数字医生分身多智能体系统技术设计文档V1

一、文档概述

1.1 项目背景

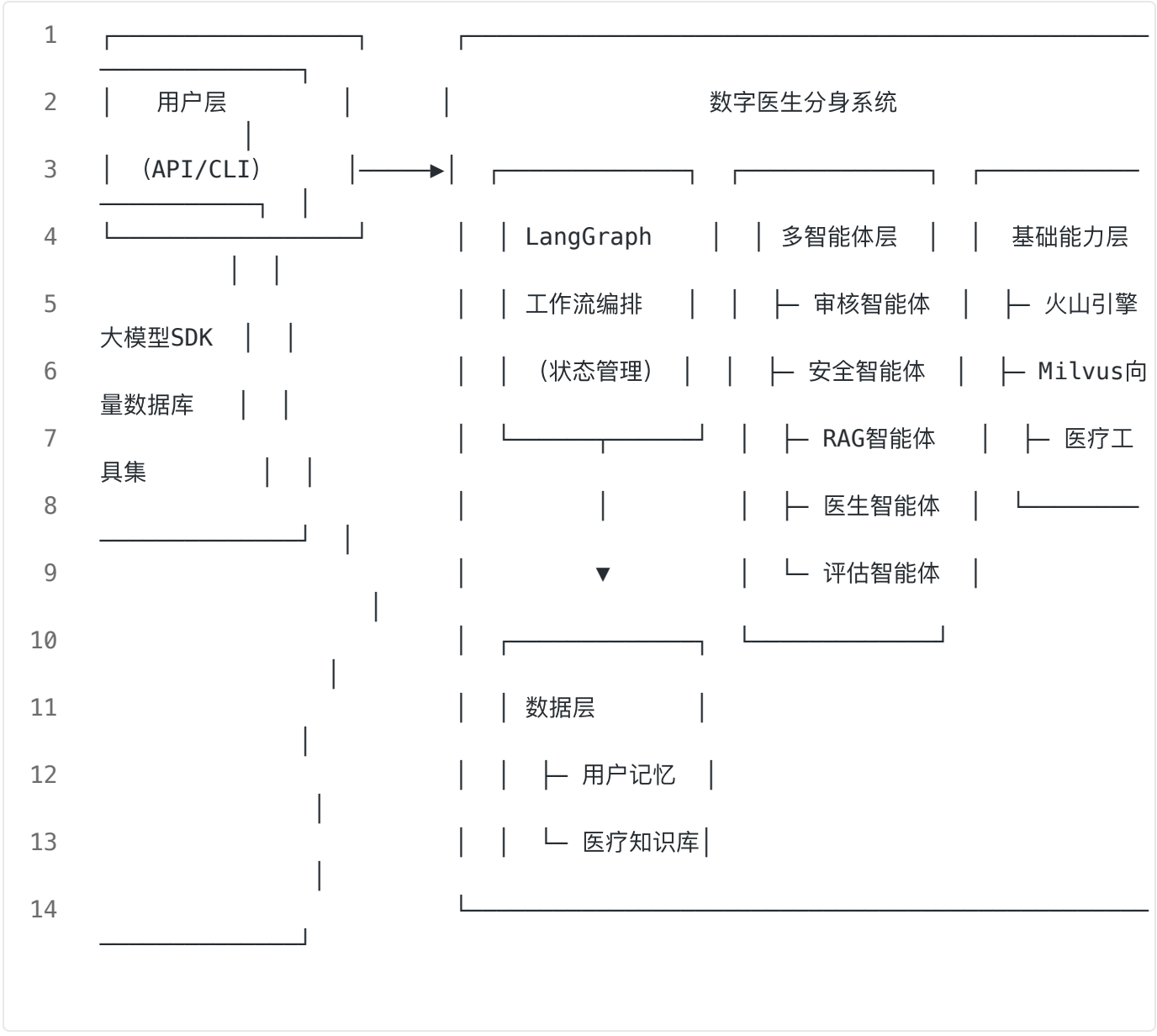
随着大模型与多智能体技术的发展，医疗领域对“智能辅助问诊”的需求日益增长。本项目基于**火山引擎大模型+LangGraph多智能体编排+Milvus向量数据库**，构建“数字医生分身”系统，实现医疗咨询的智能回复、风险控制、知识增强等核心能力,目前只限定在“三高”领域。

