ChatGPT

# Evaluating the GHSpec “Hybrid” Adapter (Option F)

## Executive Summary

The **Hybrid Approach (Option F)** proposes to integrate GitHub’s Spec-Kit with an OpenAI API-driven backend to simulate GitHub Copilot’s behavior in our experiment. In principle, Option F aims to combine Spec-Kit’s structured spec-driven workflow (for planning and task breakdown) with direct OpenAI API calls (for code generation), thereby **avoiding reliance on the actual Copilot IDE plugin**. This is intended to give us full control over the model and key, reproducible runs, and a fair basis for comparison with autonomous agent frameworks.

After careful analysis, we find that Option F has merit but **requires significant enhancement** to truly meet our goals. While it addresses some issues (API key control, consistent model usage [1] [2], and easier token tracking via OpenAI’s Usage API [3] [4]), it also **falls short on fidelity** to real Copilot usage and **risks scientific validity** if not extended to handle iterative refinement. For parity with frameworks like ChatDev and BAEs, the GHSpec adapter must not only **functionally implement** the Spec-Kit workflow but also mirror the **methodological characteristics** of those agents (e.g. multi-turn planning, self-correction, and dynamic re-planning).

**Recommendation in brief:** Approve a **modified Option F** – a **Spec-Kit–API hybrid with iterative refinement** – rather than vanilla Option F. We will use Spec-Kit’s structured prompts (for fidelity to spec-driven development), but replace Copilot’s proprietary interface with our own **OpenAI API orchestration** using the `OPENAI_API_KEY_GHSPEC` key and the `gpt-4o-mini` model [2]. This adapter will generate code for each Spec-Kit task via the API, **track tokens exactly** via the Usage API [3] [4], and incorporate **automated self-critiquing loops** to handle errors or spec mismatches (emulating how ChatDev/BAEs iterate). The result will be a **scientifically valid** comparison point that preserves the essence of Spec-Kit’s Copilot-driven workflow while meeting our experiment’s requirements for control, fidelity, and reproducibility.

Below we provide a rigorous analysis of Option F, discuss alternatives, then present a full implementation plan and evaluation matrix.

## 1. Analysis of Option F’s Claims vs. Reality

**Fidelity to Copilot Workflow:** Option F claims to faithfully mimic GitHub Copilot’s spec-driven development assistance by using Spec-Kit’s own prompts and logic. It is true that by leveraging Spec-Kit’s commands (e.g. `/speckit.specify`, `/speckit.plan`, `/speckit.tasks`, `/speckit.implement`), we adhere to the intended **spec-driven process** [5] [6]. However, **concerns remain**:

- *IDE vs API behavior:* Copilot is normally an **IDE-based, interactive code completion agent** [7]. It works by suggesting code in real-time based on editor context, and possibly using “Custom Instructions” in `.github/prompts/` [8]. Our API-based approach will instead produce code by

making explicit calls with full prompt context. This **one-shot task implementation** may differ from the iterative, line-by-line suggestions Copilot would produce as a developer types. For example, Copilot might adjust suggestions based on partially written code or file context; our approach will need to supply that context in prompts to emulate similar behavior. Without careful prompt engineering, **Option F might ignore Copilot's incremental nature**, potentially reducing fidelity.

- *Spec-Kit multi-phase alignment:* In normal use, after `specify` and `plan`, a developer (with Copilot's help) would implement tasks sequentially, often **reviewing and adjusting after each suggestion** [9] [10]. Option F's initial conception may have been too static – e.g., generating code for each task without mid-course corrections. Real Copilot usage involves a **human-in-the-loop** to verify each change, whereas a fully automated run could blindly accept flawed outputs. Thus, **assuming Copilot's output is always correct is a flaw** in Option F. We need to integrate validation and potential re-prompting to catch mistakes a human would normally catch.

- *Spec updates and context:* Another fidelity issue is how **additional requirements (steps 2–6)** are handled. Spec-Kit is designed around evolving the spec (it supports multiple feature branches and even a `/speckit.bugfix` command for iterative improvements [11] [12]). Copilot, in practice, would rely on the developer to update the spec or code and then continue. **Option F as proposed did not clearly account for updating the specification or plan between steps.** If we simply feed each new step description to the model without updating the spec document, we deviate from how Spec-Driven Development should work (the spec is meant to be the source of truth [13] [14]). On the other hand, updating the spec and re-generating a full plan/tasks for each incremental feature might be too slow or redundant within one run. **This tension was not fully resolved in Option F**, indicating an assumption that incremental features could be handled by the same one-shot process (a mismatch with actual usage).

**Reproducibility:** Option F touts better reproducibility since it uses direct API calls under our control. Indeed, by fixing random seeds and using the same commit of Spec-Kit and model each time [15] [16], we avoid the nondeterminism of a cloud service that could update models or use non-fixed randomness. We also log everything (prompts, outputs) in our system, which is critical for reproducibility. However, two caveats:

- *Copilot's nondeterminism:* Real Copilot suggestions can vary over time or between sessions (the model can update or produce different code on different tries). Option F using the OpenAI API with a fixed model and (ideally) `temperature=0` can produce more deterministic outputs. This **improves reproducibility** but at the cost of possibly not reflecting Copilot's occasional creativity or variability. The scientific question is whether we want to emulate **Copilot's actual behavior** (which might require some randomness) or treat Copilot as a **fixed function** for fairness. Option F assumed we could treat it deterministically – a defensible approach for fair benchmarking, but we should note that it's a simplification of reality.

- *Environment differences:* In practice, Copilot's behavior might depend on factors like the IDE state or user-specific context. By removing those, we make runs consistent, but there is a slight **threat to validity**: perhaps Copilot (with its VS Code integration) has access to additional context (open files, etc.) that our API prompt might miss, or vice versa. We mitigate this by explicitly providing all relevant context to the API (the spec, plan, tasks, and current codebase), but doing so robustly is complex. Option F's basic design may not have fully described how to maintain and feed *evolving*

*code context* into each prompt. Without that, reproducibility is moot because the results might be wrong to begin with. **In summary**, Option F can be made reproducible (by controlling seeds, keys, and context), but it must carefully manage context state across steps to remain faithful.

**API Control:** This is where Option F shines: by not using the actual Copilot service, we **retain full control over which API and model** is used. The design will ensure that `OPENAI_API_KEY_GHSPEC` is the only key used for GHSpec's LLM calls [1], and we can point those calls to the exact model variant (GPT-4 or others) we choose. All frameworks will use the same model (`gpt-4o-mini` in config [2]), satisfying the **fair comparison criterion** (no one framework quietly using a more powerful model). This addresses a critical issue if we had used Copilot directly – Copilot might use a proprietary model or a mix of GPT-4 and code-specific optimizations that we cannot configure. With Option F, **the model and parameters are in our hands**, ensuring a level playing field.

That said, one **flaw** is that **Spec-Kit's Copilot integration is not via a CLI** – it's IDE-based and does not expose an API by itself [7]. So Option F implicitly requires **reimplementing how Spec-Kit drives the AI**. In other words, we are intercepting what would have been Copilot's job and doing it ourselves via OpenAI. This is fine (and indeed our intention), but it means there is **no out-of-the-box reference to validate our implementation against**. We must be careful that our prompts to the API truly mirror what Spec-Kit *would have asked Copilot to do* in an IDE. The Spec-Kit project likely has hidden prompt templates for Copilot (possibly in `.github/prompts/` files and in how the VS Code extension uses them). We should obtain those templates or guidelines and use them in our API calls to maintain fidelity. Option F in its initial form might not have accounted for needing these templates, but we consider it essential to find and use them (or construct equivalents).

**Scientific Validity:** The goal is to treat GHSpec (Spec-Kit+Copilot) as another *autonomous coding agent*, and scientifically compare it to ChatDev and BAEs. For a valid comparison, **all agents should attempt the same tasks under similar conditions**. The Hybrid Approach is a step toward that—ensuring the same model and token counting, etc.—but there are subtle validity concerns:

- *Methodological parity:* ChatDev and BAEs are *autonomous agent frameworks* that likely involve multi-step reasoning, internal critiques, and possibly multi-agent collaboration (ChatDev uses two agents communicating [17]). If our GHSpec adapter simply generates code from spec and stops, the comparison might be unfair. It might either do too **little error-checking** (making GHSpec look worse if errors aren't fixed) or skip steps that the others undertake (making GHSpec look artificially fast or cheap). Scientific rigor demands that GHSpec's process include analogous phases: e.g. if ChatDev has a built-in step to run tests and fix bugs, we should allow GHSpec to do the same (perhaps by automatically invoking a "bug-fix" round using the spec kit agent if tests fail). Option F's original description didn't detail such behavior, which is a gap we must fill to ensure validity. Without adding iterative repair, we'd be measuring a Copilot-based approach that is not allowed to retry while the other agents possibly do (an **apples-to-oranges** scenario).

- *Spec alignment:* Spec-Kit's philosophy is that the spec and tasks drive development and that the AI **"does the bulk of the writing" with the human verifying at checkpoints** [18]. In a scientific experiment, we replace the human with automation, but we must still enforce that *verification* checkpoint. For example, after each phase (spec, plan, tasks, code), we should verify outcomes against expectations (with tests or predefined criteria). Option F must be extended to include those verifications and subsequent actions (like halting or fixing) to match the careful, checkpointed

process Spec-Kit advertises [19] [10]. This not only improves fidelity but also ensures we aren't "scoring" GHSpec on unvalidated output.

