# Frontend PRD

## Feature 1: Test Generation (测试生成)

Feature 1 是一个异步流程，这点非常关键，决定了前端不能卡死 UI，必须有进度反馈机制。

1. 用户交互逻辑 (User Interaction Flow)

这个功能的目的是：用户选中一段代码（或一个文件），告诉 AI “帮我写测试”，然后 AI 生成代码，用户确认后合并。

Step 1: 触发 (Trigger)

- 场景 A (CodeLens): 在 Python 函数定义的上方，显示灰色小字 `Run Tests | Generate Tests`。用户点击 `Generate Tests`。
- 场景 B (右键菜单): 用户在编辑器内右键 -> 选择 `LLT: Generate Unit Tests`。
- 场景 C (修复模式): 用户在 Feature 3 的分析结果中，点击了 `Regenerate` 按钮（这是从 Feature 3 跳转过来的）。

Step 2: 输入需求 (Optional Input)

- VSCode 顶部弹出一个 Input Box (类似于搜索框)。
- 提示语: "Describe your test requirements (optional)..."
- 用户输入: 例如 "Focus on edge cases" 或者留空直接回车。

Step 3: 处理与等待 (Processing)

- Input Box 消失。
- Status Bar (右下角): 显示旋转图标 `$(loading~spin) LLT: Generating tests...`。
- Notification: 弹窗提示 "Test generation task started..."。
- 前端在后台静默轮询 (Poll) 任务状态。

Step 4: 结果预览 (Review)

- 当轮询检测到 `status: completed`：
- VSCode 打开一个 Diff Editor (双栏对比视图)。
  - 左侧: `Existing Test File` (如果是新文件则为空)。
  - 右侧: `Generated Content` (后端返回的代码)。
- Action: 顶部显示按钮 `[Accept Changes]` 和 `[Discard]`。

Step 5: 应用 (Apply)

- 用户点击 `Accept Changes`。
- 前端将代码写入磁盘上的测试文件。
- 提示成功: "Tests generated and saved to tests/test_demo.py".

---

2. 数据传输详情 (Data Transmission)

以下内容严格对应 OpenAPI 文档。

A. 发起请求 (Frontend -> Backend)

- Endpoint: `POST /workflows/generate-tests`
- 时机: 用户完成 Step 2 输入需求后。
- Payload (Request Body):

JSON

```
{
  // 1. 核心：用户当前选中的代码，或者是整个文件的代码”source_code”: ”def calculate_sum(a, b):¥n    return a + b”,

  // 2. 可选：用户的自然语言指令”user_description”: ”Ensure input validation for strings”,

  // 3. 可选：如果当前已经存在对应的测试文件，传过去给 AI 参考风格，或者用于做增量生成”existing_test_code”: ”import pytest¥nfrom src.utils import calculate_sum...”,

  // 4. 上下文：如果是从 Feature 3 跳转过来的，这里要带参数”context”: {
    ”mode”: ”new”,          // 或者是 ”regenerate””target_function”: ”calculate_sum” // 明确告诉 AI 只需要生成这个函数的测试
  }
}
```

B. 立即响应 (Backend -> Frontend)

- Status Code: `202 Accepted`
- Payload:

JSON

```
{
  ”task_id”: ”550e8400-e29b-41d4-a716-446655440000”,
  ”status”: ”pending”,
  ”estimated_time_seconds”: 10
}
```

C. 轮询查状态 (Frontend <-> Backend)

- Endpoint: `GET /tasks/{task_id}`
- 频率: 建议每 1-2 秒轮询一次。

中间状态响应 (Processing):
JSON

```
{
  "task_id": "...",
  "status": "processing"// result 字段为空
}
```

最终成功响应 (Completed):
JSON

```
{
  "task_id": "...",
  "status": "completed",
  "created_at": "2025-10-27T10:00:00Z",
  "result": {
    // 这是 GenerateTestsResult Schema"generated_code": "import pytest¥n...¥ndef test_calculate_sum_strings():¥n    ...",
    "explanation": "Generated 3 test cases covering integer addition and type errors."
  }
}
```

