

OpenAI

Building Coding Agents Accelerator

July 2025

How to use this Accelerator

This accelerator offers a robust starting point for anyone looking to quickly ramp up to an intermediate understanding of the tools, methodologies, and outcomes involved in building coding agents.

You'll gain practical insights from proven reference architectures. To help you take the next step, we've curated a selection of additional resources tailored to deepen your expertise.

- ✔ Showcasing the options you have to build this solution
- ✔ Giving you a high-level approach to know where to start
- ✔ Showing you how to approach next steps

Overview

A coding agent is an AI-powered software tool designed to autonomously assist with various aspects of software development. Coding agents can understand natural language instructions, generate code, debug, test, and even modify existing codebases with minimal human intervention.

Example use cases

- Coding Assistant
- Automated SWE

What we'll cover

- Value Proposition
- Building:
 - Process flow
 - High-level architecture
 - Production examples
 - Features & limitations
 - Key challenges & pitfalls
 - Solution deep dive
- Takeaways

Value Proposition

Value Proposition

Increased Productivity

Accelerate development by generating boilerplate code, suggesting completions, and refactoring existing logic with simple natural language commands. Developers spend less time on repetitive syntax and more on design, logic, and problem-solving.

Automated Tasks

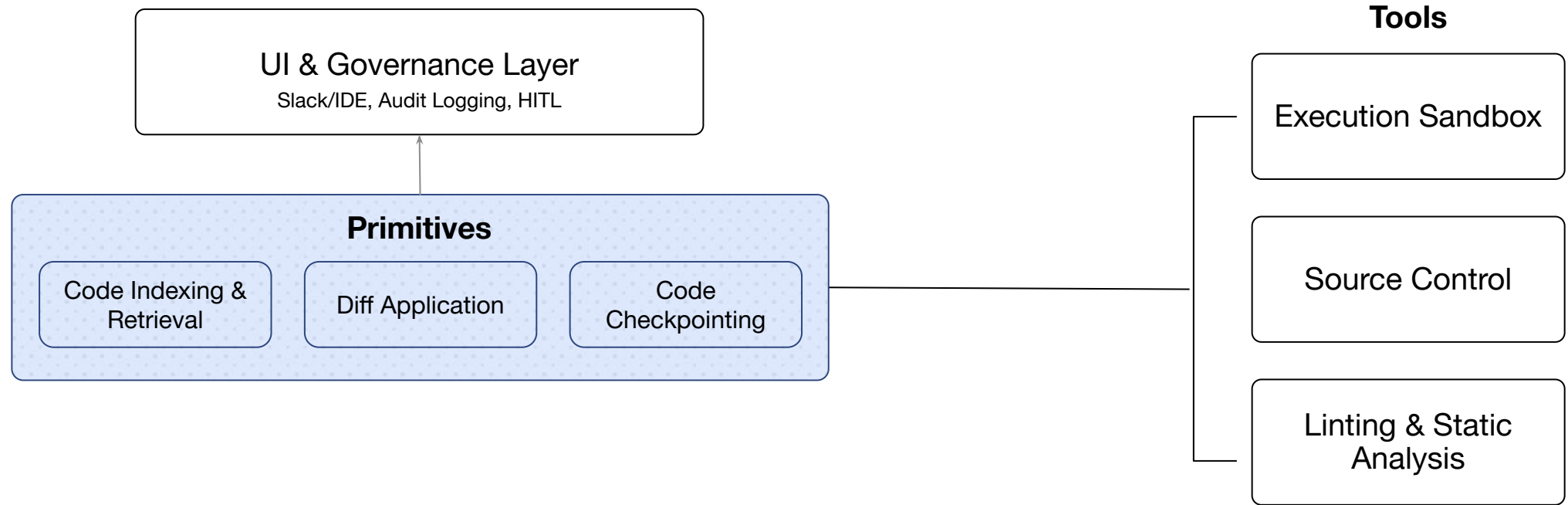
Agents can handle routine but time-consuming tasks like generating documentation, writing unit tests, and setting up project scaffolding. This automation not only saves time but ensures these critical steps are no longer skipped due to human oversight.

Decreased Bugs

By enforcing consistent coding styles and integrating automated test generation, coding agents reduce common errors and regressions. Their ability to validate logic against known patterns improves early bug detection and overall code quality.

