ChatGPT

Refactoring Plan for the New System Architecture

Overview and Motivation

The current codebase needs restructuring to better align with the envisioned system architecture and improve maintainability. The architecture documentation describes a modular system with distinct components (Frontend API, CrewAI backend, LLM layer, tools, etc.) [1] [2], but the implementation is still largely monolithic. For example, all agent definitions are currently created via factory functions in a single `agents.py` module [3]. Likewise, all task classes reside in one file. This refactoring plan aims to break the code into logical modules that mirror the architecture, making it easier to extend and manage each component.

Key goals of this refactor include:

- **Modularity:** Group related classes and functions into packages (agents, tasks, API, etc.) matching the architecture components.
- **Clarity:** Each file/folder will have a clear responsibility (e.g. one agent per file, one task per file), making the system design easier to understand.
- **Align with Design:** Implement structures (agent classes, flows, etc.) that reflect the architecture diagrams and component descriptions [4] [2].
- **Prepare for Extensions:** Facilitate future additions like an API gateway and separate flows (e.g. audit logging) without major changes to core logic.

Proposed Modular Structure

Below is the proposed structure after refactoring, with new or reorganized components:

```
healthcare-aigent/
├── src/
│   ├── agents/                    # Agent definitions (one module per agent
type)
│   │   ├── __init__.py            # Initializes the agents package (aggregates
agents if needed)
│   │   ├── preprocessing_agent.py  # PreprocessingAgent class (or factory
function)
│   │   ├── language_assessor.py    # LanguageAssessmentAgent class
│   │   ├── clinical_extractor.py   # ClinicalExtractionAgent class
│   │   ├── summarization_agent.py  # SummarizationAgent class
│   │   └── quality_control_agent.py# QualityControlAgent class
│   ├── tasks/                     # Task definitions (each task in separate
module)
│   │   ├── __init__.py
```

```
│   │   ├── preprocess_task.py      # PreprocessMedicalTextTask class
│   │   ├── assess_language_task.py # AssessPatientLanguageTask class
│   │   ├── extract_clinical_task.py# ExtractClinicalInfoTask class
│   │   ├── summarize_task.py       # GenerateSummaryTask class
│   │   └── quality_control_task.py # QualityControlTask class
│   ├── flows/                      # Workflow orchestration (future-proofing
for multiple flows)
│   │   ├── __init__.py
│   │   └── audit_logging_flow.py   # (Potential future flow for audit logging &
DB updates)
│   ├── api/                        # API Gateway components
│   │   ├── __init__.py
│   │   └── api_server.py           # FastAPI (or similar) app definition
│   ├── tools/                      # Tool implementations (unchanged structure)
│   │   ├── medical_tools.py
│   │   ├── web_tools.py
│   │   ├── database_tools.py
│   │   ├── database_interface.py
│   │   ├── logging_tools.py
│   │   └── mock_database.py
│   ├── llm/                        # LLM Layer (already exists)
│   │   ├── __init__.py
│   │   ├── base.py
│   │   ├── openai_llm.py
│   │   ├── ollama_llm.py
│   │   ├── circuit_breaker.py
│   │   └── llm_factory.py
│   ├── main.py                     # Main entry point (orchestrates the overall
process)
│   └── utils.py                    # Utility functions (if any)
├── tests/                          # Tests (will be updated to new import
paths)
└── docs/                           # Documentation (update to reflect new
structure)
```

Correspondence to Architecture:

- **Agents (`src/agents/`):** Implements the specialized agents (Preprocessing, Assessment, Extraction, Summarization, QualityControl) as separate classes or factory functions. This aligns with the class diagram which envisions distinct classes for each agent [4].
- **Tasks (`src/tasks/`):** Contains each task in its own module, matching one-to-one with agent responsibilities (preprocessing, assessment, etc.). Tasks encapsulate the logic each agent performs.
- **Flows (`src/flows/`):** Prepares for the **Flows** concept in the architecture [5]. Initially, the main conversation processing flow is handled by the CrewAI `Process.sequential` crew. We may later add separate flow controllers (e.g. an audit logging flow) here.