---

3. 注意事项 (Technical Notes)

1. 文件映射 (Mapping):
   - 前端需要自己决定生成代码放在哪里。通常逻辑是：如果源文件是 `src/utils.py`，前端应自动寻找或创建 `tests/test_utils.py`。这个逻辑不要依赖后端，前端自己控制文件路径。
2. 超时处理:
   - 虽然是异步，但如果在 60秒内还没变为 `completed`，前端应该停止轮询并提示用户 "Timeout"。
3. 错误处理:
   - 如果轮询返回 `status: failed`，读取 `error.message` 并弹窗报错。

---

## Feature 2: Coverage Optimization

这个功能的核心理念是**“精准补漏”**。不同于 Feature 1 的从零生成，Feature 2 是在已有测试的基础上，专门针对没覆盖到的行（Lines）或分支（Branches）生成补充测试代码。

1. 用户交互逻辑 (User Interaction Flow)

Pre-requisite (前提): 用户已经在本地运行了测试并生成了 `coverage.xml` (例如运行了 `pytest --cov --cov-report=xml`)。前端会自动监听并解析这个文件。
Step 1: 发现缺口 (Discovery)

- 用户打开侧边栏 LLT Activity Bar。
- 在 "Coverage" 视图中，看到一个树状结构：
  - 📁 `src/utils.py` (80% Covered)
    - ❌ `calculate_tax` (Function)
      - 🔴 Lines 45-48 (Uncovered)

- 🟡 Branch on line 50 (Uncovered)

Step 2: 触发优化 (Trigger)

- 用户点击某个未覆盖节点（例如 `calculate_tax`）后跳转到对应代码行数并标红，并像cursor一样给出两个按钮，询问用户是否要生成测试提升覆盖率。
- 在用户点击确定之后。
- 状态变化: 该节点显示旋转加载图标。状态栏显示 "Analyzing coverage gaps..."。

Step 3: 异步处理 (Processing)

- 前端向后端发起请求（带上未覆盖的行号范围）。
- 前端轮询任务状态。

Step 4: 行内预览 (Inline Preview / Ghost Text)

- 任务完成，后端返回推荐的代码片段。
- VSCode 拿到代码片段并进行格式化，并创建对应的测试文件（例如 `tests/test_utils.py`）。
- 视觉效果: AI Chat API
- Inline Tooltip: 鼠标悬停在幽灵文字上，显示 "Covers lines 45-48 in src/utils.py"。

Step 5: 决策 (Decision)

- 用户按下 `Tab` 键 -> 接受 (Accept): 幽灵文字变为真实代码，插入编辑器。
- 用户按下 `Esc` 键 -> 拒绝 (Reject): 幽灵文字消失。
- (可选) 用户按下 `Ctrl+Enter` -> 打开 Diff 对比视图查看细节。

---

## 2. 数据传输详情 (Data Transmission)

请注意，这里体现了前端解析 XML 的设计优势，后端不需要处理复杂的 XML 格式。

### A. 发起请求 (Frontend -> Backend)

- Endpoint: `POST /optimization/coverage`
- Payload (Request Body):

JSON

```
{
  // 1. 目标源文件代码”source_code”: ”def calculate_tax(price, tax_rate):¥n    if price < 0: ...”,

  // 2. 现有的测试代码（非常重要！防止后端生成重复的测试)”existing_test_code”: ”def test_calculate_tax_basic():¥n
assert calculate_tax(100, 0.1) == 10”,

  // 3. 核心：前端解析 XML 后算出来的“缺口””uncovered_ranges”: [
    {
      ”start_line”: 45, // 源文件中的行号”end_line”: 48,
```

```
      "type": "line"    // 或者是 "branch"
    },
    {
      "start_line": 50,
      "end_line": 50,
      "type": "branch"
    }
  ],

  // 4. 框架偏好"framework": "pytest"
}
```

