ChatGPT

VALIS Code Audit Report

Architecture & Design

Modular Structure: The VALIS project is well-organized into clear modules for the core engine, provider implementations, persona definitions, and integrations. The repository structure separates concerns effectively [1]. The **core** module (e.g. `core/valis_engine.py` and `core/provider_manager.py`) contains the engine logic and provider cascade orchestration. The **providers** module contains individual backend provider classes (Anthropic API, OpenAI API, a Desktop Commander integration, and a hardcoded fallback) [2]. Persona profiles are defined separately as JSON files under **personas/**, keeping persona data (like name, description, style) decoupled from code [3]. There is also an **mcp_integration** section for bridging to an external "Desktop Commander" system [4]. This separation suggests a thoughtful design where adding or modifying personas or providers can be done in isolation (e.g. adding a new persona by dropping a JSON file, or updating a provider's logic without touching core engine code).

Provider Cascade Logic: VALIS implements a cascade-of-providers strategy to ensure robust responses. Providers are tried in a priority order defined by the configuration: (1) Desktop Commander MCP (free Claude via local integration), (2) Anthropic API (Claude via cloud, paid), (3) OpenAI API (GPT, paid), and finally (4) a Hardcoded Fallback that "never fails" [5]. This ordering is built into the default config and reflects VALIS's philosophy of giving the *best possible* answer given the user's resources [6]. The `ProviderManager` class orchestrates this cascade. It initializes the providers in order and then, for each request, attempts each provider sequentially until one succeeds [7] [8]. Each provider is checked for availability via an `is_available()` method (for example, ensuring API keys exist or the local service is running) before use [9]. If a provider is not available or returns an error/timeout, the manager cleanly logs the failure and moves on to the next provider [10] [11]. This design provides fault tolerance – a failure or slow response in a higher-tier provider will automatically fall back to the next option without crashing the system. Notably, the final `HardcodedFallbackProvider` is **always available** and is designed to always return a response (even if it's a generic one) [12] [13], thereby guaranteeing that *"the engine never fails"*. This cascade approach, combined with the hardcoded fallback, ensures robustness: even in worst-case scenarios where external AI APIs are down or keys are missing, VALIS will produce some intelligent response.

Extensibility of Providers: While the provider cascade is conceptually extensible (the list of provider names comes from a config and is passed into `ProviderManager` [14] [15]), the current implementation uses a fixed factory method with hardcoded class imports for each known provider identifier [16]. Adding a completely new provider would require modifying this `_create_provider` method to handle the new name and import the appropriate class. There isn't a plugin registration system or dynamic discovery of providers – it's a static if/elif mapping. This is a slight limitation in design: the modules are separate, but the core still *knows* each provider type explicitly. In practice, it's not difficult to extend (one can add a new provider class in the providers folder and update the factory), but a more scalable design might use a base Provider interface and dynamic loading (e.g. via naming conventions or entry points). As is, the design cleanly supports the four built-ins, but adding providers beyond those requires touching core code. On the positive side, the use of a config list means the *order* or inclusion of providers can be changed without code

changes – for example, a user could disable a provider or reorder them by providing a custom config file, and the engine will respect that order when populating the cascade [17] [15].

Core Engine Responsibilities: The `VALISEngine` class serves as the central interface. It loads all persona definitions on startup and instantiates the provider cascade manager [18] [15]. This design ensures the engine is aware of all personas and providers at initialization. The engine's primary job is to accept a persona request and route it through the provider manager, adding any necessary context or post-processing. The engine also manages *session tracking* and *memory integration*: it tracks conversations by session ID to maintain continuity and to possibly feed a "neural memory" system for context [19] [20]. These features are integrated carefully – for example, the engine can augment the context passed to providers with retrieved memory snippets or session info (like conversation summary or interaction count) [21] [22]. This context-enrichment is done before calling `ProviderManager.get_response()`, meaning providers can potentially use that extra contextual info. It's a thoughtful design for extensibility: if the memory system is enabled and working, it enhances responses; if not, the engine falls back gracefully to no additional context [23]. Similarly, a "Neural Matrix Health Monitor" is conditionally integrated to monitor system health, but failure to initialize it is caught and just logged as a warning [24]. These design choices show an emphasis on graceful degradation – optional components (memory, health monitor) don't break the core engine if unavailable.