- **API (`src/api/`):** Houses the **API Gateway** component from the architecture (for authentication, routing, etc.) [6]. This will allow exposing `process_medical_conversation` via a web service (e.g. a FastAPI app) for the front-end or external systems.
- **Tools (`src/tools/`):** Remains as the collection of external integration tools (database, web search, etc.), corresponding to the **Tools** in architecture [5].
- **LLM Layer (`src/llm/`):** Contains logic for LLM selection and usage (**LLM Factory**, OpenAI and Ollama integrations, etc.), representing the **LLM Layer** [5]. This was partially introduced already and will be fully utilized after refactoring.
- **Main (`src/main.py`):** Will orchestrate the overall sequence using the CrewAI System (**CrewAI System** in architecture) [2], but with calls into the new structured modules above. It may be slimmed down as more responsibility moves into the API or flow controllers.

Detailed Refactoring Changes

1. Agents Module Refactoring

Current State: All agents are created via functions in `src/agents.py`, which returns a generic `crewai.Agent` with role, goal, and tools configured [3]. There are no specialized Agent subclasses; the system relies on tasks for specific logic.

Changes:

- **Create `src/agents/` Package:** Convert the single `agents.py` into a package directory. Within `src/agents/`, create separate modules for each agent type:
- `preprocessing_agent.py` defining `PreprocessingAgent`
- `language_assessor.py` defining `LanguageAssessmentAgent`
- `clinical_extractor.py` defining `ClinicalExtractionAgent`
- `summarization_agent.py` defining `SummarizationAgent`
- `quality_control_agent.py` defining `QualityControlAgent`

Each module will encapsulate the creation and any unique behavior of that agent.

- **Define Agent Classes:** For clarity and future extensibility, implement each agent as a subclass of `crewai.Agent`. (The architecture diagram anticipates such classes [4].) For example, `PreprocessingAgent` could inherit from `Agent` and perhaps override an `execute_task` method or additional helper methods if needed. Initially, these classes might simply call the base class constructor with the appropriate role, goal, backstory, and tool list (mirroring what the factory functions did). This change allows adding agent-specific behaviors later (e.g., custom error handling or inter-agent communication logic per agent type).

- **Factory Functions (Optional):** If retaining simple creation functions is useful (for brevity when assembling a crew), we can keep factory functions like `create_medical_preprocessor(llm)` inside each agent module or in the package `__init__.py`. However, these would now instantiate the corresponding class, e.g. `return PreprocessingAgent(llm=llm, tools=[...])` instead of returning a base Agent. Another approach is to have the class constructor itself set up the tools and attributes, so calling `PreprocessingAgent(llm)` is sufficient.

- **Update Crew Creation:** The `create_medical_crew(llm)` function (now likely moved to `agents/__init__.py` or a specific module) will instantiate each of the five agent classes and return the list. For example, `create_medical_crew` would do `[PreprocessingAgent(llm), LanguageAssessmentAgent(llm), ...]` instead of calling the old functions.

- **Remove or Deprecate `agents.py`:** Once the package is in place, the original `src/agents.py` file should be removed. All references to it (in imports throughout the codebase and tests) will be updated to the new package. For instance, `from src.agents import create_medical_crew` will change to `from src.agents import create_medical_crew` (if implemented in `__init__.py`) or explicitly import the needed classes from `src.agents.*` modules.

Rationale: This change isolates each agent's definition, making it easier to modify one agent's behavior or add a new agent without touching a monolithic file. It mirrors the conceptual model of specialized agent classes [7]. It also sets the stage for possibly attaching specific methods or state to individual agents beyond what the base `Agent` provides.

2. Tasks Module Refactoring

Current State: All task classes (`PreprocessMedicalTextTask`, `AssessPatientLanguageTask`, etc.) are defined in a single `src/tasks.py` module [8] [9]. While this is less problematic than the agents (since they are at least separate classes), the file is long and groups multiple domains together.

Changes:

- **Create `src/tasks/` Package:** Similar to agents, convert `tasks.py` into a package directory. Each task class will live in its own module for clarity:
- `preprocess_task.py` with `PreprocessMedicalTextTask`
- `assess_language_task.py` with `AssessPatientLanguageTask`
- `extract_clinical_task.py` with `ExtractClinicalInfoTask`
- `summarize_task.py` with `GenerateSummaryTask`
- `quality_control_task.py` with `QualityControlTask`

- **Preserve Task Interfaces:** The behavior and interface of each Task class remain the same; we are only relocating them. Each task class still inherits from `crewai.Task` and implements `execute()` as before. For instance, `QualityControlTask.execute()` will remain as is, including its logging of audit events, until we optionally refactor that (see Flows below).

