ChatGPT

Plan for an Autonomous Bug-Fixing Agent

Overview of the Approach

Develop an autonomous **bug-fixing agent** that learns to fix software bugs by iteratively improving its own prompts and strategies. The core idea is to use a repository's commit history as a training ground: we have a codebase (e.g. a TypeScript/React project) with 100+ historical bug-fix commits, each labeled clearly as a bug fix. The agent will **simulate going back in time** – it will check out a prior commit (just before a known bug fix), run the tests to observe a failure, and then attempt to fix the bug at that commit. By comparing its attempt with the actual historical fix, the agent can **self-evaluate and refine** its prompting strategy. Over many such iterations, the agent should converge on an effective prompting method that *consistently produces correct fixes* for all these past bugs.

This approach treats the language model not just as a static code generator but as an **autonomous problem-solving agent** that can plan, execute, and learn from feedback [1] [2]. The agent will have access to tools like a Git client (for checking out commits) and a test runner, enabling it to mimic a developer's workflow of running tests, diagnosing failures, and applying fixes. Crucially, the agent's behavior will evolve over time: it uses **LLM-driven self-reflection** to improve its prompts or "policy" after each bug attempt, gradually increasing its success rate [3]. Below, we break down the system architecture, components, and workflow in detail.

System Architecture and Components

To achieve this, we will orchestrate **multiple components (agents)** working together, each with a clear role, under a central controller. A multi-agent design can provide specialized skills and allow collaboration, which is beneficial for complex tasks like debugging [4]. The key components include:

- **Orchestrator (Controller):** A program (written in TypeScript, possibly running on Node.js) that coordinates the process. It uses Git commands to manipulate repository state, runs tests, gathers outputs, and invokes the LLM agents as needed. The orchestrator contains the main loop logic (going through commits and iterations) and ensures the overall process follows the desired sequence.

- **Bug-Fixer Agent (Programmer):** An LLM (Claude by Anthropic) acting as the "programmer" responsible for generating code fixes. This agent is prompted with information about the failing test and relevant code context, and it produces a patch or modified code intended to make the tests pass. Its **system prompt** will define the task ("You are an expert TypeScript/React developer tasked with fixing bugs..." etc.), including rules like *"do not modify tests, only code"*. The **user prompt** for each attempt will include specifics: e.g. failing test details, error messages, and the current code snippet or file content that likely contains the bug. The Bug-Fixer should propose a code change that addresses the failure.

- **Tester (Tool/Agent):** A non-LLM component (or very simple agent) whose job is to run the project's test suite at a given commit. In practice, the orchestrator itself can serve this role by invoking the test command (for example, running `npm test` or a Jest/Mocha command for the TypeScript project) and capturing the output. The test results (which test failed, error stack traces, etc.) are then fed into the Bug-Fixer agent's prompt. In a more advanced setup, one could personify this as a "Tester agent" that simply returns results, but it doesn't require an LLM – it's essentially an automated script.

- **Prompt Refiner Agent (Critic/Coach):** An LLM (could be the same Claude model or a second instance) acting in a meta capacity to improve the Bug-Fixer's strategy. This agent comes into play when the Bug-Fixer fails to solve a bug. The refiner is given additional insight – specifically, it can be shown the *actual fix* from the commit history (the "ground truth" diff) and the Bug-Fixer's unsuccessful attempts. Its job is to analyze **why the Bug-Fixer's prompt or approach fell short** and suggest adjustments to the system prompt or strategy **without revealing the exact fix**. Essentially, it produces a revised prompt or new guidelines that might lead the Bug-Fixer to success on the next try. (It's important that the refiner only provides high-level guidance or hints, not the answer directly, to ensure the Bug-Fixer still "figures out" the fix on its own and generalizes the learning.)

- **Knowledge Base (for Patterns):** Optionally, maintain a knowledge base of common bug fix patterns or rules derived from the repository's history. For example, if many past bugs were off-by-one errors, missing null checks, or wrong state updates, we can compile those patterns. This can be included in the Bug-Fixer's system prompt as a set of **guidelines or examples** ("Common bug fix patterns include X..."). Prior work has shown that providing recurring fix patterns and examples can help an LLM-based agent address bugs more effectively [5]. We might generate this pattern list from the commit history itself (perhaps via an offline analysis or even as part of the agent's preparation steps).

- **User Interface (React Frontend):** A React app can serve as a dashboard to monitor and, if needed, intervene in the process. It can display the current commit being tested, the failing test output, the diff of the agent's proposed fix versus the original code, and whether tests passed. If a **human-in-the-loop** is desired, the UI could provide prompts for a human to approve a fix, provide a hint, or adjust a parameter (especially if the agent gets stuck). In early development, having a human oversee the agent's actions is valuable for catching obviously incorrect fixes or prompt adjustments. Over time, the goal is to minimize human intervention, but the interface remains useful for transparency and debugging the agent's decision-making.