- *Token and cost tracking:* A positive aspect of Option F is using OpenAI's usage API to get exact token counts. We already implemented a unified token counting via the Usage API in the BaseAdapter [3] [4]. This means GHSpec's token usage can be captured just like ChatDev's, **ensuring accurate cost comparison** (one of our key metrics). If we had used the Copilot service, we would have no straightforward way to log token usage, since Copilot's API is closed. By using our own key and model, we can use `OPEN_AI_KEY_ADM` to fetch usage data after each step [20] [21]. This is a clear win for Option F – it upholds scientific validity in measuring efficiency. The **only caution** is that our GHSpec adapter must call `fetch_usage_from_openai()` at appropriate times (and not forget to set `_step_start_time` as noted [22]). The Hybrid Approach plan does anticipate this, but it's on us to implement it correctly (likely copying the pattern used for ChatDev adapter [23]).

In summary, **Option F partially delivers on its promises**: it gives us control over the API and model (thus helping reproducibility and fairness) [1], and it can be made to follow Spec-Kit's workflow. However, it **assumed a bit too much** equivalence between an API-driven run and actual Copilot usage. To avoid **mismatches with Copilot's interactive style**, we'll need to enhance context handling and possibly simulate an interactive environment. And to maintain scientific rigor, we must incorporate iterative error handling so that GHSpec is methodologically on par with the more autonomous frameworks. We next examine how to achieve that parity and consider alternative approaches that were on the table.

## 2. Parity with Autonomous Agent Frameworks (ChatDev, BAEs)

Our experiment demands that GHSpec (the Spec-Kit/Copilot approach) not only solves the same tasks as ChatDev and BAEs, but does so in a **comparably agentic manner**. ChatDev and BAEs are presumably systems where the AI *autonomously* plans, codes, evaluates, and iterates. For instance:

- **ChatDev:** This framework (OpenBMB's ChatDev) uses multiple AI "roles" (e.g. a Chief Architect, a Code Writer, a Tester, etc.) in a conversational loop to plan and implement features. It inherently includes **self-critiquing** – e.g., the Tester agent might point out errors and the Coder agent fixes them in subsequent rounds [17] [24]. ChatDev's adapter in our system reflects this by detecting when the AI requests clarification or identifies an issue and injecting a fixed response or triggering another cycle [17]. In tasks, we see explicit handling for "HITL detection" (clarification queries) and iterative retries [17] [25]. So ChatDev is robust: if the first attempt is incomplete, it can ask questions or try again within the same step.

- **BAEs:** Although we have less detail on BAEs in the text, given it's a "Build Autonomous Agent System" (as implied by the repo name and context), it likely uses a single agent or a specialized process to plan and code. BAEs might generate a plan, then code, then test, etc., possibly in an iterative improvement loop (similar to frameworks like AutoGPT or others). The key is that BAEs presumably **does more than one-shot code generation** – it might refine its output if tests or validations fail. We know our orchestrator will run a validation suite after each step for any framework [26] [27], so if BAEs fails a step's validation, it could incorporate a loop to fix it (depending on BAEs' capabilities). If BAEs doesn't have that inherently, we might enforce step retries in the adapter (the design mentions up to 2 retries on step failure in the runner logic [28]).

Given that, **GHSpec must be enhanced to have similar capabilities**:

1. **Iteration & Replanning:** Spec-Kit by itself produces a plan and tasks and then code. But if something goes wrong (e.g., a requirement wasn't met or a bug is present), a human would ordinarily step in and use a Spec-Kit command (like `/speckit.clarify` or propose a fix) and run the agent again. In our automation, we should mimic this by **allowing GHSpec multiple passes** when needed. For example, after implementing tasks, run our test validators. If a test fails, we can formulate a **"bug fix" command** to the AI. Although Spec-Kit's official release did not yet have a `/speckit.bugfix` (there was an open issue for it [11]), we can create a prompt pattern that says: "The following test failed or requirement wasn't met; propose a code fix." In practice, this could be as simple as appending a new task like "Fix the bug in X as revealed by test Y" and calling the implement agent on that task. This ensures GHSpec isn't unfairly penalized for first-try errors that the other frameworks would resolve through autonomy.

2. **Multi-step context carryover:** ChatDev and BAEs are likely **persistent across the six evolution steps** – they carry the state of the code and conversation forward. GHSpec must do the same. This means the GHSpec adapter should *not* treat each of the six steps as completely independent. Instead, after step 1 (initial CRUD app) is done, step 2 ("Add enrollment relationship") should be executed **in the context of the existing codebase**. Concretely, if we're using our API-based approach, the prompt for step 2's tasks should include the code (or a summary of it) from step 1, so that the model knows what to modify. We might emulate what a Copilot user would do: open the relevant files and ask for changes. This parity is crucial – otherwise GHSpec would be like starting from scratch at each step, which is neither realistic nor fair. Option F didn't explicitly detail how to persist state, but our implementation will ensure that the **workspace (repository)** is persistent through all 6 steps, and our prompts use the code in that workspace as part of the context for subsequent steps (perhaps by reading key files or diffing the required changes).

3. **Methodology vs Functionality:** It's not enough that all three frameworks *get to the end result* (functioning code) – we also care about **process metrics** (AUTR: Autonomy Ratio, ITR: iterations, etc.). If ChatDev typically solves a step in, say, 3 internal iterations and GHSpec just does 1, can we truly compare "autonomy"? Possibly not, unless GHSpec also is given the chance for multiple internal iterations. Our design can incorporate a loop: for each GHSpec step, allow up to N internal fix attempts (just as runner allows retries). Indeed, our runner already has `retry_count` and can do up to 2 retries on failure for any adapter [28]. We can leverage that: define a "failure" for GHSpec step as failing validation tests or hitting an unhandled exception in code generation, and then simply let the runner call `execute_step` again (with perhaps an augmented prompt like "second attempt"). This way the *framework-level* retry aligns with ChatDev's internal logic. We must implement GHSpec's `execute_step` such that it knows whether this is a retry and can adjust (for example, it could read a log of what went wrong and try a fix). This is advanced, but even a naive approach (re-run the same tasks – the deterministic model might give the same output though, unless we alter the prompt to include feedback) might not help. So better is to explicitly incorporate feedback as described above.

In essence, to achieve parity, the GHSpec adapter will operate more like an **autonomous agent that uses Spec-Kit's structured approach as its "thought process."** The phases might be:

- *Phase A: Spec & Plan (only at step 1)* – Get the initial spec and plan. (Subsequent steps might update these.)

- *Phase B: Task generation* – Use Spec-Kit's `/tasks` to break down the current requirement (either the initial or an incremental one).
- *Phase C: Task implementation loop* – For each task, use the LLM to write code. Possibly iterate within a task if needed (though ideally tasks are small enough to do in one go).
- *Phase D: Validation & repair* – After tasks are done (or after each, depending on strategy), run tests. If failing, treat the failures as new input (perhaps generate a "fix tasks" list or directly attempt fixes with the LLM).
- *Phase E: Move to next step feature* – repeat Phases B–D for the next feature, preserving all prior code.

By mirroring the above, GHSpec will behave in spirit like ChatDev/BAEs: reading instructions, breaking into sub-tasks, carrying out coding, verifying, and adjusting. This **methodological comparability** is crucial for our scientific claims. If we implement Option F with these enhancements, **we can credibly state that GHSpec (Spec-Kit+LLM) was given the same opportunities to self-correct and plan as the other agents**, making outcome differences truly attributable to the framework approach (structured spec vs. multi-agent chat, etc.).

## 3. Exploring Alternative Strategies

Before committing fully to the Hybrid Approach, we evaluated other implementation strategies for the GHSpec adapter. Our goal was to maximize fidelity to real Copilot usage while ensuring feasibility and scientific rigor. Below we review the main alternatives considered, along with their pros and cons:

- **A. Direct Copilot Extension Automation (VS Code + Copilot):** One idea was to literally use the GitHub Copilot VS Code extension in an automated way. For example, spin up a headless VS Code instance or use the Copilot CLI (if available) to run through the spec-driven commands. This would mean **true fidelity** – we'd be using Copilot exactly as a human user would, with the IDE providing context and suggestions. However, this approach is **fraught with practical issues**:
- *No official headless API:* Copilot is not offered as a public API for code generation tasks outside the IDE environment [29] . Any automation would be hacky – e.g., using UI scripting to paste commands and accept suggestions, which would be extremely brittle and hard to reproduce.
- *Inconsistent outputs:* Because we cannot set Copilot's randomness or ensure the same suggestion each time, two runs might differ (hurting reproducibility). We also cannot guarantee using the exact same model version across time – GitHub might update Copilot's backend models without notice.
- *Lack of token insight:* We'd have no way to measure tokens or cost, since Copilot doesn't expose those. We'd also surrender control of `OPENAI_API_KEY_GHSPEC` – Copilot uses its own credentials. This violates our requirement of full API control and comparability of cost.
- *Time and complexity:* Automating an entire six-step scenario through an IDE likely exceeds reasonable complexity. It could involve waiting for suggestions, detecting when Copilot is "done" for each task, and dealing with potential Copilot timeouts or errors in understanding commands.