- **Update Imports:** Anywhere the code or tests did `from src.tasks import PreprocessMedicalTextTask` (or similar), update to import from the new package. We might add convenience imports in `tasks/__init__.py` if we want, e.g. `from .preprocess_task import PreprocessMedicalTextTask` so that `from src.tasks import PreprocessMedicalTextTask` still works after the change. Alternatively, the main code can explicitly import from individual modules.

- **Optional Consolidation:** If some tasks share significant logic or data structures, we might consider grouping them (e.g., if there were many tasks, one might group by category). However, with only five tasks, one per agent, one class per file is straightforward.

Rationale: Separating tasks into distinct modules clarifies the scope of each task and makes the code more navigable. If a developer needs to modify how clinical info extraction works, they can go straight to `extract_clinical_task.py` rather than wading through an omnibus file. This structure also parallels the agents package – reinforcing the one-to-one mapping between agent types and task types in the codebase.

3. LLM Layer Integration Updates

Current State: There are two mechanisms for obtaining LLMs in the code: - `src/llm_config.py` provides a function `get_llm()` that chooses between OpenAI or Ollama based on an environment variable and returns a LangChain `ChatOpenAI` or `ChatOllama` instance [10] [11]. - A newer `src/llm/` package exists (with `LLMFactory`, `OpenAILLM`, `OllamaLLM`, etc.), along with tests. This factory approach supports multiple LLM types and providers with a unified interface and fallback logic [12] [13].

Currently, `main.py` still uses `get_llm()` from `llm_config` [14], meaning the new LLMFactory is not yet leveraged in the main workflow.

Changes:

- **Adopt LLMFactory in Main Flow:** Refactor `process_medical_conversation` (and any other entry points) to use `LLMFactory` instead of the old `get_llm`. For example, decide on an LLM strategy for the conversation processing crew (perhaps use `LLMFactory.create_llm(LLMType.CLOUD_FAST)` or `get_llm_for_task` if tasks specify types). This ensures the **LLM Layer** component (LLM Factory) from the architecture is actually used in practice [15]. It also provides automatic provider fallback (e.g., if OpenAI key is missing or service down, fall back to local Ollama) [16] [17].

- **Environment Configuration:** Ensure that environment variables (`LLM_PROVIDER`, `OPENAI_API_KEY`, etc.) still influence the behavior appropriately. We may extend `LLMFactory` to consider `LLM_PROVIDER` to pick a default `LLMType` or modify `LLMFactory.create_llm` usage based on config. Another approach is to deprecate `LLM_PROVIDER` env in favor of a more nuanced selection via code logic. Regardless, document how to choose the provider (e.g., perhaps default to OpenAI if API key present, else Ollama – which is essentially what `get_llm()` did).

- **Deprecate `llm_config.py`:** Once the new factory is in use, the `llm_config.get_llm()` function becomes redundant. We can remove this module or keep it temporarily as a shim (calling into `LLMFactory`) to avoid breaking anything unexpectedly. Given this is an internal refactor, outright removal is fine as long as tests and docs are updated.

- **Update Tests:** Update any tests that directly used `get_llm` or expected certain LLM behavior. For example, tests might need to instantiate an LLM via the factory or mock `LLMFactory` instead of

`ChatOpenAI`. The existing tests for `LLMFactory` and connectivity will ensure the factory works; we just need to adjust others to the new interface.

Rationale: Using `LLMFactory` everywhere centralizes LLM selection logic and aligns with the design of an LLM layer that can switch providers and models dynamically [18] [19]. It reduces duplication (one unified way to get an LLM) and improves resilience (built-in fallbacks and checks for provider availability are in the factory). This change will make the system more adaptable to future LLM providers (e.g. adding Claude or others via the factory without changing core code paths).

4. Introducing an API Gateway Component

Current State: There is no dedicated API or web server in the code yet. The `main.py` simply runs a demo conversation when executed as a script, printing results to console [20] [21]. The architecture, however, envisions a Frontend with a **Web Interface** and an **API Gateway** that communicates with the backend [22] [1]. The plan is to eventually have a JavaScript front-end and a Python API backend (as noted in project decisions [23]).