This architecture ensures a clear separation of concerns. The orchestrator manages state and tools, the Bug-Fixer generates code changes, and the Prompt Refiner improves the approach. Such separation mirrors what some research prototypes do – e.g., **MarsCode** uses a multi-agent pipeline with roles like "Tester" and "Programmer" to iterate on bug fixes [6] [7], and **RepairAgent** uses a single LLM agent with a tool-using loop guided by a controller [8]. Here, we explicitly use two LLM roles to incorporate a self-improvement feedback loop.

Workflow: Commit-by-Commit Iteration

The process will proceed through the repository's commit history in reverse (from latest to earlier commits), targeting commits that were bug fixes. Here's the step-by-step workflow:

1. **Identify Bug-Fix Commits:** Using Git, scan the commit history (latest ~100 commits) for those labeled as bug fixes (e.g. commit messages containing "fix" or referencing a bug ID). For each such commit, note its hash and its parent commit hash. These pairs (parent -> fix commit) represent scenarios where the parent has a bug (tests failing) and the fix commit resolves it (tests passing). The plan is to test the agent on each such bug scenario in sequence.

2. **Initialize Agent Prompt (Iteration 0):** Before tackling any bugs, set up the initial **system prompt** for the Bug-Fixer agent. This prompt establishes the agent's role and general instructions. For example: *"You are an expert software engineer specializing in TypeScript/React. You are given a codebase and a failing test. Identify the cause of the failure and modify the code (without changing any tests) to fix the bug. Provide the patched code. Only output the code changes necessary, without any extraneous explanation."* Include any general rules (e.g., coding style guidelines, not introducing regressions, etc.). Initially, this prompt may be basic. We expect to refine it over time.

3. **Loop Over Bug Scenarios:** For each identified bug-fix commit (from newest to oldest):

4. **a. Checkout Buggy Commit:** Use the orchestrator to check out the **parent commit** (the state *before* the fix was applied). This puts the repository in a state where the bug is present.
5. **b. Run Tests:** Trigger the test suite on this commit. Likely use a script or CLI (`npm test`) to run all unit tests. Since this commit is known to have had a failing test (otherwise a bug-fix commit wouldn't exist), we expect at least one test to fail. Capture the test results output.
6. **c. Analyze Test Failure:** Parse the output to identify which test(s) failed and the error message or stack trace. Often, the failing test name, error type, and possibly a stack trace with a file/line number will be available. This information is crucial context. For instance, if a test assertion failed saying "Expected X but got Y in function Z", or a stack trace points to a specific file, the agent should use that to zero in on the faulty code.
7. **d. Prepare Bug-Fixer Prompt:** Invoke the **Bug-Fixer agent (Claude)** with a prompt containing:
    - The **system prompt** (current version of it, which may have evolved).
    - A **user prompt** describing the situation: e.g. "One test is failing with the following error..." followed by the relevant excerpt of the test output (assertion message, stack trace). If the failing test code is short or the error points to a specific function/file, include those code snippets for context. The prompt might say: *"Fix the bug so that all tests pass. Do not modify any tests. Only modify application code. The failing test is in* `UserModel.test.ts` *line 42: it expected an email validation to reject invalid addresses, but the assertion failed. The error was 'Email validator accepted invalid input'. The relevant code is in* `UserModel.ts` *(see below). Please diagnose and fix the issue."* Then include the content of `UserModel.ts` or at least the function suspected to be buggy.
    - By providing both the failing test info and the likely faulty code, we focus the LLM on the right area. (In cases where it's unclear which code is to blame, the agent might have to search the codebase. We could implement simple search tools: e.g., find in repository for the test name or error message. However, since we have the actual fix commit, a pragmatic approach is to know which files were changed by the fix and use that as a hint for context. The orchestrator

can load those files from the buggy commit version and supply them as context. This is a bit of "oracle" help, but it dramatically narrows the search space. Commit history is known to contain valuable clues for bug fixing [9], so leveraging it to guide the agent's focus is acceptable as long as the agent doesn't see the *content* of the fix.)

8. **e. Generate Fix:** The Bug-Fixer LLM (Claude) receives the prompt and outputs a proposed fix. Ideally, we ask it to provide the patch in a structured way, for example: "Respond with the diff of changes in unified diff format" or "output the revised code for the modified functions only". Having a **structured output** makes it easier for the orchestrator to apply the changes. Claude is capable of understanding and producing diffs or code blocks. We might include in the system prompt an instruction about output format (e.g., only code changes in markdown format or a unified diff) [10] [11].
9. **f. Apply the Fix:** The orchestrator takes the LLM's output and applies it to the codebase. This could be done by parsing a diff or simply replacing the old code with the new code from the response. (We must be careful to validate that the LLM's suggested changes are intended as replacements or insertions in the right place. If we enforce a diff format, we can apply it programmatically. Alternatively, we could integrate a tool that allows the LLM to edit files directly if it were an agent with file system access, but for safety and simplicity, orchestrator-mediated application of changes is fine.)
10. **g. Re-run Tests:** With the proposed fix applied, run the test suite again via the Tester.