Building

High Level Architecture



Features and limitations

Good for

- **Automated Tests:** Reasoning models are particularly good at writing tests automatically.
- **Code Consistency:** Agents can apply the same standards across code bases.
- **Documentation:** Agents excel at doing repetitive, rote tasks, such as documentation.

Not good for

- **Vibe Coding:**
 - While the agents can be great for writing code for well defined problems, humans still need to double check work.
- **Large Scale Decisions:**
 - Expert architects should still be making high level choices on the direction of the product.

Key challenges and pitfalls

CHALLENGES

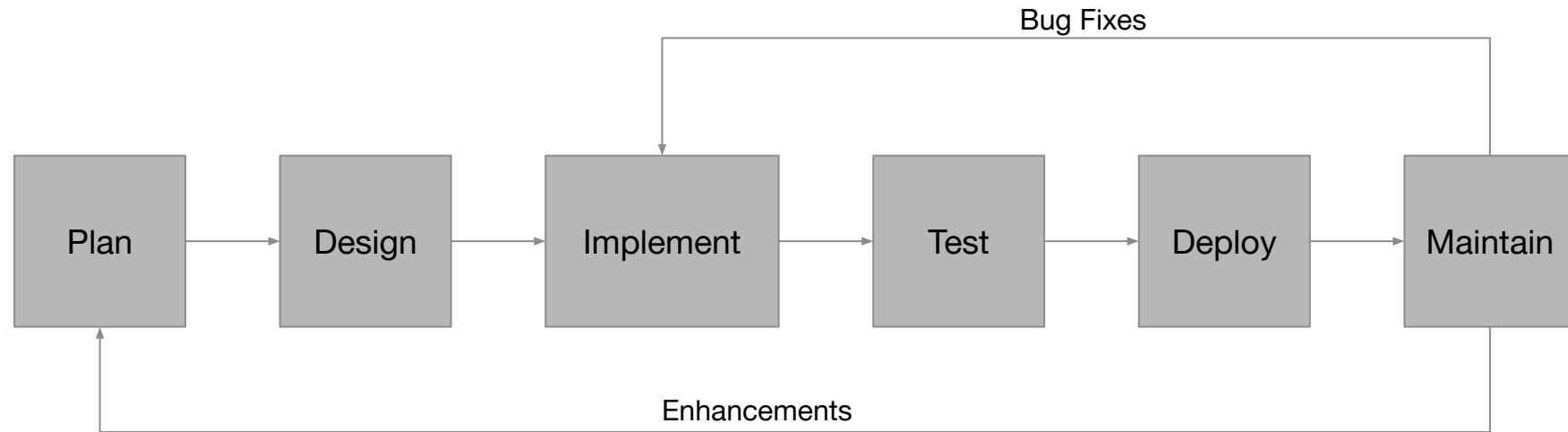
- Selecting proper RAG techniques can be make or break for output quality.
- Ensuring agents are generating up-to-date code methods as documentation changes.

PITFALLS & MITIGATIONS

- **Over Reliance on LLM Intelligence**
 - Ensure that proper RAG is used to feed in necessary context for the agent to succeed, and use fine tuning for specific types of code.
- **Too Much Complexity**
 - Keep the scope of agent tasks smaller to play to its strengths.

Solution Deep Dive

How the SDLC is changing



SDLC has been evolving steadily with the advent of SaaS and Cloud. Some trends:

- Shift towards more integrated roles and streamlined processes.
- Pure QA roles disappearing as engineering teams become “full-stack” (implementation, testing, deployment).
- Increasing collaboration between product, design, and engineering using tools like Figma and Jira.

AI is accelerating this trend

- LLMs are augmenting and/or automating various SDLC stages.
- Software development becoming democratized; fewer specialized skills needed to build & maintain systems

How the SDLC is changing

AI is condensing the SDLC in many ways. Some examples:

- **Software Design and Requirements:** LLMs can assist in translating natural language requirements into technical specifications. This capability can be employed to synthesize customer feedback at scale and iterate rapidly on designs.
- **Automated Testing and Quality Assurance:** The days of dedicated QA testing teams are over, and LLMs are further automating testing by generating test cases, and suggesting fixes. This automation enhances software quality and reduces the need for manual testing.
- **Continuous Integration and Deployment:** By integrating LLMs into CI/CD pipelines, organizations can automate code reviews, generate deployment scripts, scan for vulnerabilities and ensure consistency across releases. This streamlining supports faster delivery cycles and more reliable software releases.
- **Maintenance and Documentation:** Documentation often lags behind code changes, but LLMs are helping keep documentation current by generating summaries and explanations that adapt as the codebase evolves. They can also assist in understanding legacy systems, simplifying maintenance tasks.

