دانشگاه فردوسی مشهد

دانشکده علوم ریاضی

MultiModal Agentic RAG

درس: پروژه کارشناسی
استاد درس: دکتر جلال‌الدّین نصیری

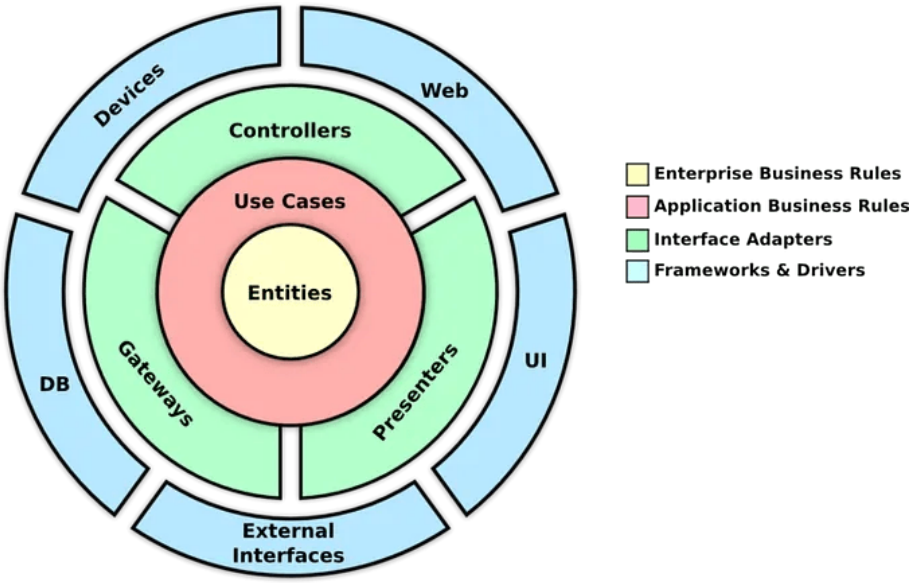
دانشجو: علی محمودجانلو

سال ۱۴۰۴

فهرست مطالب

۱ معماری سیستم

این پروژه بر اساس الگوی معماری Clean Architecture پیاده‌سازی شده است که امکان جداسازی مناسب نگرانی‌ها، مقیاس‌پذیری و نگهداری‌پذیری بالا را فراهم می‌آورد. انتخاب این معماری برای یک سیستم مبتنی بر مدل‌های زبانی بزرگ (LLM) که نیازمند انعطاف‌پذیری بالا و قابلیت تست‌پذیری است، از اهمیت ویژه‌ای برخوردار است.



شکل ۱: نمودار معماری Clean Architecture و لایه‌های آن

شکل ۱ نمودار کلی معماری Clean را نشان می‌دهد که در آن جهت وابستگی‌ها از بیرون به درون است و لایه‌های مرکزی از جزئیات پیاده‌سازی مستقل هستند.

۱.۱ دلایل انتخاب معماری Clean

انتخاب معماری Clean برای این پروژه بر اساس چندین معیار کلیدی صورت گرفته است:

- **قابلیت تست‌پذیری:** با جداسازی لایه‌های مختلف و استفاده از تزریق وابستگی (Dependency Injection)، امکان نوشتن تست‌های واحد و یکپارچه با پوشش بالا میسر می‌شود.
- **انعطاف‌پذیری در تعویض ارائه‌دهندگان LLM:** سیستم طراحی شده به گونه‌ای است که تغییر یا افزودن ارائه‌دهندگان مختلف LLM (مانند OpenAI، Cohere، یا مدل‌های متن‌باز) بدون تغییر در منطق اصلی امکان‌پذیر است.
- **مقیاس‌پذیری:** ساختار لایه‌بندی شده امکان توسعه و گسترش سیستم را در آینده تسهیل می‌کند.