Persona Handling: Persona definitions are external JSON files, which is a good design for extensibility. The engine loads all `*.json` files from the personas directory into a dictionary at startup [25]. Each persona's data (name, description, traits, etc.) is then available for use during response generation. Currently, the persona data is primarily used for identification and basic context: the engine ensures the persona exists and passes the persona's dict to the provider. The provider implementations then may use fields like persona name or id to tailor the response. For example, the Desktop Commander provider passes the persona ID to the external script to invoke the correct persona profile [26] [27], and the Hardcoded Fallback provider uses the persona's name/ID to select an appropriate tone or fallback response category [28] [29]. Because personas are defined outside of code, adding a new persona is as simple as creating a new JSON profile – no code changes required. This is a strong point in the architecture for **persona extensibility**. One small caveat is that not all persona-specific data is currently utilized by the built-in providers: e.g. the JSON contains detailed fields like tone, background, and language patterns [30] [31] which the stubbed API providers do not yet use. As those providers get implemented, one would expect them to incorporate these fields (e.g. constructing system prompts or few-shot examples using the persona's background and style). The architecture has the data available; it just needs to be leveraged in provider logic. In summary, the design around personas is flexible and future-proof: new personas can be added easily, and the engine is prepared to supply persona context to whichever provider is handling the request.

Fault-Tolerance Mechanisms: The cascade design itself is a primary fault-tolerance mechanism, but VALIS adds more resilience on top of that. The `ProviderManager` includes a form of **circuit breaker and retry logic** (dubbed "temporal stabilization" in comments [32]). It tracks consecutive failures per provider and can temporarily stop using a provider that fails repeatedly: after 3 failures, a "circuit" for that provider is opened for 5 minutes, causing the cascade to skip it until the timeout expires [33] [34]. It also defines exponential backoff delays for retrying transient errors [35] [36]. In the current code, the main `get_response()` method in `ProviderManager` does a straightforward loop with a fixed 30s timeout per provider [8] and doesn't explicitly call the more advanced `_execute_cascade()` (which contains the detailed circuit-breaker logic). This suggests that the advanced fault-tolerance features are a work in progress or an optional path not yet activated. Nevertheless, their presence indicates a forward-looking design to make the

system robust against flakiness. For instance, `_try_provider_with_retries` will catch exceptions from a provider and automatically retry if the error looks temporary (network issues, timeouts, etc.) [37] [38], only giving up after exhausting the backoff schedule or encountering a permanent error. Even though this specific method isn't currently invoked in the main flow, its existence means the groundwork is laid for more sophisticated error-handling in future iterations.

In summary, the **architecture** is modular and well-conceived for an AI persona system. The separation of core logic, providers, and persona data is clean. The provider cascade and fallback strategy provide both extensibility (multiple backends) and fault tolerance. A couple of design limitations exist – such as the static provider factory and currently unused persona attributes – but overall the design should accommodate growth (adding new personas, implementing new providers, or integrating new features like memory and health monitoring) without major refactoring. The intent to support "universal personas" across varying resource scenarios is well-reflected in the project structure and flow.

Code Quality and Async/Safety

Code Readability & Style: The codebase is generally clear and readable. It follows standard Python conventions and uses descriptive naming. There are docstrings at the module and class level explaining purpose (e.g. the top of `valis_engine.py` provides an overview of VALIS and its inspiration [39], and the `VALISEngine` class has a succinct description of its role [40]). Key methods also include docstrings (for example, `VALISEngine.get_persona_response` and the provider classes' `get_response` methods each describe their function [41] [42]). Type hints are used consistently for function parameters and return types, which improves clarity and helps with static analysis. For instance, `get_persona_response` is annotated to return `Dict[str, Any]` and to accept optional context and session_id [41], and each provider's `get_response` signature includes types for persona, message, etc. [43]. This use of type hints across the code (combined with Python's duck-typing for providers) strikes a good balance between clarity and flexibility. Logging is used extensively and at appropriate levels (info for high-level events, debug for detailed retry/circuit messages, warning for recoverable issues, error for failures) [11] [44]. This will be valuable for debugging and understanding runtime behavior.

