ChatGPT

agentic-tdd: Multi-Agent TDD CLI Tool Project Plan

Project Goal and Overview

The goal of **agentic-tdd** is to create a command-line tool that can **automate the Test-Driven Development (TDD) process** for coding katas by leveraging multiple AI agents. Instead of a single monolithic AI writing code, agentic-tdd will orchestrate **four specialized agents** – each with a distinct role in the classic TDD cycle (red-green-refactor) – to collaboratively develop a solution for a given code kata. By splitting the development workflow into dedicated roles (Tester, Implementer, Refactorer, and Supervisor), we aim to replicate the discipline of TDD and improve quality and clarity of the generated code. This multi-agent approach is expected to yield faster progress and better code quality than a single-agent method, thanks to **built-in quality checks and clear separation of concerns** [1]. Recent research on multi-agent coding systems also suggests significant gains in test success rates and code maintainability when using specialized agents versus a single agent [2]. In summary, agentic-tdd's overarching goal is to **produce a working codebase for the kata through iterative cycles of test writing, implementation, and refactoring**, all supervised by an AI agent that ensures the process stays on track.

First Milestone: CLI Execution with Custom Kata

For the first milestone, the focus is on enabling a basic end-to-end run of agentic-tdd on a user-specified kata. The user should be able to invoke the tool via a CLI command, providing the kata description file, the desired LLM model and provider, API key, and an output working directory. For example:

```
agentic-tdd ~/workspace/kata/rust/mars-rover-kata/docs/kata_rules.md \
    --model sonar-pro --provider openai \
    --api-key $PERPLEXITY_API_KEY \
    --work-dir ~/workspace/agentic-tdd-output/
```

Scope of Milestone 1: When this command is executed, agentic-tdd will load the kata description (from the Markdown file `kata_rules.md`) and initialize a new project in the specified `--work-dir`. It will then run through a predetermined number of TDD cycles (or until the kata's goals are met) using the chosen LLM model. The expected outcome is that after these cycles, the `--work-dir` contains a codebase implementing the kata's requirements (including source code and test files), with a history of git commits documenting each TDD step. The CLI should stream updates to the console so the user can **follow the red-green-refactor loop in real time**, seeing messages for each agent's actions (writing a test, making it pass, refactoring, etc.). This milestone is essentially about proving the concept: given any coding kata description and a supported LLM, the tool can drive the development from scratch to a working solution through iterative TDD steps.

Key features to implement for Milestone 1 include:

- **CLI argument parsing** for all required inputs (kata file path, model name, provider, API key, output directory).
- **Model/provider integration**: using the specified LLM (via an OpenAI-compatible API) to power the agents' reasoning and code generation.
- **Initial TDD cycle execution**: at least one full cycle of failing test -> minimal implementation -> refactor, looped a number of times.
- **Console logging** to trace each step, enabling users to watch how the solution evolves test by test.
- **Git repository setup** in the work directory: each passing implementation and refactor should be committed (with meaningful messages), so the commit history reflects the TDD progression.

Achieving this milestone will validate the core workflow of agentic-tdd and lay the groundwork for more advanced features and fine-tuning in subsequent iterations.

Development Process and Workflow

To build **agentic-tdd** itself, we will adopt a disciplined development approach mirroring the tool's philosophy. This means maintaining a clear backlog of tasks, implementing features in small increments, and ensuring quality at each step:

- **Backlog & TODO List:** We will maintain an up-to-date **TODO list** of features and fixes. This backlog will be continuously refined as we progress. Each entry should be as granular as possible (a single feature or task), which makes it easier to implement and test in isolation.
- **Small Commits:** We will implement features one by one, **committing frequently** rather than batching many changes at once. Every small feature (or fix) will result in a commit with a descriptive message. This approach ensures that if a bug is introduced, we can quickly pinpoint it via git history. It also mirrors best practices of TDD (make one test pass at a time) in our development of the tool.
- **Quality Checks Before Commit:** Before finalizing each commit, we will run **quality assurance checks**:
- *Automated tests:* As the project grows, we will write unit or integration tests for agentic-tdd's functionality. Running these tests (e.g., via `pytest` for our tool's code) before each commit ensures we don't introduce regressions.
- *Code formatting:* We will use standardized formatting (for Python, tools like `black` or `isort` can be applied; for other languages if any, their equivalents). Consistent formatting improves readability.
- *Linting/static analysis:* Running linters (like `flake8` or `pylint`) and possibly type checkers (`mypy`) helps catch errors or bad practices early. We will address any warnings where feasible before committing.
- **Incremental TDD for agentic-tdd:** We can optionally follow a TDD approach in building this tool as well. For each new feature (e.g. parsing a new CLI option or implementing an agent's behavior), we might write a failing test (if applicable), implement the minimal code to pass it, then refactor. This ensures we specify the expected behavior first and keep the design clean.
- **Continuous Integration mindset:** Although this is a local project, we treat each commit as if it's going to be integrated and potentially released. That means maintaining **build stability** at all times. If a new feature isn't working or a test is failing, we prefer to fix it before moving on, rather than leaving broken code in the main branch.

By adhering to this process, development will be more predictable and the resulting codebase more robust. It will also make collaboration or future contributions easier, since the commit history will narrate the project's evolution through small, well-defined changes.

Multi-Agent TDD Workflow and Roles

The core innovation of agentic-tdd is its **multi-agent architecture**. Three primary agents will emulate the traditional red-green-refactor cycle of TDD, and a fourth agent will supervise and guide the process. Each agent uses an LLM (Large Language Model) to decide on actions and generate code or tests. They communicate implicitly through the shared code repository and explicit signals (such as test results or error messages). This design draws inspiration from the way specialized team roles can collaborate to achieve better outcomes than a single generalist, offering benefits like parallel thinking, quality checks from different perspectives, and clear division of responsibilities [1].

The agents and their responsibilities are as follows:

1. Tester Agent (Red Phase)

Role: The Tester Agent is responsible for driving the **"red" phase of TDD** by writing a new failing test that captures the next required functionality or behavior described in the kata. This agent essentially asks: *"What's the next small capability the program should have, which isn't implemented yet?"* and then encodes that as a test.

