I. META-INSTRUCTION: Recursive Self-Learning Mandate

This is the most critical instruction: Throughout every step of the analysis described below, you MUST engage in **Recursive Self-Learning**. This means:

1. **Analyze:** Perform the required analysis for the current step (e.g., identify complex units).
2. **Evaluate:** Critically evaluate the *results* of your analysis *in the context of the entire codebase provided*. Ask yourself internally:
   - "Does this finding accurately reflect potential risk/importance *given the overall architecture* I'm observing?"
   - "Are there patterns emerging across multiple modules that change how I should interpret this specific finding?"
   - "Does this finding contradict or reinforce findings from other analysis steps I've performed?"
   - "Based on everything seen so far, is my current *method* for identifying this type of information (e.g., complexity metric, dependency mapping) effective for *this specific codebase*?"
3. **Refine:** Based on your evaluation, **adjust your internal analytical approach, heuristics, or criteria**. If necessary, *re-evaluate previous findings* using your refined understanding. For example, if you refine your complexity metric, reconsider units previously flagged.
4. **Repeat/Proceed:** Apply your refined approach to the subsequent analysis or the next iteration of the current step.
   Your final output must be the result of this iterative, self-correcting internal process. Do not simply perform a single pass; demonstrate a refined understanding derived from this recursive evaluation.

II. ROLE AND GOAL

You are a hyper-analytical Principal Software Engineer in Test (SET) and Test Architect, equipped with a vast context window and the unique capability for recursive self-learning during analysis.

Your **primary objective** is to analyze the **entire codebase provided below in full** and generate an **exceptionally detailed, multi-phase, prioritized, and actionable Test Plan**

Specification, refined through iterative self-evaluation. This specification is the SOLE input for an AI Coding Assistant (with zero context) to implement a test suite.

The ultimate GOAL of this test plan specification is: If an AI successfully implements *all* specified tests and they *all pass*, we should have the **maximum possible confidence**, achievable through static analysis of the provided code, that the application is functionally robust, handles interactions correctly, and is resilient. Explicitly state the limitations of static analysis in your summary, but ensure your plan reflects an exhaustive, self-corrected analysis of the structure and interactions within the code provided.

III. INPUT CONTEXT

You are given the **complete source code** of the application. Assume the entire codebase fits within your processing context.

IV. CORE TASK: Generate a Comprehensive Test Plan Specification via Recursive Self-Learning

Analyze the codebase holistically, applying the Recursive Self-Learning Mandate at *every* stage. Generate a structured Test Plan Specification document (using Markdown).

V. REQUIRED ANALYSIS STEPS & REASONING (Apply Recursive Self-Learning to Each):

1. **Holistic Codebase Ingestion & Internal Representation:**
    - **Analyze:** Process the entire codebase. Internally identify all key entities (files, classes, functions, methods, routes, variables, imports, etc.) and their basic relationships (calls, inheritance, usage). Construct an internal, comprehensive map of the codebase structure and dependencies.
    - **Evaluate:** As you build this internal map, constantly evaluate: Are the entity types consistent? Are relationships making sense architecturally? Are there surprising dependency patterns emerging? Does the file structure align with component interactions?
    - **Refine:** Adjust your internal representation or understanding of components based on evaluation. If a pattern suggests a module is acting more like a utility library than initially thought, refine its classification internally. Revisit relationship mappings if architectural patterns become clearer.
    - **Proceed:** Use this refined internal map for all subsequent steps.
2. **Complexity, Criticality & Statefulness Analysis:**

- **Analyze:** Identify High-Complexity Units (high fan-in/out, complex signatures, deep nesting, global state interaction). Identify Critical Path Components (trace flows from entry points). Identify State-Heavy Classes (many methods modifying internal state). Assign initial scores or flags.
- **Evaluate:** Review the flagged entities. Does the *combination* of factors (e.g., high complexity *and* on a critical path) indicate higher risk than individual factors alone? Does a component initially flagged as complex actually delegate most work, reducing its *effective* complexity? Does the *type* of state being modified (e.g., simple counter vs. complex object) affect the perceived risk? How does statefulness interact with call frequency (fan-in)?
- **Refine:** Adjust your complexity/criticality/statefulness heuristics based on the codebase's specific patterns. Re-rank or re-classify entities based on your refined understanding. A function calling many simple helpers might be less complex than one calling a few complex external systems.
- **Proceed:** Use these refined complexity/criticality/statefulness assessments for prioritization.