    - If all tests pass now: **Success!** The agent managed to fix the bug. We record this outcome (and perhaps the diff it produced) for analysis. We then proceed to the next bug scenario (go back to step 3a for the next older commit).
    - If tests are still failing (either the same failure persists or new failures arose): the fix attempt was not successful. Collect the new test output. This now forms feedback for the next iteration on this same bug.

11. **h. Inner Fix Iteration:** Allow the Bug-Fixer to iterate on the fix if it didn't succeed initially. We can loop a few times (say up to 3 attempts) where each time:

    1. The latest test results (failure info) are fed back to the Bug-Fixer LLM, possibly along with a reminder of what it tried last (or it can infer from the code state).
    2. The LLM suggests a revised fix (it might modify its previous solution or try a different approach).
    3. Apply the changes and run tests again. This is similar to an **internal refinement loop** where the LLM gets to adjust its solution based on runtime feedback (a capability demonstrated in iterative repair research [12] [2]). In practice, if the first fix attempt was close, this might resolve simple mistakes. However, often an LLM might not improve much without new guidance – it could oscillate or make different wrong changes. That's where the next step comes in.

12. **i. Prompt Refinement Loop (Meta-Learning):** If the Bug-Fixer agent fails to fix the bug after a few direct tries, we invoke the **Prompt Refiner (Critic) agent** to step in. This agent will use the historical knowledge (the actual commit that fixed the bug) to improve our approach:

    1. The orchestrator loads the *actual diff of the bug-fix commit* (i.e., what changes a human developer made to fix the bug).

2. We **do not want to simply feed this diff to the Bug-Fixer**, as that would solve the bug by copying the answer (defeating the purpose). Instead, we use it in a separate prompt to the Prompt Refiner LLM. For example, we might prompt: *"The agent attempted to fix the bug but tests are still failing. Here is what a human developer did to fix this bug (diff below), and here is what the agent tried (diff or description). Without giving away the exact solution, identify what high-level aspect the agent missed or what hint could have led it to the correct fix. Suggest how to modify the Bug-Fixer's system prompt or strategy to account for this."* Then include the actual diff as reference, maybe also the agent's last diff.
3. The Prompt Refiner (Claude) will analyze the differences. Perhaps it sees that the real fix involved adding a null-check, or adjusting a regex, or handling a missing case. It then might output something like: *"The agent's prompt should remind it to consider input validation for email addresses,"* or *"Emphasize checking off-by-one errors when dealing with array indexing."* In essence, it produces a **generalized coaching tip** or a tweak to the instructions that would help catch this category of bug. This works because LLMs can do abstraction – we leverage Claude's ability to provide natural language feedback and critique agent behavior [3].
4. Update the Bug-Fixer's system prompt (or any prompt template we use) to incorporate this advice. For instance, we might append to the system prompt guidelines a new bullet: "- Always verify input validation logic when tests failing for incorrect input handling," if that was the lesson. Over time, these accumulated guidelines form a more robust prompt. (It's akin to the agent "learning" from mistakes, without changing the model's weights – we're encoding the learning in the prompt itself).
5. Reset the code to the original buggy commit state (to undo any partial changes from failed attempts), and try the Bug-Fixer agent again *from scratch* on this bug, but now with the improved prompt. Because the prompt now contains a more pointed hint or strategy, the hope is that the LLM will generate the correct fix this time.
6. Run tests. If it passes now, great – the prompt adjustment helped. If not, we might repeat the refine loop again with any new insights (up to a certain limit, say 10 iterations total as suggested). Each iteration might add or adjust a guideline. There is a risk of overfitting extremely specific hints, so we should focus on **general heuristics** ("check boundary conditions", "consider concurrency issues", etc.) rather than anything that only applies to one bug.

13. **j. Success and Logging:** Once the agent successfully fixes the bug and tests pass, log the outcome. We should record:

    - The final diff the agent produced.
    - How many attempts it took.
    - What prompt refinements were added due to this bug.
    - Perhaps a short summary of the bug and fix for future reference. This data will be useful for evaluation and for building a knowledge base of what was learned. We will also compare the agent's diff to the actual human diff to see if it came up with a similar solution or an alternative fix. (Both are fine as long as tests pass; an alternative solution might be acceptable if it's semantically correct. However, a drastically different fix might indicate the agent found a hack; we'd examine that in evaluation.)

14. **Iterate Through All Bug Commits:** Continue the above loop for the next bug-fix commit (i.e., move further back in history). Importantly, the **prompt improvements we made carry over** to the next bugs. This means as we go back to older bugs, the Bug-Fixer agent's system prompt is progressively

enriched with all the guidelines and insights gained from later (more recent) bugs. We anticipate that over multiple examples, the prompt becomes more comprehensive and the agent's first-try success rate will increase. The ultimate goal is that by the time we've processed all 100 commits, the agent's prompt (and approach) is so well-tuned that it could handle new, unseen bugs in this codebase (or even in similar projects) in one shot. Essentially, we are **meta-training the agent through prompt evolution**.

