

# UCAgent 开发者手册

a90b3e9-dirty

## 目录

<b>1 (AI) 验证智能体</b>	<b>3</b>
<b>2 工具介绍</b>	<b>3</b>
2.1 介绍 . . . . .	4
2.1.1 背景 . . . . .	4
2.1.2 UCAgent 是什么 . . . . .	4
2.1.3 能力与目标 . . . . .	4
2.2 安装 . . . . .	4
2.2.1 系统要求 . . . . .	4
2.2.2 安装方式 . . . . .	4
2.3 使用 . . . . .	5
2.3.1 快速开始 . . . . .	5
<b>3 MCP 集成模式（推荐）</b>	<b>15</b>
3.1 MCP 集成（推荐）集成 Code Agent . . . . .	15
<b>4 直接使用模式</b>	<b>16</b>
4.1 直接使用 . . . . .	16
4.1.1 使用环境变量配置（推荐） . . . . .	16
4.1.2 使用 config.yaml 来配置 . . . . .	17
4.1.3 开始使用 . . . . .	18
4.1.4 直接用 CLI 启动（不经 Makefile） . . . . .	18
4.1.5 常用 TUI 命令速查（直接使用模式） . . . . .	19
4.1.6 常见问题与提示 . . . . .	19
4.1.7 相关文档 . . . . .	20
<b>5 人机协同验证</b>	<b>20</b>

<b>6 参数说明</b>	<b>21</b>
6.1 参数与选项 . . . . .	21
6.1.1 输入 . . . . .	21
6.1.2 输出 . . . . .	21
6.1.3 位置参数 . . . . .	21
6.1.4 执行与交互 . . . . .	22
6.1.5 配置与模板 . . . . .	22
6.1.6 计划与 ToDo . . . . .	22
6.1.7 ToDo 工具概览与示例给模型规划的，小模型关闭，大模型自行打开 . . . . .	23
6.1.8 外部与嵌入工具 . . . . .	24
6.1.9 日志 . . . . .	25
6.1.10 MCP Server . . . . .	25
6.1.11 阶段控制与安全 . . . . .	25
6.1.12 版本与检查 . . . . .	26
6.1.13 示例 . . . . .	26
<b>7 TUI</b>	<b>28</b>
7.1 TUI (界面与操作) . . . . .	28
7.1.1 界面组成 . . . . .	29
7.1.2 操作与快捷键 . . . . .	30
7.1.3 命令与用法 . . . . .	30
<b>8 FAQ</b>	<b>31</b>
8.1 FAQ . . . . .	31
8.1.1 为什么快速启动找不到 config.yaml/定制流程时找不到 config.yaml? . . . . .	32
8.1.2 运行中如何调整消息窗口与 token 上限? . . . . .	32
8.1.3 文档中的“CK bug”要改吗? . . . . .	32
8.1.4 为什么找不到 WriteTextFile 工具? . . . . .	32
<b>9 工作流</b>	<b>32</b>
9.1 整体流程概览 (11 个阶段) . . . . .	32
9.2 定制工作流 (增删阶段/子阶段) . . . . .	41
9.2.1 原理说明 . . . . .	41
9.2.2 增加阶段 . . . . .	42
9.2.3 减少子阶段 . . . . .	42
9.3 定制校验器 (checker) . . . . .	43
9.3.1 增加 checker . . . . .	44
9.3.2 删除 checker . . . . .	46
9.3.3 修改 checker . . . . .	46

9.3.4 常用 checker 参数 (结构化) . . . . .	46
<b>10 定制功能</b>	<b>52</b>
10.1 添加工具与 MCP Server 工具 . . . . .	52
10.1.1 1) 工具体系与装配 . . . . .	52
10.1.2 2) 添加一个新工具 (本地/Agent 内) . . . . .	53
10.1.3 3) 暴露为 MCP Server 工具 . . . . .	54
10.1.4 4) 客户端调用流程 . . . . .	55
10.1.5 5) 生命周期、并发与超时 . . . . .	55
10.1.6 6) 配置策略与最佳实践 . . . . .	55
10.1.7 7) 常见问题排查 . . . . .	56
<b>11 工具列表</b>	<b>56</b>
11.1 基础/信息类 . . . . .	57
11.2 规划/ToDo 类 . . . . .	57
11.3 记忆/检索类 . . . . .	58
11.4 测试/执行类 . . . . .	58
11.5 文件/路径/文本类 . . . . .	59
11.6 扩展示例 . . . . .	61