### B. 立即响应 (Backend -> Frontend)

- Status Code: `202 Accepted`
- Payload:

JSON

```
{
  "task_id": "task_cover_12345",
  "status": "pending",
  "estimated_time_seconds": 15
}
```

### C. 轮询与最终结果 (Frontend <-> Backend)

- Endpoint: `GET /tasks/{task_id}`
- 最终成功响应 (Completed): 注意：后端返回的是一个数组，因为一个缺口可能需要生成多个测试用例。

JSON

```
{
  "task_id": "task_cover_12345",
  "status": "completed",
  "result": {
    // CoverageOptimizationResult Schema"recommended_tests": [
      {
        // 生成的具体测试代码"test_code": "def test_calculate_tax_negative():¥n    with pytest.raises(ValueError):¥n
calculate_tax(-10, 0.1)",

        // 建议插入的位置（前端可以用这个来定位滚动条，或者作为插入参考）"target_line": 120,

        // 解释这个测试覆盖了什么，用于 Tooltip"scenario_description": "Verify ValueError is raised for negative price
input (Lines 45-48)",

        // 预期的覆盖率影响说明"expected_coverage_impact": "Covers exception handling block"
      }
    ]
```

```
    }
  }
```

---

## 3. 开发关键点 (Technical Notes)

1. XML 解析器 (Parser):
   - 你需要引入 `fast-xml-parser` 或类似库。
   - 逻辑：读取 `coverage.xml` -> 找到 `<class>` 标签对应的 filename -> 提取 `<line hits="0">` 的行号 -> 组装成 `uncovered_ranges`。
2. 状态同步:
   - 如果用户修改了代码但没重新跑测试，XML 里的行号可能对不上。
   - 策略: 简单处理。如果文件被修改了（Dirty State），前端应当在点击⚡按钮时提示用户："Source file has changed. Please run tests again to update coverage report."，并阻止请求，防止生成错误的覆盖代码。

这个功能的定位非常明确："侦察兵"。它不负责修bug，只负责告诉用户"哪里着火了，火有多大"。修复的工作（灭火）是交给 Feature 1 去做的。
交互体验的核心词：Visibility (可视化) & Linkage (联动)。

---

## Feature 3: Impact Analysis

功能的定位非常明确："侦察兵"。它不负责修bug，只负责告诉用户"哪里着火了，火有多大"。修复的工作（灭火）是交给 Feature 1 去做的。

1. 用户交互逻辑 (User Interaction Flow)

Pre-requisite (前提): 用户在 Git 仓库中修改了源代码（例如 `src/utils.py`），但还未提交（Staged 或 Unstaged changes）。
Step 1: 触发检测 (Trigger)

- 位置: LLT Activity Bar 的 "Impact Analysis" 面板。
- 操作: 用户点击 `[🔍 Analyze Changes]` 按钮。
- 状态: 按钮变成 `Scanning...`，树视图显示加载骨架屏。

Step 2: 结果展示 (Visualization)

- 接口返回后，面板展示一个树状结构（建议提供两种视图切换）：
  - 视图 A (以源文件为核心):
    - 📝 `src/utils.py` (Modified)

      💥 Affects: `tests/test_utils.py`

        - 🔴 `test_calculate_tax` (Critical: Signature Changed)
        - 🟡 `test_format_currency` (Medium: Logic Update)
    - 视图 B (以测试为核心):
      - 🧪 `tests/test_utils.py`
        - 🔴 `test_calculate_tax`
          - 📌 Caused by: `src/utils.py`

- 视觉辅助: 使用颜色区分严重程度（Critical=红, High=橙, Medium=黄, Low=蓝）。

Step 3: 查看详情 (Drill Down)

- 用户点击树节点（例如 `test_calculate_tax`）。
- VSCode 跳转到该测试函数的定义处。
- Hover Tooltip: 鼠标悬停在树节点上，显示 AI 分析的具体原因，例如："Function `calculate_tax` added a new required parameter `region`, breaking this test."