Changes:

- **Create** `src/api/` **Module:** Set up a new package (e.g., `api_server.py` inside `src/api/`) that uses a web framework (likely **FastAPI** for ease of integration with async tasks) to expose endpoints. The primary endpoint could be `/process_conversation` (HTTP POST) which accepts a conversation (and possibly patient ID or other context) and returns the JSON results currently produced by `process_medical_conversation()`.

- **Implement Endpoint Logic:** Inside this API, call the core processing function (from `main.py` or perhaps we move `process_medical_conversation` into a more central module accessible to both CLI and API). The API should handle request parsing, calling the backend logic, and formatting the response JSON. It will also handle concerns like authentication, request validation, and error handling as needed (for now, maybe simple or none, but structured to add later).

- **Non-Blocking Execution:** Since CrewAI's `Crew.kickoff()` is synchronous (as adjusted in the async/sync fix) [24], we can call it directly inside the FastAPI endpoint (which can be a normal def or async def). If we foresee long processing times, we might make the endpoint async and run `crew.kickoff()` in a threadpool to avoid blocking the server event loop. This can be refined once we measure performance.

- **Configuration:** Decide how the API server is run. We can add to `pyproject.toml` another script entry for running the API (e.g. `healthcare-aigent-api = "src.api.api_server:run"` if we have a `run()` function launching Uvicorn). Documentation should be updated to guide how to start the API server. In development, one might run `uvicorn src.api.api_server:app --reload` for FastAPI.

- **Integration with Frontend:** This API will serve as the layer that a future JavaScript frontend calls. For now, a simple test could be done with `curl` or a small HTML form to ensure it returns results

properly. We should also ensure CORS is allowed if a web app will call it from a different origin (using FastAPI's middleware if needed).

Rationale: Adding an API layer is crucial to transition from a console-based POC to a real application. It fulfills the **API Gateway** role in the architecture [22], handling external communication and leaving the CrewAI system to focus on processing. By structuring it in its own module, we keep a clear separation of concerns: the API code (HTTP, request/response formatting) is separate from the core logic (agents, tasks, flows). This will also make it easier to secure and scale the interface (e.g., adding auth, rate limiting can happen in the API layer without affecting the agents).

Note: If immediate implementation of the API is out-of-scope for this refactor, at least stubbing out this module or planning for it is recommended. Even without a frontend yet, a well-defined API will guide how the system can be used in production (and can be tested with integration tests or by other services).

5. Implementing Flows and Audit Logging Separation

Current State: The concept of **Flows** (for audit logging, data pipelines, etc.) is mentioned in the architecture [5], but currently not explicitly implemented. The main processing flow is essentially the Crew that runs the five tasks sequentially. Audit logging and database updates occur within the QualityControl task – for example, `QualityControlTask.execute` reads patient data and logs an audit event to the database [25]. This means the task itself handles a bit of what could be considered a separate concern (logging).

Changes (Planned/Future):

- **Separate Audit Logging Flow:** We propose to isolate audit logging and perhaps database persistence into a distinct flow that can run after or in parallel with the main conversation-processing flow. For instance, once the main `crew.kickoff()` returns results, a new lightweight process could take relevant info (patient_id, summary, quality metrics, etc.) and handle logging to the database and audit trail. This could be done with a second `Crew` or simply a procedural step, but encapsulating it as a flow (maybe using CrewAI's deterministic flow concept or just a function) keeps the concerns separate.

- **Design a Flow Controller:** We might introduce a small class or function in `src/flows/audit_logging_flow.py` that knows how to perform these steps. For now, it could simply call the existing database interface's `log_audit_event` for the quality control results and handle any errors separately from the main agent logic. In CrewAI terms, a "Flow" might correspond to using the `Process` parameter or a different Crew configured for logging tasks (the CrewAI framework may allow multiple crews handling different processes).

- **Adjust QualityControlTask:** In the interim, we may leave `QualityControlTask` as is, but eventually we can simplify it to *only* produce the quality metrics, and move the database update (`db.update_records` or `log_audit_event`) out of the task. The sequence diagram in the architecture shows the Quality Control Agent updating records in the database after verification [26]. Realizing this, we might implement that update as a separate step after the agent completes, rather than inside the agent's task logic. This will make the core NLP tasks purely focused on AI output, and the logging/persistence handled by the system infrastructure.

- **Database Operations Flow:** Similarly, if in the future there are other deterministic processes (like saving processed conversation data to a database, or retrieving additional data), those could be structured as flows. The architecture lists **Database Operations Flow** as a component, hinting that database interactions might be orchestrated separately from the AI reasoning steps.