Behavior and Capabilities:

- **Write a minimal failing test:** Using the kata description and the current state of the codebase, the Tester Agent will craft a unit test (or small set of tests) targeting the next unimplemented behavior. The test should be as simple as possible while still forcing the implementation of a new aspect of the problem. For example, if the kata is FizzBuzz, the first test might assert that for input 1, the output is "1".
- **Ensure the test fails:** Before handing off to the implementer, the Tester Agent will run the test suite (or at least the new test) to confirm that the test indeed fails. This guarantees that we are in the "red" state – the new behavior is not yet fulfilled by the code. If the test accidentally passes immediately, it means the Implementer Agent had previously implemented more than necessary (or the test is not correctly targeting a new behavior). In such a case, the Tester Agent will report this outcome to the Supervisor (because writing a passing test indicates a mistake in the process). The Tester may then try a different test or wait for guidance from the Supervisor.
- **Interaction with environment:** The Tester Agent will have the ability to **read the kata rules and existing code**, create new test files or append to existing test suites, and execute test commands. Depending on the programming language, it might run a command like `pytest` (for Python) or `cargo test` (for Rust) to verify the test fails.
- **No commit of failing tests:** To avoid breaking the repository's build, the Tester Agent will typically **not commit the failing test**. It can stage the test file (so that the Implementer Agent can see it in version control if needed), but it will wait to commit until after the Implementer makes it pass. This way, the git history remains bisectable (no commit where tests are red). The new test will eventually be committed together with the implementation that makes it pass, preserving atomic commit integrity.

After writing the failing test and confirming it indeed fails, the Tester Agent's turn is done. It then "hands over the keyboard" to the Implementer Agent for the green phase.

2. Implementer Agent (Green Phase)

Role: The Implementer Agent handles the **"green" phase of TDD**. Its job is to write the minimum amount of code required to make all tests pass (particularly focusing on the test just introduced by the Tester Agent). The guiding principle for this agent is **YAGNI (You Ain't Gonna Need It)** – implement only what is necessary to satisfy the current failing test, nothing more.

Behavior and Capabilities:

- **Minimal implementation:** Upon activation, the Implementer Agent will inspect the failing test(s) and the existing codebase. It uses the context from the kata description (which outlines the overall requirements) but should be careful *not* to preemptively implement future features. It should produce code changes focused solely on making the current test pass. For instance, if the test expects a certain output for a given input, the Implementer will hard-code or logically derive that output just for that case, if that suffices to pass the test, deferring generalization until more tests demand it.
- **Run tests to verify**: After code changes are applied, the Implementer Agent will run the test suite in the working directory (e.g., run `cargo test` for a Rust kata or `npm test` or appropriate command depending on the environment). The goal is to see **all tests green** at this point, proving that the new code satisfies the latest test without breaking any previous functionality.
- **Handling implementation failures:** If the new test still fails or if the changes caused a regression in older tests, the Implementer Agent may attempt to adjust the code. It can iterate within its turn: modifying the code and re-running tests until they all pass (or until it determines it cannot easily satisfy the test). If it gets stuck (for example, the test might require a large architectural change that the Implementer can't do alone), it will report back to the Supervisor Agent that it cannot make the test pass. The Supervisor might then decide to invoke a different strategy (perhaps asking the Refactorer Agent to intervene and restructure code before re-attempting implementation, or guiding the Tester to simplify the test).
- **Commit on success:** Once **all tests are passing**, the Implementer Agent will stage and commit the changes. The commit message can be auto-generated summarizing the implementation (e.g., "feat: make rover turn right – implement Right command to pass test"). This commit will include the test written by the Tester (now passing) and the code changes that make it pass. Committing at this point ensures the repository's history has a green build at each commit.
- **Tools and context:** The Implementer can open and edit source files, possibly using an editor tool or direct file operations. It will receive the failing test's content and any error messages from the test run as part of its context (for example, if an assertion failed saying "expected X but got Y", that can help the agent focus on what needs to change).

After successfully committing the passing code, control flows to the next agent in the cycle – the Refactorer – to clean up or improve the code.

3. Refactorer Agent (Refactor Phase)

Role: The Refactorer Agent is in charge of the **“refactor” phase of TDD**, where the codebase is improved without changing its external behavior. The goal here is to **enhance code quality, readability, and maintainability** now that the tests are green, ensuring the system remains agile for further extension. This agent ensures the solution adheres to best practices and any specific constraints mentioned in the kata (for example, some katas impose style or design restrictions like “no primitives” or “single level of indentation”).

Behavior and Capabilities:

- **Identify improvement opportunities:** The Refactorer Agent will analyze the current codebase (after the latest implementation) to spot any **code smells or violations** of the kata’s constraints. This could include things like duplicated code, functions that are too long or do too much, poor naming, inefficient algorithms that could be simple, etc. If the kata description includes explicit guidelines (e.g., *“Wrap all primitives”* or *“use descriptive class names”*), the agent will check compliance with those rules and refactor accordingly.
- **Safe refactoring:** The agent will propose changes that **do not alter the observable behavior** of the code. Examples of refactoring actions are: renaming variables or functions for clarity, extracting a piece of logic into a separate function or class, removing redundant code, simplifying complex conditionals, improving algorithmic efficiency without changing outputs, organizing files and modules more cleanly, etc. It must be careful to keep all tests passing. The test suite acts as a safety net – any behavior change would cause a test to fail and thus indicate a bad refactor.
- **Apply changes and test:** The Refactorer will modify the codebase as needed (much like an experienced developer performing cleanup). After making changes, it runs the full test suite to ensure that **all tests still pass**. If tests are green, the refactoring is considered successful. If any test fails after refactoring, that means the refactor inadvertently changed something it shouldn’t have. In that case, the Refactorer has a few strategies:
- It can try to **undo or adjust** the last changes it made, or
- It can attempt a different refactoring approach that achieves the intended improvement without breaking the test.
- We will allow the Refactorer Agent to make a limited number of attempts (configurable, default might be 3 tries) to get the refactor right. This might involve iterating with the LLM to figure out why the test failed and how to preserve behavior correctly.
- **Failure escalation:** If after the allowed number of retries the tests still fail, the Refactorer Agent will concede failure for this refactoring round. It will inform the Supervisor Agent that it cannot complete the refactor without breaking functionality. At this point, the Supervisor might decide to roll back the refactor changes (keeping the code as it was) or might intervene with a different strategy (perhaps instructing a smaller refactor, or even adjusting tests if they were too constraining, though ideally tests remain fixed).
- **Commit on success:** If the refactoring is successful (all tests stayed green), the Refactorer Agent will commit the changes to the repository. The commit message might be like “refactor: simplify movement logic in Rover class” or “refactor: extract helper for input parsing”. This commit again keeps the project in a working state but now with cleaner code.
- **Continuous improvement mindset:** On successive iterations, the Refactorer will continuously keep the codebase clean. As new features get added through red/green, there may be new opportunities to refactor (e.g., emerging duplication between test cases could be factored out, or a naive

implementation can be generalized). The agent's presence in every cycle ensures the code does not drift into chaos as complexity increases.