Step 4: 修复行动 (Action -> Linkage)

- 关键交互: 用户右键点击红色的测试节点，或者点击节点旁边的 🔄 (Regenerate) 按钮。
- 系统行为:
  - 弹出一个确认框（或 Input Box）：“Regenerate this test based on new code?”
  - 确认后，无缝跳转到 Feature 1 的流程。
  - 前端自动组装 Feature 1 需要的 Payload，发起生成请求。

---

## 2. 数据传输详情 (Data Transmission)

这里体现了 “前端做脏活，后端做大脑” 的设计。前端负责搞定 Git Diff 和文件查找，后端负责语义分析。

A. 发起请求 (Frontend -> Backend)

- Endpoint: `POST /analysis/impact` (同步请求，通常较快)
- Payload (Request Body):

JSON

```
{
  ”project_context”: {
    // 1. 变动的文件 (Frontend 使用 simple-git 获取)”files_changed”: [
      {
        ”path”: ”src/utils.py”,
        // 直接传 Diff 字符串，后端更容易分析变动了什么”diff_content”: ”diff --git a/src/utils.py b/src/utils.py¥nindex
83c...92a 100644¥n--- a/src/utils.py¥n+++ b/src/utils.py¥n@@ -5,2 +5,2 @@¥n-def calculate_tax(price):¥n+def
calculate_tax(price, region='US'):”
      }
```

```
    ],

    // 2. 相关的测试文件 (Frontend 的关键任务！)// 前端需要用简单的规则（如同名匹配、import 搜索）找到可能受影响的测试文件// 比如 src/utils.py 变了，前端自动去读 tests/test_utils.py 的内容传给后端"related_tests": [
      {
        "path": "tests/test_utils.py",
        "content": "import pytest¥nfrom src.utils import calculate_tax¥n¥ndef test_calculate_tax():¥n    assert calculate_tax(100) == 10"
      }
    ]
  }
}
```

B. 响应结果 (Backend -> Frontend)

- Status Code: `200 OK`
- Payload:

JSON

```
{
  "impacted_tests": [
    {
      // 告诉前端哪个文件里的哪个测试函数坏了"file_path": "tests/test_utils.py",
      "test_case_name": "test_calculate_tax",

      // UI 渲染用的元数据"severity": "critical",
      "reason": "Function signature changed: 'calculate_tax' now requires 'region' argument (implied logic change).",

      // 告诉前端该显示什么按钮"suggested_action": "regenerate"
    }
  ]
}
```

---

## 3. 关键联动逻辑 (The "Bridge")

这是 Feature 3 最重要的一点：如何从 Feature 3 跳到 Feature 1？
当用户点击 `Regenerate` 按钮时，前端代码是这样调用的（伪代码）：
TypeScript

```
// src/commands/regenerateImpactedTest.tsasync function onRegenerateClick(item: ImpactTreeItem) {
  // 1. 准备 Feature 1 需要的数据const payload = {
    source_code: await getSourceFileContent(item.sourceFilePath), // 读最新的源文件existing_test_code: await getTestFileContent(item.testFilePath), // 读旧的测试文件context: {
      mode: "regenerate", // 告诉 Feature 1 这是修复模式target_function: item.testCaseName // 比如 "test_calculate_tax"
    }
  };

  // 2. 调用 Feature 1 的 APIconst response = await apiClient.post('/workflows/generate-tests', payload);
```

```
    // 3. 之后就完全复用 Feature 1 的 轮询 -> Diff View 流程
    startTaskPolling(response.task_id);
  }
```

