# Building an Interface for LangChain-like LLM Orchestration in TypeScript

Large Language Models (LLMs) can now assist in coding tasks, even to the point of *self-healing* their own code. In a self-healing system, the AI iteratively fixes errors in its output until all tests pass – the kind of smooth, automated development process developers dream about. This concept is cutting-edge and relatively rare, but early experiments are promising. For example, even a basic prototype that auto-fixes failing builds could handle simple issues (like lint errors or merge conflicts) and save “hundreds of dev hours” for developers[[1]](https://www.reddit.com/r/LangChain/comments/129exd4/selfhealing_github_action_using_langchain_and/#:~:text=Love%20it%21%20I%E2%80%99m%20not%20saying,dev%20hours%20at%20big%20companies). To implement such capabilities, we need an interface that can orchestrate complex LLM workflows. One approach is to simulate the functionality of **LangChain**, a popular framework for LLM applications, by building our own lightweight chaining interface in TypeScript.

## What is LangChain and Why Simulate It?

**LangChain** is an open-source framework and toolkit for constructing LLM-powered applications[[2]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=LangChain%20is%20an%20open,com). It provides a **standard interface** to integrate language models with various tools and data, allowing developers to chain together prompts, model calls, and other operations into coherent workflows[[3]](https://www.langchain.com/langchain#:~:text=LangChain%20is%20the%20open%2C%20composable,fast%20as%20the%20ecosystem%20evolves). LangChain is available in both Python and JavaScript, catering to a wide range of development environments[[2]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=LangChain%20is%20an%20open,com). In essence, LangChain lets you break a complex task into a series of steps (a "chain"), each step handled by an LLM or a utility, and manage the flow of data between steps.

**Why simulate LangChain?** In some cases, LangChain’s official libraries might feel too heavy or not perfectly suited to a specialized project. For instance, developers have noted that LangChain can be “bloated,” prompting some to create lightweight TypeScript frameworks of their own[[4]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=). Building a custom interface gives you full control and the ability to integrate domain-specific logic. In your case (an **AI self-healing debugger**), a custom chain interface can be tailored to run code tests, apply code patches, and interact with your systems (possibly via a Model Context Protocol, as suggested by your use of the **MCP** SDK). By simulating LangChain, you’ll implement just the features you need for your LLM to reliably write and fix code.

## Core Concepts of a LangChain-Like Interface

When designing an interface similar to LangChain, keep in mind the key components and concepts LangChain provides:

* **LLM Abstraction** – A unified way to call different language models. You’ll want to create a class or function that wraps API calls to your model (OpenAI, etc.) so that your chain can invoke the LLM easily. LangChain’s value is in offering a standard model interface for any provider[[3]](https://www.langchain.com/langchain#:~:text=LangChain%20is%20the%20open%2C%20composable,fast%20as%20the%20ecosystem%20evolves), so your interface should do the same (perhaps using MCP to remain model-agnostic). For example, you might define an LLMModel interface with a method like generate(prompt): Promise<string>.
* **Prompt Templates** – Manage prompts and formatting. LangChain has prompt template classes to inject variables into prompts. In a custom interface, you can simply use template strings or small helper functions to format prompts for each step. The goal is to clearly define how each query to the LLM is constructed (including system instructions, if using chat models). For instance, LangChain’s prompt classes allow mixing system/user messages[[5]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=def%20checks_generation%28task%29%3A%20,exactly%20suits%20the%20question%20given)[[6]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=prompt%20%3D%20ChatPromptTemplate.from_messages%28%5B%20%28,exactly%20suits%20the%20question%20given); you can achieve similar structure with your own template strings or JSON for chat roles.
* **Chains (Sequential Steps)** – The core of LangChain is the ability to link multiple steps (LLM calls or functions) in sequence. Each step takes some input (possibly the previous step’s output) and produces output for the next step. Under the hood, this is essentially passing a **state** through a pipeline. As one LangChain guide notes, traditional software follows a fixed sequence, but in AI workflows the sequence can be dynamic based on model outputs[[7]](https://dev.to/mohsenkamrani/master-langchain-in-typescript-a-practical-guide-4ifc#:~:text=Each%20node%20adds%20a%20specific,one%20step%20to%20the%20next). Your interface should allow representing a sequence of operations and executing them one by one, possibly altering the flow depending on intermediate results. This could be as simple as an array of async functions that you call in order.
* **Tools/Actions Integration** – LangChain allows LLMs to use tools (e.g. web search, calculators, code execution). For a self-healing code agent, your primary “tools” are likely the ability to run code/tests and to apply patches to the codebase. You should design your chain so that not every step is an LLM call; some steps will be programmatic actions (like running a test suite, or parsing the LLM’s output into a code patch). Your chain framework can handle this by permitting arbitrary functions in the sequence. For example, one step in the chain might call the LLM to propose a code fix, and the next step runs the test suite to verify the fix.
* **Memory (State Carryover)** – In conversational applications, LangChain offers memory to retain information between turns. In a self-healing scenario, “memory” might mean keeping track of what the current code state is or what errors have been encountered so far. You may not need a complex memory module initially, but ensure your chain’s state (the data passed along) can accumulate necessary context. This could simply be an object that you pass from step to step, containing fields like the current code diff or test results.
* **Control Flow (Loops & Conditionals)** – Unlike a simple linear pipeline, an advanced chain may need loops or branching. For example, if tests fail, you want to loop back and have the LLM try another fix. LangChain supports conditionals and even recursive chains; you can implement this with standard programming constructs. Your interface might allow a step to decide the next step or to repeat a sequence. In practice, you can code this logic explicitly: e.g., after running tests, if failures remain, call the LLM again (possibly with a new prompt that includes the failure info) and repeat. This forms an **agent loop** that continues until success or a stop condition.