1.2 核心目标

- 实现**多智能体协作**：通过审核、安全校验、RAG检索、医生回复、质量评估等智能体的协同，输出专业、安全的医疗建议；
- 符合**医疗领域规范**：内置内容审核、风险校验模块，避免违规/高风险回复；
- 支持**知识增强与记忆**：通过RAG（检索增强生成）和Milvus记忆存储，提升回复的专业性与连续性；
- 遵循**工程化规范**：基于SOLID设计原则和面向接口编程：采用Poetry+LangGraph+模块化架构，保证系统可扩展、可维护。

二、系统架构设计

2.1 整体架构图



2.2 分层设计说明

(1) 工作流编排层（LangGraph）

- 核心职责：定义多智能体的协作流程、管理全局状态；
- 实现方式：通过 `StateGraph` 定义节点（智能体）、边（流程走向）、条件判断（失败终止/成功继续）；
- 关键组件： `DoctorState` （全局状态模型）、节点函数（对应智能体执行逻辑）。

（2）多智能体层

遵循**单一职责原则**，每个智能体仅负责一个功能：

智能体类型	核心职责
审核智能体	过滤敏感/违规内容（如“违禁药品”）
安全智能体	检测医疗风险内容（如“绝对治愈”）
RAG智能体	从Milvus医疗知识库中检索相关知识，增强回复专业性
医生智能体	调用火山引擎大模型，结合记忆/RAG知识生成医疗建议
评估智能体	对回复的专业性、安全性进行评分，低于阈值则重新生成

（3）基础能力层

- **火山引擎大模型SDK**：提供对话生成、文本Embedding能力；
- **Milvus向量数据库**：存储用户对话记忆、医疗知识库，支持RAG检索；
- **医疗工具集**：内置症状校验、用药建议校验等医疗专用工具。

（4）数据层

- **用户记忆**：存储用户历史对话（含Embedding向量），实现“上下文连续对话”；
- **医疗知识库**：存储专业医疗文档（如《临床诊疗指南》），通过RAG增强回复专业性。

三、核心模块设计

3.1 多智能体设计（SOLID遵循）

所有智能体基于 `BaseAgent` 接口实现，符合**依赖倒置原则**：

```python
# 核心接口定义（core/agent_interface.py）
from abc import ABC, abstractmethod
from typing import Dict, Any

class BaseAgent(ABC):
    @abstractmethod
    def execute(self, user_input: str, context: Dict[str, Any]) -> Dict[str, Any]:
        """智能体执行核心逻辑"""
        pass
```

示例：医生智能体实现

```python
class DoctorAgent(BaseAgent):
    def __init__(self):
        self.volc_tool = VolcAPITool()  # 火山引擎工具
        self.memory = MilvusMemory()    # Milvus记忆
        self.rag = MilvusRAG()          # RAG检索

    def execute(self, user_input: str, context: Dict[str, Any]) -> Dict[str, Any]:
        # 1. 加载用户记忆
        memory = self.memory.load_memory(context["user_id"])
        # 2. RAG检索医疗知识
        rag_docs = self.rag.retrieve(user_input)
        # 3. 构造Prompt调用大模型
        prompt = f"结合记忆{memory}和知识{rag_docs}回复：{user_input}"
        llm_res = self.volc_tool.call({"prompt": prompt})
        # 4. 保存对话到记忆
        self.memory.save_memory(context["user_id"], llm_res["content"])
        return llm_res
```

