E2E 自动化测试脱敏版本

一、E2E 与 Playwright 框架基础

1.1 E2E（端到端）测试核心概念

端到端（End-to-End，简称 E2E）测试是一种从真实用户的使用视角出发，对整个应用系统进行系统性、完整性验证的测试方法。它从系统的起点开始，穿过所有集成模块、网络、数据库等后台服务，最终在用户界面的终点完成闭环校验。

> 💡 **E2E 测试的核心价值：**
>
> - **真实性验证**：在最接近真实生产环境的场景下验证业务逻辑，确保前端、后端 API、数据库及第三方服务协同工作。
> - **回归保障**：模拟用户真实的操作链路（如“登录 -> 商品开单 -> 提交销售单 -> 收款入账”），防止关键业务路径因代码变更而发生故障。
> - **提高上线信心**：为研发和测试团队提供上线前的最后一道质量防线。

1.2 Playwright 框架特性

Playwright 是由微软（Microsoft）开源的新一代端到端测试框架，具有以下核心技术优势：

- **跨浏览器与平台**：单套 API 驱动 Chromium （Chrome， Edge）、Webkit （Safari） 和 Firefox，支持 macOS、Windows 及 Linux。
- **零配置自动等待（Auto-wait）**：在对元素执行操作前，自动检查可见性、可交互性等，极大缓解了随机失败问题（Flaky tests）。
- **基于 WebSocket 协议**：基于底层 Web Sockets 双向协议，执行速度远超传统基于 HTTP 协议的 WebDriver。

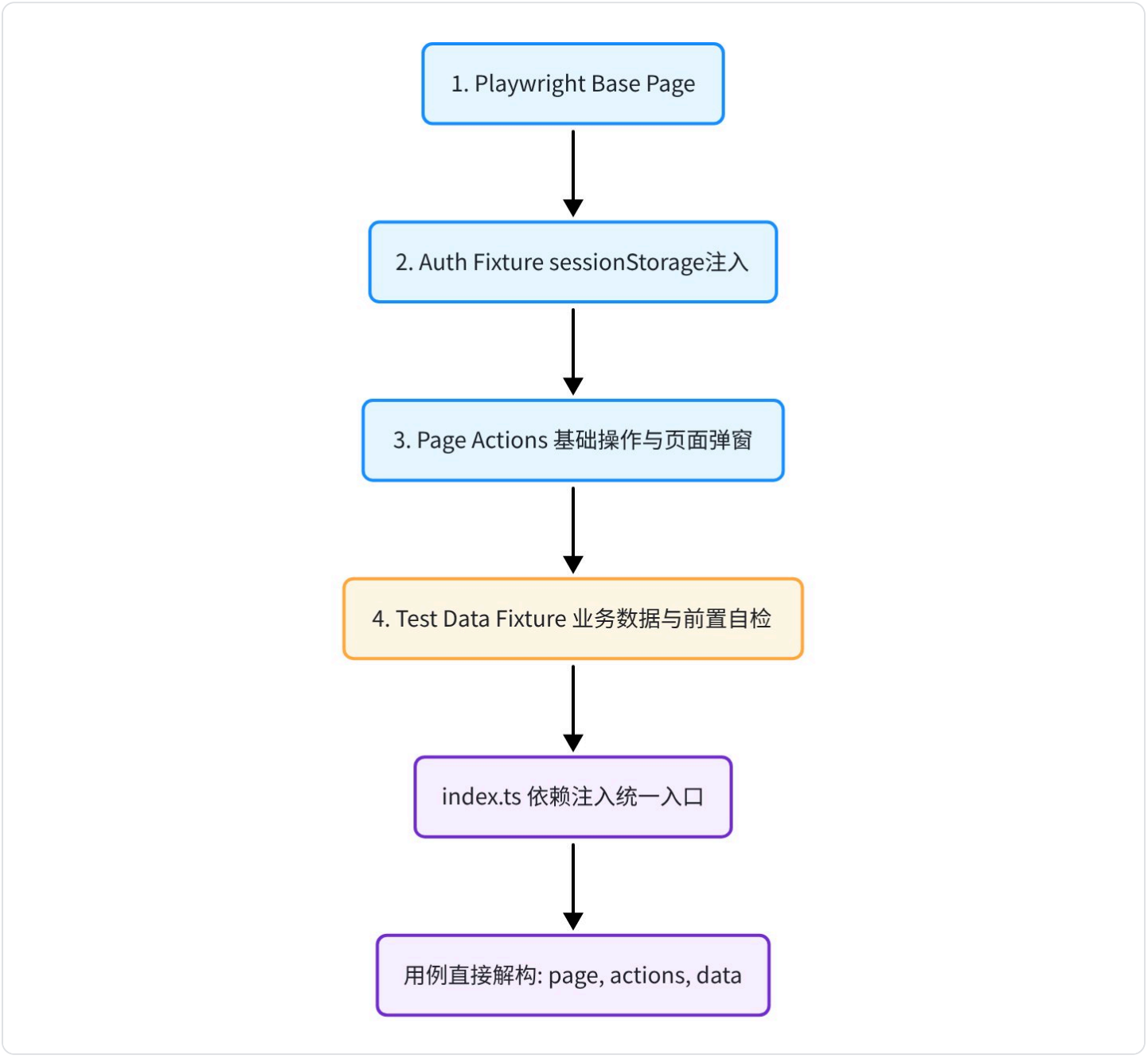
二、项目 E2E 自动化框架解析

2.1 整体架构与对标分析

本项目针对**管家婆云创业版**，采用先进的分层设计模式，完全对标并超越了官方标准，实现了高内聚与代码复用：

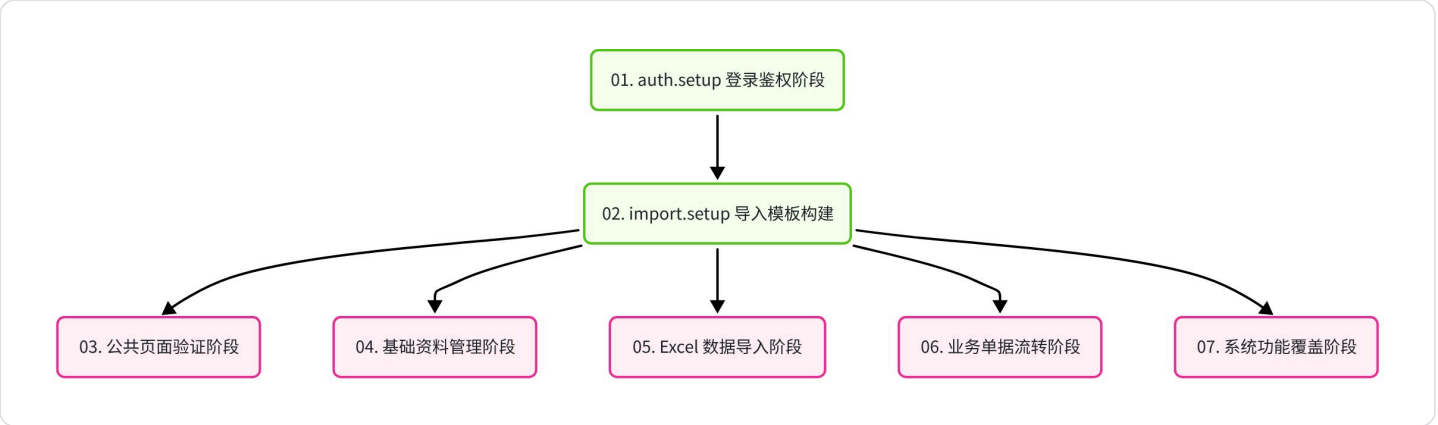
核心维度	落地状态	框架实现与代码映射分析
Test Fixtures	✅ 组合封装	构建了 **4 层依赖注入链**：`base` → `auth` → `page-actions` → `test-data` → `index.ts`，统一了测试入口。
Project Dependencies	✅ 流水线编排	在 `playwright.config.ts` 中编排了从“认证准备”到“单据流转”的 7 阶段依赖运行流。
Auto-wait	✅ 强制门禁	禁止代码中的硬等待，通过 `npm run framework:check` 静态检查拦截违规代码。
Reporters	✅ 报告	集成标准 Allure，定制了业务场景质量矩阵分析模块 `quality-reporter.ts`。