**Conclusion:** While maximally true to the user experience, this approach is **not feasible for a controlled experiment**. The risk of flakiness and inability to collect metrics rules it out.

- **B. Copilot Network/API Interception:** Another notion was to intercept the calls that the Copilot extension makes to the OpenAI service, effectively hijacking them to use our API key or to log usage. For instance, one could attempt to proxy the Copilot endpoint or simulate being the Copilot client.

This could, in theory, give us actual Copilot model suggestions but with our own key or at least visibility. However, **policy and technical barriers** make this unsuitable:

- *Terms of service:* Reverse engineering or intercepting Copilot's communication likely violates GitHub's terms and would be ethically questionable for a research experiment.
- *Compatibility:* Even if done, we cannot change the model Copilot uses (it may not be exactly `gpt-4o-mini`, and we can't point it to a different model easily). This fails the "same model" criterion.
- *Complexity:* Copilot's protocol is not public – implementing a stable interception would require significant effort and could break anytime Copilot changes its API.
- *Latency and Real-time issues:* Copilot streams suggestions as you type. Intercepting and controlling that in a batch experiment (where we want a deterministic output per full prompt) might be extremely complex.

**Conclusion:** API interception is **not practically viable or legal**. It also wouldn't easily let us do multiple planning iterations or capture token counts. We eliminated this option early.

- **C. Using an Alternate Agent CLI (Claude Code, Cursor, etc.):** Spec-Kit supports multiple AI agents (Claude, Cursor, CodeBuddy, etc.) [30] [31], some of which have CLI tools. For example, Claude Code might have a CLI that can take the `.claude/commands/*.md` and produce outputs, Cursor has a `cursor-agent` CLI, etc. [7]. One idea was to use one of these *in place of Copilot* (since Copilot lacked a CLI). Essentially, we would initialize Spec-Kit with a different `--ai` (say, `--ai cursor-agent` or `--ai codex`) which uses a CLI we can control. If that CLI could be pointed to OpenAI's API (for Codex CLI, perhaps it uses OpenAI under the hood), we might achieve a similar outcome with less custom code. This approach has some merit:
- We could leverage an **existing integration** path in Spec-Kit, meaning less work to modify Spec-Kit's flow. For example, if the Codex CLI simply calls the OpenAI Codex (or GPT-3.5) models via an API, maybe we could set the `OPENAI_API_KEY_GHSPEC` for it. It might output code into files as needed.
- Using a CLI agent might inherently provide some iterative capabilities (depending on the CLI's design).

However, there are issues: - *Mismatch in agent behavior:* Claude or Codex CLI might not use GPT-4 or might have different prompting that deviates from Copilot's style. We want to specifically assess Copilot's spec-driven paradigm, not Claude's. For fidelity, using a different agent could confound results (was it the spec-driven process or the different model that caused a performance difference?). - *Support and maintenance:* This approach introduces another moving part – we'd depend on a third-party CLI tool (like Cursor's) being installed and functioning in our environment. This adds overhead in environment setup and potential brittleness. - *Token tracking:* Unless the CLI outputs token usage (unlikely), we'd still need to fall back to the usage API via time windows to measure tokens, just as in Option F. So no advantage there. - *Parallelism with ChatDev/BAEs:* These frameworks use GPT-4 (or similar) in our config, so ideally GHSpec should too. If Codex CLI uses an older model or Claude uses Anthropic's model, we break the equivalence. Some CLI (maybe "codex") could possibly be pointed to GPT-4, but that's speculative and not documented.

**Conclusion:** Using an alternate CLI agent was **not chosen** because it undermines the "Copilot" aspect of GHSpec and complicates consistency. We prefer to stick with GPT-4 and explicitly craft the prompts rather than trust another wrapper. Still, the existence of these CLI integrations in Spec-Kit gave us confidence that

*our approach is possible*: Spec-Kit essentially does what we plan to do (format instructions and call an AI) for other agents – we just have to implement it for the OpenAI API.

- **D. Template-Only or Simplified Execution:** By "template-only", we refer to using Spec-Kit's scaffolding without actually invoking an AI at runtime. For example, Spec-Kit's `specify init` creates a project with some preset files and perhaps partially filled templates for commands. One could imagine filling in those templates manually or using them as pseudo-output. Another variant: run the `specify` command to get a spec and plan, and then **skip using AI for implementation**, instead using pre-written solution code or trivial stubs to complete tasks (essentially a dummy run). This obviously would **not answer our research questions** – it would produce no meaningful data on AI performance. We considered this only as a fallback for testing infrastructure (indeed, the plan mentions a "MockAdapter" alternative to generate synthetic data quickly for testing [32]). The MockAdapter could simulate outputs and token counts just to validate our orchestrator. While this might be useful for internal testing (ensuring the multi-framework orchestration runs without waiting on real AI), it has **no scientific value** in the final experiment. We need the real AI behavior to compare autonomy, correctness, etc. So, template-only execution is not a valid approach for the actual experiment's data collection (though a "MockAdapter path" was noted as a way to save time during dev [32]).

**Conclusion:** Rejected for experiment purposes, except as a possible development aid (not delivering actual Copilot or AI behavior).

- **E. Full End-to-End Automation via Spec-Kit CLI (all phases in one go):** This means trying to use Spec-Kit's own commands to do everything non-interactively. For example: `spec-kit specify ...; spec-kit plan ...; spec-kit tasks; spec-kit implement --tasks all` all in one script. If we supply the `--ai copilot` argument at init, the `spec-kit implement` step might stall waiting for a developer to actually use Copilot in the IDE (since Copilot is IDE-based). So that doesn't work. But if we tried `--ai claude` or others that have CLI, it could, in theory, run end-to-end automatically. We've largely covered this under option C. Another twist: we could run `spec-kit implement --tasks 1-5` with `--ai cursor-agent` for example, and it might execute using the Cursor CLI to generate code for tasks 1-5 automatically [33] [34]. Some have tried such combinations (the Reddit threads and issues suggest people mixing Spec-Kit with Claude Code etc. in practice [35]). The problem again is divergence from the Copilot focus, and ensuring the model parity. Also, since we have specific incremental steps, we'd have to run `spec-kit implement` multiple times for different task subsets or new tasks after updating the spec, which the CLI wasn't clearly designed for (spec-kit is more geared toward one feature at a time, rather than sequentially adding features in one session, which is what our 6-step scenario requires).

**Conclusion:** This is similar to approach C and suffers the same drawbacks. Not pursued further.

**Why the Hybrid Approach (Option F*) is best:** After vetting all these, the Hybrid approach – i.e., using Spec-Kit's structured methodology but driving it with **our own OpenAI API calls and custom logic** – emerged as the best solution. It balances **fidelity** (we use the actual spec, plan, and task breakdown from Spec-Kit, preserving its intended workflow [36] [37]) with **control** (we decide how and when to call the LLM, and can instrument the process extensively). It's also **feasible**: We have direct access to the Spec-Kit repository [38] and can run its code (likely via subprocess calls to the CLI, or even by importing its Python API if possible, since Spec-Kit appears to be Python-based from the repo structure). We saw in the tasks breakdown that implementing the adapter mainly involves mimicking what we did for ChatDev but tailored

to Spec-Kit [39] [40]. This is work (estimated ~23-41 hours for GHSpec adapter tasks [41]), but it's within reason. In contrast, automating a real IDE or building an interceptor would be far more time (and risk) without any guarantee of success.

The **key improvement we make over the originally proposed Option F** is to incorporate the lessons from alternatives: we will design the GHSpec adapter to allow **multi-turn interactions (like ChatDev)** and to incorporate a **validation loop (like autonomous agents)**. In effect, our recommended approach is a **Hybrid++**: use the Spec-Kit structure (for fidelity) + OpenAI API (for control) + agent-like iteration (for parity and scientific validity).

Below, we outline this recommended approach in detail, including how exactly to implement it and ensure it meets all criteria.

## 4. Recommended Implementation Strategy ("Enhanced Hybrid Adapter")

The recommended strategy is to implement a **GHSpec Adapter** that uses Spec-Kit's process under the hood, but with our own orchestration to handle prompting and iteration. This approach will ensure:

- **Full control of API key & model:** All LLM calls use `OPENAI_API_KEY_GHSPEC` and target the `gpt-4o-mini` model (or our chosen experimental model) [2] [1]. We will not rely on any proprietary Copilot endpoints.
- **Exact token tracking:** We will wrap each step's AI calls between timestamps and use the OpenAI Usage API with our admin key to fetch token counts [3] [20]. This is already supported by BaseAdapter.
- **Scientific validity through parity:** The adapter will include mechanisms for **iteration, self-critique, and re-planning** analogous to ChatDev/BAEs. E.g., if GHSpec (the LLM) asks a clarification question, we will detect it and respond with the pre-set spec clarification (just as we do for ChatDev's HITL) [42] [43]. If the code fails a test, we will allow a fix attempt (like an autonomous agent would do).
- **High fidelity to Spec-Kit workflow:** We will use Spec-Kit's actual commands wherever possible (invoking `spec-kit specify`, `plan`, `tasks`) to generate the intermediate artifacts (spec document, plan markdown, task list). Then use these artifacts as context for code generation. We will emulate Copilot's behavior by generating code **task-by-task**, possibly file-by-file, rather than one giant monolithic completion, to match how a developer would implement each task with Copilot's assistance [9]. We'll even consider using the same *prompt templates* that Spec-Kit uses for agents like Claude, adapting them for our API calls, to ensure the AI is guided in the same manner.
- **Reproducibility, error handling, extensibility:** The design will prioritize deterministic execution (fixing seeds, using temperature 0 or very low) and robust error handling (timeouts, retries). If an API call fails or times out, we catch the exception and can retry up to 3 times as per config [44]. Logs will capture every action in JSON for transparency. The code structure (using the Adapter pattern) makes this solution extensible – e.g., we can swap out the model or tweak prompts easily, or even integrate a future Copilot official API if it becomes available, by just adjusting the call in one place.