3.2 LangGraph工作流编排

通过“节点+边”定义流程，实现智能体的自动化协作：

```python
# 流程定义（graph/doctor_graph.py）
graph = StateGraph(DoctorState)
# 添加节点
graph.add_node("moderate", moderate_node)  # 审核节点
graph.add_node("safe", safe_node)          # 安全节点
graph.add_node("rag", rag_node)            # RAG节点
graph.add_node("doctor", doctor_node)      # 医生节点
graph.add_node("evaluate", evaluate_node)  # 评估节点
# 定义流程走向
graph.set_entry_point("moderate")  # 初始节点
graph.add_conditional_edges(
    "moderate",  # 从审核节点出发
    should_continue,  # 条件判断：是否有错误
    {"continue": "safe", "end": END}  # 审核通过→安全节点，失败→终止
)
# 其他节点的边定义...
```

3.3 Milvus记忆与RAG实现

（1）用户记忆模块

```python
class MilvusMemory(BaseMemory):
    def save_memory(self, user_id: str, content: str) -> bool:
        # 生成文本Embedding（火山引擎SDK）
        embedding = self.volc_tool.call({"type": "embedding", "text": cont
ent})
        # 插入Milvus
        self.collection.insert([user_id, content, embedding])
        return True

    def load_memory(self, user_id: str) -> List[str]:
        # 检索用户最近的对话
        res = self.collection.query(f"user_id == '{user_id}'", limit=5)
        return [item["content"] for item in res]
```

（2）RAG检索模块

```python
class MilvusRAG(BaseRAG):
    def retrieve(self, query: str) -> List[str]:
        # 生成查询Embedding
        query_embedding = self.volc_tool.call({"type": "embedding", "text": query})
        # 向量检索医疗知识库
        res = self.collection.search([query_embedding], limit=3)
        return [item.entity["content"] for item in res]
```

3.4 火山引擎大模型集成

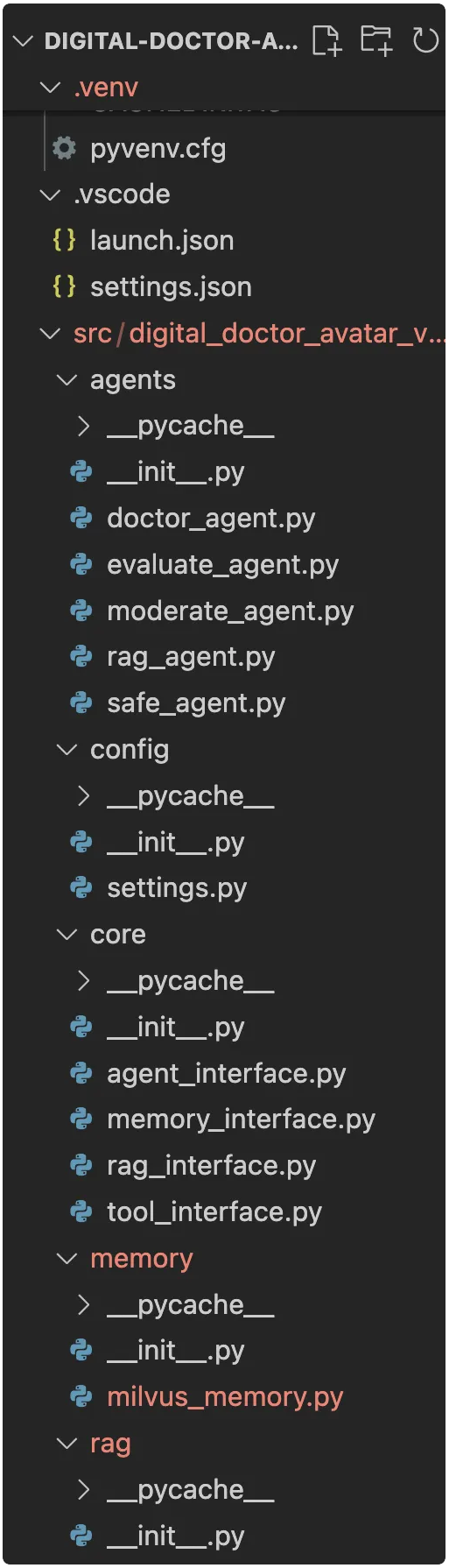
适配 `volcenginesdkarkruntime` 版本，支持对话/Embedding能力：