Asynchronous Pattern Consistency: VALIS is built on `asyncio`, and it generally handles asynchronous calls properly. The core engine method `get_persona_response` is an `async def` and uses `await` when calling into the provider manager [45]. The provider manager's `get_response` is also `async` and uses `await` for potentially long operations like checking availability and calling each provider [10]. The code uses `asyncio.wait_for` to bound the execution time of each provider call [8], which is a prudent way to prevent a hung API call from stalling the entire cascade. They also utilize `asyncio.Semaphore` to limit the number of concurrent requests being processed by providers to 10 [46], which helps avoid overwhelming external APIs or the local system. This semaphore is awaited properly using `async with self.request_semaphore` around the provider execution loop [47], ensuring that no more than 10 tasks enter that block concurrently. Similarly, an `asyncio.Lock` is used in `VALISEngine` to serialize access to certain sections – notably to protect incrementing a request counter and to manage per-session concurrency [48] [49]. The use of `async with lock:` blocks is correct and prevents race conditions on shared state like the `_request_counter` and the `_active_requests` dict.

Concurrency and Race Conditions: The code attempts to manage concurrency carefully, especially regarding multiple requests in the same session. In `VALISEngine.get_persona_response`, before

processing a request it checks if another request for the same `session_id` is already active [50]. If so, it logs a warning and even inserts a small delay (`await asyncio.sleep(0.1)`) to stagger the concurrent calls [51]. This indicates they were concerned about simultaneous requests possibly conflicting (for example, to avoid overlapping provider cascades for the same persona/session that might confuse the context or the external system). By locking and using the `_active_requests` dict, they ensure only one request per session is marked active at a time. However, this implementation has a subtle race condition: the code *overwrites* the active request entry for a session when a second request comes in after a brief sleep [51]. If the first request is still running, it may later remove the session from `_active_requests` in the `finally` block [52], inadvertently clearing the flag for the second request. This could allow a third request to start while the second is still in progress (because the guard was removed by the first). In effect, the intended serialization per session might not fully hold if multiple rapid-fire requests come in. This is a minor edge-case, but worth noting as a race condition – a more robust approach would be to queue or reject additional requests for a session instead of just sleeping, or to maintain a queue of tasks. Despite this, the likelihood of user code awaiting the first response before calling again is high, so this issue might not surface often in practice.

At the provider manager level, concurrency is handled more globally via the semaphore. All provider calls run inside the semaphore context, meaning at most 10 concurrent provider cascades system-wide. This is a reasonable default to prevent overload. The code also tracks each active request in a dict (with start time, persona, message preview) [53], which is used for logging and possibly for the health monitor, but it's not used to control execution beyond the semaphore. Since the modification of these tracking structures happens within the semaphore (thus already limited concurrency) and without internal `await`s, there's no evident race condition there. One minor point: updating shared structures like `provider_failures` and `provider_circuit_breakers` from multiple tasks could theoretically interleave, but since each modification is a simple in-memory operation protected by the GIL (and tasks only switch at await points), this is unlikely to cause inconsistency. Still, if very high concurrency were allowed, one might consider using locks around global counters or using atomic data structures. Given the concurrency is capped at 10 and operations are small, the implementation is probably fine as-is.

Await Safety and Blocking Calls: Almost all potentially blocking operations are awaited properly, but there are a couple of instances of synchronous calls in async context. For example, `DesktopCommanderProvider.is_available()` calls `subprocess.run(...)` with a timeout [54]. This is a synchronous call that will block the event loop while it runs (up to 5 seconds). Since `is_available` is awaited in the cascade, one provider's availability check could momentarily block others on the same event loop. Ideally, this should use an asynchronous subprocess (like `await asyncio.create_subprocess_exec`) or be run in a thread executor to avoid blocking the loop. That said, this check is relatively quick and only happens once per request when evaluating that provider, so the impact is minor. Similarly, loading persona JSON files at startup is done with normal file I/O [25]. This happens during engine initialization (not during an active async request), so it doesn't interfere with request handling. If the engine were initialized in the middle of an event loop with other tasks, it could block momentarily on file reads, but typically one would initialize VALIS at startup. Another area to watch is the memory integration: if `memory_client["add"]` or `query` perform I/O (e.g. database writes), they are invoked without `await` (because they might be normal functions) [55]. The code wraps them in try/except, but if those calls were long-running, they could block. It's possible these memory functions are quick or handled in their own threads; without their implementation it's hard to say, but this is something to consider for thread-safety. Overall, the majority of external interactions (provider calls, delays, etc.) use