۲.۱ لایه‌های معماری

معماری این سیستم شامل چهار لایه اصلی است که هر کدام مسئولیت‌های مشخصی دارند و قانون وابستگی (Dependency Rule) در آن‌ها رعایت می‌شود.

۱.۲.۱ لایه موجودیت‌ها (Entities)

این لایه، قلب سیستم را تشکیل می‌دهد و شامل enterprise business rules است که مستقل از هرگونه جزئیات پیاده‌سازی خارجی هستند.
مسئولیت‌های این لایه:

- تعریف core data models با استفاده از Pydantic
- Data validation و اعمال business rules
- تضمین type-safety و automatic serialization داده‌ها

به عنوان مثال، موجودیت `DocChunk` در سیستم ما شامل قوانین کسب‌وکار زیر است:

- **تولید شناسه یکتا:** هر chunk باید دارای شناسه‌ای یکتا باشد که بر اساس `document_id` و index تولید می‌شود: `{document_id}_chunk_{index}`
- **اعتبارسنجی بردار embedding:** اگر بردار embedding موجود باشد، باید از نوع `List[float]` و غیر خالی باشد
- **حفظ metadata اصلی:** تمام اطلاعات meta شامل schema_name، version و doc_items باید حفظ شوند
- **تبدیل فرمت:** موجودیت باید قابلیت تبدیل دوطرفه به فرمت Elasticsearch و Docling را داشته باشد

این قوانین مستقل از جزئیات پیاده‌سازی database یا search engine تعریف شده‌اند و در صورت تغییر فناوری زیرساخت، دست‌نخورده باقی می‌مانند.

۲.۲.۱ لایه (Use Cases)

این لایه application business logic را در بر می‌گیرد و workflow بین entities و interface adapters را هماهنگ می‌کند.
مسئولیت‌های کلیدی:

- مدیریت جریان‌های کاری پیچیده با استفاده از LangGraph
- پیاده‌سازی RAG (Retrieval-Augmented Generation) pipelines
- تولید و مدیریت embeddings
- هماهنگی با سرویس‌های خارجی از طریق interfaces

۳.۲.۱ لایه آداپتورهای رابط (Interface Adapters)

این لایه وظیفه تبدیل داده‌ها بین موارد استفاده و دنیای خارج را بر عهده دارد.
اجزای این لایه شامل:

- **Controllers:** مدیریت HTTP requests و responses
- **Presenters:** قالب‌بندی داده‌ها برای رابط کاربری یا مصرف‌کنندگان API
- **Gateways:** رابط با سرویس‌های خارجی مانند databases، vector stores و third-party APIs

۴.۲.۱ لایه فریم‌ورک‌ها و درایورها (Frameworks and Drivers)

بیرونی‌ترین لایه سیستم که شامل جزئیات پیاده‌سازی ابزارها و فریم‌ورک‌های خارجی است.
مثال‌های موجود در این لایه:

- Web frameworks (FastAPI، Flask)
- LLM providers (OpenAI، Cohere، open-source models)
- Databases و vector stores (Elasticsearch، Redis، Pinecone)

۳.۱ اصول طراحی

۱.۳.۱ قانون وابستگی (Dependency Rule)

در این معماری، وابستگی‌ها همواره از لایه‌های بیرونی به سمت لایه‌های درونی هستند. این بدان معناست که:

$$\text{Frameworks/Drivers} \rightarrow \text{Interface Adapters} \rightarrow \text{Use Cases} \rightarrow \text{Entities} \quad (۱)$$

لایه‌های درونی هیچ‌گونه اطلاعی از لایه‌های بیرونی ندارند و این استقلال امکان تغییر پیاده‌سازی‌های خارجی را بدون تأثیر بر هسته سیستم فراهم می‌آورد.