15. **Human-in-the-Loop Checkpoints:** At certain milestones or failure points, involve a human if necessary. For example, if after 10 refinement iterations on one bug the agent still can't fix it, a human engineer might review the bug and the agent's attempts to identify what went wrong (maybe the bug requires a very high-level design change that the prompt didn't cover). The human could then add a guiding rule to the prompt manually. Also, as the prompt grows, a human should review it to remove any redundant or conflicting advice. A React UI could present the current prompt and allow an expert to edit the guidelines list or rephrase instructions in a clearer way for the LLM. This ensures the prompt remains coherent and doesn't accumulate too much "prompt debt."

Throughout this workflow, **Claude** (the LLM) is central: it powers both the Bug-Fixer and the Prompt Refiner roles. Claude is a good choice here because it has a high token limit (useful for supplying code context and multiple examples in prompts) and is known for following complex instructions well (Anthropic's Claude was designed with conversational and iterative tasks in mind). By using the same underlying model for both roles, we leverage consistency in understanding. However, we must be careful in how we structure the prompts to clearly delineate the tasks (fixing vs. critiquing) so that we get focused outputs.

Using TypeScript and React for Implementation

The implementation will likely involve a **Node.js (TypeScript) backend** orchestrating the process, with an optional **React frontend** for monitoring:

- **Backend (Node + TypeScript):** Here we will implement the Orchestrator. We can use libraries like `simple-git` (a Node Git toolkit) to check out commits programmatically and manage branches. For running tests, we can spawn a child process (e.g., `npm run test`) and capture stdout/stderr. Parsing the test results can be done by looking for known patterns (for example, if using Jest, we can detect the summary of failed tests and the failure messages). The backend will also handle calls to the Claude API. We'll need to format the prompt as per Claude's expected input (system vs user messages), send it via HTTP request, and wait for the response. TypeScript will help us keep the data structures (commit lists, diffs, prompts, etc.) well-defined.

- **Managing State:** The Orchestrator will maintain state such as: current commit index, current attempt count, current prompt version (the set of guidelines and instructions we've accumulated). We might represent the prompt as a template string or a data structure where we can add "rules". For example, an array of guideline strings that we join into a prompt section each time we send to the LLM. This makes it easy to append new ones or modify existing ones via code.

- **React Frontend:** A dashboard that connects to the backend (via WebSocket or polling) can display:

- **Current Task:** which commit (perhaps its message) is being worked on, what is the failing test.

- **Agent's Attempt:** show the diff of the code the agent is proposing to change. This could be visualized side-by-side with the original code for the human to inspect.
- **Test Results:** after each attempt, indicate pass/fail (and possibly display the error message if failed).
- **Prompt Evolution:** show the current system prompt and highlight new additions made by the Prompt Refiner. This transparency can help the developer see what the agent has "learned" so far.
- **Controls:** let the human pause, intervene with a hint, or skip a bug if needed.
- **Metrics:** display counters like how many bugs fixed successfully, average attempts per bug, etc.

Using React for this interface will make it easier to manage the complex state updates in real-time and present a clean visualization. It also facilitates the human-in-the-loop aspect, as the human can interact with the process in a user-friendly way (compared to reading logs on a console).

Claude as the LLM for Bug Fixing and Critique

We choose **Claude** (from Anthropic) for the LLM agent for several reasons:

- **High Context Window:** Claude's large context window (up to 100K tokens in Claude 2) means we can feed in quite a bit of code and conversation history. For debugging, this is useful – we can include the relevant source file content, the failing test, and even some summarized history of what's been tried in this session without running out of space. This reduces the need to strictly limit context, allowing a more flexible conversation with the agent.

- **Instruction Following and Dialogue:** Claude is built with a conversational format and is good at following complicated instructions and constraints (due to Anthropic's Constitutional AI approach). We will be heavily instructing the model (e.g., "don't reveal solution, don't change tests, try simple fixes first, etc."). Claude's compliance with such instructions and its ability to produce well-structured answers (like diffs or code blocks) is advantageous.

- **Multi-step Reasoning:** Debugging a bug is a multi-step reasoning task. Claude has shown strengths in maintaining coherent reasoning over a dialogue. Even if we implement the Bug-Fixer as a single-turn prompt (with all info at once), behind the scenes Claude can reason through it. If we instead allowed a chain-of-thought (like asking the agent to output its reasoning and then an answer), Claude would likely excel. We might utilize this by prompting it to first explain what it thinks the bug is before giving the fix, though for automation we might prefer it go straight to code. Still, if needed, we could use Claude's response to have it explain its fix (perhaps via comments in code, as done in RepairAgent [5], to double-check its reasoning).

- **Refinement and Self-critique:** For the Prompt Refiner role, we need the LLM to introspect on the difference between its attempt and the actual solution. Claude is known to be quite good at **self-reflection** and critique tasks (Anthropic has demonstrated Claude's ability to analyze and improve responses when guided to do so). By using the same model for this reflective task, we keep consistency. We essentially ask Claude to look at a solution and derive a general lesson – a task akin to summarization and instruction-honing, which it can handle.