Summarizing the approach: **GHSpec Adapter will run the Spec-Kit CLI to get the spec & plan, then for each step's requirement it will generate or update tasks, and use the OpenAI GPT-4 API to implement**

**each task in the codebase, verifying as it goes.** It's a marriage of Spec-Kit's strengths (structured thinking) with the control of direct LLM usage.

## 5. Full Implementation Plan

Below is a detailed implementation plan for the recommended approach, including phases, time estimates, strategies for prompts and API orchestration, and example pseudo-code. We break the plan into logical phases aligned with tasks (T083–T091) identified in the project plan [45] [40], and add additional steps for iterative refinement features.

### Phase 1: Research & Design (Est. 3-5 hours)

**Tasks:** - T083 – *Research Spec-Kit integration:* Familiarize with Spec-Kit's CLI and file structure for Copilot agent [7]. Identify relevant prompt templates (e.g., the contents of `.github/prompts/` or any mention of Copilot in Spec-Kit's code). Research how Spec-Kit expects a developer to use Copilot (possibly reviewing Spec-Kit docs/blog posts for Copilot usage tips). - **Outcome:** A clear understanding of what inputs/outputs the Spec-Kit CLI commands produce and how to invoke them non-interactively.

**Deliverables:** - Document (internal) outlining the Spec-Kit command usage and any discovered templates for Copilot. - Decision on how to call Spec-Kit CLI from Python (likely via `subprocess.run(["specify", ...])` or by invoking its Python API through `specify_cli` module if accessible).

### Phase 2: Environment Setup & Initialization (Est. 3-4 hours)

**Tasks:** - T084 – *Spec-Kit Environment Setup:* Implement `GHSpecAdapter.start()` to set up the environment [46] [47]. This includes: - Cloning the Spec-Kit repo (already in code) and verifying the commit [16] [48]. - Setting up a Python virtual environment in `framework_dir` (if Spec-Kit has dependencies). Since our orchestrator might run Spec-Kit commands, we need Spec-Kit's requirements installed. Possibly use `pip install .` inside the cloned spec-kit directory. - Ensure the Spec-Kit CLI (`specify` command) is on PATH or directly callable via Python. We might add the spec-kit's `src` to `sys.path` to use it as a library. - Load any configuration needed (though Spec-Kit likely reads from files in the project directory). - No "service" to start for Copilot (as it's not CLI-driven), so this phase is mostly installation. We will **not** start a UI or server (the `ui_port` might remain unused, or we could open a simple HTTP server to satisfy health checks if needed).

**Deliverables:** - Working implementation of `GHSpecAdapter.start()` that results in a ready-to-use spec-kit project workspace. All necessary tools (Spec-Kit CLI, etc.) should be installed in that workspace or globally accessible. Log success or raise errors appropriately.

**Code Example (pseudo):**

```
class GHSpecAdapter(BaseAdapter):
    def start(self):
        super().start()
        # Clone and checkout already done above...
```

```
        # Set up virtual env for spec-kit
        venv_path = self.framework_dir / ".venv"
        subprocess.run([sys.executable, "-m", "venv", str(venv_path)],
check=True)
        # Activate and install spec-kit (assuming a requirements.txt or use pip
install .)
        subprocess.run([str(venv_path / "bin" / "pip"), "install", "."],
cwd=self.framework_dir, check=True)
        logger.info("Spec-kit environment set up", extra={...})
```

*(We will refine this based on actual Spec-Kit setup instructions.)*

### Phase 3: Spec & Plan Generation (Est. 4-6 hours)

**Tasks:** - This is part of what might be called *execute_step for step 1, sub-step A*. We need to take the prompt for step 1 (e.g., "Create Student/Course/Teacher CRUD app…") and feed it to Spec-Kit to generate the spec and plan. - Likely use Spec-Kit's CLI: - Run `specify` command with the description. Spec-Kit might create a `spec.md` in the project directory. - Optionally run `speckit.constitution` if needed (the plan suggests they created a constitution manually in some examples, but we have a fixed one in our config possibly). Given focus, we might skip constitution for now, unless the absence confuses the AI. - Run `speckit.plan` with a predetermined tech stack. In the Spec-Kit example, they provided a stack via `/speckit.plan` command [49]. We have a known stack (Python/FastAPI/SQLite as per our scenario). We can supply that as input to plan. Possibly our `step_1.txt` already implies it ("with Python/FastAPI/SQLite"). - If not, we might need to call `spec-kit plan "<tech stack>"` explicitly. If our scenario doesn't specify tech stack, maybe Spec-Kit's AI chooses it, but that could introduce variability. Better to be explicit to align with what ChatDev will do (ChatDev likely was told the stack too). - Run `speckit.tasks` to generate the initial task list from spec+plan. - All these commands will use the AI (Claude/Copilot) under the hood. But since we have `--ai copilot`, this is tricky – Spec-Kit will expect an IDE and do nothing for `tasks` or `plan` without human? Actually, unclear: perhaps `spec-kit specify` and `spec-kit plan` themselves use AI via the chosen agent CLI/IDE integration. - Here we hit a challenge: With `--ai copilot`, `spec-kit specify` might open VS Code or wait for user input. One way around this: - **Use an AI that has a CLI for the spec and plan phases.** We could cheat by initializing the project with `--ai claude` just for spec and plan generation, since those are pure text and Claude Code CLI could handle it automatically. Then switch to our own implementation for coding tasks. - Or we directly call the OpenAI API for spec and plan as well, constructing the prompts as Spec-Kit would. Actually, since we want to measure GHSpec end-to-end, maybe we *should* involve the model in spec and plan too (not have a predetermined plan). So yes, use GPT-4 API to generate the spec and plan. - Therefore, implement *our own version of speckit.specify and speckit.plan via API.* We can use prompts gleaned from Spec-Kit's templates for these phases. This ensures consistency (all AI work goes through our controlled model). - This essentially means not relying on `spec-kit specify` command at runtime, but rather using the prompt templates. However, generating a full spec or plan might exceed context and time; but GPT-4 should handle a spec generation. - Alternatively, if time allows, use `cursor-agent` (which presumably is a CLI hooking to GPT-4 or a similar model) as the agent for spec generation since that can run headlessly.

Considering complexity, perhaps simplest: **Use our API to do spec and plan generation, guided by Spec-Kit's approach.** We can store the results in the same files to keep everything uniform.

- In summary, the adapter's first call to `execute_step(1, prompt)` will:
- Call OpenAI API with a prompt like: "You are an AI coding assistant following the Spec-Driven Development approach. **Task**: Generate a detailed software specification for the following feature: `<step_1 description>`." – plus maybe some instructions (pulling from how Spec-Kit structures spec writing).
- Save the AI's response as `spec.md`.
- Then call OpenAI API with prompt: "Now generate a technical implementation plan given the spec and the following project constraints (stack, architecture): ..." – supply stack from scenario or default stack (we might say Python/FastAPI as desired).
- Save response as `plan.md`.
- Finally, call OpenAI to generate tasks: "List the implementation tasks required to implement the specification according to the plan, in step-by-step detail." Possibly use a format similar to Spec-Kit's tasks.md (they include IDs and phases, but we can simplify to bullet points or a checklist of tasks).
- Parse the response, format it into a `tasks.md` or a Python list of tasks in memory.

By doing steps 1–5, we effectively replicate `spec-kit specify`, `plan`, `tasks` using our model. The risk is that our model's output might differ from Spec-Kit's official output (which could impact fidelity), but since GPT-4 is quite capable, it should produce similar content if prompted well. We can cite the Spec-Kit blog's description to craft these prompts (ensuring the AI covers user journeys, etc., in spec, and covers tech design in plan) [50] [51].

**Time estimate note:** This phase is not trivial; spec generation could be complex. But given we only need to do it once per run, and mostly format the outputs, 4-6 hours of coding and testing seems reasonable.

**Deliverables:** - Code to generate spec, plan, tasks via OpenAI API (for step 1). Possibly integrated into `execute_step` when `step_num==1`. - Saved `spec.md`, `plan.md`, `tasks.md` in the workspace for record-keeping (and potential reuse when adding features). - Logging of these artifacts for traceability (maybe log the spec and plan text in orchestrator logs or archive).