Once the Refactorer Agent has committed its changes (or if it skipped due to inability to improve), control goes back to the Tester Agent to begin the next cycle of adding a new failing test for additional functionality. This **loop (test -> implement -> refactor)** repeats until the kata's requirements are fully met.

4. Supervisor Agent (Process Orchestrator)

Role: The Supervisor Agent oversees the entire process as a kind of project manager or meta-agent. It does not write code or tests directly, but it monitors the other three agents, detects impasses or missteps, and makes decisions on how to adjust the process. The Supervisor ensures that the team of agents works smoothly together and that the development stays aligned with the kata goals.

Responsibilities and Behaviors:

- **Monitoring and context:** The Supervisor keeps track of the overall progress: Which requirements of the kata have been implemented? How many tests have been written and passed? It can review the commit history or diff to understand what changes have been made. It also listens for any signals of failure from the other agents.
- **Handling Tester Agent issues:** If the Tester Agent reports that a newly written test turned out to be **already passing** (meaning the Implementer Agent had previously implemented more than what was tested), the Supervisor will note this as a process issue:
- This suggests the Implementer **over-delivered** in a prior step (violating the TDD principle of doing the simplest thing). The Supervisor might record a "penalty" or simply an observation that the implementer overshot. In practical terms, this could be used to adjust the Implementer Agent's prompting in the future (e.g., reinforce instructions to only solve exactly the failing case and nothing more).
- The Supervisor will instruct the Tester Agent to proceed by writing a different test (since the original "failing" test didn't fail, we still need a failing test to drive development). Perhaps the Tester will choose a slightly more advanced scenario that definitely isn't covered yet.
- **Handling Implementer Agent issues:** If the Implementer Agent cannot make the test pass (within a reasonable number of attempts or a time limit), it will inform the Supervisor that it's stuck. The Supervisor then decides how to unblock the situation:
- It might analyze why the test is hard to pass. For example, maybe the test expects a large piece of functionality that would require many changes at once, hinting that a design refactor is needed first.
- The Supervisor could then direct the **Refactorer Agent to perform a *preemptive refactoring*** to enable the implementation. For instance, if the architecture is too rigid to easily add the new feature, refactor it first (without the new test in place, or by temporarily disabling some tests) and then have the Implementer try again.
- Alternatively, the Supervisor might instruct the Tester to break down the requirement into smaller tests (divide and conquer) if the current test was perhaps too broad or ambitious for a single step.
- In extreme cases, the Supervisor could also adjust the test itself if it realized the test was invalid or out of scope, but ideally we try to avoid altering tests once written (unless the kata description was misunderstood).
- **Handling Refactorer Agent issues:** If the Refactorer Agent reports that it failed to refactor without breaking tests (after the allowed retries), the Supervisor has a few options:

- It might decide to **skip the refactoring for this cycle**. Perhaps the code is just good enough, or the attempted refactor was not crucial. The Supervisor can instruct to revert any partial changes and proceed to the next test. The refactoring opportunity could be revisited in a later cycle or left as a known tech debt.
- Alternatively, the Supervisor could suggest a different refactoring strategy, or even allow a partial refactor that comes with updating tests (only if the kata constraints permit changing test expectations, which is rare in TDD – typically tests are truth).
- The Supervisor will log this incident, possibly adjusting future agent behavior. For example, it might ease up on refactoring strictness if it consistently fails, or mark certain complex refactors as out-of-scope for automation.
- **Deciding when to stop:** The Supervisor Agent will also determine the **termination condition** of the TDD loop. This could be after a predefined number of cycles (e.g., 10 red-green-refactor cycles) or when it assesses that the kata goals have been achieved:
- It can compare the implemented features (in commit messages or tests) against the original kata description to see if all requirements are covered.
- If the Tester Agent cannot come up with any new failing test (because everything in the kata description has tests and passes), that is a strong indication the kata is complete. The Supervisor would then finalize the process.
- Alternatively, a user might specify a certain number of iterations to run, and the Supervisor will honor that and stop after that many cycles, possibly leaving some kata requirements unimplemented if not done yet. (For the best outcome, an intelligent approach is to continue until done, but a fail-safe count can prevent infinite loops).
- **Logging and feedback:** The Supervisor will output high-level progress updates to the console. For instance, *"Cycle 3 completed: 5 tests passing, 0 failing. Next, adding new test for edge case X."* It will also report when it intervenes, e.g., *"Supervisor: Tester's last test was already passing, likely over-implementation occurred. Reminding Implementer to stick to minimal changes."* or *"Supervisor: Implementer stuck on current test – invoking Refactorer for preparatory cleanup."* These messages help a human observer understand the decisions being made.

In essence, the Supervisor Agent is the **governor of the multi-agent system**, ensuring that the three core agents remain effective and that the overall development stays on track towards the kata's completion. It adds an extra layer of resilience: if the straightforward red-green-refactor cycle hits a snag, the Supervisor adjusts the strategy rather than leaving the process in limbo.

Technical Architecture and Stack

To implement agentic-tdd, we will use a combination of **Python**, system tools (like git and language-specific test runners), and the **LangChain** library for managing LLM interactions. Below are the key technical components and decisions:

- **Programming Language:** Python was chosen for developing agentic-tdd due to its rich ecosystem for AI/LLM integration and process orchestration. The CLI tool itself will be a Python script (or package) that can be installed and run from the terminal.
- **Multi-agent Orchestration:** We intend to utilize **LangChain** (a popular framework for building applications with LLMs) to help manage the prompts and interactions of the four agents. LangChain offers abstractions for defining agents with particular prompts and tools. Each of our agents can be modeled as a LangChain Agent (or Chain) with:

- A **prompt template** specialized for its role (Tester, Implementer, etc.). This will include a system message setting the context and role, and provide the relevant inputs (kata description, current code diff, test results, etc.).
- Access to **tools** that allow the agent to act on the environment. We will likely implement custom tools such as:
    - *Filesystem read/write:* so agents can read code files, create or modify files (for code or tests).
    - *Command execution:* to run shell commands like running the test suite or installing dependencies. LangChain has a mechanism to execute commands or we can call Python's `subprocess` module directly.
    - *Git operations:* possibly tools to commit changes or get diff (though committing can also be done directly via Git CLI).
- The agent's **stopping condition or return format** will be defined. For instance, the Implementer Agent should output the code changes it recommends (which we then apply to the filesystem), rather than directly executing them.
- **LLM Provider Abstraction:** The tool is designed to be **model-agnostic** as long as the model supports an OpenAI-compatible chat completion API. This means we can plug in different models from various providers (OpenAI, Anthropic, Cohere, Perplexity, etc.) by specifying:
- `--provider`: identifies which provider's API to use (for example, `openai`, `azure`, `anthropic`, `deepseek`, `perplexity`, etc.). Under the hood, we might use LangChain's `ChatOpenAI` class for OpenAI and OpenAI-like APIs, or if needed, specific integrations for others. The assumption is providers like Perplexity or iFlow have an endpoint that mimics OpenAI's API format. If a provider requires a different format (e.g., Anthropic uses a different API call), we can handle that by selecting the appropriate LangChain wrapper or making a custom call.
- `--model`: the specific model name/identifier to use (e.g., `gpt-4`, `sonar-pro`, `claude-2`, etc.). This will be passed to the LLM API. Some providers might not recognize the same model names, so the combination of provider and model will need to be mapped correctly in code (for instance, if provider is `openai` and model is `sonar-pro`, we might have to route that to the provider's specific naming).
- **API Key management:** We will not hardcode any API keys. The `--api-key` CLI option can be used to pass a key (like `$PERPLEXITY_API_KEY` environment variable expanded into the command), but if not provided explicitly, the tool will look for environment variables (for example, `OPENAI_API_KEY` for OpenAI, or a provider-specific env var). This allows users to configure their keys securely in their shell environment rather than writing them in code or config files. The code will likely use the standard environment variable that the provider's SDK expects (OpenAI's library looks for `OPENAI_API_KEY` by default, etc., and similarly for others if applicable).
- **Working Directory and File Management:** The `--work-dir` specified will be the sandbox where all code and test files are created and modified by the agents. If the directory does not exist, agentic-tdd will create it. If it exists and is non-empty, we have to decide how to handle it:
- Ideally, the tool should start with a clean environment for each kata. We may enforce that `--work-dir` should be empty or a fresh directory (warning or aborting if not empty to avoid overwriting user files).
- We will initialize a **git repository** in this directory (using `git init`) at the start of the run if one isn't already present. This allows the agents to make commits. We'll also create an initial commit perhaps with a README or the kata description for reference, though not strictly necessary.
- The kata description file (the Markdown input) could be copied into the work directory for reference (or the content just loaded in memory to feed into prompts). Possibly, we might create a `docs/` folder in work-dir and copy the kata MD there for completeness.

- **Test Execution:** Running tests is central to the TDD loop. We need to support executing tests for the language of the kata:
- For Python katas, we can use `pytest` or `unittest`. If using `pytest`, we ensure it's installed in the environment running agentic-tdd.
- For Rust katas, use `cargo test`. This assumes the Rust toolchain (`cargo`) is installed on the user's system. The Tester and Implementer agents must generate code and tests that fit into a Rust project structure (e.g., a `Cargo.toml` might be needed, tests can go in `tests/` or as `#[test]` functions).
- For other languages, similar approach (e.g., Java could use Maven/Gradle commands, JavaScript could use npm scripts, etc.). **In the first milestone, we might focus on one language (possibly the language of the given kata example).** The example path suggests a Rust kata, so ensuring Rust support in cycle is likely a priority. Over time, we can abstract the test-runner so that it's configurable or auto-detected.
- Implementation detail: The agents themselves might not invoke `cargo test` by writing it in their output (though they could if we gave them a shell tool). It might be easier to have our Python orchestrator run the tests and feed the results back into the agent prompts. For example, after the Tester adds a test, our code runs the tests and sees a failure output (stack trace, assertion message). We then provide that output to the Implementer agent as part of its context ("Here is the failing test and error message"). This grounds the LLM in actual feedback from running code.
- **Agent Prompt Design:** Crafting the right prompts for each agent is critical:
- The **Tester Agent's prompt** will include the kata description and possibly a summary of what's implemented so far (maybe from commit messages or an outline of the code), and instruct the LLM to output a new unit test only, targeting an unimplemented behavior. We will emphasize to **only output test code** (plus maybe minimal explanation as comments if needed) and not to reveal the solution.
- The **Implementer Agent's prompt** will include the failing test (maybe as code snippet) and context such as "All current tests are failing/passing except this new one. Implement just enough to make it pass." We'll remind it of the kata requirements and to keep changes minimal.
- The **Refactorer Agent's prompt** will include the current code (or the specific parts likely needing refactor) and possibly the kata's code quality constraints, instructing the LLM to propose changes that improve the code without changing behavior. We might provide a diff or summary of the recent implementation to focus its attention.
- The **Supervisor Agent** doesn't generate code, but we may implement its logic in Python rules rather than via LLM. The Supervisor could still use an LLM if we want it to reason about complex situations, but that might complicate things. Initially, we can encode the Supervisor's decision tree procedurally: e.g., `if test passes immediately: do X; if implementer stuck: do Y`. These rules can be refined by human developers or possibly by an LLM in future if we give it an overview of state and ask for next step (which is an interesting extension).
- **State Management:** Each agent cycle will produce some artifacts (new files, modified files, commit history entries). We need to maintain the state consistently:
- Use git not just for commits, but potentially to get **diffs** or file changes to show the agents. For instance, the Refactorer might benefit from seeing a `git diff` of the changes it intends to make or has made.
- We might keep an in-memory representation of which requirements are done. Perhaps parse the kata description into a checklist and mark off items as tests for them are added. This could be part of the Supervisor's duties.

- **Performance and Rate Limiting:** Interacting with LLMs, especially large ones, can be slow or cost tokens. We should design carefully:
- Only send the necessary information in each prompt (to save tokens).
- Possibly use function calling or structured outputs if the model supports it (for example, we could instruct the Implementer agent to output JSON with a list of file changes, but that might be complex to parse; more straightforward is to output code blocks that we can apply).
- Introduce delays or checks if needed to avoid hitting rate limits of the API (especially if running many cycles).
- Logging each prompt and response (perhaps in a debug log file) for later analysis and improvement of prompts.