## Designing the Interface in TypeScript

**1. Project Setup:** Since you’re using TypeScript/Node, initialize an NPM project and install any needed packages (for example, the OpenAI SDK or other LLM API library). If you were to use the official LangChain, it’s as easy as npm install langchain @langchain/core[[8]](https://js.langchain.com/docs/how_to/installation/#:~:text=Installation%20,starting%20point%20to%20using%20LangChain). For a custom solution, you might rely on low-level HTTP calls or a minimalist SDK. Ensure you have type definitions for the model’s responses, etc., to leverage TypeScript’s strengths.

**2. Define an LLM Client:** Create a class (or just a function) to interface with your chosen LLM. For example, an OpenAIClient class that takes an API key and has a method generateText(prompt: string): Promise<string>. This method would call the OpenAI API (via fetch or an SDK) and return the model’s completion. By abstracting this, you can later swap in different models or endpoints without changing the chain logic. The idea is similar to how LangChain provides a standard model interface for many providers[[9]](https://www.langchain.com/langchain#:~:text=Connect%20LLMs%20to%20live%20data,directly%20into%20your%20AI%20workflows).

**3. Implement a Chain Runner:** Design a way to represent a sequence of steps and execute them in order. A simple approach is to define a Chain class that holds an array of asynchronous functions (steps). Each step function can accept a state object and modify or extend it. For example, you might define type ChainStep = (state: any) => Promise<any>. The Chain.run(initialState) method would iterate through the steps, doing state = await step(state) for each. This is essentially your **pipeline** manager. It can be as flexible as needed – you might allow adding steps dynamically or even nested sub-chains.

**4. Incorporate Prompt Templates and Parsing:** For each LLM-calling step, you will likely prepare a prompt from the current state and then parse the LLM’s response into something useful. You can encapsulate this in the step function. For example, a step could: read state.problemDescription, call openAIClient.generateText(promptTemplate(state.problemDescription)), then put the response into state.proposedSolution. It’s wise to keep prompt strings outside the code (or at least in a clearly marked section) so they can be refined easily. LangChain’s usage of prompt templates is a good model: you feed the LLM a structured prompt and it returns a structured answer[[5]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=def%20checks_generation%28task%29%3A%20,exactly%20suits%20the%20question%20given)[[10]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Return%20a%20numbered%20list%20of,answered%20in%20a%20logical%20way). You might also define **output parsers** – e.g., if the LLM returns a JSON or list, convert it to a native object/array.

**5. Add Tool Steps (for Tests & Code Execution):** Some steps in your chain will perform actions like running tests or applying a code patch. These are not LLM calls, but regular code that your interface should support. For example, one step function could spawn a child process to run npm test or use a testing API, then record the result (pass/fail) in the state. Another step might take a patch (diff) from the LLM’s output and apply it to the code (perhaps via git or directly editing files). Make sure these functions are robust, since they deal with real-world side effects. In a chain, these tool steps allow the LLM to interact with its environment, which is essential for an agent that writes and tests code.

