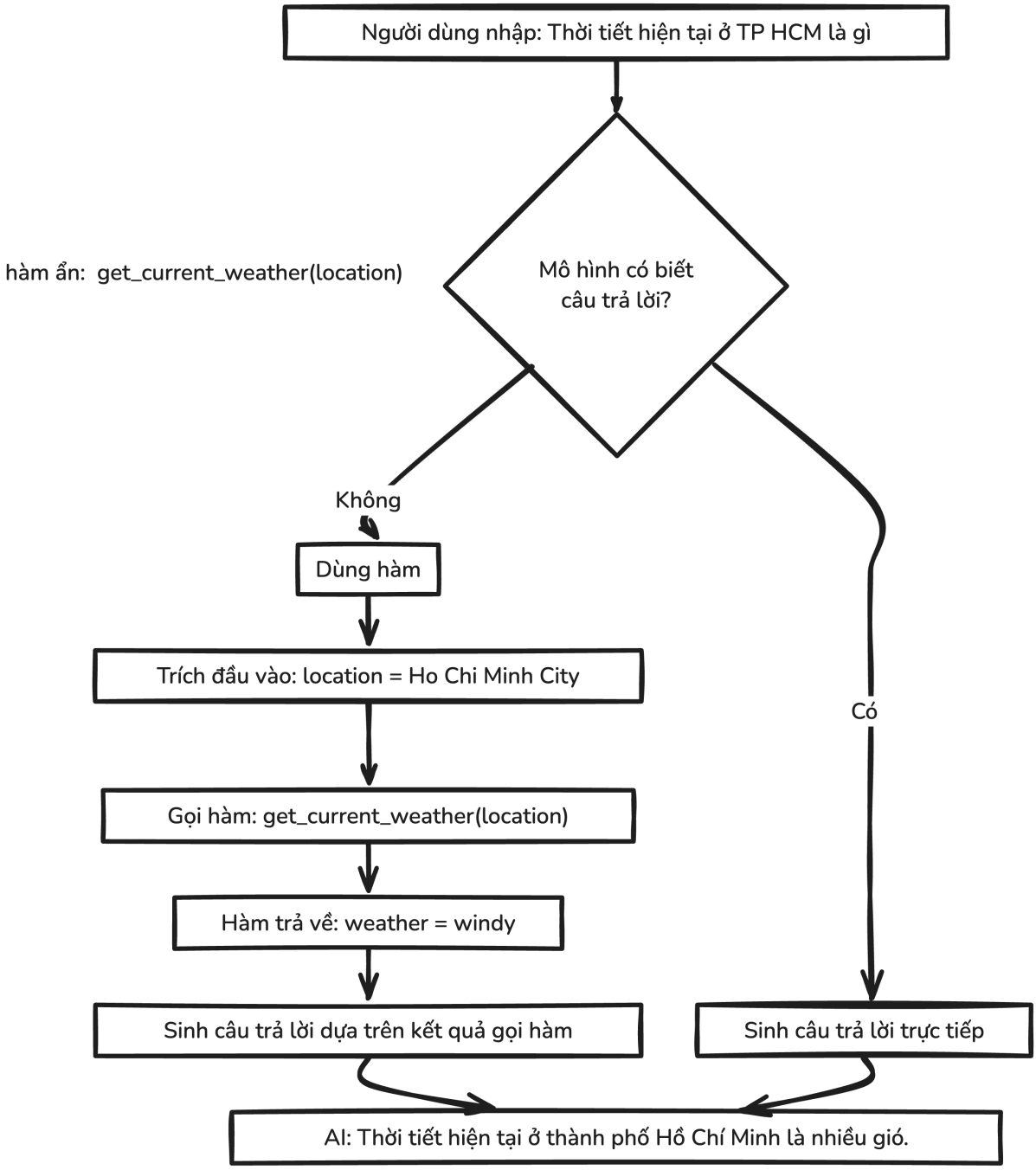
Bảng 1: Minh họa về cách chatbot hoạt động dựa trên mô hình ngôn ngữ