In summary, the architecture is one of a **coordinator script (Python)** that uses **LLM agents via LangChain** to generate content, applies that content to a **local code repo with real build/tests**, and uses the **results of those tests to inform the next LLM call**. All four agents operate in the same environment (the working directory) but with clearly defined turns and roles. By using Python and existing tools, we avoid reinventing wheels like git operations or test frameworks, and by using LangChain we can focus on the logic of prompts and agent behavior rather than low-level API calls. The design is modular, allowing swapping out the LLM model or provider as needed.

CLI Interface and Configuration

The agentic-tdd tool will be accessible via a command-line interface. This makes it straightforward to integrate into different development setups and possibly CI pipelines in the future. The CLI will accept several options and arguments to configure the run. Here's a detailed breakdown of how the CLI is designed:

Basic Usage: `agentic-tdd <kata_description.md> [--model MODEL_NAME] [--provider PROVIDER] [--api-key KEY] [--work-dir DIRECTORY] [--max-cycles N]`

- **Positional Argument: Kata Description Path** – The first argument should be the path to the kata rules/description file (in Markdown format). This file contains the problem statement and any specific rules or constraints for the kata. The tool will read this file and use its content to guide the Tester Agent (and to provide context to other agents). Example: `~/Documents/katas/mars-rover-rust/kata_rules.md`.
- We will ensure the content is read at startup. If the file is not found or unreadable, the tool will error out gracefully, informing the user.
- There's an assumption that the description is in Markdown, but it could be any text file – we might not need to do anything Markdown-specific except maybe ignore formatting. We could strip out code blocks or special formatting if needed, but likely the raw text is fine for the LLM.
- `--model MODEL_NAME`**:** (Optional, default could be something like `gpt-4` or whichever we consider best by default) This specifies the identifier of the LLM model to use. For example: `--model gpt-4` or `--model claude-instant` or `--model code-cushman-001`. The list of allowed model names depends on the provider's API.
- `--provider PROVIDER`**:** (Optional, default `openai`) The name of the LLM provider or API to use. Examples could be `openai`, `azure`, `anthropic`, `perplexity`, `deepseek`, `iflow`, etc. This option informs how we instantiate the API client.

- If `openai` (default), we use OpenAI's API endpoint. If another provider that uses the same format (OpenAI-compatible), we might just change the base URL or the API host.
- For providers with different formats (like Anthropic uses a different API call), we might internally handle it by using a different branch in code or using LangChain's Anthropic integration. We will document which providers are supported in the first version (likely focusing on those easily used via LangChain).
- `--api-key KEY`**:** (Optional) The API key for the chosen LLM provider. In practice, the user is expected to supply this securely. If not provided, the tool will look for an environment variable. We prefer environment variables to keep secrets out of command history:
- For OpenAI, `OPENAI_API_KEY` env var can be used. For other providers, we will define analogous env var names or possibly allow a generic `AGENTIC_TDD_API_KEY`.
- In the example command, the user wrote `--api-key $PERPLEXITY_API_KEY`. This suggests the user had set an environment variable `PERPLEXITY_API_KEY` and is passing it. We will not log this value anywhere, and if printing configuration, we should mask or omit the actual key.
- Alternatively, we might support reading from a config file or keyring in future, but environment is simplest to start.
- `--work-dir DIRECTORY`**:** The directory where the kata code will be developed. This acts as the **working project directory**. If not provided, a default could be to create a temp directory or use the current directory (but for safety, requiring it might be better).
- Example: `--work-dir ../agentic-tdd-kata` – if this is relative, it will be resolved relative to current directory.
- Behavior: If the directory doesn't exist, we create it. If it exists and contains files, we issue a warning. We might require an empty directory to avoid conflicts or accidentally overwriting user data. Possibly we could allow continuation if it's a previously started session (to be considered later).
- We will perform `git init` in this directory if no git repository is found. If it's already a git repo (maybe the user prepared something), we can use it.
- We'll set up necessary language-specific initialization if needed. For example, if it's a Rust kata, ensure a `Cargo.toml` exists (we could generate one based on kata name), so that tests can run. If it's Python, maybe create a `pyproject.toml` or requirements file if needed. This might be beyond milestone1 (for now possibly assume the user's kata description or setup includes how to structure the project).
- `--max-cycles N`**:** (Optional) This parameter (or a similar one) can define how many Red/Green/Refactor cycles to perform at maximum. If not set, the Supervisor will run until completion criteria are met. If set, the tool will stop after N cycles regardless. This is a safeguard to avoid infinite loops or runaway costs. For instance, `--max-cycles 5` would perform at most 5 iterations (meaning 5 new tests added). After that, it would stop and perhaps inform the user that more cycles may be needed to fully complete the kata.
- For milestone 1, we can implement a simple version of this. For example, default N=10 if not provided.
- If the kata is finished earlier (no new tests to add), the loop breaks naturally even if N is larger.
- **Additional config options (future):** We might include flags like `--language` to explicitly set the programming language if it cannot be inferred, or `--verbose` to control the level of console output, etc. Initially, we can try to infer language from context (e.g., the kata description might mention it, or the file path includes language as in the example).

Console Output: The CLI will print informative messages as the process runs, to keep the user informed. We will design the output to be readable and structured:

- Each major phase could be prefixed with the agent name, for example:
- `Tester: Adding test to check rover rotates right when command 'R' is given...`
- `Implementer: Implementing minimum logic for 'rotate right'...`
- `Refactorer: Refactoring navigate() function for clarity...`
- `Supervisor: New test passed immediately, adjusting strategy...`
- We might use different colors for each agent's logs to improve clarity (if console supports it).
- After each cycle, we can output a short summary: *"Cycle 4 complete. Tests passed: 12, New test added: 'rover turns right'."*
- If running in a non-interactive environment or if verbosity is turned down, we might only show high-level info or final outcome (e.g., "All kata requirements implemented in X cycles, code at ./agentic-tdd-kata").

Environment Setup: Users must ensure that any needed runtime for the code is available. For example, if it's a Rust kata, Rust's compiler and cargo should be installed. If it's Python kata, the appropriate Python version and test frameworks should be present. In the future, agentic-tdd might set up virtual environments or Docker containers to isolate and manage dependencies, but in the first version we rely on the user's environment.

Failure handling in CLI: If something goes wrong (for instance, if the LLM API fails or returns an error, or if a command like running tests fails due to environment issues), the tool should catch that exception and print an error message. Depending on the failure, it might either abort the entire process or inform the Supervisor Agent to handle it. We will implement basic error-catching around external calls (API and subprocesses) to ensure the CLI doesn't just crash without explanation.