**6. Enable Conditional Looping:** Design your chain or surrounding logic to handle iterations. A straightforward way is using a loop in normal code: after a full cycle of “propose fix -> apply -> test”, check the state to see if tests passed. If not, modify the state (e.g. include the error messages or failed test names) and run the chain (or part of it) again. This is essentially what the self-healing algorithm does. One guide breaks it down into seven steps including test generation, sub-task creation, answer checking, and answer fixing[[11]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Let%E2%80%99s%20divide%20our%20development%20process,into%207%20parts). You can implement these in order, and then loop back to the appropriate step on failure. For instance, you might loop steps 3–6 until step 5 (checking the final answer/tests) succeeds. By structuring your chain into modular steps, it’s clear where to re-enter the loop. Make sure to include a safeguard (like a max iterations or a manual abort) to avoid infinite loops if the model gets stuck.

**7. Logging and Monitoring:** As you build the interface, include logging at each step for transparency. LangChain provides tracing tools (LangSmith) in its ecosystem, but for a custom setup, simple console logs or a debug logging facility can help track what the LLM was asked and how it responded at each step. This is crucial for debugging your *self-healing* agent, as you'll want to inspect prompts and outputs to refine them.

## Example: Simple Chain Implementation (TypeScript)

Below is a **simplified example** of how you might start implementing a LangChain-like interface in TypeScript. This example outlines a chain with two steps: one LLM call to suggest a solution, and one tool action to verify that solution. It uses a dummy OpenAI client for illustration:

// 1. Define an interface for our LLM model client  
interface LLMClient {  
 generateText(prompt: string): Promise<string>;  
}  
  
// 2. A simple OpenAI API wrapper (dummy implementation for illustration)  
class OpenAIClient implements LLMClient {  
 constructor(private apiKey: string) {}  
 async generateText(prompt: string): Promise<string> {  
 // In a real implementation, call OpenAI API via fetch or OpenAI SDK:  
 // const response = await fetch("https://api.openai.com/v1/chat/completions", { ... })  
 // return response.choices[0].message.content;  
 console.log("LLM Prompt:", prompt);  
 // Dummy response for illustration:  
 return Promise.resolve("dummy code or solution");  
 }  
}  
  
// 3. Define the type for a chain step function  
type ChainStep<State> = (state: State) => Promise<State>;  
  
// 4. Implement a Chain class that runs an array of steps  
class Chain<State extends object> {  
 constructor(private steps: ChainStep<State>[]) {}  
 async run(initialState: State): Promise<State> {  
 let state = { ...initialState };  
 for (const step of this.steps) {  
 state = await step(state); // pass state through each step  
 }  
 return state;  
 }  
}  
  
// 5. Example usage: define a state interface for a simple self-healing attempt  
interface FixCodeState {  
 problem: string; // description of the issue to fix  
 proposedSolution?: string; // code or patch suggested by the LLM  
 testsPassed?: boolean; // whether tests passed after applying the solution  
}  
  
// 6. Instantiate the LLM client  
const llmClient = new OpenAIClient("OPENAI\_API\_KEY");  
  
// 7. Define the chain steps  
const proposeFixStep: ChainStep<FixCodeState> = async (state) => {  
 // Create a prompt instructing the LLM to fix the problem  
 const prompt = `You are an AI developer. Fix the following issue:\n${state.problem}\nProvide the patched code.`;  
 const solution = await llmClient.generateText(prompt);  
 // In a real scenario, you would parse the solution (e.g., extract code patch)  
 return { ...state, proposedSolution: solution };  
};  
  
const testSolutionStep: ChainStep<FixCodeState> = async (state) => {  
 if (!state.proposedSolution) return state;  
 // Here you would apply the solution to the codebase and run tests.  
 // For illustration, let's assume we run tests and determine the result:  
 const testsPassed = Math.random() > 0.5; // dummy pass/fail result  
 console.log("Applying solution and running tests... Result:", testsPassed ? "PASS" : "FAIL");  
 return { ...state, testsPassed };  
};  
  
// 8. Create a chain with the steps  
const fixChain = new Chain<FixCodeState>([ proposeFixStep, testSolutionStep ]);  
  
// 9. Run the chain with an initial problem description  
(async () => {  
 let state: FixCodeState = { problem: "The app crashes on launch due to a null pointer exception." };  
 for (let attempt = 1; attempt <= 5; attempt++) { // allow up to 5 attempts  
 console.log(`\nAttempt ${attempt}:`);  
 state = await fixChain.run(state);  
 if (state.testsPassed) {  
 console.log("Issue resolved by the AI in attempt", attempt);  
 break;  
 } else {  
 console.log("AI solution failed, re-attempting with feedback...");  
 // Prepare feedback for the next loop (e.g., include error logs in state.problem for next prompt)  
 state.problem += "\n(Previous attempt failed. Fix the bug differently.)";  
 // Loop will try again with updated state.problem  
 }  
 }  
})();