Chức năng gọi hàm/công cụ (function/tool calling) của mô hình ngôn ngữ lớn

Mô hình ngôn ngữ ở dạng thuần túy nhất không có khả năng lấy được thông tin ở thời gian thực. Vì vậy ta không thể nào đặt câu hỏi mong chờ kết quả của thời gian thực. Các nhà cung cấp mô hình hiện nay đã cung cấp thêm cho mô hình khả năng gọi hàm. Gọi hàm gồm các bước trừu tượng như sau:

1. Trích đầu vào cho một hàm
2. Gọi hàm
3. Sinh ra câu trả lời cho người dùng dựa trên đầu ra của hàm

(Theo dõi hình minh họa ở dưới)



Hình 3: Minh họa về cách gọi hàm của mô hình ngôn ngữ lớn

Vì sao chọn phương pháp này phù hợp hơn để giải quyết bài toán thay cho phương pháp huấn luyện mô hình

Nhược điểm:

- Độ chính xác hội tụ sớm sau một thời gian ngắn sử dụng: Vì chúng ta sẽ không huấn luyện mô hình, nên cũng không thể tăng độ chính xác theo thời gian khi kích thước bộ dữ liệu tăng lên.
- Phụ thuộc lớn vào khả năng viết prompt *hay*.

Ưu điểm:

- Không cần nghĩ đến việc huấn luyện: RAG sử dụng các mô hình ngôn ngữ lớn (LLMs) đã được huấn luyện trên kho dữ liệu khổng lồ, không yêu cầu một tập dữ liệu gắn nhãn cụ thể.

- Linh hoạt yêu cầu bài toán: Vì các LLM được huấn luyện dựa trên nhiều ngữ cảnh khác nhau nên có thể thích ứng được với nhiều loại yêu cầu.
- Dễ hiện thực: Không cần phải xây dựng chuỗi huấn luyện.
- Có thể dễ dàng khắc phục nhược điểm: Không thể tăng độ chính xác theo thời gian có thể được bỏ qua nhờ các kĩ thuật:
  - Việc viết lời gợi (prompting) hay hơn
  - Viết lời gợi kèm theo ví dụ (khoảng 2-5 ví dụ): Few-shot Prompting

2.4 Công nghệ sử dụng

2.4.1 RAG

Về cơ bản, để hiện thực được một kiến trúc học tăng cường, thì ta chỉ cần các bước sau (có thể lồng ghép các bước sau nhiều lần):

- Chuẩn bị lời gọi: soạn prompt, chuẩn hóa dữ liệu, đóng gói thành lời gọi (request) qua API của nhà cung cấp mô hình ngôn ngữ lớn (ví dụ: OpenAI, các dịch vụ như Groq, Huggingface,...).
- Xử lý kết quả trả về của API.

Tuy nhiên, vì mỗi nhà cung cấp có một cấu trúc riêng về API, giả sử số nhà cung cấp là n. Ta phải thực hiện quy trình trên với độ phức tạp O(n). Ta có thể sử dụng thư viện (thư viện) Langchain để giúp đưa bài toán này về O(1).

1. Langchain và Python
   (tác giả sử dụng Python phiên bản 3.12 và Langchain phiên bản 0.3) a. Giới thiệu về Langchain:
   Langchain là một framework giúp nhà phát triển có thể nhanh chóng tạo ra mô hình học tăng cường một cách nhanh chóng bằng cách chuẩn hóa quy trình tạo kiến trúc học tăng cường. Bên cạnh đó, Langchain đóng gói và trừu tượng hóa bước gọi API đến phần lớn các nhà cung cấp lớn trên thị trường. Vì thế, người dùng chỉ cần xây dựng kiến trúc và yêu cầu Langchain gọi dịch vụ. Nhờ đó mà nhà phát triển kiến trúc học tăng cường có thể thử nghiệm trên nhiều loại mô hình ngôn ngữ lớn khác nhau.
   a. Python
      Langchain được viết dựa trên Python nên ta sẽ sử dụng ngôn ngữ Python

   cho dự án này.