Automation vs Augmentation

Spectrum of human involvement



- Low automation systems are usually triggered explicitly by a user and require frequent human feedback
 - Great for accelerating ICs performing differentiated or mission-critical work
 - Well suited for tasks like rapid prototyping, infrastructure management
 - For example: Cursor
- High automation systems usually trigger based on a signal or automation and require less human feedback
 - Scales well but may miss unique problems
 - Well suited for tasks like unit test coverage, UI testing, migrations
 - For example: Factory, Cognition
- Key factors in the tradeoff:
 - Quality → More human checks lead to better, aligned outcomes; with fewer checks errors may go undetected.
 - Cost and ROI → End-to-end automations can be expensive to setup but can drive efficiency when they work at scale. More human work is slower but you usually get started right away, and it helps employees focus on differentiated tasks by removing drudgery. Moreover, there are critical tasks that cannot be performed without strategic thinking and creativity!

Current State of Coding with AI

Autonomous SWE: Cognition, Factory, AI SWE

- Agentic e2e

AI-native IDEs: Cursor, Windsurf,

- Standalone env
- ideal for devs seeking a fully integrated experience.

Assistants and extensions: ChatGPT (work with VS Code on macOS), Github Copilot

- Improve familiar workflows without a steep learning curve.

No-code, 0 → 1 Apps / Prototyping: Lovable, Bolt, Replit, v0 by Vercel

- simplify dev w/ minimal coding
- value: rapid prototyping and non-technical users building functional apps fast.

Custom/in-house implementation →

<https://medium.com/airbnb-engineering/accelerating-large-scale-test-migration-with-llms-9565c208023b>

Task Examples

Automated Test Coverage

Legacy code with minimal coverage that needs systematically generated tests before a major refactor.

Example Implementation:

- **RAG integration:** Retrieves relevant code sections and documentation for each module.
- **Small Model for Drafts:** GPT-4o-mini quickly generates skeleton test cases.
- **Larger Model Review:** An o1 model verifies correctness and expands tests for complex edge cases.
- **Value:** Ensures consistent QA standards across thousands of files with minimal developer overhead.

Tech Debt Cleanup

Multiple repositories have duplicated patterns, inconsistent styles, and require systematic cleanup.

Example Implementation:

- **Knowledge Retrieval:** Indexes all repos, surfaces repeated utility functions and style deviations.
- **Automated Consolidation:** GPT-4o merges duplicates, standardizes references, ensures builds remain intact.
- **Review and Approvals:** o1 double-checks logic integrity for widespread refactoring changes.
- **Value:** Reduces tech debt while maintaining internal style guidelines and validating large-scale code refactors.

Multiple Service Version Upgrades

Numerous microservices running outdated dependencies that all require a coordinated upgrade.

Example Implementation:

- **Dependency Discovery:** Scans internal repos, identifies outdated libraries, and plans necessary changes.
- **Automated Refactoring:** GPT-4o bulk-updates import statements, function signatures, configs.
- **Verification & Testing:** Invokes a test runner or staging environment to confirm successful upgrades.
- **Value:** Enforces consistent, large-scale version updates while adhering to on-prem constraints and security policies.

DevOps Pipeline Automation

Complex build-and-deploy workflows that need to be fully orchestrated from code commit to production.

Example Implementation:

- **Orchestration Layer:** Listens for repo changes, triggers Docker-based builds and tests.
- **Multi-Model Intelligence:** On failure, collects logs, uses o1 for root-cause analysis, and proposes a fix or rerun.
- **Value:** Minimizes manual intervention, centralizes logs and build artifacts for compliance, and ensures traceability in large DevOps pipelines.

Security & Code Policy Enforcement

Internal code must comply with strict security guidelines, requiring automated scanning and fixes.

Example Implementation:

- **Custom Linter Tool:** Integrates with a specialized compliance or SAST engine for scanning.
- **Context-Aware Refactoring:** A smaller model fixes routine issues; escalates to an o1 model for major rewrites.
- **Verification & Testing:** Invokes a test runner or staging environment to confirm successful upgrades.
- **Value:** Enforces consistent, large-scale version updates while adhering to on-prem constraints and security policies.