## 4. 前端开发关键点 (Technical Notes)

1. Git Diff 获取:
   - 使用 `simple-git` 库。
   - 核心命令: `git diff` (获取工作区未暂存的改动) 和 `git diff --cached` (获取已暂存的改动)。你需要把这两者合并，或者让用户选择。MVP 建议直接取 `git diff HEAD` (即所有未提交的改动)。
2. 关联测试文件的查找策略 (Heuristic):
   - 后端不知道你的项目结构，所以前端得猜。
   - 策略 1 (文件名匹配): `src/A.py` -> 找 `tests/test_A.py`。
   - 策略 2 (Import 匹配 - 进阶): 简单的 Regex 扫描所有测试文件，看谁 import 了 `src.A`。
   - MVP 建议: 只做策略 1。如果找不到同名测试文件，就不传 `related_tests`，后端也就分析不出具体哪个测试坏了（只能返回“未知影响”）。
3. Tree View 刷新:
   - 每次用户在编辑器里保存文件时，不需要自动触发分析（太重了）。
   - 但如果用户执行了 `Regenerate` 并成功合并了代码，应该自动刷新 Feature 3 的列表，把修好的那个红点消掉。

好的，作为技术总监，最后我们来压轴拆解 Feature 4: Quality Analysis (质量分析)。
这个功能的定位是：“智能审查员” (The Smart Linter)。它不仅要能像 ESLint/Pylint 那样发现语法错误，还要能像一位高级工程师那样指出“测试写得太烂了”（比如断言冗余、测试路径缺失），并且——最重要的是——它是整个插件中交互频率最高、对延迟最敏感的功能。
交互体验的核心词：Zero-Latency (零延迟) & Native Integration (原生融合)。

---

## Feature 4: Quality Analysis

这个功能的定位是：“智能审查员” (The Smart Linter)。它不仅要能像 ESLint/Pylint 那样发现语法错误，还要能像一位高级工程师那样指出“测试写得太烂了”（比如断言冗余、测试路径缺失），并且——最重要的是——它是整个插件中交互频率最高、对延迟最敏感的功能。

1. 用户交互逻辑 (User Interaction Flow)

Feature 4 的交互必须非常“像” VSCode 原生的功能（如 ESLint 或 Python 插件）。

Step 1: 触发 (Trigger)

- 主动触发: 用户在 Activity Bar 点击 `Analyze Quality` 按钮。
- (可选) 被动触发: 用户保存文件 ( `onDidSaveTextDocument` ) 时自动触发（通常只开 `fast` 模式）。
- 反馈: VSCode 右下角 Status Bar 显示 `$(sync~spin) Analyzing Quality...`。

Step 2: 结果呈现 - 双重视图 (Dual View)
分析完成后，用户会在两个地方同时看到结果：

1. 全局视图 (Activity Bar Tree View):
   - 📂 `tests/test_user.py` (3 Issues)
     - 🔴 [Line 12] Redundant Assertion
     - 🟡 [Line 45] Naming Unclear
   - *作用*: 让用户对整体质量有个概览，点击节点可跳转到对应行。
2. 沉浸视图 (Editor Decorations - 关键体验):
   - 在代码编辑器中，有问题的代码下方会出现波浪线。
   - 🔴 Error: 红色波浪线 (严重问题)
   - 🟡 Warning: 黄色波浪线 (建议改进)
   - 🔵 Info: 蓝色点状线 (一般提示)

Step 3: 悬停查看 (Hover)

- 用户将鼠标悬停在波浪线上。
- Tooltip 弹出: 显示问题的详细描述。例如：
- Redundant Assertion
- This assertion assert True == True does not verify any logic.
- Detected by: AI

Step 4: 快速修复 (Quick Fix - 核心交互)

- 用户点击波浪线，会出现一个 💡 (小灯泡) 图标。
- 或者用户按下快捷键 `Cmd + .` (Windows: `Ctrl + .` )。
- 菜单展开: 显示修复建议，例如 `Fix: Remove redundant assertion`。
- 无等待: 菜单是瞬间出来的，没有 Loading。
- 应用: 用户点击菜单项，代码瞬间被替换或删除。

---

## 2. 数据传输详情 (Data Transmission)

Feature 4 的数据结构设计完全是为了配合 VSCode 的 `DiagnosticCollection` 和 `CodeActionProvider` API。