4 层依赖注入链设计（Test Fixtures）拓扑图：



2.2 项目结构与流水线编排

项目的核心配置使用 Playwright 提供的项目依赖机制（`dependencies`），构建了一个多阶段的管道流，确保测试的前置条件被严格满足：



2.3 核心能力模块解析

2.3.1 验证码识别：通义千问-VL

在 login.ts 中，项目实现了一种极具弹性的智能登录助手：

- **拦截与抓取**：页面加载时，自动抓取验证码图片的 base64 编码数据。
- **AI 视觉识别**：将 base64 数据发送给阿里云百炼平台托管的 **通义千问-VL （Qwen-VL-Plus）** 视觉大模型进行字符识破。
- **失败自愈**：内置了 3 次重试闭环。若 AI 识别错误导致登录失败，会自动刷新验证码并重新调用 AI 二次识别。

2.3.2 业务场景覆盖度追踪

项目在 `tests/coverage/page-coverage.ts` 中声明了各个模块（如：商品信息、往来单位等）的核心场景与证据项，通过 `quality-reporter.ts` 自动在测试完成时计算业务覆盖率，生成直观报表。

2.3.3 大模型智能错误归因

集成 LLM**大模型**，在用例失败时，结合 `afterEach` 钩子与 `failure-analysis.ts`，自动分析步骤日志和错误堆栈，精准归类为 `业务系统Bug`、`测试脚本问题`、`测试数据问题` 或 `测试环境问题`，并输出修复建议。

2.3.4 结构化用例管理与场景清单

为了实现业务测试场景的规范化管理以及精细化覆盖统计，项目在 all-pages-e2e.json 中将所有的 E2E 业务用例以结构化 JSON 数据形式进行集中管理。这使得自动化测试不仅是代码，也是可被程序解析和度量的元数据：

- **场景清单元数据**：该文件声明了每个核心业务用例（如收支类型、仓库信息、员工信息等）的描述、优先级、前置条件、具体的业务操作步骤（如 navigate， click， fill， verify 等）及预期结果（Expected Result）。

图 2.1：结构化测试用例 JSON 配置（all-pages-e2e.json）


```json
{
  "name": "商品信息真实全功能",
  "description": "验证商品搜索、新增、编辑、复制、启停、删除、导入、导出和分页。",
  "priority": "high",
  "prerequisites": ["已登录", "DeepSeek 数据已生成", "商品导入模板已生成"],
  "steps": [
    {"step_number": 1, "action": "navigate", "description": "打开商品信息", "target": "商品信息页", "locator": null, "data": "/basic/product"},
    {"step_number": 2, "action": "fill", "description": "填写 DeepSeek 商品表单并保存", "target": "新增商品表单", "locator": null, "data": "DEEPSEEK_PRODUCT"},
    {"step_number": 3, "action": "verify", "description": "执行编辑、复制、启停和删除", "target": "商品行操作", "locator": null, "data": "DEEPSEEK_PRODUCT"},
    {"step_number": 4, "action": "upload", "description": "上传动态商品 Excel", "target": "导入商品", "locator": null, "data": "product-import.xlsx"},
    {"step_number": 5, "action": "click", "description": "导出商品表格", "target": "导出表格", "locator": null, "data": null}
  ],
  "expected_result": "商品 CRUD、状态、导入导出均真实生效。",
  "verification_points": [{"type": "element_visible", "description": "动态商品行可见", "locator": null, "expected": true}],
  "page_elements": [],
  "tags": ["basic", "product", "crud", "import", "export"]
},
{
  "name": "往来单位真实全功能",
  "description": "验证客户和供应商页签内的新增、编辑、删除、导入、导出和搜索。",
  "priority": "high",
  "prerequisites": ["已登录", "DeepSeek 客户和供应商数据已生成"],
  "steps": [
    {"step_number": 1, "action": "navigate", "description": "打开往来单位", "target": "往来单位页", "locator": null, "data": "/basic/partner"},
    {"step_number": 2, "action": "click", "description": "切换客户和供应商页签", "target": "业务页签", "locator": null, "data": null},
    {"step_number": 3, "action": "verify", "description": "分别执行新增、编辑、删除重建", "target": "单位表单和行操作", "locator": null, "data": "DEEPSEEK_PARTNERS"},
    {"step_number": 4, "action": "upload", "description": "分别导入客户和供应商 Excel", "target": "导入", "locator": null, "data": "DYNAMIC_PARTNER_XLSX"},
    {"step_number": 5, "action": "click", "description": "分别导出客户和供应商", "target": "导出", "locator": null, "data": null}
  ],
  "expected_result": "客户与供应商数据独立且全部操作成功。",
  "verification_points": [{"type": "element_visible", "description": "动态单位行可见", "locator": null, "expected": true}],
  "page_elements": [],
  "tags": ["basic", "partner", "crud", "import"]
},
{
  "name": "结算项目真实全功能",
  "description": "验证实账和免账项目新增、编辑、删除及重新创建。",
  "priority": "high",
  "prerequisites": ["已登录", "DeepSeek 结算数据已生成"],
  "steps": [
    {"step_number": 1, "action": "navigate", "description": "打开结算项目", "target": "结算项目页", "locator": null, "data": "/basic/settlement-item"},
    {"step_number": 2, "action": "click", "description": "添加实账并保存", "target": "添加实账", "locator": null, "data": "DEEPSEEK_ACCOUNT"},
    {"step_number": 3, "action": "click", "description": "添加免账并保存", "target": "添加免账", "locator": null, "data": "DEEPSEEK_EXEMPTION"},
    {"step_number": 4, "action": "verify", "description": "编辑、删除并重建两个项目", "target": "项目操作", "locator": null, "data": "DEEPSEEK_SETTLEMENT"}
  ],
  "expected_result": "实账和免账项目 CRUD 全部成功。",
  "verification_points": [{"type": "element_visible", "description": "编辑后的项目可见", "locator": null, "expected": true}],
  "page_elements": [],
```


- **测试场景查询与统计**：执行命令 `npm run test:list`，系统会调用 Playwright 的列用例指令（`playwright test --list`）检索所有当前活跃的自动化测试用例，并触发执行 `scripts/coverage-summary.ts` 脚本，输出总页面数、覆盖场景数和业务模块统计摘要。