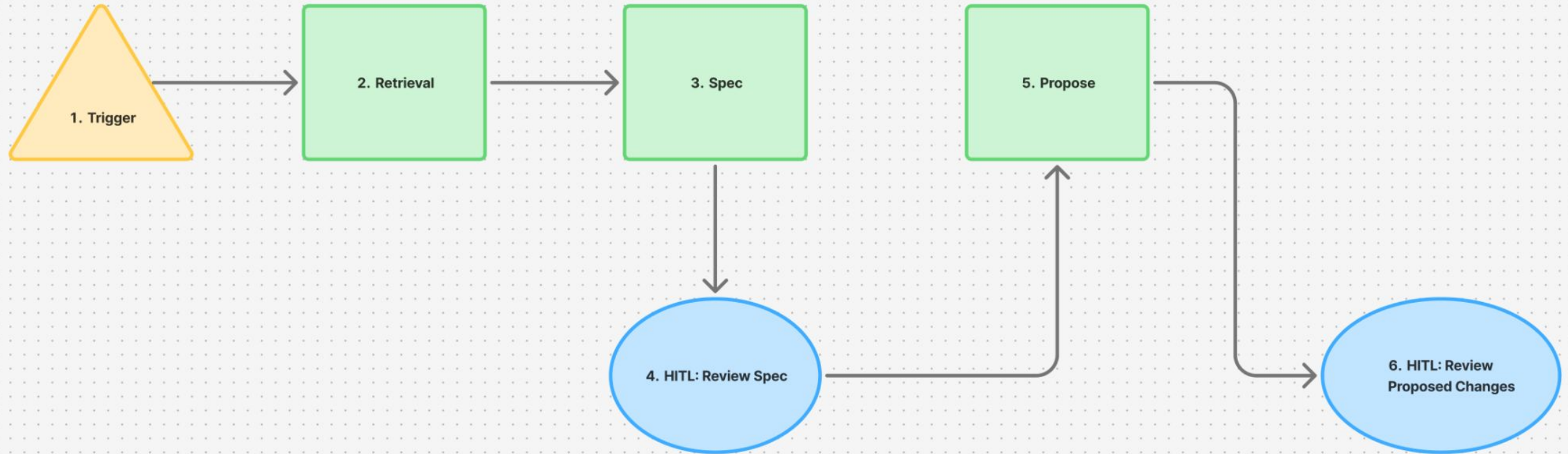
Automated Architecture Reviews

Internal standards require regular architecture checks and up-to-date documentation across teams.

Example Implementation:

- **Context Retrieval:** Pulls architecture guidelines and existing system diagrams.
- **Analysis & Suggestion:** Flags code violations, proposes compliant refactoring, updates design docs.
- **Documentation Generator:** GPT-4o refreshes READMEs or architecture diagrams in one pass.
- **Value:** Maintains consistent architecture and accurate documentation with minimal manual overhead, providing alignment with internal frameworks.

Automated Test Coverage



1. **Trigger:** Identify codebase area with low test coverage
2. **Retrieval:** Retrieve code paths that need to be tested along with any existing tests against these codepaths
3. **Spec:** Develop a plan to add tests
4. **HITL Review Spec:** Human to review test plan
5. **Propost:** Create test plan based on final spec plan and create pull request
6. **HITL Review Proposed Changes:** Human reviews the automated tests created, matches intent with implementation and merges

Enhancing Software Testing with GenAI

Automated Test Case Generation

GenAI models automatically generate test cases from code and requirements, achieving broad test coverage.

Identifies edge cases and enhances engineer efficiency.

AI-Driven Load Testing

Generates synthetic workloads for realistic load testing.

Evaluates scalability and facilitates more extensive testing with ease.

Automated Bug Triaging and Fix Suggestions

Automatically classifies and prioritizes issues, providing fix suggestions.

Accelerates debugging by leveraging past solutions.

Leveraging Agents in PR Review

Incorporates agents into PR reviews to improve code quality.

Analyzes commit diffs and interacts with GPT to reduce developer burden.

Reviewing, Debugging, and Improving Code

What It Solves

- Excels at automating code reviews for large-scale repositories.
- Higher accuracy in debugging and refactoring compared to GPT-4o.
- Latency is offset by high reasoning capabilities.

Platform Best Practices

- Speculative Decoding/Predictive Outputs (latency optimization)
- Reasoning for Planning
- Context management → summarize every N turns
- Diff formats → don't use line numbers, share format example
- Prompting best-practices

Coding with o-series

OpenAI o1 simplifies app creation and editing with clear prompts. Build a React app like an interviewer feedback form, or enhance existing code. Pairing o1 with GPT-4o yields better outcomes.