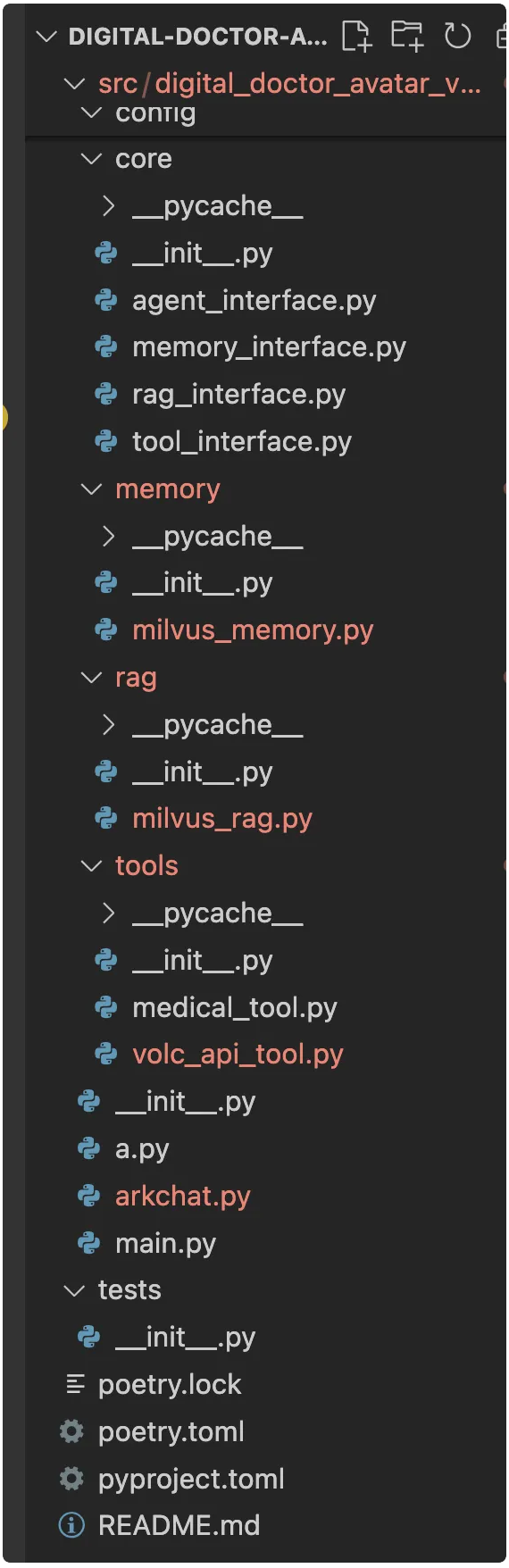
```python
class VolcAPITool(BaseTool):
    def __init__(self):
        self.client = Ark(api_key=VOLC_CONFIG["api_key"], base_url=VOLC_CONFIG["base_url"])

    def call(self, params: Dict[str, Any]) -> Dict[str, Any]:
        if params.get("type") == "embedding":
            # 生成Embedding
            res = self.client.embeddings.create(model="text-embedding-v1", input=params["text"])
            return {"success": True, "embedding": res.data[0].embedding}
        else:
            # 对话生成
            res = self.client.chat.completions.create(
                model="doubao-seed-1-6-flash",
                messages=[{"role": "user", "content": params["prompt"]}]
            )
            return {"success": True, "content": res.choices[0].message.content}
```

四、工程化实践

4.1 项目结构（Poetry+src布局）

```
digital-doctor-avatar/
├── src/
│   └── digital_doctor_avatar/  # 包根名
│       ├── core/               # 核心接口层
│       ├── agents/             # 多智能体实现
│       ├── tools/              # 工具集（火山引擎、医疗工具）
│       ├── memory/             # 记忆模块
│       ├── rag/                # RAG模块
│       ├── graph/              # LangGraph编排层
│       └── config/             # 配置文件
├── .venv/                      # Poetry虚拟环境
├── main.py                     # 入口文件
└── pyproject.toml              # Poetry配置
```