**Example (pseudo-code for spec generation):**

```
if step_num == 1:
    # Step 1: Generate spec using API
    spec_prompt = (
       "You are an expert software spec writer. Write a detailed specification
for the project described below.\n"
       f"Description: {command_text}\n\n"
       "Include user stories, acceptance criteria, and any assumptions.")
    spec_response = openai.ChatCompletion.create(model=self.model,
messages=[...], temperature=0.2)
    spec_md = spec_response['choices'][0]['message']['content']
    (self.framework_dir / "specs/001-baes-experiment-framework/
spec.md").write_text(spec_md)
```

```
    # Similarly plan:
    plan_prompt = (
       "Based on the above specification, produce a technical implementation
plan. "
       "Specify tech stack (use Python 3.11, FastAPI, SQLite as given),
architecture, and steps.")
    plan_response = openai.ChatCompletion.create(model=self.model, ...)
    plan_md = plan_response.content
    ... write plan.md ...
    # Tasks:
    tasks_prompt = "Now list all implementation tasks as bullet points, broken
down by feature or phase..."
    tasks_resp = openai.ChatCompletion.create(...)
    tasks_list = parse_tasks(tasks_resp.content)
    save_to_tasks_md(tasks_list)
```

*(The actual prompt will be more structured, and we'll include spec & plan content in the messages for tasks generation.)*

### Phase 4: Task Execution & API Orchestration (Est. 8-12 hours)

**Tasks:** - T086 – *Spec-kit Command Execution:* This is the core – executing each step's tasks via the AI. We break it into sub-steps: - **Task Parsing:** Take the `tasks.md` (or tasks list from Phase 3) and parse out individual tasks. If tasks are phrased like "Implement X in file Y", extract that info. We may have a list of tasks like: 1. Create data models for Student, Course, Teacher in `models.py`. 2. Implement CRUD API endpoints using FastAPI for each model in `main.py`. 3. ... etc. We can identify for each task which files or components it affects (maybe by convention or by parsing if the task description mentions a file path [52]). If not explicitly given, we may have to infer or just open relevant files generically. - **Orchestration of LLM calls per task:** For each task, prepare a prompt to generate the needed code. This prompt should provide *context* and *instructions*. For example: - Include relevant parts of the spec and plan that relate to this task (maybe the whole spec/plan, or a summary, to ensure the model doesn't go of track). - Include the current code (if any) in the file that needs to be modified. If creating a new file, maybe provide an empty template or import statements. - Instruction: "Write the code for task: ... Ensure it integrates with the existing codebase. Only modify the specified file(s)." - Perhaps ask for the full content of the file or a diff patch as output. A diff might be safer to apply onto current code, but parsing diffs can be complex. Outputting full file content is easier to write to disk (overwrite). - We must be cautious about context window – by step 6, the project codebase might be big. We might not feed all files, only those relevant to the task (which is why tasks should ideally mention what to work on). - If context is too large, summarization strategies or splitting tasks further might be needed, but since tasks are granular, hopefully each touches at most one or two files, so context remains manageable.

- **Applying the AI output to the codebase:** Once the AI returns code for a task, write it to the appropriate file(s) in the workspace. If it's supposed to modify an existing file, we can either replace the file with the new content from AI (assuming the AI was given the old content, it should return the updated version), or intelligently merge the changes. Simpler path: trust the AI to output the whole updated file content (this is the approach that the Cursor editor model and others use – they prompt

the AI with a file and an instruction, and get a full new file). We'll implement that approach for reliability.

- **Iteration & self-check per task (micro-level):** Suppose the AI output is obviously incomplete for a task (e.g., it says "...continue in next message" or it outputs an error). We should detect that. If it's a truncation (rare with GPT-4 at moderate sizes, but possible), we can prompt again or with a hint "you stopped mid-code, please continue." This is a low-level fix. Also, if we have any *static analysis* or quick checks, we could run them after each task (e.g., does the code compile? if we had a quick way to syntax-check Python, we could do that). However, doing full tests after each task might be overkill and time-consuming. Possibly better to wait until the entire step's tasks are done, then run tests. That's what Spec-Kit hints at: "Verify Frequently: Test each implementation before moving forward" [9], but in practice, running a full test suite after every small task might slow things too much. We could compromise by at least ensuring the server starts or basic syntax is correct after each task (maybe by running `pytest --collect-only` or something lightweight).

- **Combine multiple tasks into one API call?** Spec-Kit CLI has `spec-kit implement --tasks 1-5` example [33], suggesting it might do a batch of tasks in one go (maybe for parallel or just convenience). For us, it's safer to do one at a time sequentially, because the model focusing on one task at a time is likely to be more accurate and easier to troubleshoot. We will implement sequential single-task prompts by default. If time allows, we could identify tasks marked `[P]` (parallelizable) [53] and possibly do them in parallel threads or as a single combined prompt (but that complicates token tracking and ordering, so probably skip parallel execution – sequential is fine since within one framework run there's no concurrency anyway as per design [54]).

**Deliverables:** - Implementation of `execute_step(step_num, command_text)` in GHSpecAdapter that: - On step 1: runs Phase 3 (spec, plan, tasks generation). - On subsequent steps: reads the new requirement (from `command_text` which orchestrator passes in) and updates the spec/plan or directly generates new tasks. - This is tricky: We need to incorporate the new requirement into the existing spec. Two ways: 1. **Spec augmentation approach:** Append the step description to the spec (e.g., add a new section "Feature X:" in spec.md) and possibly run the `/tasks` command again or just manually generate tasks for that feature via API (similar to how we did for step1 tasks). This keeps spec as a living document. 2. **Direct tasking approach:** For incremental features, perhaps skip directly to tasks: feed the model the spec.md (so it knows context of current system) + plan.md (if architecture changes) + the new feature description, and prompt: "Given the existing project and the new requirement, list the tasks to implement this new requirement." This could yield tasks focused only on the delta. This might be more efficient than re-planning everything. - We will likely implement option (b) – generate incremental tasks via API. This keeps the process similar (we use AI to break down the feature) but doesn't require re-running the entire planning phase. - Ensure tasks are clearly separate from old ones (maybe use a new tasks file or append labeled as Phase 2 tasks in tasks.md). - Executes tasks as above, applying changes. - At end, returns a result dict including success status, duration, hitl_count (clarifications asked), tokens used (which we'll fill via token counting), etc. We may also include a `retry_count` (the base adapter expects that) which for GHSpec might usually be 0 unless we internally retried anything.

- A mechanism to detect HITL (Human-In-The-Loop) prompts from the model. E.g., if the model says "I need clarification on X", or poses a question about requirements, we treat that like ChatDev's clarification queries. Our plan: we have a fixed **clarification text** (in `expanded_spec.txt`) which presumably is a detailed spec covering all ambiguities [43]. If the model asks anything, we log it and

respond with that text (same as ChatDev) [55] [56]. This makes execution deterministic and fair (the model doesn't really get to diverge; we provide a consistent answer).

- Implementation: maybe scan the model's message for `?` or certain phrases after initial spec generation. However, since we ourselves generate the spec from a presumably complete description, the model *asking for clarification* might be rare. But it could happen on later steps ("Should I implement X this way or that?"). We will monitor outputs. If detected, call `handle_hitl(query)` which returns the fixed expanded spec (which likely reiterates requirements in detail) [43]. Then include that info in the context and continue the generation.
- Logging & token counting around each API call. For each call, record start time in `self._step_start_time` [57], and after completion, use `self.fetch_usage_from_openai(...)` to get token counts [20]. Sum up across tasks if multiple calls per step (or perhaps call it after each sub-call and accumulate). In the result, set `tokens_in` and `tokens_out`.

**Time complexity:** This is the largest phase, implementing logic, testing on a small scenario, adjusting prompts. It might take a full day or more (8-12 hours).

**Code Snippet (high-level):**

```
def execute_step(self, step_num: int, command_text: str):
    self.current_step = step_num
    self._step_start_time = int(time.time())
    hitl_count = 0
    try:
        if step_num == 1:
            # Phase 3: generate initial spec, plan, tasks
            generate_spec_plan_tasks(command_text)
            tasks = parse_tasks_file()
        else:
            # Incorporate new requirement
            append_to_spec_if_needed(command_text)
            tasks = generate_tasks_for_new_feature(command_text)
        for task in tasks:
            logger.info(f"Executing task: {task['description'][:50]}...",
extra={...})
            # Prepare prompt with context
            prompt = build_prompt_for_task(task, spec_file, plan_file,
current_codebase)
            response = call_openai_api(prompt)
            if asks_for_clarification(response):
                hitl_count += 1
                clarification = self.handle_hitl(response)
                # Re-call API with clarification added
                prompt_with_clarification = prompt + "\nAdditional info: " +
clarification
                response = call_openai_api(prompt_with_clarification)
            apply_response_to_codebase(response, task)
```

```
        # Run validations for this step (API/UI tests via validator module)
        validation_results = self.run_validations(step_num)
        success = validation_results["all_passed"]
        # If not success, we could decide to attempt auto-fix (replanning)
        if not success:
            # prepare bugfix tasks or directly attempt fix
            bug_tasks =
analyze_failures_and_create_fix_tasks(validation_results)
            for btask in bug_tasks:
                # similar loop as above
                ...
            success = self.run_validations(step_num)["all_passed"]
        # Token usage via usage API:
        tokens_in, tokens_out = self.fetch_usage_from_openai('OPEN_AI_KEY_ADM',
self._step_start_time, model=self.config.get('model'))
        return {
            "success": success,
            "duration_seconds": time.time() - self._step_start_time,
            "hitl_count": hitl_count,
            "tokens_in": tokens_in,
            "tokens_out": tokens_out,
            "retry_count": 0  # framework internal retries not counted here
        }
    except Exception as e:
        # handle exceptions, possibly similar structure
        ...
        raise
```