Rationale: Separating flows for logging and data management adheres to the **Separation of Concerns** principle and matches the architecture's division of Crews vs. Flows [5]. It will make the system more auditable and easier to maintain. For example, if a database call fails, that error could be caught in the logging flow without interrupting the main conversation analysis flow (or vice versa). In a production scenario, these flows could even be handled by different services or threads (one handling real-time AI response, another ensuring data is saved and logged properly). While this might not be fully implemented in the current refactor, the code is now structured to allow adding such flows cleanly.

6. Documentation and Configuration Updates

As we implement the above changes, documentation must be updated:

- **Architecture Docs:** Update `docs/architecture.md` if needed to reflect any name changes (e.g., if we introduce new class names or modules). The high-level design remains the same, but the *class diagram* could be updated to show actual class names (e.g. `PreprocessingAgent` class exists now) and perhaps the new module grouping. The *component diagram* remains valid, though we might note that API Gateway is now partially implemented.

- **Technical Implementation Doc:** The `docs/technical_implementation.md` and other design docs should be revised to match the new project structure. For instance, the project structure snippet in that doc (which currently lists `agents.py`, `tasks.py` etc.) should be changed to the new layout [27] [28]. Also, descriptions of how logging is done or how LLMs are selected should be updated to reflect the refactored approach (LLMFactory usage, etc.).

- **Setup/Installation Docs:** If any new steps are needed to run the API (e.g., installing uvicorn, or running a different command), include that in `README.md` or a dedicated `docs/setup.md`. Given we already have a dependency management via `uv` and an entry point for the CLI, we might add FastAPI to dependencies and ensure `uv sync --dev` covers it.

- **Testing:** All tests need to pass after reorganization. Update import paths in tests to point to new modules. If any tests assumed certain module-level variables or behaviors (like using `src.tasks.QualityControlTask` directly), adjust them to the new structure. Run the full test suite to catch any issues. The modular design should not change functionality, so tests should largely still be valid if imports are corrected. New tests might be added for the API endpoint (if implemented) and for any new flow logic.

- **Changelog/Migration Note:** Since this is a significant structural change, it's wise to document it in a `CHANGELOG.md` or at least in the PR description. Although this is an internal project, future contributors (or our future selves) should know that files moved and classes were introduced as of this refactor.

Work Breakdown Structure

To implement the new architecture, the work can be broken down into the following categories and tasks:

1. **Agents Module Refactor**
2. [ ] Create `src/agents/` directory and move agent-related code from `src/agents.py` into it.
3. [ ] For each agent type (Preprocessing, Language Assessment, Clinical Extraction, Summarization, Quality Control), create a module and define a corresponding class inheriting `crewai.Agent`. Ensure each sets up the proper role, goal, backstory, and tools.
4. [ ] Implement `create_medical_crew` (either in `agents/__init__.py` or a separate module) to assemble the list of agent instances using the new classes.
5. [ ] Update all imports in the project referring to `src.agents` to use the new package structure (including in `main.py` and tests).
6. [ ] Remove the old `agents.py` file once everything is migrated and working.
7. **Tasks Module Refactor**
8. [ ] Create `src/tasks/` directory and split the task classes from `src/tasks.py` into separate modules as described.
9. [ ] Ensure each task class is moved without altering its logic. Include any import adjustments needed (e.g., if tasks used each other or tools).
10. [ ] If needed, add imports in `tasks/__init__.py` for convenience (optional).
11. [ ] Update `main.py` and other modules/tests to import tasks from the new package structure.
12. [ ] Remove `tasks.py` after confirming all references are updated.
13. **Integrate LLMFactory & Remove llm_config**
14. [ ] Modify `process_medical_conversation` (in `main.py`) to obtain the LLM through `LLMFactory`. Determine the appropriate `LLMType` or use `get_llm_for_task` if tasks require different models (initially, likely one model for all).
15. [ ] Pass the created `llm` object into agent or task initialization as needed (the current design passes one `llm` to all agents and tasks).
16. [ ] Remove usage of `src/llm_config.get_llm()`. If `llm_config.py` is no longer needed, delete it.
17. [ ] Run LLM unit tests and integration tests to ensure the new code path works. Adjust tests that might have been directly calling `get_llm`.
18. [ ] Update documentation (especially any installation or config instructions that mentioned `LLM_PROVIDER` usage via `llm_config`).
19. **API Gateway Implementation**
20. [ ] Add FastAPI (or chosen framework) to project dependencies in `pyproject.toml` (if not already included).