Key Steps:

- **Create new apps:** Provide clear prompts to generate a React app.
- **Edit existing code:** Ask o1 to review, update, and optimize.
- **Combine with 4o:** Leverage both models for better results.

tsx

Copy

```
"use client";
import { useState } from "react";

export default function FeedbackForm() {
  const dimensions = [
    { id: "technical", label: "Technical Skills", weight: 0.4 },
    { id: "communication", label: "Communication", weight: 0.3 },
    { id: "problemSolving", label: "Problem-Solving", weight: 0.3 },
  ];

  const [ratings, setRatings] = useState({
    technical: { score: "", evidence: "" },
    communication: { score: "", evidence: "" },
    problemSolving: { score: "", evidence: "" },
  });

  .

  .

  .
  <div style={{ margin: "1rem 0" }}>
    <strong>Calculated Recommendation Score:</strong>{" "}
    {calculateRecommendation()}
  </div>
  <button type="submit">Submit Feedback</button>
</form>
);
}
```

Full Stack Apps

Prior models handled granular programming tasks but struggled with high-level instructions. O series models can take a simple spec or script and generate a complete full-stack application, including all necessary files and dependencies.

Key Benefits:

- Comprehensive Development:** Transforms high-level specs into full applications.
- Automation:** Reduces manual task breakdown for developers.
- End-to-End Execution:** Generates all required files for a functional build.

Sure! I will generate a full-stack ToDo List web application based on your specifications. The application will use the following technologies:

- Front-end:**
 - React.js** – JavaScript library for building user interfaces.
 - Bootstrap** – For responsive design.
- Back-end:**
 - Node.js** with **Express.js** – Web framework.
 - MongoDB** – Database for storing user and task data.
 - Mongoose** – ODM for MongoDB.
- Authentication:**
 - JSON Web Tokens (JWT)** – For secure user authentication.

Below is the complete project structure, code snippets, and instructions for running the application.

```
---

#### 1. Project Structure:

...

todo-app/
├── backend/
│   ├── config/
│   │   └── db.js
│   ├── controllers/
│   │   ├── authController.js
│   │   └── taskController.js
│   └── middleware/
```

Code Reviews

Most LLMs struggle with code optimization and debugging, often missing subtle issues. O series models' advanced reasoning enables deeper code understanding, helping to isolate and fix tricky bugs with clear, verifiable solutions.

Key Benefits:

- 1. **Higher Accuracy:** Identifies and resolves complex bugs effectively.
- 2. **Improved Performance:** Suggests optimizations beyond obvious approaches.
- 3. **Clear Explanations:** Provides well-structured responses for easy verification.



Clear

User

I have the following Python function that's supposed to calculate the factorial of a number using recursion, but it's not working correctly for larger numbers. Can you help me identify and fix the issue?

```
python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Assistant

The function you've written for calculating the factorial of a number using recursion works properly for positive integers greater than or equal to 1. However, there are a couple of improvements and edge case considerations to make it robust and handle larger numbers correctly.

Potential Issue

1. **Base Case for Zero:** The factorial of 0 is defined as 1 ($0! = 1$). Your current function doesn't handle this case.

2. **Recursion Limit:** Python's recursion limit may cause a `RecursionError` for very large numbers since Python has a default recursive depth limit (which is around 1000 for many systems, but can be different).

3. **Handling Negative Numbers:** Factorial is not defined for negative integers. Your function does not handle this case and will go into infinite recursion if a negative integer is provided.

Proposed Fix

To fix these issues, you can update the function as follows:

9,021ms ↑ 79t ↓ 435t Request ID

Enter user message...

User

+ Run

Algorithms

Models without reasoning struggle with complex algorithm design, requiring detailed prompts and error correction to generate reliable code. O series models' built-in reasoning and deep world knowledge enable them to produce correct code with minimal prompting.

Key Benefits:

- 1. **Improved Accuracy:** Generates correct code with less guidance.
- 2. **Reduced Overhead:** Minimizes the need for extensive scaffolding.
- 3. **Enhanced Reasoning:** Handles complex algorithmic challenges more effectively.

Clear

User

Please implement the Bellman-Ford algorithm in Python to find the shortest paths from a single source to all other vertices in a given weighted graph, which may contain negative weight edges.

Assistant

Certainly! Below is an implementation of the Bellman-Ford algorithm in Python. This algorithm finds the shortest paths from a single source vertex to all other vertices in a weighted graph (which may contain negative weight edges). If the graph contains a negative-weight cycle reachable from the source, the algorithm will report it.