图 2.2：`npm run test:list` 用例清单控制台输出（前页）



图 2.3：`npm run test:list` 控制台统计与覆盖度摘要（后页）



2.3.5 业务需求到测试场景的 Specs 转换

为了让测试场景的设计具备高可追溯性并契合业务，项目建立了基于 `specs/*.md` 的**需求追踪与自动化规格规范**，作为业务 PRD 与代码实现的桥梁：

- **宏观需求追踪矩阵**：在 aicyberp-requirements-traceability.md 中，清晰映射了系统各大需求模块（如资料管理、采购、销售、财务扩展、增值服务等）与文档编号，并明确定义了各模块当前覆盖状态（已覆盖、覆盖缺口、后续补充计划）及执行分层（P0 主回归、P1 需求补充、P2 独立端能力）。这极大保证了 E2E 覆盖的广度与质量可见性。
- **微观测试场景契约**：以基础资料模块为例，在 master-data.md 中，以高度结构化的规范为每个页面（如商品信息、往来单位）规划了每个操作序列的前置条件（prerequisites，如 `ensurePrerequisites` 校验）、数据隔离策略（如 workerIndex 隔离）、是否可并行，以及断言要点。这为测试的自动化实现提供了详尽的规格蓝图。

三、项目结构与本地开发指南

3.1 核心目录与关键文件分工

项目核心文件结构

```
cyberp_test/
├── specs/                    # 业务功能规格说明书目录（Markdown 文件）
├── tests/                    # 自动化测试核心代码目录
│   ├── setup/                # 预备执行阶段目录（测试依赖准备）
│   │   ├── auth.setup.ts     # UI 登录抓取登录态，持久化为 .auth/user.json
│   │   └── import.setup.ts   # 测试所需的 Excel 数据导入模板初始化构建
│   ├── e2e/                  # 核心测试场景用例文件（spec.ts）
│   │   ├── master-data/      # 商品信息、往来单位等基础资料 CRUD 测试
│   │   └── documents/        # 采购、销售及收款等单据完整闭环流转测试
│   ├── fixtures/             # 统一 Fixtures 依赖注入链
│   ├── reporters/            # 自定义执行报告处理器（quality-reporter.ts）
│   └── coverage/             # 业务场景检测指标 definition（page-coverage.ts）
├── scripts/                  # 辅助开发与流水线工具脚本
├── playwright.config.ts      # Playwright 测试引擎核心配置文件
├── package.json              # 工程依赖、配置及快捷指令
└── .env                      # 环境变量、阿里云百炼 Key 与后台服务凭证
```

3.2 本地运行与调试

3.2.1 本地初始化

1. 运行 `npm install` 安装依赖。
2. 运行 `npx playwright install chromium` 下载浏览器内核。
3. 复制并配置 `.env` 文件，填入 `TEST_PHONE`、`TEST_PASSWORD` 和 `DASHSCOPE_API_KEY`。

3.2.2 测试运行调试

1. 运行 `npm test` 进行全量回归。
2. 运行指定文件：`npx playwright test tests/e2e/master-data/master-data-crud.spec.ts`
3. 调试模式：运行 `npx playwright test --ui` 开启交互界面。

四、自动化开发与调试实战

4.1 DOM 探索与定位器选择

在自动化测试用例开发中，第一步通常是“寻找并定位页面上的元素”。在本项目中，必须遵循以下核心准则：

- **用户视角优先**：禁止使用不稳定的深层 CSS 选择器或绝对路径 XPath（如 `/html/body/div[1]/div/div[2]/...` 或 `.ant-btn-primary`）。因为页面结构的微小变化极易导致选择器失效，从而产生不稳定的测试（Flaky tests）。
- **推荐定位器语法**：
  - 使用页面角色（Role）：`page.getByRole('button', { name: '确认' })`
  - 使用展示文本（Text）：`page.getByText('操作成功')`
  - 使用表单标签（Label）：`page.getByLabel('商品名称')`
- **DOM 探索实战**：本地调试时，运行 `npx playwright test --ui`。在弹出的 Playwright UI 界面的上方，点击 **"Pick Locator"** 按钮。将鼠标悬停在浏览器的任意页面元素上，Playwright 会在右下角自动生成最符合用户视角的定位器代码，直接复制到测试脚本中即可。

4.2 脚本编写与维护实战

4.2.1 案例 1：基础商品 CRUD 脚本开发

在用例文件中，必须从 `tests/fixtures` 导入 `test` 和 `expect`。这会自动完成系统登录、免密状态共享，无需重复编写登录逻辑。测试用例通过链式注入 `{ page, actions, data }` 获得操作能力：

1. 新增商品弹窗填写的原子封装助手

```
async function createProductThroughUi(
  page: Page,
  actions: PageActions,
  data: RuntimeBusinessData,
) {
```

```
  // 1. 点击“新增商品”按钮打开弹窗
  await page.getByRole('button', { name: /新增商品/ }).click();
  const current = actions.dialog();

  // 2. 通过 placeholder 智能填充大模型生成的业务随机数据，规避脏数据冲突
  await current.getByPlaceholder('请输入商品名称').fill(data.product.name);
  await current.getByPlaceholder('请输入商品编号').fill(data.product.code);
  await current.getByPlaceholder('请输入规格型号').fill(data.product.model);
  await current.getByPlaceholder('请输入基础单位').fill(data.product.unit);
  await current.getByPlaceholder('请输入品牌').fill(data.product.brand);
  await current.getByPlaceholder('请输入库存上
限').fill(String(data.product.stockUpper));
  await current.getByPlaceholder('请输入库存下
限').fill(String(data.product.stockLower));
  await current.getByPlaceholder('请输入产地').fill(data.product.origin);
  await current.getByPlaceholder('请输入备注').fill(data.product.remark);

  // 3. 填入预设的采购价、销售价及零售价（使用 nth 索引）
  const prices = current.getByRole('spinbutton');
  await prices.nth(2).fill(String(data.product.purchasePrice));
  await prices.nth(3).fill(String(data.product.salePrice));
  await prices.nth(4).fill(String(data.product.retailPrice));

  // 4. 提交保存
  await actions.submitDialog();
}
```

2. 核心商品测试用例逻辑（tests/e2e/master-data/master-data-crud.spec.ts）

```
import { test, expect } from '../../fixtures';

test('商品：新增与列表落库校验', async ({ page, actions, data }) => {
  // 1. 路由导航至基本资料 the 商品页面
  await actions.openPage('/basic/product');

  // 2. 搜索确认该商品是否已存在（避免前置步骤已创建而导致重复新增失败）
  await actions.search(/搜索商品名称、编号/, data.product.code);
  let row = page.getByRole('row').filter({ hasText: data.product.code
}).last();

  if (!(await row.isVisible().catch(() => false))) {
    // 3. 商品不存在，通过 UI 执行完整新增流程
    await createProductThroughUi(page, actions, data);

    // 4. 重新搜索并获取新创建的商品行
    await actions.search(/搜索商品名称、编号/, data.product.name);
```

```
      row = page.getByRole('row').filter({ hasText: data.product.name }).last();
      if (!(await row.isVisible().catch(() => false))) {
        // 5. 兼容自愈：如果前置数据在生命周期内已被更新，则使用 updatedName 进行回退搜索
        await actions.search(/搜索商品名称、编号/, data.product.updatedName);
        row = page.getByRole('row').filter({ hasText: data.product.updatedName
}).last();
      }
    }

    // 6. 最终结果断言
    await expect(row).toBeVisible();
  });
```

4.2.2 案例 2：UI + API 双重验证

在大规模自动化测试中，纯 UI 断言有时会由于前端渲染延迟导致测试失败。因此，本项目在 `tests/e2e/reports/reports-system.spec.ts` 中引入了 UI 与 API 的双重验证：

- **动作触发**：模拟用户在 UI 表格行上点击“查看明细”按钮。
- **网络层强校验**：拦截并捕获该操作触发的后台 API 请求。如果请求的 HTTP 状态码非 200，或者接口返回的 JSON 数据中业务码 `code !== 'A00000'`，测试会**立刻报错**并还原业务层面的详细错误，拒绝无效的前端等待。
- **UI 层渲染断言**：当前端 API 成功返回数据后，最终断言界面上的 Drawer 抽屉或 Modal 弹框是否已正常呈现。