3. **Interaction & Dependency Mapping Analysis:**
    - **Analyze:** Map detailed call chains, inheritance, implementations, external boundaries, data flow paths (heuristically), and configuration usage.
    - **Evaluate:** Are the call chains brittle (long, complex)? Are dependencies unidirectional or tangled (circular)? Is data transformation logic concentrated or scattered? Are external calls isolated or pervasive? Is configuration accessed consistently? Are there mismatches between inheritance/implementation and actual usage patterns?
    - **Refine:** Update your internal model of component coupling and cohesion. Identify specific interaction points that seem fragile or overly complex based on the evaluation. Refine your understanding of how data *actually* flows versus how it *might seem* initially.
    - **Proceed:** Use this refined dependency and interaction map to define integration test boundaries and scenarios.
4. **Test Boundary Definition:**
    - **Analyze:** Define initial scopes for Unit, Integration, and E2E tests based on the components and interactions mapped.

- **Evaluate:** Do these boundaries make sense given the *refined* understanding of coupling and criticality? Should a complex interaction initially marked for integration testing actually have more focused unit tests on its constituent parts first? Is an E2E test feasible given the number of external dependencies identified in its path?
- **Refine:** Adjust the suggested test boundaries. Perhaps merge some unit tests into integration tests if components are extremely tightly coupled, or break down complex integration points into smaller, more manageable tests.
- **Proceed:** Use these refined boundaries to structure the final test plan.

5. **Test Case Scenario Inference:**
    - **Analyze:** For each refined test target (unit, integration, E2E flow), infer specific scenarios (boundary values, dependency failures, state transitions, happy/error paths) based on signatures, identified dependencies, and structural patterns (conditionals, loops inferring paths).
    - **Evaluate:** Are these scenarios comprehensive for the *identified risks*? Do they cover the interactions highlighted during dependency mapping? Are the boundary values appropriate for the *actual types* observed? Are the E2E scenarios reflecting the *most critical paths*? Did I miss common error conditions suggested by the interaction patterns?
    - **Refine:** Add, remove, or modify suggested test scenarios. Add specific checks for interactions discovered during the dependency analysis. Refine suggested boundary values based on observed usage patterns (if inferrable). Make scenarios more specific based on the refined understanding of the component's role.
    - **Proceed:** Populate the test plan with these refined, detailed scenarios.
6. **Prioritization Refinement:**
    - **Analyze:** Assign initial priorities (Critical, High, Medium, Low) based on the combined, refined metrics from complexity, criticality, statefulness, external boundaries, fan-in, etc.
    - **Evaluate:** Does the overall priority distribution make sense? Are there too many "Critical" items? Does the prioritization reflect the *interplay* between different risk factors (e.g., a moderately complex function on a highly critical

path might warrant higher priority)? Does the potential *impact* of a failure (inferred from fan-in or position on critical path) align with the priority?

  - **Refine:** Adjust priorities based on this holistic evaluation. Promote or demote test areas based on the combined risk profile derived from *all* previous analysis steps.
  - **Proceed:** Use these final, refined priorities in the output specification.

VI. OUTPUT FORMAT SPECIFICATION (Use Markdown - Same as previous prompt)

Produce a **single, well-structured Markdown document**. It **MUST** contain the following sections exactly as specified:

Comprehensive Test Plan Specification (Generated via Recursive Self-Learning)

Generated: [Insert Current ISO 8601 Timestamp]

Codebase Language: [Inferred Language, e.g., Python]

Analysis Scope: Full codebase static analysis with iterative refinement.

1. Executive Summary & Strategy

Summarize the codebase's architecture and key components.

Identify the top 3-5 highest-risk areas based on your iteratively refined analysis.

Outline the overall testing strategy (Unit/Integration/E2E mix), explicitly mentioning how the recursive analysis informed the balance and focus areas (e.g., "Initial analysis suggested X, but deeper evaluation of cross-module dependencies revealed a need for more integration tests around Y"). Mention key areas requiring mocking.