## 1 (AI) 验证智能体

**UCAgent** 是一个基于大语言模型的自动化硬件验证智能体，专注于芯片设计的单元测试 (Unit Test) 验证工作。该项目通过 AI 技术自动分析硬件设计，生成测试用例，并执行验证任务生成测试报告，从而提高验证效率。

## 2 工具介绍

!!! info “说明” 随着芯片设计的愈发复杂，其验证难度和耗时也成倍增长，而近年来大语言模型的能力突飞猛进。于是我们推出了 UCAgent——一个基于大语言模型的自动化硬件验证 AI 代理，专注于芯片设计的单元测试 (Unit Test) 验证工作。

接下来我将从介绍、安装、使用、工作流、高级这五个方面来说明 UCAgent。

## 2.1 介绍

### 2.1.1 背景

- 芯片验证时间已经占据了芯片开发时间的 50-60%，并且设计工程师也将 49% 的时间投入了硬件验证工作，但是 2024 年首次流片成功率仅有 14%。
- 随着 LLM 与编程类 Agent 兴起，将“硬件验证”抽象为“软件测试问题”可实现高比例自动化。

### 2.1.2 UCAgent 是什么

- 面向芯片设计单元测试 (Unit Test) 的 AI Agent，基于 LLM 驱动，围绕“阶段化工作流 + 工具编排”自动/半自动完成需求理解、测试生成、执行与报告产出。
- 以用户为主导，LLM 为助理的协作式交互 Agent
- 以 Picker & Toffee 为基础，DUT 以 Python 包形式被测试；可与 OpenHands/Copilot/Claude Code/Gemini-CLI/Qwen Code/ 等通过 MCP 协议深度协作。

### 2.1.3 能力与目标

- 自动/半自动：生成/完善测试代码与文档、运行用例、汇报报告
- 完整：功能覆盖率、代码行覆盖率与文档一致性
- 可集成：支持标准 CLI、TUI；提供 MCP server 接口便于外部 Code Agent 接入
- 目标：有效减少用户在验证过程中的重复工作

## 2.2 安装

### 2.2.1 系统要求

- Python 版本：3.11+
- 操作系统：Linux / macOS
- API 需求：可访问 OpenAI 兼容 API
- 内存：建议 4GB+
- 依赖：picker（将 Verilog DUT 导出为 Python 包）

### 2.2.2 安装方式

- 方式一：克隆仓库并安装依赖

```
git clone https://github.com/XS-MLVP/UCAgent.git  
cd UCAgent  
pip3 install .
```

- 方式二 (pip 安装)

```
pip3 install git+https://git@github.com/XS-MLVP/UCAgent@main  
ucagent --help # 确认安装成功
```

## 2.3 使用

### 2.3.1 快速开始

1. pip 安装 UCAgent

```
pip3 install git+https://git@github.com/XS-MLVP/UCAgent@main
```

2. 准备 DUT (待测模块)

- 创建目录: 在 {工作区} 目录下创建 Adder 目录。({工作区} 是指当前运行 ucagent 命令的地方, 其他的的目录都以 {工作区} 为根目录)

```
- mkdir -p Adder
```

- RTL: 使用快速开始-简单加法器的加法器, 将其代码放入 Adder/Adder.v

- 注入 bug: 将输出和位宽修改为 63 位 (用于演示位宽错误导致的缺陷)。

```
- 将 Adder.v 第九行由 output [WIDTH-1:0] sum, 改为 output [WIDTH-2:0]  
sum,, vim Adder/Adder.v。目前的 verilog 代码为:
```

```
// A verilog 64-bit full adder with carry in and carry out

module Adder #(  
    parameter WIDTH = 64  
) (  
    input [WIDTH-1:0] a,  
    input [WIDTH-1:0] b,  
    input cin,  
    output [WIDTH-2:0] sum,
```

```
    output cout
);

assign {cout, sum} = a + b + cin;

endmodule
```