UI + API 联动双重断言示例

```
async function assertReportDetail(
  page: Page,
  detailButton: Locator,
  label: string,
  detailApiPath: string,
) {
  // 1. 设置 API 拦截 Promise，等待特定明细请求响应
  const responsePromise = page.waitForResponse(
    (response) => response.url().includes(detailApiPath),
    { timeout: 15_000 },
  );

  // 2. 点击明细按钮触发接口请求与 UI 渲染
  await detailButton.click();

  // 3. 强拦截网络接口响应进行第一重校验
  const response = await responsePromise;

```

```
    // 3.1 校验 HTTP 状态
    if (!response.ok()) {
      throw new Error(`${label} 明细接口返回 HTTP ${response.status()} 异常。`);
    }

    // 3.2 校验后端业务状态码
    const payload = await response.json().catch(() => undefined);
    if (payload?.code && payload.code !== 'A00000') {
      throw new Error(
        `${label} 明细接口返回业务层错误: code=${payload.code},
  message=${payload.message ?? ''}`
      );
    }

    // 4. 第二重校验：在确认后端数据到达后，断言 UI 层的 Modal 弹窗正常呈现
    const modal = page
      .locator('[role="dialog"]:visible, .ant-modal-content:visible')
      .last();
    await expect(modal).toBeVisible();
  }
```

4.3 AI 协同开发中的 Bug 发现与 Trace 归因

在 E2E 自动化测试的用例编写与场景覆盖（特别是使用 AI 编程辅助工具如 Qoder/TRAE 协同开发）过程中，AI 不仅能加速脚本编写，还能敏锐地捕获并深度定位系统的潜在缺陷。以下是项目推进全模块场景覆盖时的一个真实实战案例：

4.3.1 AI 辅助的覆盖现状审计与规划

在覆盖率梳理阶段，AI 助手通过遍历分析 page-coverage.ts 定义、specs 计划和实际运行的 spec 脚本，自动审计并梳理出了详尽的“已覆盖”与“未覆盖”业务模块状态表，并据此给出了优先级分阶段推进的测试开发建议：

图 4.3：AI 自动梳理的 E2E 覆盖缺口分析表



根据对项目需求文档、测试计划、覆盖率定义文件和实际 e2e 测试脚本的完整分析，结论是：**当前 e2e 测试场景覆盖不全。**

下面是详细的覆盖情况分析。

一、已覆盖模块（核心主链路，35 个页面）

这部分在 page-coverage.ts 中有明确定义，对应的 spec 脚本也已实现：

模块	需求编号	覆盖页面数	覆盖状态	对应脚本
资料管理	ZL-001~006	6/6	✅ 完整	master-data-crud.spec.ts
采购管理	CG-001~004	4/4	✅ 完整	purchase.spec.ts
销售管理	XS-001~005	5/5	✅ 完整	sales.spec.ts
库存管理	KC-001~007	7/7	✅ 完整	inventory.spec.ts
财务管理(基础)	CW-001~008	7/8	⚠️ 部分	finance.spec.ts
系统管理	XT-002~004	3/4	⚠️ 部分	system-management.spec.ts
增值服务(入口)	US 部分	2/2	⚠️ 仅入口	value-added-services.spec.ts
认证	-	1	✅ 完整	authentication.spec.ts
导入	-	-	✅ 完整	imports.spec.ts
报表	-	-	✅ 完整	reports-system.spec.ts

已覆盖的 35 个页面都有完整的脚本实现，且关键业务链路做到了 UI 操作 + 后端 API 联合断言（库存、应收应付、资金流水、明细接口）。

二、未覆盖模块（需求已定义，但无自动化脚本）

以下模块在 AICybERP-docs 中有完整需求文档，在 specs/ 目录下也有测试计划，**但尚未生成 e2e 测试脚本**：

4.3.2 真实 Bug 捕获：付款单作废 API 调用异常

在开发财务管理模块的“付款单作废”场景时，E2E 测试运行到 UI 页面点击“作废”时频繁失败。通过捕获底层的网络请求并配合 Trace 分析，AI 最终发现并归因出了一个隐藏的前端路由/接口调用 Bug：

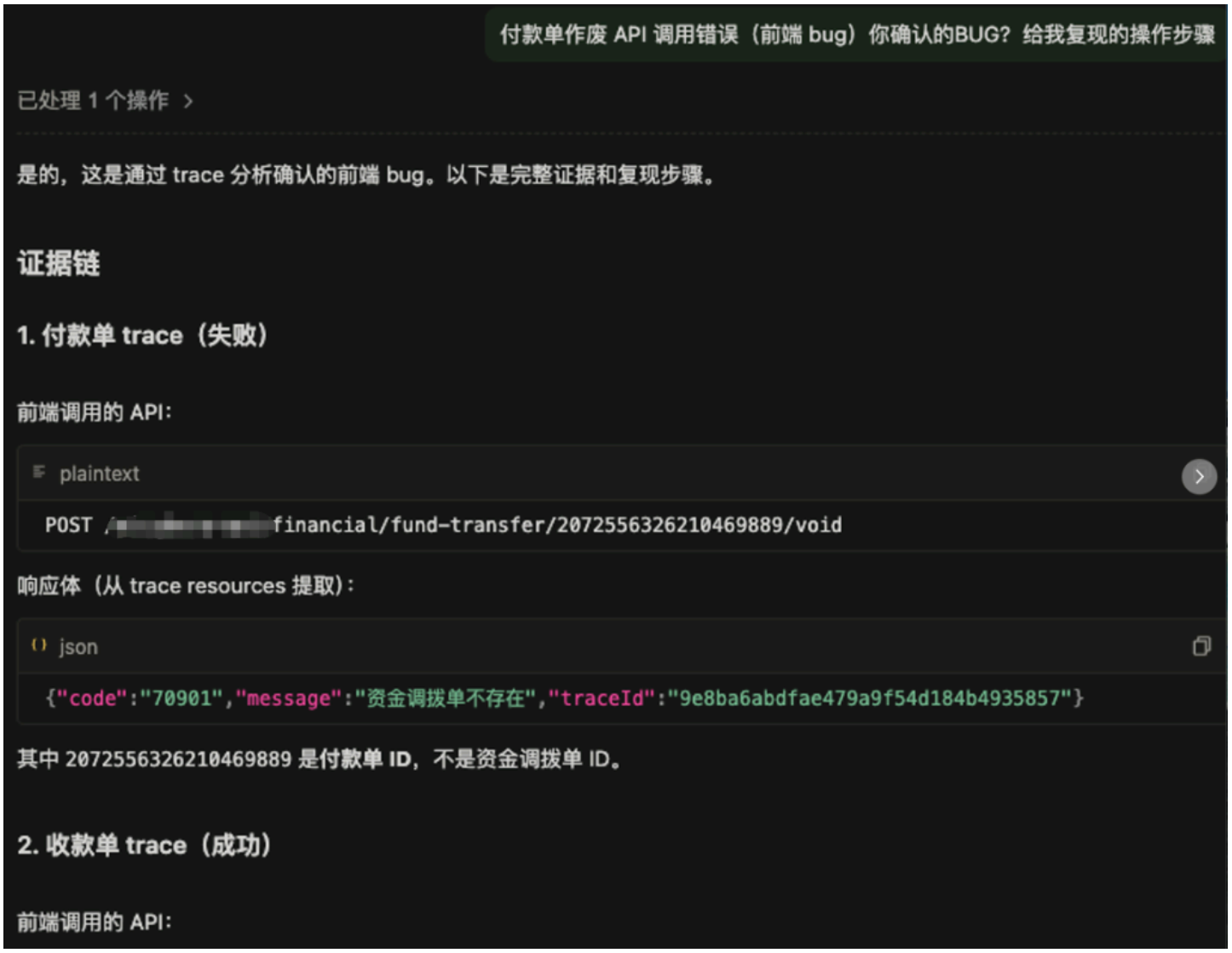
- **缺陷现象描述**：前端“作废”按钮在付款单详情页被点击时，错误地向后台发送了 `POST /financial/fund-transfer/{id}/void`（资金调拨单作废接口）请求，且把付款单 ID 作为资源标识符传入，导致后端校验失败，返回 `code: 70901`，提示“资金调拨单不存在”错误。而在收款单页面，前端则能正确调用 `PUT /financial/receipt-orders/{id}/void` 接口。
- **测试自愈与规避手段**：在提交 Bug 给开发团队的同时，AI 在测试框架中实现了一个名为 `voidFinanceDocument` 的 API Helper，在测试脚本中直接发起正确的 `PUT /financial/payment-orders/{id}/void` 后端网络请求，从而保证测试脚本能够绕过前端界面上的 Bug 阻碍，继续验证后端的作废落库与流水账逻辑。