۲.۳.۱ اصل وارونگی وابستگی (Dependency Inversion Principle)

برای حفظ قانون وابستگی، از interfaces و کلاس‌های انتزاعی استفاده می‌شود. به این ترتیب:

- Interfaces در لایه‌های هسته (Use Cases) تعریف می‌شوند
- پیاده‌سازی‌های واقعی در لایه‌های بیرونی (Interface Adapters یا Frameworks) قرار می‌گیرند
- لایه‌های سطح بالا به انتزاع وابسته‌اند، نه به پیاده‌سازی‌های خاص

مثال کاربردی: در سیستم ما، یuse caseای به نام agentic_rag ماژولی از نوع EmbeddingServiceInterface دریافت می‌کند. این use case توابعی مانند embed_single را فراخوانی می‌کند، بدون آنکه بداند پیاده‌سازی واقعی از چه ارائه‌دهنده‌ای استفاده می‌کند (مانند OpenAI، Google GenAI، Azure، یا مدل‌های متن‌باز). این امر به این دلیل امکان‌پذیر است که:

- Use case تنها به contract تعریف شده در interface وابسته است
- تعویض ارائه‌دهنده embedding بدون تغییر در منطق کسب‌وکار به راحتی امکان‌پذیر است

۴.۱ سیستم Dependency Injection

برای مدیریت وابستگی‌ها و پیکربندی سیستم، از کتابخانه `dependency-injector` استفاده شده است. این سیستم مزایای زیر را فراهم می‌آورد:

۱.۴.۱ مزایای Dependency Injection

- **Centralized configuration:** تمام تنظیمات و وابستگی‌ها در یک container مرکزی مدیریت می‌شوند
- **تسهیل تست:** امکان جایگزینی وابستگی‌های واقعی با mocks برای تست‌نویسی
- **مدیریت singleton:** کنترل lifecycle منابع گران‌قیمت مانند database connections
- **انعطاف‌پذیری:** تغییر configuration در runtime بدون تغییر کد

انواع providers در کتابخانه dependency-injector:

- **Factory:** برای ایجاد نمونه جدید در هر بار استفاده. این نوع provider هر بار که درخواست می‌شود، یک instance جدید از کلاس مورد نظر ایجاد می‌کند. برای اشیائی مناسب است که stateless هستند یا هر بار نیاز به تنظیمات تازه دارند.
- **Singleton:** برای نگهداری یک نمونه واحد در طول عمر برنامه. این provider فقط یک بار شیء را می‌سازد و در تمام درخواست‌های بعدی همان نمونه را بازمی‌گرداند. برای منابع گران‌قیمت مانند اتصالات database، HTTP clients یا configuration objects مفید است.
- **Resource:** برای مدیریت چرخه حیات اشیاء پیچیده که نیاز به راه‌اندازی و پاکسازی دارند. این provider قابلیت‌های lifecycle management مانند initialization، cleanup و مدیریت منابع سیستمی را فراهم می‌آورد. برای سرویس‌هایی که نیاز به graceful shutdown دارند، ایده‌آل است.
- **Configuration:** برای مدیریت و تزریق مقادیر پیکربندی مانند environment variables، تنظیمات API، یا پارامترهای سیستم. این provider امکان خواندن مقادیر از فایل‌های config، متغیرهای محیطی یا منابع خارجی را فراهم می‌آورد و آن‌ها را در سراسر سیستم در دسترس قرار می‌دهد.

۵.۱ یکپارچگی با فریم‌ورک‌های LLM

یکی از چالش‌های اصلی این پروژه، یکپارچگی با فریم‌ورک‌های مختلف LLM بوده است. معماری طراحی شده این امکان را فراهم می‌آورد که:

- از چندین ارائه‌دهنده LLM به صورت همزمان استفاده شود
- جریان‌های کاری پیچیده با LangGraph مدیریت شوند
- عملیات RAG به صورت ماژولار پیاده‌سازی شوند
- تغییر یا ارتقای فریم‌ورک‌ها بدون تغییر منطق اصلی امکان‌پذیر باشد