To integrate Claude, we'll use its API. We may have two API endpoints or calls – one for the Bug-Fixer (with the code and test context) and one for the Prompt Refiner (with diff comparison context). We need to ensure not to mix up the conversations. Each bug scenario might be a separate session or conversation

with Claude to avoid any leakage of solution from the refiner call to the fixer call. The orchestrator will explicitly manage these calls: e.g., call Claude with prompt X to get fix, separately call with prompt Y to get critique. We won't rely on multi-turn memory across those roles (to be safe), but within one attempt's conversation we might allow multi-turn if needed (like if the agent asks for clarification – though ideally, we design prompts so it doesn't need to ask questions interactively, since we want automation). If we do find multi-turn is needed (say the agent wants to see another file, etc.), the Orchestrator could detect that in Claude's response and act accordingly (similarly to how tool-using agents work [8], though that adds complexity). Initially, we aim for a straightforward single-turn fix suggestion given all info.

Self-Improvement via Prompt Evolution

A central aspect of this system is **continuous learning without modifying the model itself**, i.e., learning via prompt engineering adjustments and strategy improvements. Each bug the agent fails to fix on the first try is an opportunity to refine its approach. Over time, the prompt itself becomes a repository of distilled best practices for bug fixing. Essentially, the agent's prompt starts simple and gradually **evolves into a sophisticated instruction set** tailored to debugging this project (and possibly generalizable beyond it).

This method is inspired by ideas in recent research on self-improving agents: - The agent uses its own failures and the ground truth outcomes as feedback, akin to how one might fine-tune, but here done in natural language space (prompts) rather than gradient updates [13]. - Approaches like **PromptBreeder** and others have explored letting LLMs mutate and improve their prompts through iterative processes [3]. We apply that here by having an LLM (Prompt Refiner) explicitly propose prompt mutations after analyzing mistakes. - The advantage is data-efficiency and interpretability: we can see each rule the agent "learned" in plain language. For example, after a few bugs, the prompt might include rules like *"When dealing with numeric calculations, check for off-by-one errors"* or *"If a test expects an error to be thrown, ensure the code actually throws it rather than just logging"*. These are human-understandable heuristics that improve the agent's performance.

We need to be careful to keep these prompts **general** and not overfit to one specific bug. The instruction "if test X fails, do Y" is too specific and not useful elsewhere. Instead, general lessons ("handle date parsing for invalid formats") can apply to multiple scenarios. The human overseeing the process can validate that new guidelines are phrased generally. If a guideline seems too narrow, the human might generalize it further before adding.

In addition to prompt rules, the agent could also improve how it uses tools or the order of steps: For instance, it might learn that "running tests after each small change" is good (which we already enforce), or if the project has linter/type-check errors, those should be fixed first, etc. If such patterns emerge, we incorporate them into the workflow or prompt.

Another form of self-improvement is internal: we could instruct the Bug-Fixer agent to **reflect on the test output deeply** before coding. For example, part of its prompt could say "explain to yourself what the root cause might be before writing the fix." This can trigger Claude to internally reason (possibly outputting a summary of the bug in a comment) which might improve the quality of the fix. RepairAgent, for example, has the LLM produce a hypothesis of the bug cause and even comment its reasoning in the code [14] [5]. This not only can improve the solution but also produce a natural language explanation that we can log. We might not want to commit those comments, but they could be stripped after confirming the tests pass.

Human-in-the-Loop and Orchestration Strategy

While the goal is full autonomy, having a **human in the loop** initially will greatly enhance the reliability of the system:

- **During Development & Testing:** A human can monitor how the agent is performing on each bug and step in if something goes awry (e.g., the agent misunderstands the test and deletes it instead of fixing code – our instructions forbid test changes, but the human should watch for any rule breaking or destructive changes the agent might attempt). The human can also provide clarifications if the test failure is too ambiguous. For example, if the test output isn't clear, a human could annotate the prompt with an extra hint like "(The failing test indicates that the function doesn't handle empty input correctly)". We could also incorporate a mechanism for the agent to ask for help. If after several tries it's failing, the agent (Claude) could be prompted to say "I am not sure what to do; maybe I need a hint about X." The orchestrator can surface that message on the UI for a human to see and respond.

- **In Prompt Refinement:** The suggestions given by the Prompt Refiner agent might sometimes be off-base or too verbose. A human should review the new prompt guidelines before appending them. This prevents accumulating misleading advice. Since these guidelines are in natural language, a human engineer can vet whether they make sense. Over time, as confidence in the system grows, this review could be loosened or automated by validating that each new rule actually helped fix the bug it was intended for (if not, maybe discard that rule).

- **Fine-Tuning Data Verification:** If we plan to use the data from this process to fine-tune models later (discussed below), human oversight is critical to label outcomes correctly (success/failure), filter out bad attempts, and ensure the solutions are truly correct. For instance, an agent's fix that passes the tests might still be semantically wrong in some subtle way that tests didn't cover. A human might catch that by inspecting the diff and decide whether to count that as a "successful fix" in the training data.