图 4.4：付款单作废 API 异常描述

3. 付款单作废 API 调用错误（前端 bug，影响付款单测试）

- 前端「作废」按钮对付款单错误调用了 POST /financial/fund-transfer/{id}/void（资金调拨单作废接口），API 返回 {"code":"70901","message":"资金调拨单不存在"}
- 收款单前端正确调用 PUT /financial/receipt-orders/{id}/void，所以收款单测试通过
- 修复：新增 voidFinanceDocument helper，直接调用正确的 PUT /financial/payment-orders/{id}/void

图 4.5：付款单作废 API 异常 Trace 证据链



4.4 财务扩展场景覆盖开发示范

为了让其他测试人员清晰借鉴“如何从零设计并落地一个复杂业务场景的自动化用例”，项目建立了标准的“**Specs 计划设计 → 联动断言规划 → Playwright 脚本生成**”三步走交付工作流。以下以财务扩展（CW-009 ～ CW-012）模块的开发过程为例进行完整复盘：

4.4.1 步骤 1：Specs 契约构建

在编写代码前，首先在 finance-extended.md 中定义该模块的测试范围、页面清单及场景依赖。以“财务期间管理”页面为例，我们明确规划了“开启活跃期间”和“结账期间与单据锁定”两个核心测试场景的前置条件（Prerequisites）、数据隔离与并行策略，以及具体的操作序列与断言点，使其成为可追踪的测试蓝图。

4.4.2 步骤 2：API + UI 联动断言设计

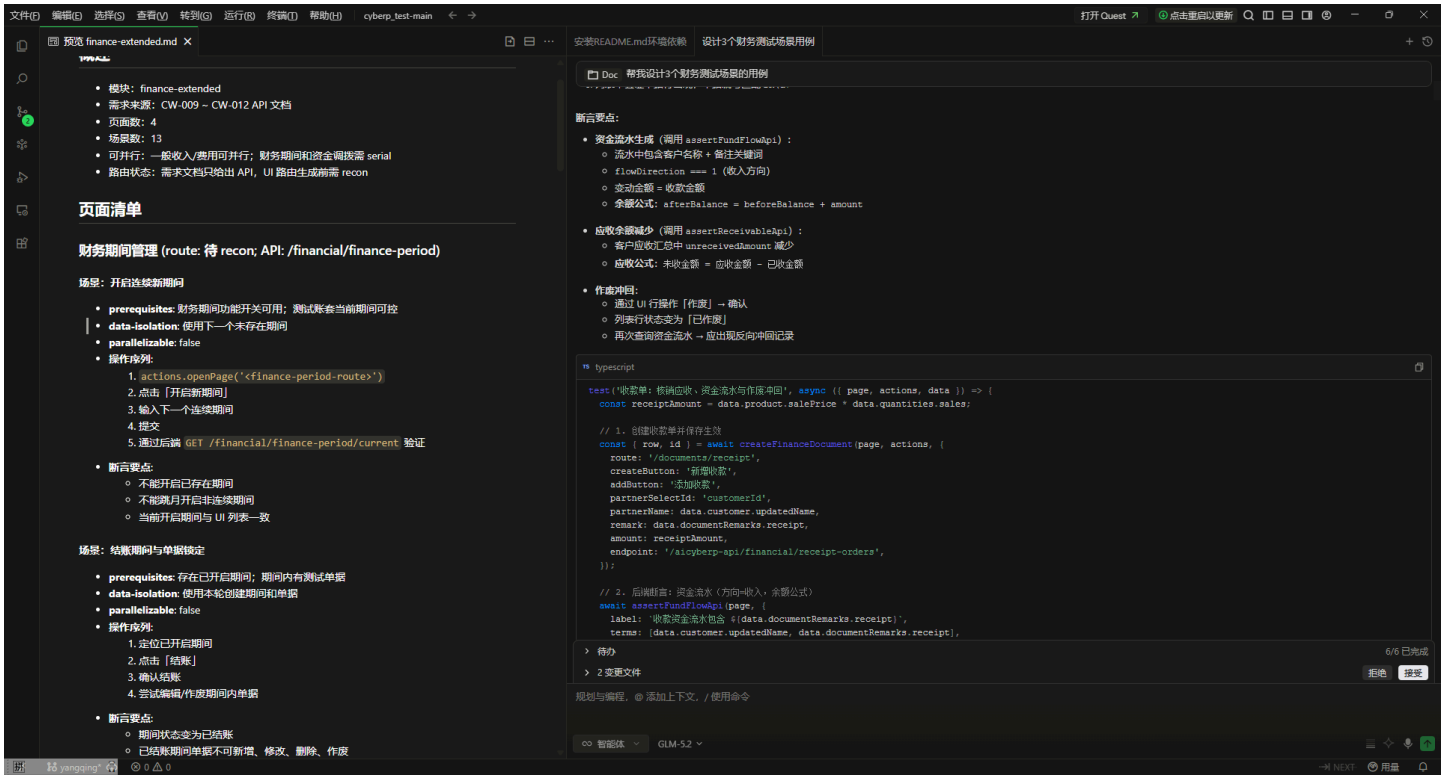
在具体场景（如收款单/付款单）的用例设计中，AI 助手协同设计了比普通 UI 校验更深层的后端网络断言，确保系统的数据完整性：