In this snippet, we set up a Chain that has two steps. The first step calls the LLM to propose a fix for the given problem, and the second step simulates applying that fix and running tests. We loop up to 5 attempts, feeding back a note into the problem description if the test fails, prompting the LLM to try a different solution in the next iteration. In a real implementation, you’d capture actual test failure messages and include them in the prompt for the next attempt (making the feedback more informative than the placeholder used above).

**Key points in the example:**

* The **LLM client (OpenAIClient)** is abstracted. It could be replaced with any model provider. This aligns with LangChain’s goal of provider flexibility[[12]](https://www.langchain.com/langchain#:~:text=Evolve%20your%20stack%20without%20vendor,without%20rewriting%20your%20entire%20application).
* Each **chain step** is an async function that reads and writes to a shared state. We use the state to carry the problem description, the proposed solution, and test results between steps.
* We included a simple **feedback loop** around the chain execution. This loop is external to the Chain class for simplicity, but you could also build looping into the chain definition. The loop updates the state.problem with feedback from the failed attempt before rerunning the chain. This realizes the self-healing behavior: the AI gets multiple chances to correct its mistakes, guided by test outcomes.
* Logging (console.log) is used to trace progress. In practice, you might integrate with a more sophisticated logging or monitoring system.

## Tips and Next Steps

Developing a LangChain-like interface is an iterative process. Start small: get a basic chain working with one LLM call and one simple tool, then expand. Here are some tips and next steps:

* **Start with Simple Use-Cases:** Before tackling full auto-debugging, test your chain idea on a simpler scenario (e.g. a chain that just asks the LLM a two-step question). This helps validate that your chaining mechanism works and that the LLM responses can be parsed as expected.
* **Leverage Existing Libraries Selectively:** You don’t have to reinvent the wheel for everything. For example, you could use **LangChain.js** components in parts of your project if it saves time – it’s modular, so you might import just the prompt template or output parser utilities. However, given that you’re aiming for a custom solution (and possibly using MCP), you might keep dependencies minimal. There are also smaller libraries focusing on prompt templating or function chaining which you could evaluate.
* **Incorporate JSON Schemas and Validation:** Notably, you have JSON schemas (like selfhealing.schema.json and patch-envelope.schema.json) in your project. You can use these to have the LLM output a structured format (e.g. a patch envelope JSON) and then validate it against the schema. This is a smart way to enforce the AI’s output format. You could add a chain step that, after the LLM proposes a patch, validates the JSON against your schema (using a JSON schema validator library) before applying it. This ensures the *self-healing agent* produces well-formed patches and follows the rules (like circuit breaker policies) defined in your schema.
* **Testing and Refinement:** As you build the interface, continually test it with real examples. Observe where the LLM might need better prompting or where the chain logic might need adjustments. Building a self-healing agent is an experimental endeavor – it will likely take many prompt tweaks and logic refinements to get reliable results. Keep an eye on edge cases (e.g. the LLM proposing a fix that doesn’t actually address the bug, or passing tests incorrectly). Each failure is an opportunity to improve the prompts or add constraints for the next round.
* **Community and Resources:** Since this is a novel area, consider keeping up with communities (like the LangChain forum or Reddit) to see how others are faring. For instance, one Medium guide provides a full **recursive reasoning framework** for a self-healing agent, with steps to generate tests, break down tasks, and iteratively fix the answer[[11]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Let%E2%80%99s%20divide%20our%20development%20process,into%207%20parts). Such resources can offer inspiration for structuring your own solution. You might not follow them verbatim, but they can spark ideas (for example, automatically generating unit tests for a given bug using the LLM, which then guide the fixing process).