From an orchestration perspective, integrating human input means our system should allow pausing and resuming, and possibly a mode to accept a human-proposed fix or prompt tweak. Practically, the orchestrator could await a "green light" from the UI after showing an agent's diff before applying it, if we want a manual check. Alternatively, run in an automatic mode that logs everything for later review, which is useful when running overnight experiments.

Continuous Improvement and Fine-Tuning Strategy

After running this agent on many historical bugs, we will have a wealth of information: - A list of prompt adjustments that were made, along with notes on which bugs prompted them. - Transcripts of attempts per bug (initial prompt, model output, test results, refined prompt, etc.). - The final diffs the agent produced and whether they matched the human diffs or not. - Cases where the agent struggled and why.

Initially, simply **writing these insights to markdown files** is a great way to build an internal knowledge base. For each bug scenario, we can create a Markdown report that includes: - The commit identifier and description of the bug. - The failing test and error. - The agent's final fix diff (and possibly intermediate diffs).

- The actual fix diff from history. - Any new prompt guidelines learned. - How many tries it took. - Observations (did the agent's fix differ significantly from the actual fix? Did it introduce a different approach?).

These markdown files serve two purposes: **documentation and data for future training**. They are easily readable by humans to understand how the agent is learning, and they can be parsed by scripts if we decide to extract a structured dataset for model fine-tuning.

Fine-Tuning the Models Over Time: Once we've accumulated enough cases (perhaps after going through all 100 commits), we can consider training a specialized model or fine-tuning an existing one (like a smaller LLM or even Claude itself if that becomes possible) on this data: - One approach is to fine-tune a model to directly output code fixes given a bug description. For example, create (prompt, code fix) pairs from our logs and train a model to imitate the successful fix. However, this alone might not teach the *process* of debugging. - A better approach might be to fine-tune the model on the **process**: incorporate the idea of chain-of-thought and tool use. We could assemble trajectories (state -> action -> result sequences) from the logs to train the model to perform the whole sequence in one go. This is complex and maybe unnecessary if prompt-based learning suffices with an external orchestrator. - We can also fine-tune on the **prompt refinement** knowledge: basically turning the learned prompt into a model prior. For instance, if we have 100 guidelines that were added, a fine-tuned model might internalize those so it doesn't need them explicitly in the prompt (making it generalize better or at least make the prompt shorter). - Anthropic's Claude might not be openly fine-tunable by users, but we could use an open-source model (like LLaMA 2 or similar with code understanding) and fine-tune it on our bug-fix transcripts. That model could then serve as a self-contained bug fixer agent (with hopefully lower cost than API calls). The prompt evolution strategy is essentially a way to generate training data for such a fine-tune in a domain-specific way, which is powerful because it's *empirically grounded in real bug fixes*.

It's worth noting that our approach is a form of **non-gradient, natural language adaptation**. The literature has shown that letting agents self-reflect and update their prompts can yield significant performance gains without any weight updates [13]. By writing our results to markdown and curating them, we ensure that if we do fine-tune, we're using high-quality, validated information.

In the short term, even without model fine-tuning, the evolving prompt might be enough to handle most bugs ("prompt engineering as training"). In the long term, fine-tuning could consolidate these gains and possibly allow the agent to handle more complex bugs or larger codebases because the model itself becomes more specialized.

Evaluation Metrics and Success Criteria

We will measure the agent's performance and improvement using several metrics:

- **Test Passing Rate:** This is the primary success criterion. A fix is only good if it makes all the tests pass (and we did not alter any tests). So for each bug scenario, we check if the agent eventually achieved a passing test suite. We can track the fraction of bugs (out of the 100) that the agent fixed successfully *without human intervention*. As the prompt improves, this fraction should increase. We might start with low success and ideally reach near 100% on the training set of bugs (since those are historical ones it learned from). More importantly, we can then try some *unseen* bug (if available) to see if the prompt generalizes.

- **Number of Attempts (Efficiency):** How many attempts (LLM calls and code iterations) did it take per bug? This includes both fix attempts and prompt refinement cycles. Fewer is better. We expect this to drop over time – earlier bugs might take many tries and prompt tweaks, whereas later ones (with a mature prompt) might be one-shot or only a couple of tries needed. We will monitor average attempts per bug over the timeline as a measure of the agent's efficiency gains.

- **Diff Quality – Size and Scope:** We will evaluate the **diff size** of the agent's fixes compared to the human fixes. Ideally, an automated fix should be minimal and targeted (changing only what's necessary to fix the bug), just like a good human fix. If the agent's diff is significantly larger than the human's, it might indicate it's doing unnecessary changes or possibly overfitting. We'll measure lines changed: e.g., if the human fix was +5/-2 lines and the agent did +30/-10, that's a flag. Smaller, focused diffs are preferred. We might score diffs by how closely they match the human diff in size and even content:

- We can use a simple similarity metric (like overlap of changed functions or similarity of patch hunks) to gauge **semantic correctness**. If the agent's fix is different but all tests pass, that's okay, but we need to be sure it didn't just paper over the symptom. For instance, if a test expected an error thrown and the agent simply removed that part of the test, that would be a bad fix (though our instructions forbid altering tests). Or if the agent fixed the symptom in an incomplete way, another test might catch it. So, running the *full test suite* is important to ensure semantic correctness as far as tests can tell.