DIGITAL-DOCTOR-A...
.venv
pyvenv.cfg
.vscode
launch.json
settings.json
src / digital_doctor_avatar_v...
agents
__pycache__
__init__.py
doctor_agent.py
evaluate_agent.py
moderate_agent.py
rag_agent.py
safe_agent.py
config
__pycache__
__init__.py
settings.py
core
__pycache__
__init__.py
agent_interface.py
memory_interface.py
rag_interface.py
tool_interface.py
memory
__pycache__
__init__.py
milvus_memory.py
rag
__pycache__
__init__.py


DIGITAL-DOCTOR-A...
src/digital_doctor_avatar_v...
config
core
__pycache__
__init__.py
agent_interface.py
memory_interface.py
rag_interface.py
tool_interface.py
memory
__pycache__
__init__.py
milvus_memory.py
rag
__pycache__
__init__.py
milvus_rag.py
tools
__pycache__
__init__.py
medical_tool.py
volc_api_tool.py
__init__.py
a.py
arkchat.py
main.py
tests
__init__.py
poetry.lock
poetry.toml
pyproject.toml
README.md

4.2 依赖管理（Poetry）

pyproject.toml

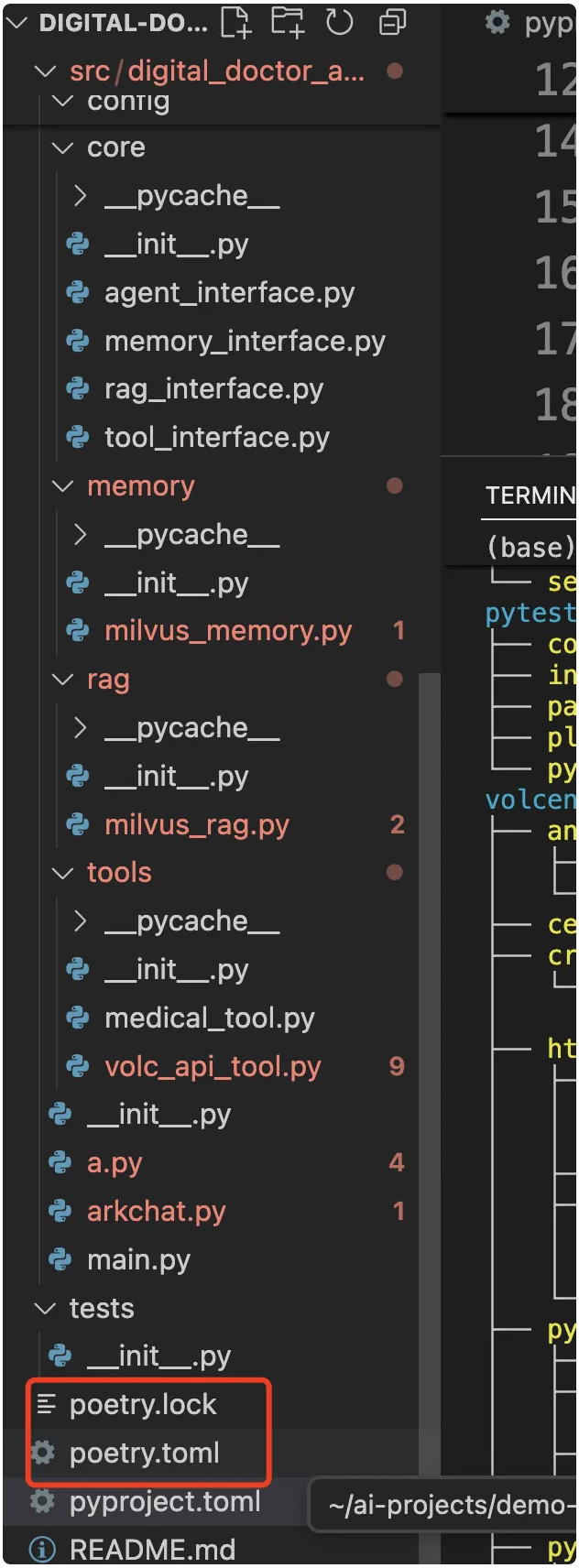
```toml
[project]
name = "digital-doctor-avatar-v1-demo"
version = "0.1.0"
description = ""
license = "MIT"
authors = [
    {name = "qinlinsen",email = "1540340840@qq.com"}
]

readme = "README.md"
requires-python = ">=3.11"
dependencies = [
    "volcengine-python-sdk[ark] (>=4.0.42,<5.0.0)",
    "pymilvus (>=2.6.5,<3.0.0)",
    "langchain (>=0.3.0,<0.4.0)",
    "langchain-core (>=0.3.0,<0.4.0)",
    "langchain-community (>=0.3.0,<0.4.0)",
    "langsmith (>=0.3.0,<0.4.0)",
    "langgraph (>=0.3.0,<0.4.0)",

]

[tool.poetry]
packages = [{include = "digital_doctor_avatar_v1_demo", from = "src"}]
[tool.poetry.dependencies]
# Python 版本 >= 3.10 是兼容 LangChain 0.3.x 所必需的
python = ">=3.11.0, <3.12.0"

# 使用波浪号 (~) 将 LangChain 核心包严格限制在 0.3.x 版本范围
# 这意味着安装 >=0.3.0 且 <0.4.0 的最新版本
#langchain = "~0.3.0"
#langchain-core = "~0.3.0"

# 社区包同样限制在 0.3.x，确保完全兼容
#langchain-community = "~0.3.0"

#langsmith = "~0.3.0"




requests = "^2.25.0"

#duckduckgo_search==8.0.4

[tool.poetry.group.dev.dependencies]
# 可选的开发/测试依赖项
```

```
pytest = "^8.0.0"
ipykernel = "^6.20.0" # 方便在 Jupyter 中运行

[build-system]
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"
```