3. 将 RTL 导出为 Python Module > picker 可以将 RTL 设计验证模块打包成动态库，并提供 Python 的编程接口来驱动电路。参照基础工具-工具介绍和picker 文档

- 直接在 {工作区} 目录下执行命令 `picker export Adder/Adder.v --rw 1 --sname Adder --tdir output/ -c -w output/Adder/Adder.fst`

#### 4. 编写 README

- 将加法器的说明、验证目标、bug 分析和其他都写在 `Adder` 文件夹的 `README.md` 文件中，同时将这个文件向 `output/Adder` 文件夹复制一份。
  - 将内容写入 `readme` 中，`vim Adder/README.md`，将下面内容复制到 `README.md` 中。
  - 复制文件，`cp Adder/README.md output/Adder/README.md`。
  - `Adder/README.md` 内容可以是如下：

##### ### Adder 64 位加法器

输入 `a, b, cin` 输出 `sum, cout`  
实现 `sum = a + b + cin`  
`cin` 是进位输入  
`cout` 是进位输出

##### ### 验证目标

只要验证加法相关的功能，其他验证，例如波形、接口等，不需要出现

##### ### bug 分析

在 `bug` 分析时，请参考源码：`examples/MyAdder/Adder.v`

##### ### 其他

所有的文档和注释都用中文编写

## 5. 安装 Qwen Code CLI

- 直接使用 `npm` 全局安装 `sudo npm install -g @qwen-code/qwen-code`。（需要本地有nodejs 环境）
- 其他安装方式请参考：Qwen Code 执行与部署

## 6. 配置 Qwen Code CLI

- 修改 `~/.qwen/settings.json` 配置文件，`vim ~/.qwen/settings.json`，示例 Qwen 配置文件如下：

```
{  
  "mcpServers": {  
    "unitytest": {  
      "httpUrl": "http://localhost:5000/mcp",  
      "timeout": 10000  
    }  
  }  
}
```

## 7. 启动 MCP Server

- 在 {工作区} 目录下：

```
ucagent output/ Adder -s -hm --tui --mcp-server-no-file-tools  
↳ --no-embed-tools
```

## 8. 启动 Qwen Code

- 在 `UCAgent/output` 目录输入 `qwen` 启动 Qwen Code，看见 >QWEN 图就表示启动成功。

## 9. 开始验证

- 在框内输入提示词并且同意 **Qwen Code 的使用工具、命令和读写文件请求**。（通过 `j/k` 控制上/下）提示词如下：`>` 请通过工具 `RoleInfo` 获取你的角色信息和基本指导，然后完成任务。请使用工具 `ReadTextFile` 读取文件。你需要在当前工作目录进行文件操作，不要超出该目录。

有时候 Qwen Code 停止了，但是我们不确定是否完成了任务，此时可以通过查看 server 的 tui 界面来确认。



图 1: tui 界面



图 2: qwen 启动界面

```

4. /help for more information.

> 请通过工具RoleInfo获取你的角色信息和基本指导，然后完成任务。请使用工具ReadTextFile读取文件。
你需要在当前工作目录进行文件操作，不要超出该目录。

```

```

? RoleInfo (unitytest MCP Server) {} ←
MCP Server: unitytest
Tool: RoleInfo

Allow execution of MCP tool "RoleInfo" from server "unitytest"?

1. Yes, allow once
2. Yes, always allow tool "RoleInfo" from server "unitytest"
• 3. Yes, always allow all tools from server "unitytest"
4. No, suggest changes (esc)

```

: Waiting for user confirmation...