21. [ ] Create `src/api/api_server.py` with a FastAPI app instance and define at least one endpoint (e.g., POST `/process_conversation`) that calls `process_medical_conversation` and returns the result.
22. [ ] For now, use the in-memory `process_medical_conversation` function; later this could be extended to use a persistent store or handle multiple requests concurrently.
23. [ ] Write a simple integration test for the API (e.g., using `starlette.testclient` to post a sample conversation and check the response structure).

24. [ ] Document how to run the API server in the README or a new doc (e.g., "API Usage").

25. **Flows & Logging (Design for Future)**

26. [ ] (Future) Outline how an `AuditLoggingFlow` or similar would be invoked after obtaining results. For this refactor, possibly just create a placeholder function or note in code for where to integrate it.
27. [ ] (Future) Refactor `QualityControlTask` to remove direct DB logging. Perhaps have it return the QC results, and then outside the crew execution, call a logging function with those results.
28. [ ] (Future) Implement `flows/audit_logging_flow.py` if choosing to formalize the flow. This could be as simple as a function `run_audit_log_flow(results)`.

29. [ ] (Future) Extend tests to ensure that moving the logging outside tasks doesn't break overall outcomes.

30. **Test and Document**

31. [ ] Verify all tests pass after the refactor. Fix any broken import paths or adjust expected data if anything changed slightly.
32. [ ] Run an end-to-end manual test: execute `healthcare-aigent` CLI (which calls `main.py`) to ensure it still prints the results correctly. Also, test the new API if implemented.
33. [ ] Update `docs/architecture.md` (if needed) and `docs/technical_implementation.md` with the new file structure and any changed technical decisions (e.g., note that we now have agent subclasses, and we use FastAPI for the API layer).
34. [ ] Increment version number or date in docs if that practice is used to mark the update.

Each of these tasks can be developed and reviewed incrementally. For instance, the Agents and Tasks refactoring (steps 1 and 2) can be done first, since they mostly involve moving code. LLM integration (step 3) can follow, ensuring the system still runs. Step 4 (API) can be developed and tested independently once the core is stable. Steps under 5 are forward-looking and can be implemented in a subsequent iteration or alongside if time permits. Documentation and testing are ongoing as part of each step.

Conclusion and Next Steps

By undertaking this refactoring, we will achieve a cleaner separation of concerns that closely matches the intended architecture. The codebase will be easier to navigate: one can find agent logic in the `agents` package, task logic in `tasks`, LLM handling in `llm`, and so on, reflecting the high-level design. This modular structure will make it simpler to **extend functionality**, such as adding new agent types or integrating additional tools, without overhauling unrelated parts of the code.

Critically, these changes prepare the project for scalability and collaboration. With an **API layer** in place, front-end developers or other services can integrate with the AI backend. With clearly defined agent and task modules, multiple developers can work in parallel (e.g., one can refine the ClinicalExtractionAgent while another works on the SummarizationAgent module).

After this refactor, the immediate next steps would be to thoroughly test the system end-to-end in its new structure and fix any minor issues introduced during the moves. Then attention can shift to remaining planned features: completing the LangFlow proof-of-concept integration (if still needed), enhancing the API (adding authentication, more endpoints), and implementing the separate flows for auditing and database operations as discussed.

In summary, this refactoring brings the implementation closer to the architecture vision, paving the way for a robust, maintainable healthcare multi-agent system. All functionality that existed before should continue to work as before (we are not changing what the agents do, just **how** the code is organized), but the system will be far more ready for future growth.

---

1 2 4 5 6 7 15 22 26 architecture.md
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/docs/architecture.md

3 agents.py
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/src/agents.py

8 9 25 tasks.py
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/src/tasks.py

10 11 llm_config.py
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/src/llm_config.py

12 13 16 17 18 19 llm_factory.py
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/src/llm/llm_factory.py

14 20 21 main.py
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/src/main.py

23 progress.md
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/docs/progress.md

24 migration_summary.md
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/docs/migration_summary.md

27 28 technical_implementation.md
https://github.com/pkuppens/healthcare-aigent/blob/776541147fcd3c6df6c2da7b1f10f7fda217833e/docs/technical_implementation.md