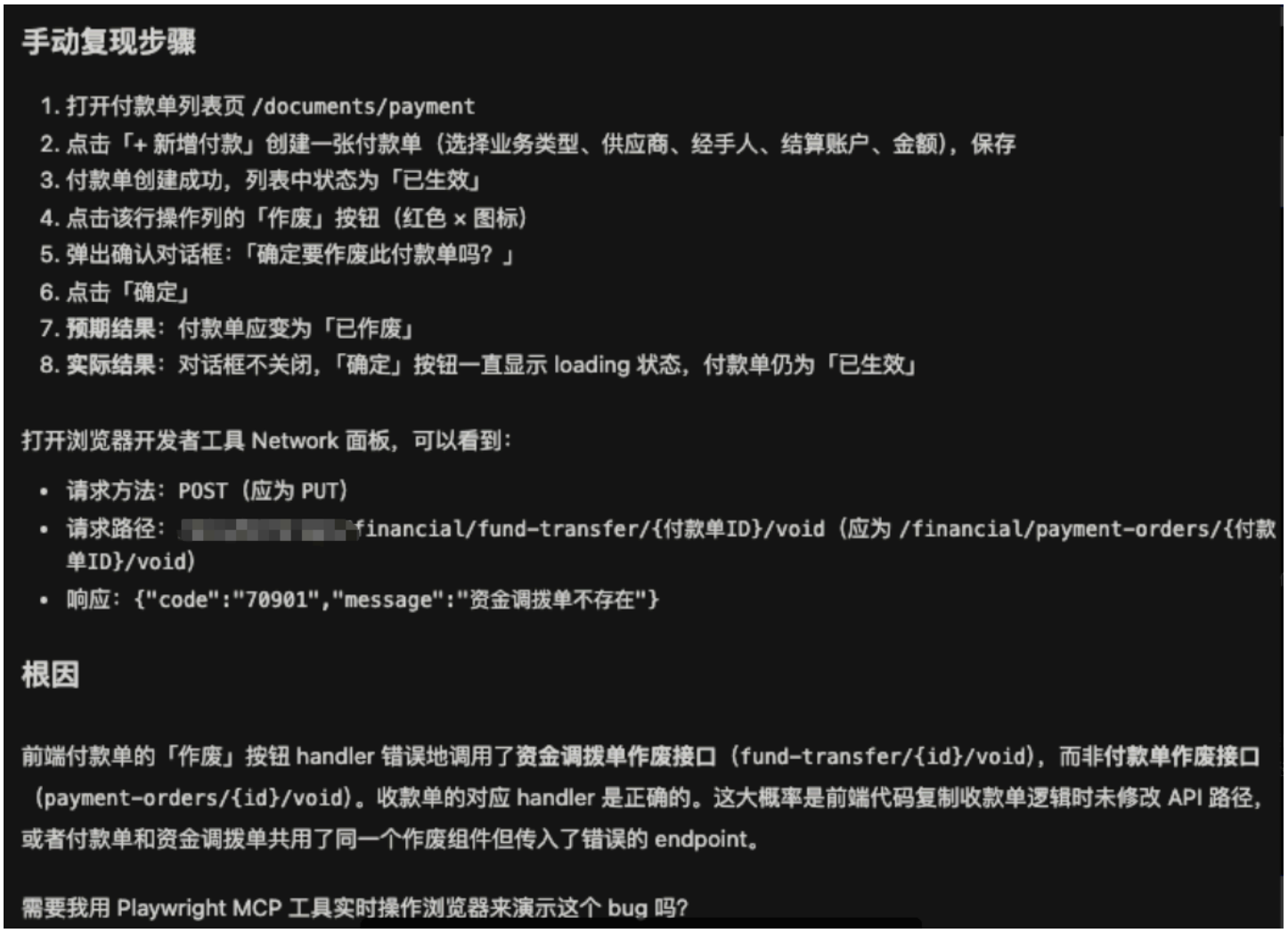
- **资金流水校验**：UI 提交后，调用后端 `assertFundFlowApi` 接口，强校验生成的资金流水日志中是否包含对应的客户名称、正确的资金走向方向（`flowDirection === 1` 表示收入）、准确的收款金额，以及验证期末余额是否符合数学公式（`afterBalance = beforeBalance + amount`）。
- **应收应付账款校验**：调用 `assertReceivableApi` 验证客户应收汇总的未收金额是否按对应收款数额等比例扣减。
- **作废与反向冲回校验**：模拟点击“作废”后，不仅断言 UI 行状态更新为“已作废”，还要再次查询接口验证流水账中已自动生成反向冲回记录。

4.4.3 步骤 3：自动化脚本实现

有了上面的契约和断言规划，AI 助手和开发人员能够直接转化为高内聚的 Playwright TypeScript 测试代码。用例通过封装好的辅助方法（如 `createFinanceDocument`）在页面上完成单据的创建与生效，并直接发起 `assertFundFlowApi` 后端断言，形成了完整的闭环：

图 4.6：财务扩展 Specs 与 AI 协同用例开发步骤对照图





五、AI 协同测试开发规范

随着 AI 编码辅助工具的普及，如何保证 AI 编写的测试脚本不发生劣化（如自动产生长 CSS/XPath 定位、偷偷插入硬超时 `waitForTimeout` 或跳过失败的断言）成为新的挑战。本项目在 `claude/skills/` 目录下定义了一套高水准的 **Agent 协同测试开发契约**，使 AI 编程助手能无缝接入、标准作业并持续产出健壮用例。

> 💡 **Agent 协同测试开发的三大核心原则：**
>
> - **只修脚本，不修业务**：自愈过程仅限于定位器（Locators）和超时时间修正，绝对禁止 AI 修改业务代码或软化断言门禁。
> - **强制遵循定位器优先级契约**：禁止生成脆弱的 CSS/XPath 绝对定位，强制遵守用户视角（Accessibility）的链式选择。
> - **失败类型强制硬归类**：测试用例失败时，必须将错误归纳为 `script`（脚本）、`data`（数据）、`environment`（环境）或 `business-bug`（业务缺陷）四类之一。

5.1 4 Agent Skill 职责定义

在 `claude/skills/` 中定义了如下 4 类 Agent 契约：

Skill 契约名称	执行的工作流与命令	核心准则与防线限制
recon （页面探索）	运行 `npx tsx claude/skills/_shared/scripts/page-recon.ts <url>`	用于探索未知页面的 DOM 结构，并提出用户视角定位器（getByRole 等）的最优选择建议。
generator （代码生成）	将 `specs/{feature}.md` 的业务描述转化为 `.spec.ts` 用例。	禁止生成 `waitForTimeout()`；必须使用 web-first 断言；必须在 project-context 范围内复用已有的 Fixture。
healer （测试自愈）	捕获失败 Trace，修改失败用例并重新运行 3 次直至通过。	**禁止作弊**：禁止使用 `test.skip`、`test.fixme` 或直接删除断言来换取绿色通过。如有业务 Bug，必须上报。
planner （场景规划）	依据 PRD 规划 spec 场景、边界状态与前置资料准备。	规范测试覆盖边界，预防长链条场景中的死锁与冲突。

5.2 契约体系的复用方式

本项目的 Agent Skills 体系不仅可以用于本地，任何复用该框架的测试团队均可直接复制 `claude/skills/` 文件夹：

1. **复用文件夹**：拷贝 `claude/skills/` 至新项目的根目录。
2. **AI 发现与自适应**：任何现代 AI 编程助手（如 Cursor、Claude Code、Gemini-IDE）加载工作区时，会自发读取这些 Skill 契约，并按照规定的 Locator 契约和 Healer 守则执行测试的编写与自愈修改，实现 AI 时代的测试规范化。