4.3 开发模式配置

`用poetry进行依赖管理简单便捷和前沿。`

只需关注文件pyproject.toml和poetry.lock即可


DIGITAL-DO...
src/digital_doctor_a...
config
core
__pycache__
__init__.py
agent_interface.py
memory_interface.py
rag_interface.py
tool_interface.py
memory
__pycache__
__init__.py
milvus_memory.py 1
rag
__pycache__
__init__.py
milvus_rag.py 2
tools
__pycache__
__init__.py
medical_tool.py
volc_api_tool.py 9
__init__.py
a.py 4
arkchat.py 1
main.py
tests
__init__.py
poetry.lock
poetry.toml
pyproject.toml
README.md
~/ai-projects/demo-
TERMIN
(base)

五、系统扩展与优化方向

1. **医疗知识库增强**：接入公开医疗数据库（如PubMed），自动同步最新诊疗指南；对公司内部的“三高”优质文档进行录入
2. **多轮对话优化**：通过LangGraph的 `MemorySaver` 实现对话状态持久化，支持长对话；
3. **异步节点支持**：将RAG/Embedding节点改为异步，提升系统并发能力；
4. **可视化监控**：集成LangSmith实现智能体执行流程的监控与调试；
5. **合规性增强**：对接医疗行业合规接口，确保回复符合《互联网诊疗管理办法》。增加Human In The Loop(HITL)
6. **多模态支持**：增加对多模态的支持包括音频，图片和视频。
7. **MDT专业诊断**：MDT结合目前的im-service模块提取出相关工具进行MDT，实现跨学科团队的诊疗。
8. **通用工具提取**：viatris-patient, doctor-service 及其blood-pressure-service需抽出供agent调用的通用工具。

六、总结

本系统通过**多智能体协作+RAG知识增强+LangGraph流程编排**，实现了医疗咨询场景下的专业、安全、可扩展的数字医生分身。系统遵循工程化规范，既保证了医疗场景的专业性，又具备良好的可维护性与扩展性，可作为医疗AI辅助系统的基础架构。