proper `await`. The code does not appear to spawn background tasks without awaiting them (no use of `asyncio.create_task` that could go unmanaged), avoiding common pitfalls of orphaned coroutines.

Exception Handling: Robust error handling is implemented throughout the code. At the top level, `VALISEngine.get_persona_response` wraps the entire provider cascade call in a try/except so that any unexpected exception bubbles up as a controlled error message rather than crashing [56]. Within the cascade, each provider call is protected: the `ProviderManager.get_response` loop catches `asyncio.TimeoutError` for timeouts and general `Exception` for other errors from each provider [57]. When a provider fails, the error is logged and recorded, but the loop continues to the next provider [11]. This means one provider's failure won't halt the cascade – exactly what we want for fault tolerance. The code distinguishes between provider returning a result with `"success": False` (an expected failure mode where the provider handled the error and gave a message) versus the provider raising an exception; both cases result in trying the next provider, but exceptions are logged at error level with stack info for developers [57]. This layered approach (provider methods themselves often catch and return errors as dicts, plus the manager catches anything unhandled) makes the system resilient. For example, the DesktopCommander provider wraps its entire logic in a try/except and will return a dict with `"success": False` and an error message if anything goes wrong internally [58]. The cascade then treats that as a normal failure case. Another example is the HardcodedFallback provider's get_response: it catches **any** exception in its logic and still returns a `"success": True` with a safe fallback message [13], essentially ensuring that even if the fallback logic had a bug, the user still gets some response. This is a clever fail-safe ("even the fallback has a fallback") and demonstrates the defensive programming style used throughout VALIS.

One thing to note is that two of the provider classes (Anthropic and OpenAI) are currently stub implementations that immediately return a failure indicating "not yet implemented" [59] [60]. The cascade will handle these as just another provider returning `success=False` and move on. The stubs do check for API key presence in `is_available` [61] [62], so if no key is set, the provider will be skipped entirely. If a key *is* set, the cascade will attempt the provider, get the "not implemented" error, and then proceed to the next. The error messages for these stubs are clearly worded, so a developer or user can understand why those providers were skipped. As a matter of code quality, the presence of these stubs suggests that network call implementations (and their proper exception handling) are on the roadmap. When those are written, similar try/except patterns will need to be applied (e.g. catching `aiohttp.ClientError` or HTTP errors and returning a graceful message).

Async/Await Patterns: The code avoids common async pitfalls like mixing blocking calls (mostly) and uses context managers (`async with`) effectively for locks and semaphores. It also uses `await asyncio.sleep()` appropriately (only in a controlled way for timing tweaks). There is no misuse of low-level loop controls or anything that stands out as unsafe. The cascade uses `asyncio.wait_for` for a timeout, which is a correct pattern, but it's worth verifying that 30 seconds is an appropriate default. In a scenario where an API might legitimately take longer for a complex query, 30s could cut it off – however, given user-facing latency concerns, 30s is already quite high. For now, this is hardcoded [8]; making it configurable might be a future improvement, but not a critical issue. Another pattern worth mentioning is resource cleanup: the DesktopCommander provider creates a temp file for context and ensures it's deleted after use [63] [64]. This cleanup is done right after reading the process output, which is good practice to avoid littering temp files. It's done synchronously (no await, just an os.unlink), but that is fine.

Thread Safety: Since the code is single-threaded async (no threads spawned explicitly), thread safety mostly isn't a concern except for interactions with any external libraries that might use threads. The logging and data structures used are all standard for single-thread use. The only slight worry is the `subprocess.run` call in an async method, as mentioned, which blocks the main thread briefly. This doesn't introduce threading issues, just potential latency.