六、测试大盘与效果展示

测试运行完毕后，Allure 大盘提供了全方位的可视化数据，包含了通过率趋势、执行日志、失败截图以及 AI 归因诊断等能力。

图 6.1：Allure 测试大盘仪表盘总览（241 个测试用例全景展示）

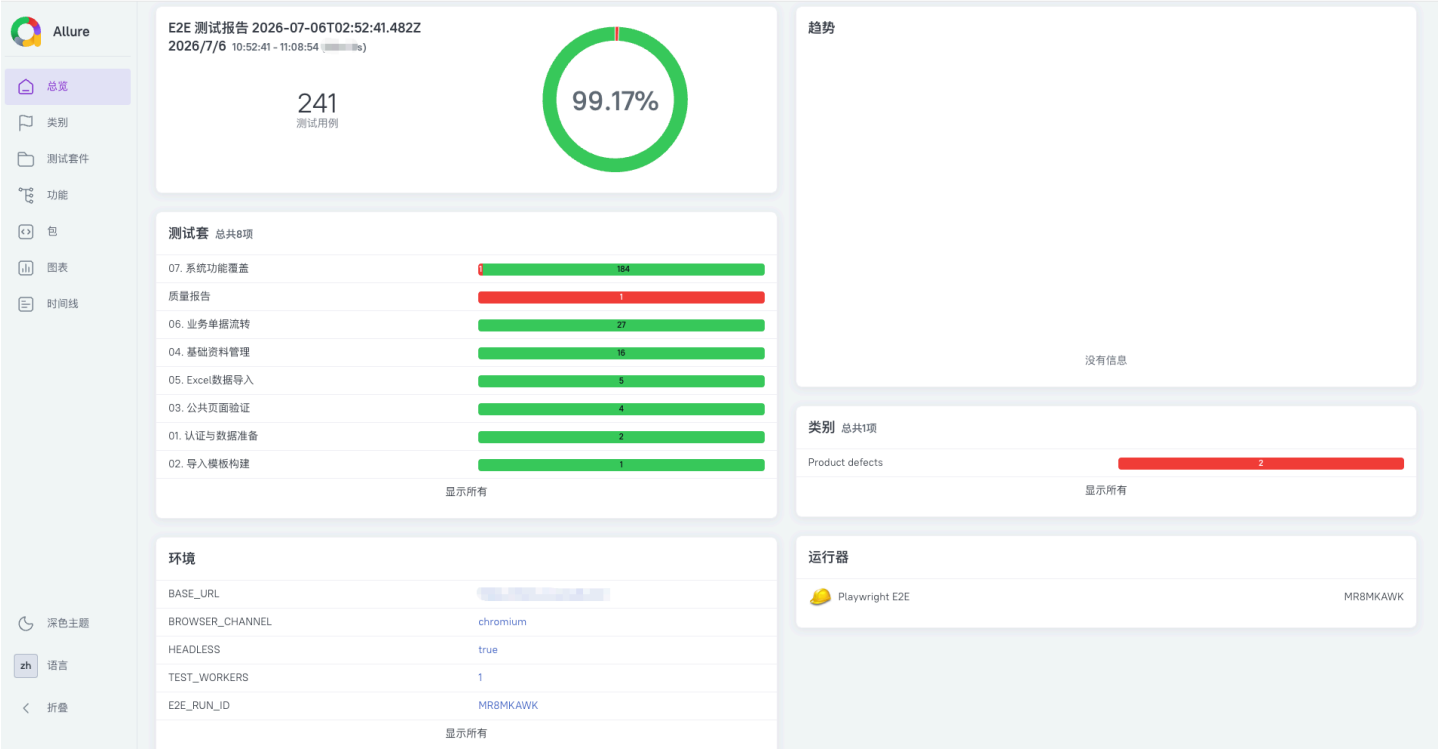
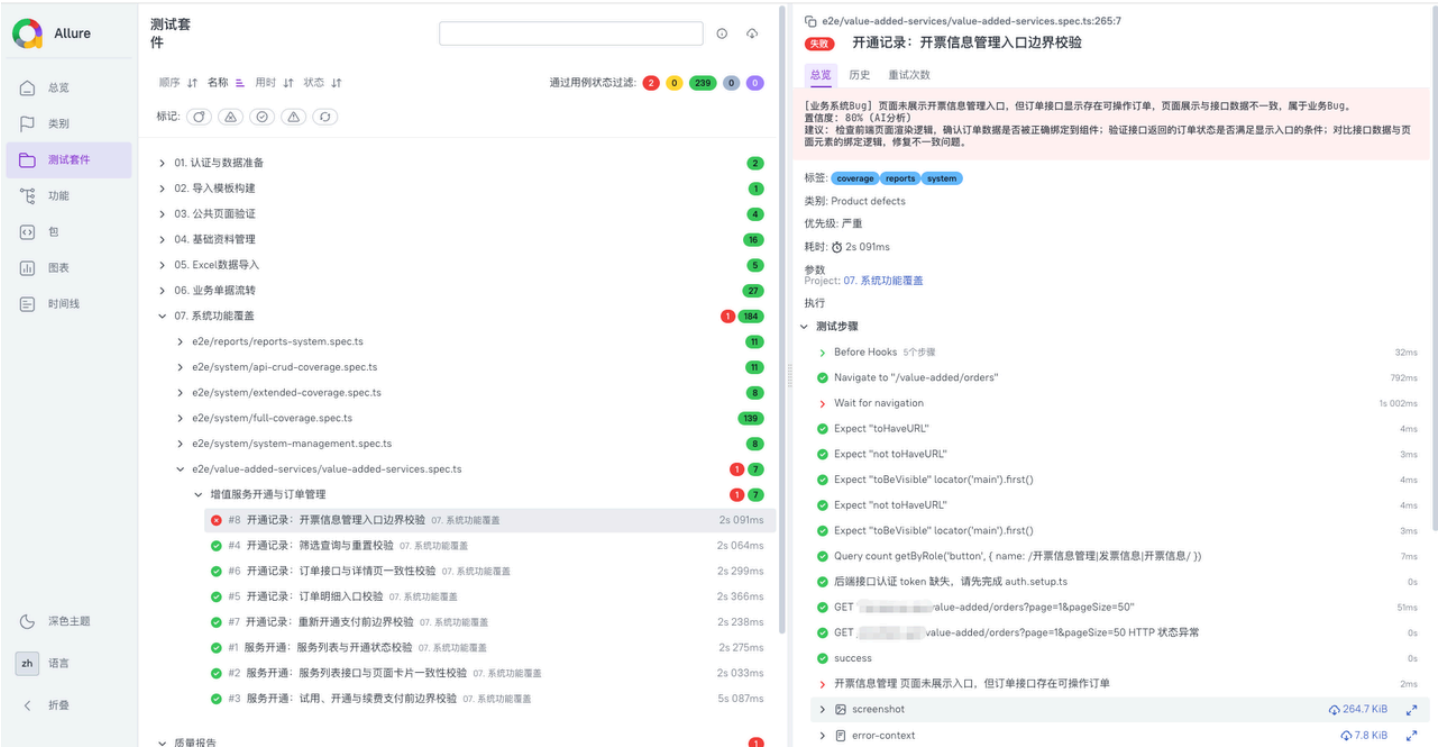


图 6.2：Allure 测试大盘失败用例的 AI 智能分析与原因判定



当用例失败时，系统会自动将步骤日志链和错误堆栈发送给大模型（如 gml5.2），自动判定为 `业务系统Bug`、`测试脚本问题` 或 `环境问题`，并给出修复建议，减少定位时间。

七、常见问题与避坑指南

> 💡 **问题 1：运行测试报错，登录配置缺失：DASHSCOPE_API_KEY**

- **排查根因**：`auth.setup.ts` 执行登录时需要通过通义千问-VL 模型破解图形验证码。如果本地未创建 `.env` 文件，或 `DASHSCOPE_API_KEY` 未配置，就会触发断言中断。
- **解决对策**：确保根目录下有 `.env` 文件，并添加 `DASHSCOPE_API_KEY=你的API_KEY`。