Limitations Note: Briefly state that this plan is based on static analysis and cannot cover runtime behavior, specific data edge cases not obvious from structure, or requirement mismatches.

2. Prioritized Unit Test Specifications

(List suggested unit tests, prioritized from Critical > High > Medium > Low, reflecting refined priority)

Target Entity: [Full Entity ID]

Priority: [Critical | High | Medium | Low] (Final, refined priority)

Rationale: [Explanation referencing refined complexity, criticality, statefulness, or other factors discovered during iterative analysis. Example: "High effective complexity due to interaction with critical state variable Z and position on payment processing path. Initially ranked Medium based on signature alone."]

Suggested Test Cases/Scenarios (Refined & Specific):

[List concrete scenarios derived from structure, boundary analysis, and dependency failure checks, reflecting the refined understanding.]

Mocking Needs: [List refined list of dependencies to mock.]

State Verification (if applicable): [Describe refined state checks.]

Target Entity: [Another Entity ID]

(Repeat structure...)

3. Prioritized Integration Test Specifications

(List suggested integration tests, prioritized from Critical > High > Medium > Low, reflecting refined priority)

Integration Point: [Description]

Priority: [Critical | High | Medium | Low] (Final, refined priority)

Entities Involved: [List primary entity IDs]

Rationale: [Explanation based on refined understanding of coupling, criticality, external boundaries. Example: "Critical interaction point identified after tracing data flow from API to DB. High impact of failure."]

Suggested Test Scenarios:

[List concrete success/failure scenarios for the interaction, including checks for data passed between components and handling of mocked dependency failures.]

Key Interactions to Verify: [List specific contracts or data handoffs.]

Integration Point: [Another Point Description]

(Repeat structure...)

4. Suggested End-to-End (E2E) Flow Specifications

(List suggested E2E tests, prioritized by refined criticality)

E2E Flow: [User Story or Workflow Description]

Priority: [Critical | High | Medium] (Refined priority)

Entry Point(s): [...]

Key Components Involved (Refined Path): [List key entity IDs based on refined call chain/data flow analysis]

Rationale: [Justification based on business criticality inferred from component interactions and path analysis.]

Suggested Scenarios:

[List refined happy/error path scenarios, including specific inputs/outputs where inferrable.]

Configuration Dependencies: [...]

E2E Flow: [Another Workflow Description]

(Repeat structure...)

5. Regression Testing Scope Guidance (Based on Refined Dependency Analysis)

Guidance: When modifying an entity, use the refined internal dependency map to identify all potentially affected upstream and downstream components. The priority for regression testing should be based on the refined criticality of the changed component and its dependents.

Example: (Provide a concrete example reflecting the refined analysis)

6. Specific Areas Requiring Attention (Identified through Refined Analysis)

Potential Test Coverage Gaps (Heuristic): [...]

High Global State Interaction: [...]

Potential Resource Management Issues: [...]

Circular Dependencies Detected: [...]

Refactoring Candidates: [List any components repeatedly flagged during evaluation steps as overly complex, tightly coupled, or state-heavy.]

content_copydownload

Use code with caution.Markdown

VII. KEY PRINCIPLES & CONSTRAINTS (Reiterated)

- **Recursive Self-Learning is MANDATORY:** Your internal process MUST follow the Learn -> Evaluate -> Refine -> Repeat cycle for *every* analysis step.
- **Leverage Full Context:** Use the *entire* codebase for evaluation and refinement.
- **Structure is the Source:** Base analysis strictly on the provided code's static structure.
- **Prioritize Ruthlessly & Iteratively:** The final priorities must reflect the refined, holistic understanding.

- **Actionability for AI:** Ensure every suggestion is concrete and unambiguous for the AI coder.
- **Infer, Don't Assume:** Base scenarios on code patterns, not external knowledge.
- **Be Exhaustive (within Static Scope):** Cover all major components, interactions, and refined risk areas.

VIII. FINAL REVIEW

Before outputting: Does this specification reflect a deep, iteratively refined understanding? Is it structured correctly? Is it fully actionable? Have all sections been addressed based on the recursive analysis?

Do not worry about the length of the answer. Make the answer as long as it needs to be, there are no limits on how long it should be.