In summary, the CLI is designed to be straightforward: the user points agentic-tdd at a kata description and provides the AI model details, and then the multi-agent system takes over. The configuration options allow flexibility in choosing AI backends and controlling execution, while the tool handles the heavy lifting of driving the TDD workflow.

Implementation Backlog and Task Breakdown

To build agentic-tdd, we'll break down the work into a series of manageable tasks. Below is the **backlog of features and tasks** required to achieve the first milestone and beyond, in a logical implementation order:

1. **Project Initialization**: Set up the Python project structure.
2. Create a new repository (if not already done) and scaffold a Python package (e.g., `agentic_tdd/` directory with `__init__.py`).
3. Set up a basic `pyproject.toml` or `setup.py` if needed for dependencies. We'll include **LangChain** and any LLM SDKs (OpenAI SDK, etc.), as well as testing frameworks (for the tool's tests) and possibly shell utility libraries.
4. Initialize version control (git) for our own project and make an initial commit of the scaffold.
5. **CLI Argument Parsing**:

6. Use Python's `argparse` or the `click` library to define the CLI interface.
7. Implement options: `--model`, `--provider`, `--api-key`, `--work-dir`, `--max-cycles`, etc., and the positional `kata_path`.
8. Make sure help texts are clear (`agentic-tdd --help` should describe usage).

9. Test this component by running the CLI with various arguments to ensure it captures inputs correctly. This can be a standalone commit (e.g., "feat: add CLI with arguments parsing").

10. **Loading Kata Description & Environment Setup**:

11. Read the content of the provided kata Markdown file into memory.
12. Validate that `--work-dir` is usable: create directory if not exists, or empty it/init if it exists. Initialize a git repository inside it if not present.
13. Possibly commit the kata description file into the repo as a reference (optional).
14. Determine the language or any meta-info from the kata if possible. (For milestone 1, this might be manual or assumed. We could parse the file name or content for clues like "This is a Rust kata").
15. If needed, create basic project files (for Rust, `cargo init` could be run to create Cargo.toml and src directory; for Python, maybe nothing special or create a basic structure).
16. Ensure we have a mechanism to run tests: e.g., if Rust, verify `cargo test` works (initially it will run zero tests in an empty project).

17. This stage sets the groundwork for the agents to start working. Commit any baseline files (like a Cargo.toml) as "chore: initialize project structure".

18. **Integrate LLM Model (Proof of Concept)**:

19. Before implementing all agents, do a quick integration test with the chosen model to ensure we can get a response. For instance, use LangChain or direct API call to have the model respond to a trivial prompt.
20. This involves configuring the API key and endpoint based on `--provider`. Use LangChain's `ChatOpenAI` for OpenAI or adjust for others. We might start with OpenAI as default to verify connectivity.
21. Handle errors (e.g., invalid key, network issues) gracefully with messages.

22. Once verified, structure our code to easily create an LLM client object that can be passed to agent prompt functions.

23. **Tester Agent Implementation**:

24. **Prompt Template**: Design the prompt for the Tester. It should include the kata description (or a summary of what's remaining to implement), and possibly reference the last implemented behavior to avoid repetition. The system message could be: *"You are a Test Writer AI following TDD. You have the following kata description and the current codebase. Write a new unit test for the next required behavior that is not yet implemented. The test should fail initially."* Include any kata rules about testing (some katas specify certain testing approaches).

25. **Functionality**: After getting the test code from the LLM:
    - Save the test to a file in the work-dir (e.g., in Rust, create a new test module or add to `tests/` directory; in Python, maybe create a new `test_xyz.py` file or append to an existing one).
    - Run the tests: execute the test command for the project. Capture the result:
    - If the test run fails (which we expect, at least the new test should fail), that's good. We might parse the output to get the specific failure message or at least confirm it failed.
    - If the test run passes all tests (unexpected), that means the new test did not actually introduce a failing condition.
        - In this case, trigger a Supervisor routine (which for now could be a simple flag or exception indicating "TestAlreadyPassing").
        - We likely won't commit this test since it didn't drive a new code change. Possibly log it and ask the Tester to write a different test.
    - If the test failed as expected, we proceed. Possibly **stage** the new test file with `git add` (but do not commit).

26. **Evaluation**: We should manually test this agent by giving it a simple kata and seeing if it produces a reasonable test. For initial development, we can simulate with a very simple kata (like "function that returns 1 when input is 1") to validate the logic.

27. **Implementer Agent Implementation**:

28. **Prompt Template**: Construct the Implementer's prompt with context. It will likely include:
    - The kata description (for understanding context).
    - The text of the failing test (or at least what the test is expecting vs getting).
    - Possibly the output of the last test run (e.g., "expected X, got Y" or stack trace). This can guide the LLM to know what is failing.
    - The current codebase or at least the relevant files content, so it knows where to insert or update code.
    - Instruction: *"Only make the minimal changes necessary to the code to make the above test pass (while keeping all other tests passing). Do not introduce unrelated functionality."*
29. **Functionality**: When we receive the code changes (likely the LLM will provide new code snippets or modifications):
    - We might need to apply these changes. If the LLM outputs full file content, we can overwrite the file. If it outputs a diff or just a snippet, we might have to merge it in. This is a tricky part: ensuring changes are applied correctly. Possibly, we can prompt the LLM to output the whole file content for any file that changes, enclosed in markers, or a unified diff format. Alternatively, use a simpler approach: give it the file and ask for an updated version of the file with changes.
    - Once changes are applied to the filesystem, run the tests again.
    - If tests pass now: great. Proceed to commit.
    - If tests still fail (including the possibility of new failures introduced): we can loop back into the Implementer prompt, perhaps informing it of the new failure. (We should decide a cut-off to avoid infinite loop if the LLM keeps failing; maybe give it a couple of attempts, then escalate to Supervisor).
    - Commit the changes (including the test file). Commit message can be generated. We might generate a simple message like "Implement [feature] to pass [test]" or allow the LLM to suggest a commit message. Simpler: form a message from the test name or assertion.

30. **Testing**: Try this on a known failing test scenario to see if it works. We could have a dummy test like `expect function f(2) == 2` fails (since f not implemented) and see if the implementer writes f to return 2.