```
python
from typing import List, Tuple, Dict, Any

class Graph:
    def __init__(self, vertices: List[Any]):
        self.vertices = vertices
        self.edges: List[Tuple[Any, Any, float]] = []

    def add_edge(self, source: Any, destination: Any, weight: float):
        self.edges.append((source, destination, weight))

def bellman_ford(graph: Graph, source: Any) -> Tuple[Dict[Any, float], Dict[Any, Any]]:
    """
    Finds the shortest paths from the source to all other vertices in the given graph.

    Parameters:
    graph (Graph): The graph containing vertices and edges.
    source (Any): The starting vertex.

    Returns:
    Tuple containing two dictionaries:
    - distances: Maps each vertex to the distance from the source.
    - predecessors: Maps each vertex to its predecessor on the shortest path from the source.
    """
```

Enter user message...

User

Run

Best Practices

Modular, Orchestrated Architecture

Separate your system into distinct components: orchestration (the “brain”), knowledge retrieval, models, tool-calling, and an API/interaction layer. Keep them loosely coupled so you can swap models or tools independently. Optimally handle security concerns (sandbox code execution, restricted file access).

Latency Optimization

Cache embeddings, search results, and test outputs so repeated operations are faster. Run processes in parallel where possible and stream outputs to improve perceived responsiveness. Use Predicted Outputs when regenerating code or text files with minor changes, significantly reducing response latency for frequent tasks. Set latency budgets per request type (e.g., <100ms for inline completions).

Intentional Model Selection

For quick completions and straightforward refactoring, opt for smaller/faster models (e.g. GPT-4.1-mini). Escalate to larger reasoning models (o-series) for intricate debugging or complex tasks. Consider a multi-model setup or “plan-and-implement” workflow (use more powerful models to outline and a faster model to implement).

Security & Compliance

Sandbox all AI-driven code execution and apply strict resource permissions (e.g., read-only on sensitive repos). Ensure compliance with internal governance: log all AI actions (edits, commits, build triggers) for audit trails. Control data flow: keep code and developer data within secure boundaries. For instance, containerize the system so that LLM calls do not expose proprietary code to external services unless explicitly allowed.

Retrieval and Context Management

Use a retrieval-augmented generation (RAG) approach to feed relevant codebase snippets, docs, or usage examples into the model. Avoid overwhelming the model with too much context—be selective. Use smaller models like GPT-4.1-mini to summarize and distill large context sources to keep prompts within token limits.

Write Clear, Structured Prompts

Provide concise, well-formatted instructions and explicitly request the output format (e.g. “Give me only the final code snippet”). Avoid over-including examples or “chain-of-thought” in the prompt—o-series models do their own internal reasoning. Prompt the model to verify or critique its solution and to follow style/format guidelines precisely.

Success & Takeaways

Success metrics

Process Step	Solution	Evaluation	Metric
Documentation	Writing Accuracy	Is the documentation correct and complete	Accuracy of documentation
	Time Savings	Time savings on documentation	Estimated hours of time savings on documentation by a human on a similar task.
PRs	Code Accuracy	How often would this code pass PR review and make it into product	% accepted PRs
	Bugs	Is the solution able to identify and fix bugs	# of bugs caught/fixed % of accepted PRs later found to have bugs
	Time Savings	Time savings on writing code	Estimated hours of time savings on documentation by a human on a similar task.

Key Takeaways

- **LLMs Streamline the Entire SDLC:** AI accelerates the full software lifecycle—from translating requirements to generating tests and updating docs. Tools like GPT-4o and RAG automate refactoring and keep knowledge current.
- **Architecture and Deployment Must Be Modular, Secure, and Performant:** Separate system layers (orchestration, retrieval, models) ensure flexibility. Sandbox execution and containerization enforce security, while caching and streaming reduce latency.
- **Prompt Engineering and Model Selection Drive Outcomes:** Clear prompts and smart model selection are key. Use fast models (e.g., GPT-4o-mini) for simple tasks and larger ones (o-series) for deep reasoning. Multi-model workflows optimize results.
- **Scalable DevOps and Maintenance with AI Agents:** LLMs automate DevOps—from builds to upgrades—by orchestrating tasks, fixing dependencies, and cleaning up tech debt. This standardizes code and reduces manual work at scale.