2.4.2 Mô hình ngôn ngữ lớn (LLM)

Vì Langchain cho phép dễ dàng thay đổi linh hoạt nhiều mô hình ngôn ngữ mà không cần phải xây dựng lại kiến trúc học tăng cường, tôi lựa chọn các mô hình ngôn ngữ được cung cấp bởi Meta (Llama 3.2 và các phiên bản tương tự), Google (Gemini, Vertex,...) vì họ cung cấp API miễn phí đối với cá nhân nhà phát triển. Đồng thời tôi cũng sẽ sử dụng các model được huấn luyện thêm (finetune) cho tác vụ gọi hàm (function calling) trên nền tảng Huggingface. Sau khi xây dựng kiến trúc học tăng cường hoạt động tốt thì sẽ cân nhắc chuyển sang dùng các model của OpenAI (GPT-4o,...) hay Anthropic (Claude Sonnet 3.5,...).

2.4.3 Thư viện Fastapi:

Để hiện thực nhanh chóng microservice này, ta có thể sử dụng thư viện FastAPI của Python, thư viện này cho phép nhà phát triển nhanh chóng viết ra dịch vụ bởi hướng tiếp cận đơn giản và hướng dẫn sử dụng kĩ, cùng với đó là hệ sinh thái tốt và hỗ trợ lập trình async qua uvicorn server async (ASGI server) hoặc multi-processing (uvicorn workers).

3 Hiện thực

3.1 Xây dựng mô hình học tăng cường (RAG)

3.1.1 Prompt

3.1.1.a Prompt cơ bản

```
from langchain_core.messages import SystemMessage
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from pydantic import BaseModel

messages = [
    SystemMessage(
        content="Extract shipping job details from PDF text. "
        "If any value is missing, return default value. "
        "Use field descriptions for guidance, and please use enum values "
        "if provided",
    ),
    ("human", "This is what you have to extract from: {context}")
]
simple_extraction_prompt = ChatPromptTemplate.from_messages(messages)
```

3.1.1.b Prompt thực tế

Dùng hàm `create_dynamic_messages()` để tạo prompt trong Runtime một cách linh hoạt. Cho phép người dùng thay đổi prompt mà không cần phải ngưng server để sửa mã nguồn và chạy lại. Bên cạnh đó, người dùng sẽ thêm những lời hướng dẫn (tips) cho mô hình trở nên thông minh hơn.

```
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from pydantic import BaseModel


class PromptDefinition(BaseModel):
    system: str
    tips: list[str]


def create_dynamic_messages(
    prompt_definition: PromptDefinition, need_examples: bool = False
) -> ChatPromptTemplate:
    messages = [
        SystemMessage(content=prompt_definition.system),
    ]
    messages.extend(
        [HumanMessage(content=f"Tips: {tip}") for tip in prompt_definition.tips]
    )
    if need_examples:
        messages.append(MessagesPlaceholder("examples"))
    messages.append(("human", "This is what you have to extract: {context}"))

    return messages
```

từ đây ta có thể chỉnh sửa file `prompts.json` như sau:

```
{
  "system": "Extract shipping job details from PDF text. If any value is missing,
return default value. Use field descriptions for guidance, and please use enum
values if provided.",
  "tips": [
    "If a job lacks an ETD, it's likely IMPORT/RAIL_INBOUND; if it lacks an ETA,
it's likely EXPORT/RAIL_OUTBOUND. A job with rail info is a RAIL job. If unsure,
use IMPORT",
    "vessel and voyage often go together, first part is vessel name and second
part is voyage code",
    "Use datetime format YYYY-MM-DD for dates, timezone should be in UTC",
    "Container weights often include only grossWeight, but, it can sometimes
include netWeight or tareWeight, or both. Formula: netWeight = grossWeight -
tareWeight."
  ]
}
```

Mô hình sẽ đọc file `prompts.json` và gọi hàm `create_dynamic_messages()`:

```
from json import load
prompts_definition = load("prompts.json")
parsed_prompts_definition = PromptDefinition.model_validate(prompts_definition)
prompt = ChatPromptTemplate.from_messages(
    create_dynamic_messages(parsed_prompts_definition, need_examples)
)
```

3.1.2 Schema

Yêu cầu output có kết quả như sau:

```
{
  "jobType": "IMPORT",
  "shipmentType": "FCL",
  "referenceNumber": "string",
  "vessel": "string",
  "voyage": "string",
  "etd": "2024-06-11T10:25:40.834Z",
  "eta": "2024-06-11T10:25:40.834Z",
  "agentClient": "string",
  "consignClient": "string",
  "warehouseClient": "string",
  "accountReceivableClient": "string",
  "jobContainers": [
    {
      "containerNumber": "string",
      "sealNumber": "string",
      "tare": 0,
      "net": 0,
      "grossWeight": 0,
      "doorType": "string",
      "dropMode": "string",
      "containerSize": "string"
    }
```

```
  ]
}
```