31. **Refactorer Agent Implementation**:

32. **Prompt Template**: Provide the Refactorer with:
    - The kata constraints (especially if they include design rules).
    - Possibly a summary of the current code (or the full codebase if small, or the specific files that might need refactoring).
    - Emphasize: *"Refactor the code to improve readability/maintainability/design without changing behavior. All tests must remain passing. List what you will change and then show the updated code."*
33. **Functionality**: After getting the refactored code suggestions:
    - Similar to Implementer, apply the changes to the files. (The LLM might output entire refactored files or just snippets; we'll likely encourage it to output full files for clarity).
    - Run tests to ensure they still pass.
    - If pass, commit the refactoring with message like "refactor: [description]".
    - If fail, we have a strategy: we can either:
        - Attempt to run another round with the Refactorer agent: possibly providing the test failure output and ask it to fix the refactor.
        - Or revert the changes and report failure to Supervisor (depending on how complex to automate fixing a broken refactor).
    - We'll implement a loop for a limited number of retries (e.g., 3). Use a counter, and if still failing after 3 attempts, revert to last known good state (we can use git to discard changes) and flag Supervisor.
    - If no refactoring is needed (maybe the code is already simple), the Refactorer might output that it's fine or make a trivial change. If it essentially does nothing significant, we might choose not to commit an empty change (or commit a minor style fix).

34. **Testing**: We should verify on a scenario. Possibly, after implementing a couple of features, see if the refactorer can combine two similar functions or rename a variable.

35. **Supervisor Logic and Orchestration**:

36. Now that the three core agents are in place, implement the loop in our main function:
    - for i in 1..max_cycles:
    - Call Tester Agent. If Tester reports "done" (no new test idea) or if test fails to fail (paradoxical as that sounds), break or handle via Supervisor logic.
    - Call Implementer Agent to handle the failing test. If unable to pass the test, break and let Supervisor decide (maybe call Refactorer out-of-order or abort).
    - Call Refactorer Agent to clean up. If fails, decide via Supervisor: either try again or skip.
    - If kata seems complete (no failing tests and perhaps all requirements covered), break out early.
    - The Supervisor's intelligence initially can be simplistic rule-based:

- Maintain a flag for “implementer overdid last time” and adjust prompts accordingly (e.g., if a test came out passing immediately, we can modify the system message of the Implementer for next time to be more cautious).
- If implementer fails, maybe call refactorer then re-attempt implementer.
- Ensure to not get stuck in an infinite loop.
- Print relevant Supervisor messages to console as we go.

37. As this is the heart of the process, we will test it end-to-end on a simple kata:
    - Use a known simple kata (e.g., “return 1 for input 1, return 2 for input 2” type of trivial function) to see the cycle works: Tester writes a test, Implementer makes it pass, Refactorer maybe does nothing, loop ends when no more obvious behavior to test.
    - Debug any issues in the sequencing, like ensuring each agent sees the latest code, etc.

38. This step is likely a major integration point and will yield the first working prototype of agentic-tdd’s full cycle.

39. **Git Integration (Commit History)**:

40. By now we’ve touched on staging and committing in agents, but ensure that:
    - The git commits are being created correctly in `--work-dir` repo. Use `gitpython` library or shell out to `git` commands to add and commit.
    - Create informative commit messages. Possibly prepend with the agent role, e.g., “Test: ensure rover rotates right” for tester-added tests, “Feat: implement rotate right command” for implementer, “Refactor: extract orientation logic” for refactorer. A consistent format can be nice.
    - If any commit fails (e.g., user has git config issues), handle that gracefully.
    - After the run, the user can `cd` into the work directory and inspect the commit log to see the development story. This is a key feature, essentially producing a self-documented progression.

41. We will test that commits are made as expected and the repository is usable.

42. **Logging and User Feedback**:

    - Refine console output to be user-friendly, as described earlier. Possibly add a verbosity flag or quiet mode.
    - Ensure that critical issues (like an agent failing or an API error) are clearly reported.
    - Maybe summarize at the end: e.g., "Kata completed in 5 cycles, 5 tests added, solution in `work-dir`."

43. **Internal Testing & QA**:

    - Write a few automated tests for our agentic-tdd code (for example, a test for the CLI argument parsing, a test for the function that runs a dummy agent to ensure we parse outputs correctly).
    - Run linters/formatters to clean up code style.
    - Test on at least one real kata description to see how it performs. We may not get a perfect solution, but we can check that the cycle flows without crashing and does something sensible.

    - Prepare example outputs to include in documentation.

44. **Documentation**:

    - Create a README.md for the project explaining how to install and use agentic-tdd.
    - Document the CLI usage, all options, and possibly an example of a session (with example console output truncated for brevity).
    - Mention which providers/models have been tested.
    - Warn about the limitations (like costs of API calls, need for environment for target language, etc.).
    - Also include in documentation the concept of the multi-agent TDD approach for context (some users may be interested in the theory behind it).

Each of these tasks will result in one or more git commits. By following the backlog in order, we'll ensure that at each stage we have a working increment of the tool (e.g., after step 5, we should be able to run agentic-tdd to just add a test and stop, etc.). This iterative approach de-risks the development and allows testing along the way.

Throughout, if new insights or problems arise, we'll update the TODO list. For example, if we discover the need for a config file or to handle timeouts for LLM calls, we'll add tasks for those.

Future Enhancements and Nice-to-Have Features

Once the basic version of agentic-tdd is working (as outlined in the milestone and tasks above), there are many enhancements and extensions we can plan for future versions:

- **Integration with GitHub Copilot and Advanced Models:** As more powerful coding assistants become available (e.g., GitHub Copilot's upcoming GPT-5 based models or other specialized code models), we want to allow using them in agentic-tdd. This could involve:
- Providing **Copilot's API integration** (if Copilot for individuals gets an API or via Azure OpenAI endpoints for certain models). The CLI might accept `--provider copilot` or a specific model name that we map to the Copilot service.
- Possibly even running an instance of Copilot locally or as a service that agentic-tdd can query. We'd need to research how to interface, as Copilot currently integrates via editor plugins rather than direct API, but Microsoft may provide endpoints for certain partners.
- The benefit is to leverage training that Copilot/GPT-5 might have on code contexts, which could improve generation quality.
- **Support for More Agent Roles or Modified Workflows:** The current design has four agents. We might experiment with additional agents or variations:
- An **Architect Agent** that, at the very start, reads the kata and plans a high-level strategy (maybe writes some pseudo-code or outlines modules). This could provide guidance to subsequent cycles or create a backlog of tasks that the Tester then follows. The Reddit example used an Architect for planning [3], which could be analogous for complex projects.
- A **Documentation Agent** similar to the "Scribe" role [4], which could generate documentation or explanations for the code after each cycle or at the end. For instance, updating a README with usage instructions or commenting the code. This isn't strictly TDD, but it could be a useful addition for learning-oriented katas or for completeness.