In summary, **code quality** is solid: it's well-documented, logically organized, and uses Python async features correctly for the most part. **Async and concurrency safety** have been carefully considered with locks, semaphores, and timeouts, though a couple of minor flaws (like the session concurrency logic and one synchronous call) exist. These do not fundamentally break the system but are areas for improvement. The extensive use of try/except and logging at all layers greatly aids in making the system robust and maintainable – errors are caught and reported rather than causing crashes or unhandled rejections. The code style is consistent, and the inclusion of docstrings and type annotations indicates a focus on maintainability.

Additional Observations and Potential Issues

Beyond architecture and basic code quality, a few specific issues and code smells were identified during the audit:

- **Incomplete Implementations:** As noted, the **AnthropicProvider** and **OpenAIProvider** are essentially placeholders. They always return `{"success": False, "error": "... not yet implemented"}` without making real API calls [59] [60]. This is a known gap rather than a hidden bug – presumably these will be implemented later. Until then, VALIS's cascade will never actually get a real response from those providers. This isn't a runtime error, but users should be aware that currently only the Desktop Commander (if available) and the hardcoded fallback produce answers. The presence of these stubs is a sign of a **"lazy" or deferred implementation**. In an audit, we flag this to ensure it's tracked: the code is structured to support these providers, but they are non-functional right now. One positive aspect is that the design clearly allows these to be filled in with actual API integration when ready (the patterns for key checking and method signatures are already in place).

- **Hardcoded Values and Configurability:** Several operational parameters are hardcoded in the code, which could reduce flexibility. For example, the provider concurrency semaphore is fixed at 10 threads [46], the circuit breaker triggers after 3 failures and lasts 5 minutes [65], and the retry delays are fixed to [1, 2, 4] seconds [35]. The timeout per provider is fixed at 30 seconds [8]. None of these values are currently configurable via the external config file (the config supports provider ordering and logging level, but not these internals [17]). While the chosen values are reasonable defaults, different deployments might want to tune them (for example, allow only 1 concurrent request in a low-resource environment, or shorten the timeout for snappier responses). Hardcoding them isn't immediately dangerous, but it's an **anti-pattern** for a library intended to be widely usable. Encapsulating these in the config or at least module-level constants would improve maintainability. On the flip side, the code does allow overriding the list of providers and some flags like `enable_memory` through a config JSON, which is good [17]. Extending that mechanism to cover these other parameters would be a natural next step.

- **Tight Coupling in Provider Initialization:** The use of a static factory method `_create_provider` with explicit imports means the core code knows about each provider class [16]. This is a bit of a **code smell in extensibility** – if one wanted to add, say, a new `AzureOpenAIProvider`, one must edit `provider_manager.py` to handle that name. A more decoupled approach would be to use a naming convention or registry. For example, provider classes could all inherit from a `BaseProvider` and be registered in a dict by name, or the code could dynamically import a module based on the name string. As it stands, the design is serviceable for the intended providers but not open for extension by third parties without modifying core. This doesn't affect functionality for now, but it does slightly contradict the open philosophy of "universal" personas, since adding new intelligence sources isn't plug-and-play. It's an area to consider refactoring if the project anticipates many new providers.

- **Persona Extensibility and Fallback Logic:** The persona system is flexible in terms of data, but we noticed that the **HardcodedFallbackProvider** doesn't have entries for all personas currently listed. It has tailored responses for "jane", "coach_emma", and "billy_corgan" [66] [67], but the personas "advisor_alex" and "guide_sam" (which are present in the JSON profiles) are not explicitly handled. In the fallback code, any persona that isn't recognized by name defaults to using "jane"'s response patterns [28]. This means if the user asks for Alex or Sam, the fallback will still produce a generic HR-flavored answer as if it were Jane. This is likely an oversight. It doesn't break the code (no errors), but it *does* reduce the quality of persona extensibility: adding a new persona JSON alone might not yield a unique persona behavior unless the fallback provider is also updated to include persona-specific responses or patterns. Ideally, the fallback could at least use the persona's description or tone from the JSON to craft a response. Currently, it relies on hardcoded canned replies. This can be marked as a minor design issue – **personas are easy to add, but their unique voice might not come through unless the AI providers use their data**. As the API-backed providers get implemented, we expect this to improve (those will likely incorporate persona background/tone into prompts). The fallback could similarly be enhanced by reading some cues from the persona definition (to avoid needing manual additions for each new persona).