۶.۱ استفاده از Pydantic

در لایه موجودیت‌ها، از کتابخانه Pydantic برای تعریف data models استفاده شده است. این انتخاب به دلایل زیر صورت گرفته است:

- **Automatic validation:** داده‌ها به صورت خودکار در زمان ایجاد object اعتبارسنجی می‌شوند

- **Type Safety:** تضمین صحت data types در زمان توسعه
- **Serialization:** تبدیل خودکار به JSON و سایر فرمت ها
- **Automatic documentation:** تولید schema برای APIها

۷.۱ مدیریت Error و Logging

سیستم دارای یک لایه یکپارچه برای error handling و logging است که:

- از multi-level logging پشتیبانی می‌کند
- خطاها را به صورت ساختاریافته ثبت می‌کند
- امکان request tracing را فراهم می‌آورد
- با سیستم‌های monitoring خارجی قابل یکپارچگی است

به عنوان مثال، میتوان از Prometheus برای جمع‌آوری متریک ها و از Grafana برای visualization استفاده کرد تا به اهداف زیر رسید:

- **Metrics Collection:** Response times، تعداد درخواست‌ها، نرخ خطا، و استفاده از منابع
- **Health Checks:** بررسی وضعیت سرویس‌های LLM، database connections، و vector stores
- **Alerting:** اطلاع‌رسانی خودکار در صورت بروز مشکل از طریق Slack یا Email

سیستم logging پیاده‌سازی شده در این پروژه، از قابلیت automatic rotation و مدیریت log levels مختلف برخوردار است. شکل ۲ نمونه‌ای از خروجی سیستم logging را نشان می‌دهد که شامل timestamp، نام module، log level، و message مربوطه است.

```
2025-10-13 21:02:09 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:79 - _switch_to_next_api_key() - Switched to API key index 0
2025-10-13 21:02:09 - multimodal_rag.frameworks.google_genai_base_service - WARNING - google_genai_ba
se_service.py:83 - _switch_to_next_api_key() - Cycled back to token 0, waiting 60 seconds before cont
inuing
2025-10-13 21:03:09 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:115 - _execute_with_retry_and_token_switching() - Trying embeddings_for_content with API k
ey index 0
2025-10-13 21:03:11 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:122 - _execute_with_retry_and_token_switching() - Successfully executed embeddings_for_con
tent with API key index 0
2025-10-13 21:03:11 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:115 - _execute_with_retry_and_token_switching() - Trying embeddings_for_content with API k
ey index 0
2025-10-13 21:03:13 - multimodal_rag.frameworks.google_genai_base_service - WARNING - google_genai_ba
se_service.py:134 - _execute_with_retry_and_token_switching() - Rate limit error with API key index 0
, switching immediately
2025-10-13 21:03:13 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:79 - _switch_to_next_api_key() - Switched to API key index 1
2025-10-13 21:03:13 - multimodal_rag.frameworks.google_genai_base_service - INFO - google_genai_base_
service.py:115 - _execute_with_retry_and_token_switching() - Trying embeddings_for_content with API k
ey index 1
```

شکل ۲: نمونه output سیستم logging

۸.۱ نتیجه‌گیری معماری

معماری Clean برای این پروژه MultiModal RAG، بستری مناسب برای توسعه یک سیستم scalable، maintainable و flexible فراهم آورده است. جداسازی واضح لایه‌ها، استفاده از dependency injection، و رعایت اصول SOLID، امکان توسعه پایدار و با کیفیت این سیستم را در بلندمدت تضمین می‌کند. این رویکرد معماری به‌ویژه برای سیستم‌های مبتنی بر LLM که نیازمند flexibility بالا در تعویض providers و models هستند، مناسب است.

۲ References

References

[۱] C. Ribeiro، “Sports scheduling: Problems and applications،” International Transactions in Operational Research، vol.۱۹، pp.۲۲۶–۲۰۱، Jan. .۲۰۱۲