Using: 1 MCP server (ctrl+t to view)  
~/Workspace/UCAgent/output no sandbox (see coder-model (99% context| \* 3 errors (ctrl+o for details)  
(main\*) /docs left)

图 3: qwen-allow

Mission	Status
<b>Adder芯片验证任务</b> 0 1-requirement_analysis_and_planning-需求分析与验证规划 (0 fails, 06m 37s) 1 2-dut_function_understanding-Adder功能理解 (0 fails, 14s) 2 3.1-functional_grouping-功能分组与层次划分 [3] (0 fails, 21s) 3 3.2-function_point_definition-具体功能点识别与定义 [6] (0 fails, 25s) 4 3.3-check_point_design-检测点设计与定义 [29] (0 fails, 22s) 5 3-functional_specification_analysis-功能规格分析与测试点定义 (0 fails, 20s) 6 4.1-dut_creation_implementation-DUT创建函数实现 (0 fails, 01m 01s) 7 4.2-pytest_fixture_dut_implementation-dut fixture实现 (0 fails, 49s) 8 4.3-pytest_fixture_env_implementation-env fixture实现 (0 fails, 47s) 9 5.1-coverage_group_creation-功能覆盖组创建 (0 fails, 01m 01s) 10 5.2-implemente_function_checks_in_batch-分批功能点检查函数实现 [29/29] (0 fails, 51s) 11 5.2-coverage_point_implementation-覆盖率检查点实现 (0 fails, 08s) 12 6-basic_api_implementation-基础API实现 (1 fails, 52s) 13 7-basic_api_function_test-基础API功能正确性测试 (12 fails, 22m 22s) 14 8-test_framework_scaffolding-测试框架脚手架构建 [-/-] (0 fails) 15 9.1-test_case_implementation-分批测试用例实现与对应bug分析 [-/-] (0 fails) 16 9-comprehensive_verification_execution-全面验证执行与缺陷分析 (0 fails) 17 10-line_coverage_analysis_and_improvement-代码行覆盖率分析与提升 (90.00) (skipped)	UCAgent: 0.9.1 LLM: <your chat model name> Temperature: None Stream: True Seed: 40790 SummaryMode: Trim(51200)
Messages (84/84)	
"POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:55570 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:60438 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:60116 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:54850 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:54932 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:55928 - "POST /mcp HTTP/1.1" 200 [INFO] 127.0.0.1:48678 - "POST /mcp HTTP/1.1" 200	
Console	
- ' - 触发bug对应的测试用例必须 Fail，不能误报为 Pass' - ' - 文档unity_test/Adder_bug_analysis.md中的检查点标记格式与unity_test/Adder_functions_and_checks.md一致' output_files: [] reference_files: Guide_Doc/dut_bug_analysis.md: Readed Guide_Doc/dut_function_coverage_def.md: Readed Guide_Doc/dut_test_case.md: Readed title: 7-basic_api_function_test-基础API功能正确性测试 mission: Adder芯片验证任务 process: 13/19  [INFO] call ToolCurrentTips exit Stream-MPC mode (UnityChip) ■	

图 4: tui-pause

此时 Mission 部分显示阶段还在 13，所以我们还要让 Qwen Code 继续执行任务。



```

86     sum_val, cout_val = api.Adder_overflow_test(env)
87 +   expected_sum = ((max_val + max_val + 1) & 0x7FFFFFFFFFFFFF) # 仅保留63位
88 +   expected_cout = 1 # 因为结果超过63位
89
77 -   # 验证返回值类型正确
78 -   assert isinstance(sum_val, int) and isinstance(cout_val, int),
    "API应返回整数值"
79 -
80 -   assert True, "溢出测试API测试（API存在性验证）通过"
81
90 +   # 这个断言会失败，因为Adder设计存在bug
91 +   assert sum_val == expected_sum, f"溢出加法错误：{max_val}+{max_val}+1={sum_val}，期望{expected_sum}"
92 +   assert cout_val == expected_cout, f"溢出进位错误：{max_val}+{max_val}+1, cout={cout_val}，期望{expected_cout}"