- **Concurrency Control Quirk:** As discussed, the approach to handling concurrent requests in the same session uses a time-based workaround (sleep) which isn't foolproof [50]. This is a bit of an **anti-pattern** because it doesn't truly synchronize or queue the requests; it just delays the second request slightly. If the first request is lengthy, the second will still proceed after 0.1s, potentially overlapping. The use of a lock here is almost moot because they release it during the sleep. A cleaner approach would be to keep the lock held until the first request is done or to implement a queue or refusal mechanism. The current solution may have been a quick fix to avoid immediate simultaneous calls. This won't cause a crash, but it could lead to odd behavior (e.g., two overlapping answers for the same session persona). It's an edge case, but worth flagging as a code smell.

- **Use of Memory and External Integrations:** The engine conditionally integrates with an external memory module and a health monitor [68] [24]. It appends to `sys.path` and tries to import a memory manager. This is a bit hacky but understandable if the memory module isn't packaged. It's wrapped in a try/except so it fails gracefully if the module isn't present [68]. One could consider this a **code smell** (manipulating `sys.path` at runtime), but given it's only done once on init and clearly logged, it's acceptable for an internal extension point. Ideally, if this project grows, the memory subsystem would become a more formal plugin or package. The current implementation doesn't pose safety issues; it's more about maintainability.

- **Performance Considerations:** Instantiating a new `VALISEngine` for every `ask_persona` call (as shown in the README quick start) [69] could be inefficient if an application plans to handle many requests. The convenience function `ask_persona` creates a fresh engine (thus re-loading personas and re-initializing providers each time) [70]. This is fine for simple scripts or tests, but in a long-running application, one would likely want to create the engine once and reuse it. The code allows that – you can instantiate `VALISEngine` and call `get_persona_response` repeatedly. The documentation might want to clarify this to users to avoid performance issues. This isn't a bug in code, just a usage note: repeated re-init has overhead (reading files, etc.), although it's not huge (just a few JSON loads and object inits).

- **Testing and Reliability:** We noticed a `tests/` directory listed in the project structure [71]. A review of tests (if present) would be useful to gauge code coverage and catch any issues, but since this audit is focused on the code itself, we didn't go deep into tests. Assuming tests exist, that's a good sign for code quality focus. If not, adding tests for the cascade behavior (especially edge cases like all providers failing, or memory context injection) would be beneficial.

In conclusion, the additional issues found are relatively minor and fixable. The **major positives** – modular design, robust error handling, and clear async patterns – outweigh the negatives. The points above (hardcoded values, stub implementations, minor concurrency oddities, and fallback personalization gaps) should be addressed as the project matures to improve extensibility and polish. None of these are critical failures; they are typical in a young project balancing rapid development with robustness. Addressing them would elevate the code quality further and reduce technical debt.

Overall, VALIS demonstrates good software engineering practices for an AI system: clear separation of concerns, defensive programming, and asynchronous concurrency handling. With some refinements to the noted areas, it will be well-positioned as a reliable "universal AI persona engine" that is both extensible and resilient in production environments.

---

1 2 3 4 5 6 69 71 README.md
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/README.md

7 8 9 10 11 16 32 33 34 35 36 37 38 43 44 46 47 53 57 65 provider_manager.py
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/core/provider_manager.py

12 13 28 29 66 67 hardcoded_fallback.py
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/providers/hardcoded_fallback.py

14 15 17 18 19 20 21 22 23 24 25 39 40 41 45 48 49 50 51 52 55 56 68 70 valis_engine.py
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/core/valis_engine.py

26 27 42 54 58 63 64 desktop_commander_provider.py
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/providers/desktop_commander_provider.py

30 31 jane.json
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/personas/jane.json

59 61 anthropic_provider.py
https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/providers/anthropic_provider.py

60 62 openai_provider.py

https://github.com/Loflou-Inc/VALIS/blob/316792abc81b5f222a6cfcf77603ec0b57a53f1a/providers/openai_provider.py