The above pseudo-code highlights how we incorporate HITL, tasks loop, validations, and even an auto-bugfix attempt. Some parts like `run_validations` and `analyze_failures_and_create_fix_tasks` we have to implement: - `run_validations(step_num)`: likely calls our existing `Validator` on the workspace for that step. The orchestrator might also call the validator outside the adapter, but in design it says *adapter returns results and orchestrator validates each step* [26] [27]. Actually, the Orchestrator might handle calling validation after each step and merge it into metrics. So maybe adapter doesn't need to call validation, just leave that to orchestrator. We should confirm orchestrator flow: It likely does:

```
result = adapter.execute_step(n, prompt_n)
validation = validator.validate_step(n)
metrics = collector.collect(..., validation, api_usage, etc.)
```

So, we might not embed validation in adapter; we can rely on orchestrator to handle it. In that case, the adapter's `success` can simply reflect whether the step "ran to completion" from the AI perspective (perhaps always True unless it crashed, because even if tests fail, the step technically executed and yielded code – orchestrator will catch correctness issues via validation). Possibly we define `success` in adapter context as "the framework produced some output without critical errors". ChatDev's adapter likely sets success based on process exit codes [25]. Since our GHSpec doesn't have a single process exit code, we can

treat it as success if no exception and we produced code. We'll let validation judge the quality. - The metric AUTR (Autonomy) likely counts whether it needed HITL or human help. We record `hitl_count` for that (each clarification increments it). - If we want the adapter to attempt a bug fix, that's extra autonomy which could be seen as part of the framework's work (and in fairness, ChatDev/BAEs do it internally). Implementing that inside adapter is plausible. But orchestrator's retry logic (T022) might be simpler: it can just treat a test failure as a step failure and *re-run execute_step* (maybe with the same prompt) up to 2 retries [28]. That would re-generate code from scratch though, not likely to magically fix anything unless randomness helps. A smarter approach is to do targeted fixes as above with context of failure – that requires inside knowledge (which only the adapter or a specialized agent can do). - ChatDev presumably would incorporate test failures into its conversation and try again *within* the step. BAEs might do similar. So yes, arguably GHSpec adapter should handle it *within* execute_step rather than rely on orchestrator's blind retry. - We will do as pseudo-code shows: if validation fails, gather failure info (maybe the assertion messages or error logs) and prompt the model: "Given this spec and current code, the following test failed: [error]. Please fix the code to address this." This is essentially a `/speckit.bugfix` command we implement ourselves. - We have to be careful not to enter an infinite loop. We can limit fix attempts (say 1 fix attempt per step, to mirror ChatDev's 2 clarification limit [17] or orchestrator's 2 retries). - If after a fix attempt tests still fail, we mark the step as failed (or let orchestrator handle final failure).

**Phase 4 deliverables:** - Completed `execute_step` logic with subroutines for prompting and applying code. - Verified on a simple example repository (maybe create a dummy spec and see if it writes files). - Unit tests for parsing tasks and integrating output.

### Phase 5: Health Check & Shutdown (Est. 1-2 hours)

**Tasks:** - T089 – *Spec-kit Health Checks:* Implement `health_check()` to verify the framework is "up" [58]. In our case, since there's no persistent service, health check can be trivial: check that the spec-kit workspace exists and perhaps that the CLI is callable. Or always return True after start since everything runs in-process. We might simulate a health endpoint: maybe after start, we could start a minimal HTTP server just to satisfy an HTTP check (but that's overkill). Simpler: we know orchestrator will call `health_check()` periodically during a step to ensure the adapter hasn't hung. We can implement it to return True if the current process is alive and no issues flagged. Since GHSpec's work is done within the `execute_step` call itself (which is blocking), we might not need complex health checks. Just ensure `health_check()` returns True whenever called (or perhaps check that the `workspace_path` is still present as a trivial check). - T090 – *Spec-kit Graceful Shutdown:* Implement `stop()` to clean up [59]. Since we have no long-running subprocess (except maybe the Spec-Kit venv or any threads), there's not much to kill. We should: - If we installed a venv or any background processes (none likely, unless we launched VS Code which we are not), we terminate them. - Perhaps remove big objects or temp files if needed. But likely nothing beyond what BaseAdapter's isolation cleanup will handle outside. - Mark in logs that spec-kit framework stopped [60]. - These are straightforward, minimal implementations.

**Deliverables:** - `health_check()` returns True if `.framework_dir` exists and maybe a quick self test (optional). - `stop()` ensures no lingering processes (if none, then just log and pass).

### Phase 6: Testing & Validation (Est. 4-6 hours)

**Tasks:** - T091 – *Spec-kit Integration Testing:* Create tests to simulate a single-step run or a full run through all 6 steps [40]. Since doing all 6 with actual AI could be slow, an integration test might run just 1-2 steps with a

very trivial scenario to see that the adapter produces output and that token counting works, etc. - Specifically, we can write a test that uses a **small prompt** (like “Create a hello world app”) and run GHSpec adapter through `execute_step(1, "...") ... execute_step(6, "...")` and verify: - spec.md and plan.md were created. - Some code files were created in workspace (like a main.py). - `metrics.json` has non-zero tokens for GHSpec run. - No exceptions or timeouts. - Additionally, test that `handle_hitl` indeed returns the expected fixed text from `expanded_spec.txt` when called (we might simulate a query string). - Possibly, test the scenario where we intentionally force a validation failure and see if our bugfix logic kicks in. For example, if step 4 is “Add input validation” and we know the AI might not do it correctly initially, see if after running tests our adapter tries a fix. - Ensure determinism for testing by maybe stubbing the openai API calls with a fake that returns preset strings (for unit tests). For integration, one can call the real API but that’s slower and depends on API availability – might not be suitable in automated tests.

**Deliverables:** - Test logs or artifacts demonstrating a successful run. - Possibly adjust any prompt or logic that misbehaves based on test results (e.g., if the model output was not applied correctly, fix that).

**Note on Estimated Hours:** Summing up phases: - Phase 1: ~4h, Phase 2: ~4h, Phase 3: ~5h, Phase 4: ~10h, Phase 5: ~2h, Phase 6: ~5h. Total ~30 hours (which fits the earlier estimate of 23-41 hours for T083-T091 [61]). We should allocate some buffer for prompt tuning, so roughly **35 hours** of implementation and testing for the GHSpec adapter.

## Prompt Parsing & Transformation Strategy

This deserves special focus, as prompt design is critical:

- **Spec & Plan Prompts:** We will base these on the Spec-Kit’s methodology. For specification, instruct the model in a way similar to how Spec-Kit’s philosophy is described: focus on *what and why*, not how [50]. For plan, focus on technical choices and constraints [51]. We might even include bullet points “Remember to include: [list from blog: user journeys, success criteria, etc.]” to guide the model. These prompts essentially transform the user’s one-line request (our step description) into a multi-page spec and a structured plan, respectively. By doing this via the API, we’re translating the natural language prompt into the richer context that Copilot (with Spec-Kit) would have had.
- **Tasks Prompt:** Similarly, the tasks breakdown prompt will transform the plan into actionable steps [37]. We may feed the entire plan text to the model with an instruction: “Produce an ordered list of development tasks needed to implement the above plan. Each task should be clear and implementable.” Possibly include guidelines like Spec-Kit’s (MVP first, etc.) [62] [63].
- **Code Generation Prompt (per task):** This is the most frequent prompt and must be carefully structured:
- **System message** (if using ChatCompletion): We can set a system role that says: “You are GitHub Copilot, a coding assistant. You will be provided with a software specification, technical plan, and current code context. Follow the task instructions precisely, producing correct and concise code. Use the project’s style and frameworks as described.” This helps steer the model to behave like Copilot.
- **User message**: Contain context:
    - The spec (possibly truncated or summarized if very long).
    - The plan (or relevant portion for this task).
    - The specific task description.
    - The content of the file to be modified, if it exists (maybe only the part relevant; or entire file if small; or class/function signatures).

    - Perhaps an explicit instruction: “Implement the above task. If modifying existing code, incorporate seamlessly. If new, create the necessary code. Provide the full updated code for the file(s) in markdown blocks.” We might ask it to output markdown with filenames, e.g., "`python\n# File: models.py\n<code>`" if multiple files. But simpler: one task likely one file – we can constrain it: “All changes are in `X.py`, provide the full content of `X.py` after implementing the task.”
- **Temperature**: use 0 (for determinism). Possibly a slight raise (0.2) for creative tasks like spec writing, but for coding, 0 to ensure consistency and no flakiness.
- We will also include a guard: “Do not stray from the task. If information is missing, *assume* a reasonable solution (do not ask questions).” This is to avoid HITL triggers unless absolutely necessary. Copilot rarely asks for clarification; it usually just guesses. We want our model to do the same unless it’s truly ambiguous.

- **Clarification Handling:** If the model still asks a question, `handle_hitl` provides the entire expanded spec (which likely answers most questions as it’s a comprehensive requirement doc) [43]. This is inserted and the task prompt re-run, ideally yielding an answer. We count that as a HITL event.