3.1.2.a Schema cơ bản

Ta có thể định nghĩa một schema cố định như sau:

```
from pydantic import BaseModel, Field
class Container(BaseModel):
    containerNumber: Optional[str] = Field(default=None, description="")
    sealNumber: Optional[str] = Field(default=None, description="")
    dropMode: Optional[str] = Field(
        default=None,
        description="",
        enum=[
            "Sideloader Wait Unpacking/Packing",
            "Standard Trailer-Drop Trailer",
            "Standard Trailer Wait Unpacking/Packing",
            "Sideloader",
        ],
    )
    grossWeight: int = Field(default=0, description="")
    doorType: str = Field(default="any", enum=["any", "rear", "fwd"])
class Job(BaseModel):
    jobType: str = Field(
        ...,
        description="Classify job type based on context, possible values: "
        "IMPORT, EXPORT, MISC, RAIL_INBOUND, RAIL_OUTBOUND",
        enum=["IMPORT", "EXPORT", "MISC", "RAIL_INBOUND", "RAIL_OUTBOUND"],
    )
    cusRefId: Optional[str] = Field(
        default=None, description="customer referal id")
        vessel: Optional[str] = Field(
        default=None, description='Vessel name (e.g., "MV Oceanic")'
    )
    vessel: Optional[str] = Field(
        default=None, description='Vessel name (e.g., "MV Oceanic")'
    )
    voyage: Optional[str] = Field(
        default=None, description='Voyage name (e.g., "VOY123")'
    )
    portOfLoading: Optional[str] = Field(
        default=None, description="loading port location"
    )
    portOfDischarge: Optional[str] = Field(
        default=None, description="discharge port location"
    )
    eta: Optional[datetime] = Field(default=None, description="")
    etd: Optional[datetime] = Field(default=None, description="")
    agentName: Optional[str] = Field(default=None, description="")
    consigneeName: Optional[str] = Field(default=None, description="")
    consignorName: Optional[str] = Field(default=None, description="")
    warehouseName: Optional[str] = Field(default=None, description="")
    accountReceivableName: Optional[str] = Field(default=None, description="")
    # Lồng kiểu Container trong kiểu Job
```

```
    jobContainers: List[Container] = Field(
        default=None,
        description="A list container information according to the schema",
    )
```

3.1.2.b Schema linh động

Giống như Prompt, để người dùng có thể thoải mái tùy chỉnh nội dung prompt, tôi đã hiện thực một thư viện Python cho phép đọc định nghĩa của schema theo cấu trúc của `ModelDefinition` cơ bản như sau (xem source code của thư viện tại: github.com/qmi03/pydantic_dynamic_model):

```
class ModelDefinition(BaseModel):
    model_name: str = Field(..., pattern="^[a-zA-Z_][a-zA-Z0-9_]*$")
    fields: List[FieldDefinition]
```

Lớp `ModelDefinition`
- `model_name` (kiểu: `string`): Định nghĩa tên của schema, và
- `fields` (kiểu: `[FieldDefinition`): Định nghĩa danh sách các miền dữ liệu ở thuộc tính `fields`.

Trong đó, lớp `FieldDefinition` được định dạng như sau:

```
class FieldDefinition(Frozen):
    name: str = Field(..., pattern="^[a-zA-Z_][a-zA-Z0-9_]*$")
    base_type: Union[SimpleType, "ModelDefinition"]
    wrappers: List[WrapperType] = Field(default_factory=list)
    required: bool = True
    default: Optional[Any] = None
    description: Optional[str] = None
    validator_defs: List[FieldValidatorDef] = Field(default_factory=list)
    enum_values: Optional[List[str]] = None
    metadata: Dict[str, Any] = Field(default_factory=dict)
```

bao gồm:

- `name` (kiểu: `string`) Tên của miền dữ liệu.
- `base_type` (kiểu: `SimpleType` hoặc `ModelDefinition`): kiểu dữ liệu gốc, gồm các giá trị cơ bản như “string”, “integer”, “float”, “boolean”, “datetime”, “date”, và thuộc kiểu `ModelDefinition`, cho phép dữ liệu được lồng. Ví dụ, trong trường hợp của chúng ta: kiểu Container được lồng trong kiểu Job.
- `wrapper_type` (kiểu: [`WrapperType`]): Kiểu dữ liệu lồng lấy kiểu dữ liệu gốc (base_type). Các kiểu lồng hỗ trợ: là `list`, `dict`, `enum`. Nhập danh sách rỗng nếu không có kiểu lồng.
- `required` (kiểu: `bool`): Đánh dấu liệu trường này có bắt buộc không. Mặc định là True.
- `default` (kiểu: `Optional[Any]`): Giá trị mặc định của trường nếu có.
- `description` (kiểu: `Optional[str]`): Mô tả ngắn gọn về trường dữ liệu.
- `enum_values` (kiểu: `Optional[List[str]]`): Giá trị liệt kê cho các trường kiểu enum.
- `metadata` (kiểu: `Dict[str, Any]`): Metadata liên quan đến trường.

Hàm tạo ra schema linh động: `def create_dynamic_model(model_definition: ModelDefinition) → type[BaseModel]`

Ví dụ cách sử dụng hàm trong hệ thống hiện tại:

1. Đọc định nghĩa của schema trong `schema.json` và chuyển về dạng `ModelDefinition`
2. Gọi hàm `create_dynamic_model()`

```
import json
schema_definition = json.loads("./schema.json")
parsed_schema_definition = ModelDefinition.model_validate(schema_definition)

schema = create_dynamic_model(parsed_schema_definition)
```

```
// Ví dụ ngắn của schema.json
{
  "model_name": "BaseTransportJobInformation",
  "fields": [
    {
      "name": "jobType",
      "base_type": "string",
      "wrappers": [],
      "required": true,
      "default": null,
      "description": "Classify job type based on context, possible values: IMPORT,
EXPORT, MISC, RAIL_INBOUND, RAIL_OUTBOUND",
      "validator_defs": [
        {
          "validator_type": "custom",
          "params": {
            "customFunctionDef": "def validate(cls, v):\n    valid_types =
['IMPORT', 'EXPORT', 'MISC', 'RAIL_INBOUND', 'RAIL_OUTBOUND']\n    if v.upper()
not in valid_types:\n        raise ValueError(f'Invalid jobType: {v}')\n    return
v.upper()\n    "
          },
          "error_message": "Invalid job type"
        }
      ],
      "enum_values": [
        "IMPORT",
        "EXPORT",
        "MISC",
        "RAIL_INBOUND",
        "RAIL_OUTBOUND"
      ],
      "metadata": {}
    },
    {
      "name": "voyage",
      "base_type": "string",
      "wrappers": ["optional"],
      "required": false,
      "default": null,
      "description": "Voyage name (e.g., \"VOY123\")",
      "validator_defs": [],
      "enum_values": null,
      "metadata": {}
    },
    {
```

```
      "name": "accountReceivableName",
      "base_type": "string",
      "wrappers": ["optional"],
      "required": false,
      "default": null,
      "description": "",
      "validator_defs": [],
      "enum_values": null,
      "metadata": {}
    },
    {
      "name": "jobContainers",
      "base_type": {
        "model_name": "Container",
        "fields": [
          {
            "name": "grossWeight",
            "base_type": "float",
            "wrappers": [],
            "required": true,
            "default": null,
            "description": "",
            "validator_defs": [],
            "enum_values": null,
            "metadata": {}
          },
        ]
      },
      "wrappers": ["optional", "list"],
      "required": false,
      "default": null,
      "description": "A list container information according to the schema",
      "validator_defs": [],
      "enum_values": null,
      "metadata": {}
    }
  ]
}
```