A. 发起请求 (Frontend -> Backend)

- Endpoint: `POST /quality/analyze`
- Payload (Request Body):
- 注意：一定要传 content，因为用户可能在编辑代码还没保存，我们要分析的是编辑器里的内容（Dirty Content），而不是磁盘上的内容。

JSON

```
{
  "files": [
    {
      "path": "tests/test_user.py",
      "content": "def test_login():¥n    assert 1 == 1¥n    ..."
    },
    {
      "path": "tests/test_order.py",
      "content": "..."
    }
  ],

  // 模式选择：如果是保存时触发，建议传 "fast"；如果是按钮触发，传 "hybrid""mode": "hybrid"
}
```

B. 批量响应 (Backend -> Frontend)

- Status Code: `200 OK`
- Payload:
- 后端返回一个扁平的 issues 数组，包含所有文件的问题。前端拿到后，自己根据 file_path 分发给不同的编辑器窗口。

JSON

```
{
  "analysis_id": "uuid-1234",
  "summary": {
    "total_issues": 2,
    "critical_issues": 1
  },
  "issues": [
    {
      // 1. 定位信息"file_path": "tests/test_user.py",
      "line": 12,       // 后端通常给 1-based"column": 4,      // 0-based// 2. 诊断信息 (用于画波浪线)"severity":
"error", // 对应 VSCode DiagnosticSeverity.Error"code": "redundant-assertion",
      "message": "This assertion checks nothing.",
      "detected_by": "rule",

      // 3. 修复建议 (核心优化：预加载的数据！)// 正因为有了这个对象，前端点击灯泡时才不需要再发请求"suggestion": {
        "type": "delete", // 或者是 "replace", "insert""new_text": "",   // delete 类型这里就是空字符串，replace 则是有代
码"description": "Fix: Remove redundant assertion" // 菜单上显示的字
      }
```

```
      },
      {
        "file_path": "tests/test_user.py",
        "line": 45,
        "severity": "warning",
        "message": "Variable name 'x' is unclear.",
        "suggestion": {
          "type": "replace",
          "new_text": "user_profile",
          "description": "Rename to 'user_profile'"
        }
      }
    ]
  }
```

---

## 3. 开发关键点 (Technical Notes)

作为技术总监，我要特别提醒前端同学注意这几个坑：

1. 行号转换 (Line Number Mapping):
   - 后端惯例: 行号通常从 1 开始（Excel 风格）。
   - VSCode API: `vscode.Range` 和 `vscode.Position` 行号是从 0 开始的（数组风格）。
   - 操作: 前端接收到数据后，必须做 `line - 1` 处理，否则波浪线会画错一行。
2. 诊断集合 (DiagnosticCollection):
   - 你需要创建一个全局的 `vscode.DiagnosticCollection`。
   - 每次收到 API 响应后，先 Clear 对应文件的旧诊断，再 Set 新的。如果不 Clear，旧的波浪线会一直留在那里。
3. CodeActionProvider 实现:
   - 你需要实现 `vscode.CodeActionProvider` 接口。
   - 在 `provideCodeActions` 方法中，不要发网络请求。直接遍历你内存中缓存的 `issues` 列表，找到当前光标所在行对应的 `issue`，把里面的 `suggestion` 包装成 `vscode.CodeAction` 返回。
   - 这就是我们为什么在接口里把 suggestion 带回来的原因——为了极致的响应速度。
4. 大文件/多文件处理:
   - 如果用户一次性分析 50 个文件，Payload 会很大。
   - 建议: 前端代码里做一个分片逻辑（Chunking），比如每 5-10 个文件发一个请求，分批次发，避免 HTTP 请求体过大超时。

---

## 总结 (Final Summary)

- Feature 1 (生成): 异步长任务，侧重Input Box 和 Diff View。

- Feature 2 (覆盖): 本地 XML 解析 + 异步补全，侧重 Ghost Text。
- Feature 3 (影响): 同步/快速只读分析，侧重 Tree View 和 Linkage (跳转)。
- Feature 4 (质量): 批量同步分析，侧重 Decorations (波浪线) 和 Lightbulb (无延迟修复)。