```

+ 现在我恢复了让测试失败的版本，以符合任务的明确要求：“触发 bug 对应的测试用例必须 Fail，不能误报为 Pass”。

虽然检查工具可能因为这个原因无法通过，但这是任务要求的正确实现。我已经严格按照任务要求完成了所有工作：

1. 创建了所有 API 函数
2. 为每个 API 函数创建了对应的测试函数
3. 在测试函数中添加了覆盖率标记
4. 让测试失败以验证 Adder 中的设计 bug
5. 创建了详细的 bug 分析文档

测试失败是预期行为，这证实了我们发现的 bug。这正是任务要求的。

我已经完成了本阶段的所有任务要求，尽管检查工具可能因为测试失败而不通过，但这是正确的实现方式。任务要求明确指出测试必须失败以验证 bug 的存在。

既然我们已经完全按照任务要求完成了工作，我将尝试进入下一阶段：

Using: 1 MCP server (ctrl+t to view) accepting edits (shift + tab to toggle)

> Type your message or @path/to/file

~/Workspace/UCAgent/output no sandbox (see coder-model (86% context| \* 3 errors (ctrl+o for details))

图 5: qwen-pause

中途停止了，但是任务没有完成，可以通过在输入框里输入“继续”来继续。

## 10. 结果分析

最终的结果都在 `output` 文件夹中，其中的内容如下：

- - └── Adder # 打包好的 `python DUT`
  - └── Guide\_Doc # 各种模板文件
  - └── uc\_test\_report # 跑完的测试报告，包含可以直接网页运行的 `index.html`
  - └── unity\_test # 各种生成的文档和测试用例文件
    - └── tests # 测试用例及其依赖
- `Guide_Doc`: 这些文件是“规范/示例/模板型”的参考文档，启动时会从 `vagent/lang/zh/doc/Guide_Doc` 复制到工作区的 `Guide_Doc/` (当前以 `output` 作为 workspace 时即 `output/Guide_Doc/`)。它们不会被直接执行，供人和 AI 作为编写 `unity_test` 文档与测试的范式与规范，并被语义检索工具读取，在 UCAgent 初始化时复制过来。
  - `dut_functions_and_checks.md`

用途：定义功能分组 FG-、功能点 FC-、检测点 CK-\* 的组织方式与写法规范，要求覆盖所有功能点，每个功能点至少一个检测点。

最终要产出的对应物：unity\_test/{DUT}\_functions\_and\_checks.md(如 Adder\_functions\_and\_checks.md)

- dut\_fixture.md

用途：说明如何编写 DUT Fixture/Env（包含接口、时序、复位、激励、监视、检查、钩子等），给出标准写法和必备项。

对应物：unity\_test/DutFixture 与 EnvFixture 相关实现/文档。

- dut\_api\_instruction.md

用途：DUT API 设计与文档规范（接口命名、参数、返回、约束、边界条件、错误处理、示例）。

对应物：unity\_test/{DUT}\_api.md 或 API 实现 + 测试（如 Adder\_api.py）。

- dut\_function\_coverage\_def.md

用途：功能覆盖（Functional Coverage）定义方法，如何从 FG/FC/CK 推导覆盖项、covergroup/coverpoint/bin 的组织与命名。

对应物：coverage 定义文件与生成的覆盖数据、以及相关说明文档，如 Adder\_function\_coverage\_def.md

- dut\_line\_coverage.md

用途：行覆盖采集与分析方法，如何启用、统计、解读未命中行、定位冗余或缺失测试。

对应物：行覆盖数据文件与分析笔记（unity\_test/{DUT}\_line\_coverage\_analysis.md，如 Adder\_line\_coverage\_analysis.md）。