- We might also run additional static analysis or lint checks to ensure the fix doesn't introduce new warnings or errors. This is a secondary check on quality.

- **Generalization of Prompt:** Although harder to quantify, an important outcome is how general our learned prompt is. After training on 100 known bugs, can the agent fix a brand new bug (for example, a new issue that arises in the codebase) without further help? We could simulate this by holding out some commits as a "test set" for the agent – though given all commits are used for training here, we might instead introduce a synthetic bug or use a bug from a different project to test transfer. If the agent performs well, that's a strong sign of successful generalized learning. The ultimate measure of success is if this autonomous agent can be run on real bug reports or failing tests in the future and fix them **without any hardcoded solutions**, relying only on its learned prompt strategy.

- **Cost and Speed:** We should track how many LLM tokens are used and how long each bug fix takes. This system could be token-intensive (each attempt might feed a lot of code into Claude, and multiple attempts per bug). The RepairAgent paper reported about 270k tokens per bug for GPT-3.5 [15]. We might try to be more efficient by limiting context to relevant info. Still, understanding the computational cost is important for practical deployment. If each bug costs a lot, we may optimize by caching results or eventually fine-tuning a cheaper model.

- **Manual Evaluation of Fixes:** In addition to test results, a human developer can review the agent's fixes for a sample of bugs. They can rate whether the fix is *readable, maintainable, and truly fixes the root cause*. This subjective metric ensures our agent isn't just appeasing tests in a hacky way but writing quality code. For example, if a test expected an exception for invalid input, a proper fix is to add input validation and throw an error. A "hack" fix might be to change the test's expected value or

force an error in an unnatural way. Our constraints and metrics aim to catch these (test integrity and diff size), but a human eye is the final judge of semantic correctness.

By analyzing these metrics, we'll identify where the agent needs improvement. For instance, if diff size is consistently larger than humans', maybe the prompt needs to encourage minimal changes or the agent needs a better understanding of the project's design to not over-correct. If attempts per bug remain high for certain types of bugs (e.g., concurrency issues or UI rendering issues), that indicates a domain where our prompt guidelines might be lacking specific rules.

Orchestration and Agent Collaboration

Tying it all together, here's how the **multi-agent orchestration** works in practice:

- The **Planner/Orchestrator** (not an LLM, just our code) drives the sequence of actions: check out commit, run tests, call Bug-Fixer, apply diff, call Tester, etc. This deterministic backbone ensures the process flows logically and nothing is missed.

- The **Bug-Fixer (Programmer agent)** and **Tester** work in a loop much like a developer running tests and editing code. In fact, this resembles a REPL for debugging: test fails -> change code -> test again -> repeat. This dynamic is exactly what MarsCode's dynamic debug workflow described: the Tester provides feedback to the Programmer, who then modifies code until the issue is resolved [6] [7]. We are automating the Programmer's role with the LLM.

- The **Prompt Refiner (Critic agent)** acts as a mentor overseeing the Programmer's learning. After the programmer-agent has "given up" (or hit a limit), the critic examines the solution and the attempts. This separation of roles is helpful because the mindset for critique is different from generation. By switching to a critic role, the LLM can analyze in a more detached manner. It's similar to how code review works: a second perspective (often even run by the same person or model but in a different mode) can spot why the initial approach failed.

- We should ensure there's a clear protocol for information flow between these components. For example:

- The Programmer agent **must not** see the actual fix diff. Only the Critic sees it. The Critic's output is filtered to remove any direct solution. If the Critic accidentally outputs a chunk of the solution code (which we should caution it against in the prompt), the orchestrator should not pass that to the Programmer. Possibly, the orchestrator can vet the Critic's advice to ensure it's abstract. We can include instructions in the Critic's prompt like "Do not reveal the exact code changes needed, only suggest improvements in the guidance."

- The Programmer agent's attempts and the Critic's guidance should both be logged. If the Critic suggests adding "Consider edge cases with empty inputs" to the prompt, we add it and note that it came from bug X. If later this guideline seems to cause any issues or is no longer needed, a human can prune it.

- **Single vs. Multiple LLM Instances:** We could use one instance of Claude for both roles by resetting its conversation and giving it the appropriate prompt each time. However, conceptually it's two

agents. In the future, one could imagine two models specialized differently (one fine-tuned for code-gen, one for analysis). Initially, using the same model is fine – just ensure each call is isolated. The orchestrator essentially plays the role of a facilitator ensuring the right prompt goes to the right role at the right time.