- **Bugfix Prompt:** If tests fail, we create something like:

- Context: spec + failing test details + relevant code snippet that failed (if available).
- Instruction: “The implementation above does not pass all tests (details given). Identify the issue and fix the code. Provide the updated code.” This essentially instructs the model to be a debugger and patch generator.
- We might run this at a high temperature or allow more tokens, as debugging can be hard. But GPT-4 should manage at temp 0 too if the info is clear.

Throughout, we will transform internal representations (like a Python list of tasks) into human-language prompts and vice versa. E.g., after tasks generation, parse the text list into a Python list for easier iteration.

We should also emphasize **extensibility**: our prompting strategy should be data-driven as much as possible (maybe stored templates in a JSON or a file) so we can tweak prompts without changing code. But given time, hardcoding in code with clear structure is acceptable.

### API Orchestration Logic

Our use of the OpenAI API will involve: - The **ChatCompletion** endpoint (preferred, as we can use system messages and better control conversation style). We’ll use `gpt-4` (or `gpt-4-0613` if specific version, “o-mini” likely alias for an OpenAI mini model accessible). - The adapter will orchestrate multiple calls in a sequence (spec -> plan -> tasks -> multiple implement calls). We must ensure to **respect rate limits** or pacing – e.g., not blasting too fast. But with sequential calls and small project, likely fine. - **Iteration**: If a call fails (API error or timeout), we have retry logic from config (3 attempts) [64]. We will wrap each call with a retry loop (using either our own or use the orchestrator’s global retry by raising an error – but better to catch inside adapter so we can log partial progress). - We will accumulate the tokens from each call in the step and sum them, or simply rely on usage API at step end to give us the sum (our time window covers the whole step execution, which spans multiple calls, and since no concurrent frameworks, that usage query will accurately sum all calls in that window [65] [66]). - For **self-critique**: beyond tests, we might incorporate the model’s own feedback. Perhaps after generating code, ask the model “Do you see any potential issues with the above code relative to the spec?” as a separate call. However, that doubles calls and might not yield

much beyond what tests catch. Given time, we skip explicit self-critique prompts; we rely on actual test results to drive fixes (which is more concrete). - **Parallel vs Sequential**: We choose sequential code generation to avoid complexity. The orchestrator anyway runs frameworks sequentially, and within GHSpec, tasks sequentially. This is simpler and aligns with how a single developer would code tasks one after another. - **Monitoring**: We'll use logging at each significant point (task started, API call made, etc.) with run_id and step info, so that we can trace the process later in logs [67]. This also helps if something hangs – though since our calls are synchronous, less risk of stray background processes.

### Code Examples for Key Components

- **Template Extraction:** For example, reading the expanded spec file for HITL:

```
if self.hitl_text is None:
    hitl_path = Path("config/hitl/expanded_spec.txt")
    self.hitl_text = hitl_path.read_text(encoding='utf-8').strip()
```

This mirrors the code in adapter [43]. We already have that stub.

- **Prompt construction:** A snippet for building a prompt for a coding task might look like:

```
def build_prompt_for_task(task, spec_text, plan_text):
    file = task.get('file') or infer_file_from_task(task['description'])
    current_code = Path(file).read_text() if Path(file).exists() else ""
    user_prompt = f"""Spec:\n{spec_text}\n\nPlan:\n{plan_text}\n
    You are working on task: "{task['description']}".
    File: {file}\nCurrent content:\n```\n{current_code}\n```\n
    Instruction: {task['description']}. Implement this in the above
file."""
    return [{"role": "user", "content": user_prompt}]
```

And then call `openai.ChatCompletion.create(messages=prompt, ...)`. (We'd actually supply system message separately, etc.)

- **Error handling example:** After getting `response`, if `response['choices'][0]['message']['content']` is empty or indicates an error:

```
content = response_msg.strip()
if not content:
    logger.warning("Empty response for task, retrying with higher
temperature...", extra={...})
    # maybe retry with temp 0.5 or break tasks further
```

Or if `asks_for_clarification(response_msg)`:

```
if "?" in response_msg and any(kw in response_msg.lower() for kw in ["not
sure", "unclear", "what is"]) :
    clarification_text = self.handle_hitl(response_msg)
    hitl_count += 1
    prompt.append({"role": "assistant", "content": response_msg})
    prompt.append({"role": "user", "content": f"Clarification:
{clarification_text}"})
    response = openai.ChatCompletion.create(messages=prompt, ...)
    content = response['choices'][0]['message']['content']
```

This effectively provides the answer and continues the conversation (we include the model's question and a user answer).

- **Applying code to file**: If `content` contains markdown, we might need to parse out the code part. We can write a small parser that finds triple backtick sections and file names if present. Or simpler: if we prompted one file at a time, we can assume `content` is exactly the file text. Then:

```
with open(file, 'w') as f:
    f.write(content)
```

If multiple files in one output (less likely with our approach), parse accordingly.

- **Token usage**: Already via `fetch_usage_from_openai` which we'll call like:

```
tokens_in, tokens_out = self.fetch_usage_from_openai(
    api_key_env_var='OPEN_AI_KEY_ADM',
    start_timestamp=self._step_start_time,
    end_timestamp=int(time.time()),
    model=self.config.get('model')
)
logger.info(f"GHSpec step {step_num} used tokens: in={tokens_in},
out={tokens_out}", extra={...})
```

This uses the usage API which sums across calls [68] [69]. No need for parsing logs like ChatDev does [25].

By providing these code structures, we ensure clarity on how each piece will be implemented.

## Scientific Framing in Paper

When writing about this method in a paper or report, we should emphasize:

- **What is being measured:** We will state that we measure the performance of a **Spec-Driven AI coding approach (GHSpec)** versus other autonomous coding agents on a controlled software evolution task. We measure multiple dimensions: code correctness/quality (via our CRUD and UI tests, CRUDe and ESR metrics), efficiency (tokens, time), and *autonomy* (how much human intervention or hints were needed – ideally none, since we supply a fixed spec and clarifications). We specifically look at whether a structured, spec-first approach yields better or worse outcomes than agentic planning or human-like dialogues (ChatDev/BAEs).
- **Method description:** "We integrate GitHub's Spec-Kit with a custom adapter that uses the OpenAI GPT-4 model to emulate GitHub Copilot's role in spec-driven development [7]. This adapter executes the Spec-Kit workflow fully automatically: generating a formal specification from requirements, a technical plan, and a breakdown of coding tasks, then implementing those tasks sequentially [70] [37]. At each stage, the AI has access to the evolving codebase and specification, ensuring context alignment. We intercept any clarification questions the AI raises and feed a predetermined comprehensive answer to maintain determinism (thus, no human input during runs) [43]. We track tokens and time for each step to compare efficiency."
- **Limitations:** We should note that *this approach does not use the actual GitHub Copilot service*, but rather an approximation via OpenAI's API. This was necessary for experimental control, but it means our GHSpec agent may not capture some proprietary behaviors of Copilot. However, since we use the same GPT-4 model that presumably powers Copilot, and the same prompts a Copilot-based Spec-Kit workflow would use, we expect the outcomes to be comparable in nature [71] [72]. Another limitation: The spec and plan quality depends on the AI's ability – in practice, a human might iteratively refine the spec or plan if they were suboptimal, but our automation currently does a single pass. This could disadvantage the GHSpec approach if the initial spec/plan has blind spots. We mitigate this by using a strong model and providing as much context as possible.
- **Threats to Validity:**
- *Internal validity:* There is a risk that errors in our adapter (e.g., prompt design or application of code) could affect GHSpec's performance. We addressed this with extensive testing and by using standardized templates from Spec-Kit where possible. Still, our emulation may not be perfect – for example, the AI might sometimes produce code that a real Copilot (with more training on GitHub repos) would avoid or vice versa. We acknowledge this and treat our GHSpec results as reflecting an "idealized Copilot-driven process" under controlled conditions.
- *External validity:* Our findings apply to the specific scenario and frameworks tested. The GHSpec approach might perform differently on other tasks or scales. Also, we assume a scenario with clearly defined requirements; in real development, requirements might be fuzzier and a human would iteratively clarify them with the AI – our deterministic setup doesn't fully replicate that interactive nuance.
- *Construct validity:* We measure "autonomy" by counting how often the AI needed additional information (which we limit to fixed clarifications). Since we provide the same clarifications to each framework when they ask, we believe the comparison is fair. If GHSpec rarely asks questions due to its upfront spec, whereas ChatDev might discuss more, we interpret that as GHSpec being more autonomous (needing fewer clarification cycles) – but one could argue the metrics need careful interpretation. We clarify that autonomy (AUTR) in our context doesn't mean better or worse quality by itself.

- *Reproducibility:* We will note that all frameworks were run multiple times and exhibited low variance in outputs due to deterministic settings (especially GHSpec and ChatDev with fixed random seeds and our careful control) [15]. Where nondeterminism existed (e.g., BAEs if it has some randomness), we ran sufficient trials and used statistical tests to ensure differences are significant [73] [74].
- **Comparative framing:** We'll describe GHSpec (Spec-driven, single-agent) vs ChatDev (multi-agent collaborative) vs BAEs (whatever its approach is, likely single-agent iterative). This situates our contribution in exploring structured planning vs emergent planning in AI coding.