Finally, remember that this is cutting-edge work – **“are we going to see this fly?”** – With the rapid progress in LLM capabilities, there’s good reason to be optimistic. Early adopters have shown that even partial self-healing (fixing small errors) is feasible and valuable[[1]](https://www.reddit.com/r/LangChain/comments/129exd4/selfhealing_github_action_using_langchain_and/#:~:text=Love%20it%21%20I%E2%80%99m%20not%20saying,dev%20hours%20at%20big%20companies). By carefully designing your LangChain-like interface in TypeScript, you are setting up a strong foundation for an AI-assisted development workflow. With patience and experimentation, you just might get that self-healing code system off the ground, bringing a smile to every developer’s face. 🚀

**Sources:**

* Pranav Marla, *“The Guide to building Self-Healing Agents using LLMs,”* *Medium*, 2023. – Describes a seven-step reasoning framework for an LLM agent that can test and fix its answers[[11]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Let%E2%80%99s%20divide%20our%20development%20process,into%207%20parts)[[13]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=1,tests%20pass%2C%20return%20the%20answer), using LangChain to orchestrate prompts and model calls for each step.
* LangChain Documentation – *LangChain: One interface, integrate any LLM.* Explains the purpose of LangChain as a standard interface for connecting LLMs to tools, data, and other components[[3]](https://www.langchain.com/langchain#:~:text=LangChain%20is%20the%20open%2C%20composable,fast%20as%20the%20ecosystem%20evolves)[[14]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=LangChain%20is%20an%20open,com). Provides context on why frameworks like LangChain are useful for building dynamic AI workflows.
* Reddit discussion on self-healing code agents – Highlights developer enthusiasm for LLMs auto-fixing code. Notably, a GitHub project demonstrated a “self-healing” CI action using LangChain and OpenAI, showing that even if such agents can’t fix complex bugs yet, they can handle simpler issues (like linting errors) and save time[[1]](https://www.reddit.com/r/LangChain/comments/129exd4/selfhealing_github_action_using_langchain_and/#:~:text=Love%20it%21%20I%E2%80%99m%20not%20saying,dev%20hours%20at%20big%20companies). This underscores the potential of the approach and the importance of frameworks to support it.

[[1]](https://www.reddit.com/r/LangChain/comments/129exd4/selfhealing_github_action_using_langchain_and/#:~:text=Love%20it%21%20I%E2%80%99m%20not%20saying,dev%20hours%20at%20big%20companies) Self-healing GitHub action using Langchain and OpenAI : r/LangChain

<https://www.reddit.com/r/LangChain/comments/129exd4/selfhealing_github_action_using_langchain_and/>

[[2]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=LangChain%20is%20an%20open,com) [[4]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=) [[14]](https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/#:~:text=LangChain%20is%20an%20open,com) I got sick of how bloated LangChain was so I made my own lightweight Typescript framework for building AI Agents (with full MCP support) : r/LangChain

<https://www.reddit.com/r/LangChain/comments/1j92n8a/i_got_sick_of_how_bloated_langchain_was_so_i_made/>

[[3]](https://www.langchain.com/langchain#:~:text=LangChain%20is%20the%20open%2C%20composable,fast%20as%20the%20ecosystem%20evolves) [[9]](https://www.langchain.com/langchain#:~:text=Connect%20LLMs%20to%20live%20data,directly%20into%20your%20AI%20workflows) [[12]](https://www.langchain.com/langchain#:~:text=Evolve%20your%20stack%20without%20vendor,without%20rewriting%20your%20entire%20application) LangChain

<https://www.langchain.com/langchain>

[[5]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=def%20checks_generation%28task%29%3A%20,exactly%20suits%20the%20question%20given) [[6]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=prompt%20%3D%20ChatPromptTemplate.from_messages%28%5B%20%28,exactly%20suits%20the%20question%20given) [[10]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Return%20a%20numbered%20list%20of,answered%20in%20a%20logical%20way) [[11]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=Let%E2%80%99s%20divide%20our%20development%20process,into%207%20parts) [[13]](https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee#:~:text=1,tests%20pass%2C%20return%20the%20answer) The Guide to building Self-Healing Agents using LLMs | by Pranav Marla | Medium

<https://medium.com/@pranav.marla/the-guide-to-building-self-healing-agents-using-llms-0827dc5fc8ee>

[[7]](https://dev.to/mohsenkamrani/master-langchain-in-typescript-a-practical-guide-4ifc#:~:text=Each%20node%20adds%20a%20specific,one%20step%20to%20the%20next) Master LangChain in Typescript - A Practical Guide - DEV Community

<https://dev.to/mohsenkamrani/master-langchain-in-typescript-a-practical-guide-4ifc>

[[8]](https://js.langchain.com/docs/how_to/installation/#:~:text=Installation%20,starting%20point%20to%20using%20LangChain) Installation - LangChain.js

<https://js.langchain.com/docs/how_to/installation/>