3.1.3 Extraction Chain

Ghép tất cả thành kiến trúc học tăng cường.

```python
from langchain.prompts import ChatPromptTemplate
from pydantic import BaseModel
from pydantic_dynamic_model import ModelDefinition, create_dynamic_model
from src.db.main import get
from src.lib.example import Example, tool_example_to_messages
from src.lib.models import groq_chatmodel as default_chat_model
from src.lib.prompts import PromptDefinition, create_dynamic_messages

def default_chain():
    prompts_definition = get("prompt")
    schema_definition = get("schema")
```

```
    parsed_prompts_definition =
PromptDefinition.model_validate(prompts_definition)
    parsed_schema_definition = ModelDefinition.model_validate(schema_definition)

    prompt = ChatPromptTemplate.from_messages(
        create_dynamic_messages(parsed_prompts_definition, need_examples)
    )
    schema = create_dynamic_model(parsed_schema_definition)

    chain = prompt | default_chat_model.with_structured_output(schema=schema)

    return chain
```

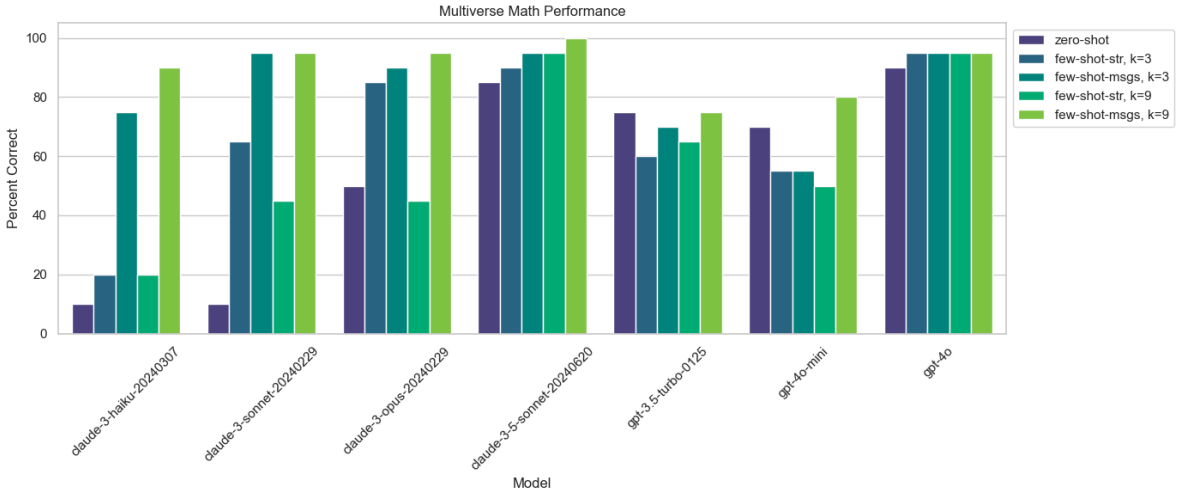
Sau đó người dùng có thể gọi bằng cách truyền dữ liệu không cấu trúc như sau:

```
input = get_input()
chain = default_chain()
output = chain.invoke({"input" = input})
```

3.1.4 (Mở rộng) Truyền thêm ví dụ vào model

Để mô hình có thể hoạt động tốt trên lượng dữ liệu chưa từng gặp phải. Ta sẽ sử dụng phương pháp Few-Shot Prompting (tạm dịch: Vài nhát ví dụ). Theo bài này của Langchain, cho thấy:

- Đối với model một số model, chỉ cần cho thêm 3 ví dụ giúp model tăng độ chính xác lên 6 lần.



Hình 4: So sánh hiệu năng của kết quả trả về khi gọi ví dụ.
Tham khảo từ:

https://blog.langchain.dev/few-shot-prompting-to-improve-tool-calling-performance/

Áp dụng vào RAG

Vì cấu trúc các mẫu pdf không thống nhất, và thay đổi tùy theo doanh nghiệp/đất nước, ta có thể cho người dùng xác định tên doanh nghiệp, từ đó model sẽ lựa chọn ngẫu nhiên trong tập dữ liệu mẫu và cho vào ví dụ giúp model trả lời thông minh hơn.