- **Error Handling:** We should prepare for cases like:

- The LLM returns an invalid patch or something that doesn't compile. Our test run might reveal a compile error or TypeScript type error rather than a test assertion. In such cases, treat it as a test failure (or separate step to handle compilation). Possibly add a guideline like "Ensure the code compiles and types pass" after any such event.
- The LLM might misunderstand the format and give a long explanation or incomplete diff. The orchestrator should be ready to handle that (perhaps by reprompting: "Please output only the code changes").
- Git conflicts or inability to apply diff – if the diff is ambiguous or doesn't apply, we might have to fall back to letting the LLM output full file content for replacement. Our system should handle patching robustly.
- Timeouts or API failures – have retry logic or slow down if needed.

By carefully orchestrating these interactions and maintaining a feedback loop, we essentially create a mini **autonomous debugging agent** that learns on the fly. Each component (tester, programmer, critic) plays a part similar to roles in a software team, and the orchestrator is the project manager ensuring things move along.

Future Extensions and Conclusion

Once the system works on the given repository, we can extend it in various ways: - Incorporate other tools: for example, a static analysis tool or a documentation search tool. The agent could query the project's documentation or comments for hints. Commit messages themselves might contain clues ("Fix null pointer in handleSubmit" – this could be used to guide the agent). We avoided giving commit messages to the Bug-Fixer to make it earn the fix, but perhaps the commit message (since it's natural language) can be given to the Critic to help form better advice. - Apply to other codebases: test the generality by taking the learned prompt (and perhaps a fine-tuned model) to a different project with a set of known bugs. - Continuous learning: As new commits get added to the repository (new bug fixes), periodically run the agent on them to further refine its prompt/model. This could integrate into CI: whenever a developer fixes a bug, the agent tries to fix that bug from the prior commit; if it fails, that's a new learning opportunity. Over time, the agent could start catching bugs before the human even writes the fix (truly becoming a coding assistant). - Limitations to address: Some bugs might be very complex (spanning multiple files, or requiring understanding long-range effects). Our approach currently depends on the failing test pointing to the issue. We might need to enhance the agent's ability to **localize bugs** in more complex scenarios. Techniques from Code Researcher (which searches commit history and codebase for relevant context) could be integrated – e.g., the agent could search for similar error messages in commit history or look at when a function was last modified [16] [17]. This can enrich the context for tricky bugs.

In summary, **yes, we will use multiple agents (or agent roles) orchestrated in a careful loop** to build a self-improving bug fixer: - A **Programmer agent** (Claude) to propose code fixes, - A **Tester tool** to evaluate

them, - A **Critic agent** (Claude) to analyze failures against ground truth and refine strategy, - All coordinated by a TypeScript orchestrator, with a React UI for monitoring and human feedback.

This multi-agent collaborative setup is designed to incrementally approach the performance of a human developer in debugging tasks [4]. By learning from each bug in the commit history, the agent's capabilities grow with experience, much like a human gaining expertise over time. The combination of prompt-based learning and (eventually) model fine-tuning will ensure that the agent not only memorizes those specific fixes, but truly **learns how to fix bugs** in a generalizable way. The careful tracking of metrics like test success, diff correctness, and prompt evolution will let us verify that the agent is on the right track and guide its development moving forward.

With this plan, implementing a real-life autonomous bug-fixing agent becomes feasible. It is a challenging project, but by breaking it down into these components and feedback loops, we leverage the strengths of LLMs (code understanding and generation, natural language reasoning) while mitigating their weaknesses (lack of grounding, tendency to go off-track) through tooling and iterative refinement. The end result will be a system that not only fixes bugs but continually gets better at doing so.

Sources:

- Bouzenia et al., *"RepairAgent: An Autonomous, LLM-Based Agent for Program Repair,"* 2023 – describes an agent that uses an LLM with tools to iteratively fix bugs [2] [8].
- Microsoft (Raza et al.), *"Code Researcher: Deep Research Agent for Large Systems Code and Commit History,"* 2025 – highlights the value of using commit history in automated bug resolution [9].
- MarsCode Agent – blog post on a multi-agent framework for automated bug fixing, illustrating the collaboration between tester and programmer agents in a loop [6] [7].
- Robeyns et al., *"A Self-Improving Coding Agent,"* 2023 – demonstrates an agent that improves itself via self-reflection and code edits, providing context on prompt optimization and meta-learning for agent behavior [13] [3].
- Additional insights were drawn from the above and adapted to this specific use-case of commit-based bug fix training.

---

[1] [2] [5] [8] [10] [11] [12] [14] [15] RepairAgent: An Autonomous, LLM-Based Agent for Program Repair
https://arxiv.org/html/2403.17134v2

[3] [13] A Self-Improving Coding Agent
https://arxiv.org/html/2504.15228v2

[4] [6] [7] MarsCode Agent: Powered by LLM for Automated Bug Fixing - DEV Community
https://dev.to/marscode/marscode-agent-powered-by-llm-for-automated-bug-fixing-2j7a

[9] [16] [17] microsoft.com
https://www.microsoft.com/en-us/research/wp-content/uploads/2025/06/Code_Researcher-1.pdf