❗ **问题 2：macOS 环境下 Chromium 浏览器启动闪退或报错 MachPortRendezvous**

- **排查根因**：部分 macOS 版本与 Playwright 默认下载的 Chromium 内核存在兼容冲突或沙箱隔离限制。
- **解决对策**：在本地 `.env` 文件中添加 `BROWSER_CHANNEL=chrome`，强制使用本地安装的官方 Chrome 浏览器运行测试。

💡 **问题 3：登录助手频繁报“验证码错误”失败**

- **排查根因**：图形验证码带有干扰线，视觉大模型偶尔会将相似字符看错（如 0 与 o，l 与 1）。
- **解决对策**：框架已内置 3 次自动点击刷新并让 AI 重新识别的机制。如果经常全部失败，请检查百炼服务网络或在 `.env` 中配置 `HEADLESS=false` 启动有头模式进行人工观察。

✅ **问题 4：本地调试卡顿或多用例运行响应过慢**

- **排查根因**：本地并发 Workers 数量过大（多核 CPU 会默认拉起多个浏览器并发执行），或开启了有头模式。
- **解决对策**：本地调试时，建议将 `.env` 中的 `TEST_WORKERS` 强制设定为 `1`，并将 `HEADLESS` 设为 `true`（默认值），以降低本地 CPU 载荷。

💡 **问题 5：测试执行完后，如何直观查看失败原因、录像及截图?**

- **解决对策**：请在测试结束后，于控制台直接运行：

```
代码块

1   npm run allure:serve
```

八、快速应用路线图

其他项目测试开发人员若要快速复用这套“智能免密 + AI 诊断”的测试架构，建议遵循以下路线图在自己的项目中落地：

阶段 1：全局状态共享与加速

重点移植文件与模块：`playwright.config.ts`、`tests/setup/auth.setup.ts`

快速配置与应用技巧：在全局配置中配置 `storageState`，将登录态一次性序列化落盘，其他所有业务测试用例直接免密读取登录，提升 80% 测试执行效率。

阶段 2：UI 操作原子化解耦

重点移植文件与模块：`tests/fixtures/page-actions.fixture.ts`、`tests/fixtures/index.ts`

快速配置与应用技巧：将系统常用的输入框输入、下拉选择、Dialog 确认、Tab 切换等通用交互行为统一封装在 `actions` 下，测试脚本中只描述业务路径，降低 50% 脚本维护工作量。

阶段 3：引入智能报错归因

重点移植文件与模块：`tests/support/analysis/failure-analysis.ts`、`tests/fixtures/index.ts`（`afterEach` 挂载）

快速配置与应用技巧：在项目的 `afterEach` 钩子中，捕获 `testInfo.errors` 并传递给本地的正则匹配规则（快速判断网关/环境故障），若判断为用例失败则调用 DeepSeek 对话接口输出归因附件。

九、用例迭代维护指南

在系统的日常迭代中，前端界面的调整、后端业务逻辑的变更都会导致原有的 E2E 自动化测试用例发生失效或需要扩展。为了确保测试套件的健壮性，本章将以真实的业务场景**“采购退货：关联采购单保存过账校验”**（位于 purchase.spec.ts 第 120-159 行）为例，详细说明如何进行日常的迭代维护与更新。

9.1 用例原型分析

采购退货用例源码分析

```
1  test('采购退货：关联采购单保存过账校验', async ({ page, actions, data }) => {
2    // 1. 前置依赖：通过 UI 创建并过账一张采购单，获取单号
3    const { purchaseOrderNo } = await createPostedPurchaseOrder(
4      page, actions, data, uniqueRemark(data.documentRemarks.purchase,
   data.runId, 'return-source')
```

```
    );
    // 2. 路由跳转至采购退货开单页
    await actions.openPage('/documents/purchase-return/create');
    // 3. 填写单据头，关联前置采购单
    await createDocumentHeader(page, [
      ['供应商', data.supplier.updatedName],
      ['出库仓库', null],
      ['经手人', data.employee.updatedName],
      ['关联采购单', purchaseOrderNo],
    ], data.documentRemarks.purchaseReturn);
    // 4. 添加退货商品与价格
    await addProduct(page, data.product.updatedName,
data.quantities.purchaseReturn, data.product.purchasePrice);
    // 5. 点击“保存过账”按钮保存单据
    const purchaseReturnNo = await saveDocument(page, actions, /保存过账/,
'/documents/purchase-return', data.runId);
    // 6. 联动断言：校验后端 API 接口返回的退货单明细、关联单号及金额
    await assertPurchaseReturnApi(page, purchaseReturnNo, purchaseOrderNo,
data.supplier.updatedName,
      data.quantities.purchaseReturn, data.product.purchasePrice *
data.quantities.purchaseReturn
    );
});
```

9.2 三类典型迭代维护场景

场景 1：前端 UI 变更（按钮文案/DOM 结构变化）

- **变更现象**：前端开发将底部的“保存过账”按钮重命名为“确认过账并存盘”。此时，执行测试会由于找不到包含 `/保存过账/` 正则的按钮而报错超时。
- **维护动作**：
  a. 定位到用例第 147 行。
  b. 将 `saveDocument` 方法中的第三个参数正则定位器修改为 `/确认过账并存盘/`。
  c. 本地运行 `npx playwright test tests/e2e/documents/purchase.spec.ts -g "采购退货"` 验证通过。

场景 2：业务逻辑变更（增加表单必填字段）

- **变更现象**：业务需求调整，要求采购退货单必须选择“退货原因”才能保存，否则保存时前端会弹出“请选择退货原因”的校验错误，导致测试在保存步骤挂起。
- **维护动作**：
  a. 打开测试计划说明书，在对应的 Specs 场景描述中补充该字段需求。

b. 修改脚本中的 `createDocumentHeader` 列表参数，在数组中追加一项 `['退货原因', '质量不合格']`，例如：

```
代码块
1  await createDocumentHeader(page, [
2    ['供应商', data.supplier.updatedName],
3    ['关联采购单', purchaseOrderNo],
4    ['退货原因', '质量不合格'], // 新增必填字段
5  ], data.documentRemarks.purchaseReturn);
```

c. 重新运行测试，确认单据头顺利填充并保存成功。

场景 3：接口级断言微调（后端 API 字段调整）

- **变更现象**：后端重构，退货单明细接口中返回的总金额字段名由 `totalAmount` 变更为 `settlementAmount`。此时，自定义的联动断言方法 `assertPurchaseReturnApi` 在提取响应字段时会读取到 `undefined` 导致断言失败。
- **维护动作**：
  a. 无需修改具体的业务 Spec 用例脚本，直接定位到项目底层公共断言实现文件 `tests/support/assertions/purchase-assertions.ts`。
  b. 修改 `assertPurchaseReturnApi` 中提取和验证金额的 JSON 路径，将 `payload.totalAmount` 修正为 `payload.settlementAmount`。
  c. 此举保证了底层断言库自适应接口重构，同时所有调用该断言的测试脚本无需做任何重复性修改。

十、协作与开发门禁规范

- **测试数据动态隔离**：在迁移复用时，要求所有的测试用例对新增数据（如新建商品、单据、往来单位）都带有 `data.runId` 指纹（或当前毫秒时间戳），从根本上隔离并发 Workers 导致的数据互锁和重复校验报错。
- **断言前必须加可见性验证**：在执行任何点击、输入等操作后，下一步必须加显式断言检查（如 `await expect(locator).toBeVisible()`），使代码库不含任何 `page.waitForTimeout()` 硬等待，确保执行效率最优化。
- **双人 Code Review 门禁**：为了防止框架腐化，建议将 E2E 脚本纳入开发工作流，每次 UI 变动需要同步更新 Actions 库并审核通过后，方可合并代码。