Including such discussion in the paper ensures readers understand the context and limitations of our GHSpec adapter results.

## Evaluation Matrix

Finally, we present a matrix comparing the **Hybrid Approach (Option F)** and our **Enhanced GHSpec Adapter (Recommended)** against other alternatives for key criteria:

Let's compare three columns: - **Option F (Original Hybrid)** – as initially proposed (API-driven, but perhaps without iterative enhancements). - **Enhanced Hybrid (Recommended)** – our improved design as described. - **Full Copilot IDE Automation** – a representative alternative for fidelity (just to illustrate why we didn't pick it).

| Criteria | Option F: Basic Hybrid (API + Spec-Kit) | Enhanced Hybrid (Recommended) | Actual Copilot IDE Automation |
|---|---|---|---|
| **API Control (Key & Model)** | **High** – Uses our API key and chosen model, avoiding external dependencies [1]. All calls are under experimenter control. | **High** – Same as Option F. Additionally, we ensure the *exact same model* as other frameworks (`gpt-4o-mini`), making comparison fair [2]. | **Low** – Relies on Copilot's service using its own API and model. Cannot specify model version or guarantee same parameters. No access to raw API; would violate control. |
| **Token Tracking Accuracy** | **High** – Each API call can be logged; usage API gives aggregate tokens [3] [4]. No reliance on parsing logs (which for Copilot aren't available) – this was a key motivation for Option F. | **High** – Unchanged: we use the Usage API to sum tokens per step [20] [21]. Every prompt and response token is counted. We also record tokens per task internally for finer analysis. | **None/Low** – Copilot doesn't expose token counts. We could only roughly estimate based on context size. No reliable way to measure tokens used by Copilot suggestions. |

| Criteria | Option F: Basic Hybrid (API + Spec-Kit) | Enhanced Hybrid (Recommended) | Actual Copilot IDE Automation |
|---|---|---|---|
| **Scientific Validity** (comparable methodology) | **Moderate** – Follows the spec-plan-code paradigm, but *original Option F lacked iterative repair*. Could lead to GHSpec failing tests that others fix, skewing results. Also potentially less dialog (no multi-turn) than ChatDev, affecting autonomy metric interpretation. | **High** – Incorporates self-critique and repair loops, aligning with autonomous agent behavior (multi-turn if needed). All frameworks can clarify and retry, so GHSpec now can too, making comparisons apples-to-apples. We preserve the structured workflow while adding these agent-like capabilities. | **Moderate** – It would reflect actual Copilot usage, but measurement issues and inability to enforce identical conditions hurt validity. E.g., Copilot might use a more powerful model or context length, giving unfair advantage unless we treat differences qualitatively rather than quantitatively. |
| **Implementation Feasibility** | **High** – Requires coding the adapter but within reach (estimated ~30-40 hours [61]). Leverages existing Spec-Kit logic and OpenAI API, which we are familiar with. No need for unsupported tools. | **High** – Slightly more complex than Option F but still feasible. We add validation handling and prompt engineering – well within our team's expertise. The plan accounts for ~35 hours, which is manageable. The design reuses patterns from ChatDev adapter (HITL handling, etc.) [17] [24]. | **Low** – Very difficult. Setting up a headless VS Code with Copilot, or automating UI, is time-intensive and brittle. High chance of failures during runs. Would likely exceed our time budget and still not guarantee consistent results. |

| Criteria | Option F: Basic Hybrid (API + Spec-Kit) | Enhanced Hybrid (Recommended) | Actual Copilot IDE Automation |
|---|---|---|---|
| **Fidelity to Spec-Kit/Copilot Workflow** | **Moderate** – Adheres to Spec-Kit's structure (spec, plan, tasks), but actual code generation is via direct API calls in one-shot manner, not exactly how Copilot suggests in IDE. Might ignore some Copilot quirks (like partial suggestions, continuous integration into editor). However, uses the same high-level steps a Copilot user would, so broadly faithful. | **Moderate** – We maintain fidelity in overall process. Additionally, by using Spec-Kit templates and providing file-by-file context, we simulate how Copilot would fill in code in an IDE. Still not 100% (no real-time IDE events), but the model's output is expected to be similar to Copilot's completion if given the same context. The structured approach of Spec-Kit is fully preserved [6]. | **Highest** – This would be literally using Copilot, so by definition it's faithful. It captures Copilot's actual suggestion timing, style, and any hidden optimizations. If fidelity were the *only* concern, this wins. But it fails in other aspects (control, measurement). |
| **Reproducibility** | **High** – Every run can be made deterministic (same prompts yield same outputs with temperature 0). External factors minimized. We can run multiple trials easily. | **High** – Same as Option F; plus we fix seeds and have commit-hash pinned Spec-Kit [16] [48]. Even the expanded spec for HITL is fixed, ensuring identical clarification each time. Randomness is minimized. Any remaining nondeterminism (e.g., minor variations in language) can be countered by multiple runs, but likely negligible. | **Low** – Copilot suggestions can vary. We cannot set a seed. Also, if GitHub updates the model or context window changes, results differ over time. Running the experiment later might produce different code. Hard to isolate variance vs true performance differences. |

| Criteria | Option F: Basic Hybrid (API + Spec-Kit) | Enhanced Hybrid (Recommended) | Actual Copilot IDE Automation |
|---|---|---|---|
| **Code Quality & Correctness** | *(This depends on AI performance, but as an approach metric:)* Possibly **Moderate** – The code quality will hinge on GPT-4's ability to follow spec. Without iterative fixes, initial errors might remain, affecting quality metrics. But structured spec might guide model to produce higher-quality code upfront (compared to ad-hoc prompting). | **High** – By enabling an iteration to fix bugs, final code quality should improve (fewer failing tests, higher CRUDe coverage). The structured approach likely yields organized code (Spec-Kit enforces good project structure and documentation) [75] [76] . We anticipate this approach to score well on maintainability and completeness, given spec-driven nature, with minimal human input. | **Variable** – Actual Copilot might produce slightly different code, perhaps more idiomatic due to training on GitHub data. Quality might be high, but without someone to correct mistakes, Copilot might leave some subtleties wrong. In an automated run, we can't guarantee Copilot will self-correct – it's really a tool for a human, not an autonomous system. So if left alone, quality might suffer (e.g., it might not run tests on its own unless instructed via CLI, which it doesn't have). |

*(Table Key: "High/Moderate/Low" are relative assessments of each approach on that criterion.)*

From this matrix, it's clear that the **Enhanced Hybrid approach** best satisfies the combination of requirements: we retain control and measurability while adopting agent-like robustness, at only a minor cost to perfect fidelity. The original Option F was a solid starting point, but our enhancements significantly improve its scientific rigor. Meanwhile, more "faithful" alternatives like using Copilot directly were deemed impractical due to lack of measurement and control.

## Conclusion

We recommend **approving the Enhanced Hybrid Approach** for implementing the GHSpec adapter. It should be adopted with the outlined enhancements rather than the basic Option F as initially proposed. This strategy will enable us to execute the Spec-Driven Development workflow fully autonomously and fairly compare it with other AI frameworks. We have provided an implementation roadmap to achieve this, and our evaluation indicates this approach will meet all key objectives – **fidelity** (to the extent needed), **reproducibility, API control, valid methodology, and extensibility** – thus ensuring the experimental results are sound and insightful.

---

[1] API_KEY_ARCHITECTURE.md
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/docs/API_KEY_ARCHITECTURE.md

2 44 64 experiment.yaml
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/config/experiment.yaml

3 4 68 69 base_adapter.py
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/src/adapters/base_adapter.py

5 6 13 14 18 19 36 37 50 51 70 Spec-driven development with AI: Get started with a new open source toolkit - The GitHub Blog
https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

7 8 29 AGENTS.md
https://github.com/github/spec-kit/blob/7b55522213fa7e855a08cc2c5c65c6d9de78cf46/AGENTS.md

9 10 12 33 34 62 63 Revolutionizing AI-Powered Development: A Complete Guide to GitHub’s SpecKit | by Abhinav Dobhal | Sep, 2025 | Medium
https://medium.com/@abhinav.dobhal/revolutionizing-ai-powered-development-a-complete-guide-to-githubs-speckit-a85a39f0e2ee

11 New slash command /bugfix · Issue #619 · github/spec-kit
https://github.com/github/spec-kit/issues/619

15 plan.md
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/specs/001-baes-experiment-framework/plan.md

16 38 42 43 46 47 48 55 56 58 59 60 67 ghspec_adapter.py
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/src/adapters/ghspec_adapter.py

17 24 25 28 32 39 40 41 45 52 53 61 75 76 tasks.md
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/specs/001-baes-experiment-framework/tasks.md

20 21 22 23 54 57 65 66 token_counting_implementation.md
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/docs/token_counting_implementation.md

26 27 73 74 architecture.md
https://github.com/gesad-lab/baes_experiment/blob/35357337ad48993ef2809692f674e5f62b5900c6/docs/architecture.md

30 31 49 71 72 GitHub - github/spec-kit: Toolkit to help you get started with Spec-Driven Development
https://github.com/github/spec-kit

35 Anyone tried GitHub's Spec-Kit with Claude Code? - Reddit
https://www.reddit.com/r/ClaudeCode/comments/1nb8mi9/anyone_tried_githubs_speckit_with_claude_code/