- A **Parallelization or Multi-threading of agents**: In future, some parts could run in parallel (for example, planning ahead while implementation is going on). However, TDD is inherently sequential for a single feature. Still, if multiple sub-problems are independent, an advanced version might tackle two test-implement-refactor threads concurrently and then merge, but that's complex and likely out of scope.
- **Persistent Learning and Memory:** Currently, each run of agentic-tdd starts fresh with just the kata description. In the future, we could allow the system to **learn from past runs** or have a knowledge base:
- If the kata has been done before, the tool could recall what tests were written or what pitfalls occurred (though this treads close to just retrieving a known solution, which might defeat the purpose of practicing the kata).
- However, maintaining a memory of how agents performed (e.g., implementer overshooting often) might help adjust prompts globally.
- **Better Error Recovery:** Future versions should handle unexpected scenarios more gracefully:
- If the LLM produces invalid code (syntax errors), the implementer or tester could catch that (from compiler errors) and automatically ask the LLM to correct the syntax.
- If an API call fails mid-run (network issue), the Supervisor could retry that step rather than abort everything.
- Implement timeouts for each agent's operation to avoid hanging (especially if an LLM gets stuck or a test hangs).
- **Multi-language Support:** Extend the tool to comfortably work with many languages:
- This means having knowledge of how to run tests in each language, how to structure projects, and possibly tailoring prompts to language specifics (for example, a Tester agent writing JUnit tests vs. PyTest tests vs. Rust tests are different styles).
- We can maintain a mapping of language -> test command and maybe a template project skeleton. Possibly infer language from file extension of kata description or a flag.
- Even within a language, different frameworks (like in JS, jest vs mocha) could be user-specified.
- In the long term, agentic-tdd could become polyglot, but testing that thoroughly will be an effort.
- **Agent Prompt Optimization:** With more usage, we will refine the prompts to reduce token usage and improve relevance:
- Use few-shot examples in prompts if needed (like showing an example of a good test output format to the Tester agent).
- Use shorter descriptions or references once the context is established (e.g., we might not need to send the entire kata text every single time if it's large; we could send a summary or just the remaining tasks).
- Possibly use the function-calling feature of OpenAI to get structured output (e.g., get a JSON of changes to apply rather than plain text diff – though plain code may be simpler).
- **User Interface Improvements:** While the CLI is the primary interface, we could consider:
- A web dashboard that visualizes the progress (each cycle, the tests and results, the code changes). This could be great for education, watching an AI solve a kata step by step.
- Integration with IDEs: Perhaps a VSCode extension that runs agentic-tdd in the background and updates the files in the editor as it goes, allowing a user to watch or even intervene.
- Hooks for user feedback: maybe a mode where it pauses after each cycle for the user to review and approve before continuing (semi-automated pair programming mode).
- **Inspiration from Research and Community Projects:** The field of AI-assisted software development is evolving rapidly. We will keep an eye on new frameworks or success stories:
- For example, the *RefAgent* research [2] we cited shows the power of multi-agent refactoring; we might adopt some of its strategies for our Refactorer Agent (like self-reflection steps).

- The multi-agent orchestration like in the Claude Code workflow [5] demonstrates benefits of specialization; we can continue to justify and tune our agent roles accordingly.
- If any open-source projects appear that attempt similar goals (like AutoGPT-like coding agents, or LangChain "workflows"), we can draw on their lessons or even integrate with them.
- **Scalability and Performance:** As models get larger and codebases get larger through the kata, performance might suffer:
- We might introduce caching of LLM responses for identical prompts to avoid recomputation.
- Use streaming outputs from the LLM to show partial results (some APIs allow stream tokens; we could print the test as it's being written, for instance).
- If cost is a concern, allow using smaller/cheaper models for some agents (maybe Tester can use a smaller model, Implementer uses a larger one for code quality, etc., configurable by user).
- **Safety and Ethics:** Ensure that the tool doesn't produce any harmful or illegal content. Generally, code katas are benign problems, but we should still respect the usage policies of models (e.g., avoid asking the model to output license-restricted code, etc.). Incorporating an open-source LLM entirely offline could be a future option for users concerned about data privacy (like using CodeLlama locally).

All these future improvements aim to make agentic-tdd more powerful, flexible, and user-friendly. The initial version focuses on the core loop of multi-agent TDD, and if successful, it opens the door to a new way of automating software development tasks. By continuing to iterate on this tool, we align with the vision of AI agents as collaborators in software engineering, where they can take on structured roles in a development methodology and reliably produce quality software [6] [7].

Conclusion

We have outlined a comprehensive plan for building **agentic-tdd**, a multi-agent TDD CLI tool. By following this plan – starting with a clear goal, implementing a first milestone focusing on core functionality, using a meticulous development process, and detailing the roles of each agent – we can incrementally develop a system where AI agents work together to solve programming katas. The design is heavily influenced by TDD best practices and recent innovations in AI-assisted coding, combining them to ensure that each bit of functionality is grounded in a test and improved through refactoring.

The success of agentic-tdd could demonstrate that complex software tasks can be broken down and delegated to specialized AI collaborators working in concert. In effect, we are treating the AI agents as members of a development team, each with limited scope but together covering the whole lifecycle of writing code. Early indications from similar multi-agent approaches show promise in terms of quality and speed [1] [2], so this project also serves as an exciting experiment at the frontier of software engineering and artificial intelligence.

By implementing all the elements in this plan step by step, we will create a tool that not only automates kata solutions but also serves as a blueprint for how AI can participate in structured software development processes. Each commit and each passing test in the development of agentic-tdd will bring us closer to a future where human developers can specify a goal and constraints, then oversee as a team of AI agents writes, tests, and refines the code – truly realizing a new level of automation in software development.

---

[1] [3] [4] [5] How I Built a Multi-Agent Orchestration System with Claude Code Complete Guide (from a nontechnical person don't mind me) : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1l11fo2/how_i_built_a_multiagent_orchestration_system/

[2] [2511.03153] RefAgent: A Multi-agent LLM-based Framework for Automatic Software Refactoring
https://arxiv.org/abs/2511.03153

[6] [7] Agentic Software Engineering: Foundational Pillars and a Research Roadmap | by Dixon | Sep, 2025 | Medium
https://medium.com/@huguosuo/agentic-software-engineering-foundational-pillars-and-a-research-roadmap-952410205d8e