3.2 Viết dịch vụ Backend xử lý dữ liệu

3.2.1 Các API endpoints

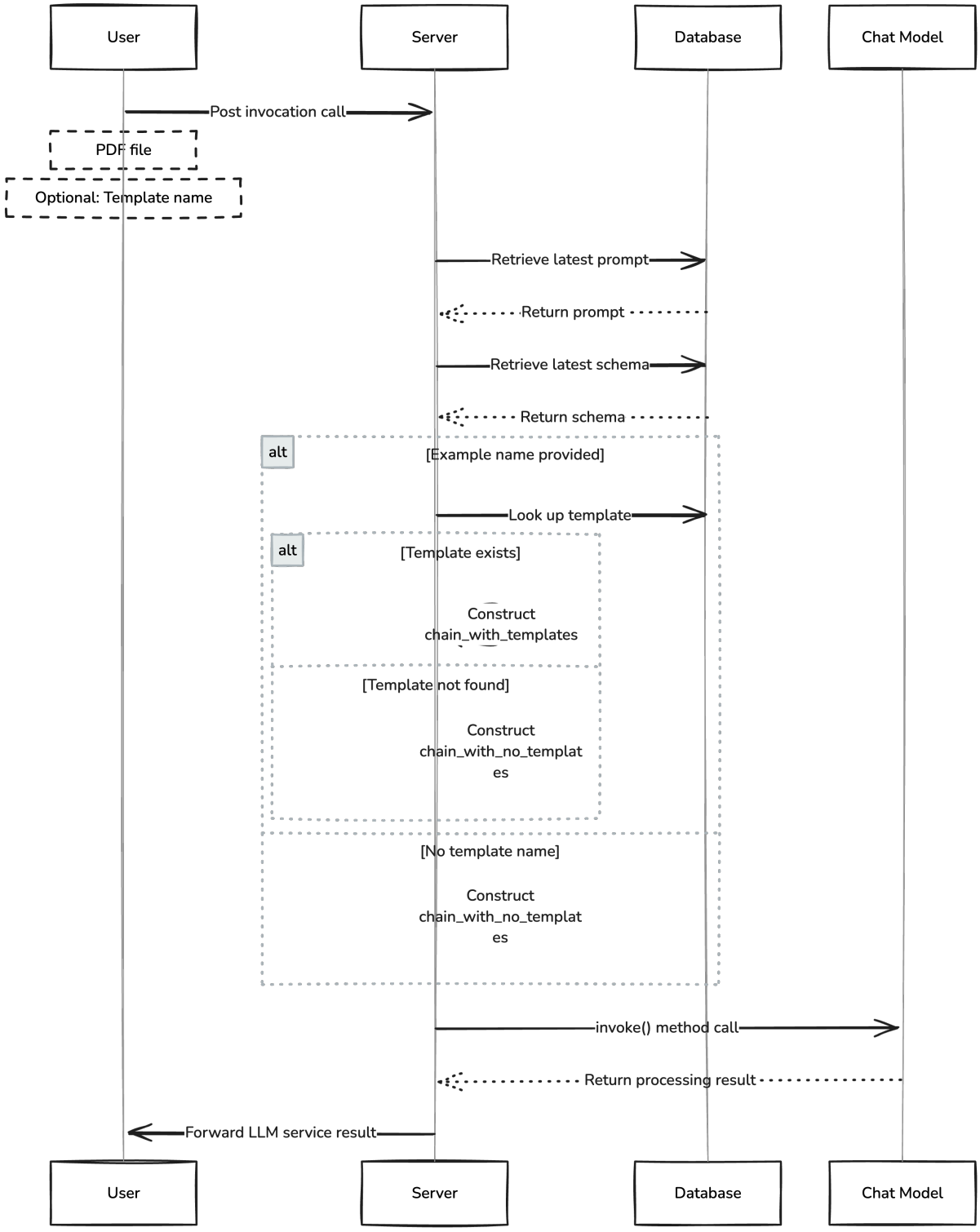
3.2.1.a /api/ai/invoke

1. Mô tả

Hệ thống sẽ có 1 endpoint API chính cho người dùng là `/api/ai/invoke`, cụ thể như sau:

- Tham số:
  - file: File pdf đầu vào.
  - template_name (Optional[str]): Tên các mẫu ví dụ để model trả lời thông minh hơn.
- Trả về: Dạng json đã qua xử lý

2. Sequence Diagram



Hình 5: Sequence diagram của endpoint `/api/ai/invoke`

4 Kết luận

TÀI LIỆU THAM KHẢO