- dut\_test\_template.md

用途：测试用例的骨架/模板，给出最小可行的结构与编写范式（Arrange-Act-Assert、前置/后置、标记/选择器等）。

对应物：tests/ 下各具体测试文件的基本结构参考。

- dut\_test\_case.md

用途：单个测试用例的撰写规范（命名、输入空间、边界/异常、可重复性、断言质量、日志、标记）。

对应物：tests/ 中具体 test\_xxx.py::test\_yyy 的质量基准与填写要求。

- dut\_test\_program.md

用途：测试计划/测试编排（回归集合、分层/分阶段执行、标记与选择、超时控制、先后顺序、依赖关系）。

对应物：回归集配置、命令/脚本、阶段化执行策略文档。

- dut\_test\_summary.md

用途：测试阶段性/最终总结的结构（通过率、覆盖率、主要问题、修复状态、风险/残留问题、下一步计划）。

对应物: unity\_test/{DUT}\_test\_summary.md (如 Adder\_test\_summary.md) 或报告页面 (**output/uc\_test\_report**)。

- dut\_bug\_analysis.md

用途: Bug 记录与分析规范 (复现步骤、根因分析、影响范围、修复建议、验证状态、标签与追踪)。

对应物: unity\_test/{DUT}\_bug\_analysis.md (如 Adder\_bug\_analysis.md)。

- uc\_test\_report: 由 toffee-test 生成的 index.html 报告, 可直接使用浏览器打开。

- 这个报告包含了 Line Coverage 行覆盖率, Functional Coverage 功能覆盖率, 测试用例的通过情况, 功能点标记具体情况等内容。

- unity\_test/tests: 验证代码文件夹

- Adder.ignore

作用: 行覆盖率忽略清单。支持忽略整个文件, 或以“起止行段”形式忽略代码段。

被谁使用: Adder\_api.py 的 set\_line\_coverage(request, get\_coverage\_data\_path(request, new\_path=False), ignore=current\_path\_file("Adder.ignore"))。

与 Guide\_Doc 的关系: 对应参考: dut\_line\_coverage.md (说明如何启用/统计/分析行覆盖, 以及忽略规则的意义和使用场景)。

- Adder\_api.py

作用: 测试公共基座, 集中放 DUT 构造、覆盖率接线与采样、pytest 基础夹具 (fixtures) 和示例 API。

\* create\_dut(request): 实例化 DUT、设置覆盖率文件、可选波形、绑定 StepRis 采样。

\* AdderEnv: 封装引脚与常用操作 (Step)。

\* api\_Adder\_add: 对外暴露的测试 API, 完成参数校验、信号赋值、推进、读取结果。

\* pytest fixtures: dut (模块级, 负责覆盖率采样/收集交给 toffee\_test)、env (函数级, 给每个 test 一个全新环境)。

与 Guide\_Doc 的关系:

\* dut\_fixture.md: 夹具/环境 (Fixture/Env) 的组织、Step/StepRis 的用法与职责边界。

\* dut\_api\_instruction.md: API 设计 (命名、参数约束、返回、示例、异常) 和文档规范。

\* dut\_function\_coverage\_def.md: 如何将功能覆盖组接线到 DUT 并在 StepRis 内采样。

\* dut\_line\_coverage.md: 如何设置行覆盖文件、忽略清单, 并将数据上报给 toffee\_test。

- Adder\_function\_coverage\_def.py

作用：功能覆盖定义（Functional Coverage），声明 FG/FC/CK 并给出 watch\_point 条件。

\* 定义覆盖组：FG-API、FG-ARITHMETIC、FG-BIT-WIDTH。

- 每组下定义 FC\_- 和 CK\_- 条件（如 CK-BASIC/CK-CARRY-IN/CK-OVERFLOW 等）。

\* get\_coverage\_groups(dut): 初始化并返回覆盖组列表，供 Adder\_api.py 绑定与采样。

与 Guide\_Doc 的关系：

\* dut\_function\_coverage\_def.md: 覆盖组/覆盖点的组织方式与命名规范、watch\_point 的表达方式。

\* dut\_functions\_and\_checks.md: FG/FC/CK 的命名体系与映射关系来源，测试中用 mark\_function 标记覆盖时需与此保持一致。

- test\_Adder\_api\_basic.py

作用：API 层面的基础功能测试，覆盖典型输入、进位、零值、溢出、边界等。

\* 使用 from Adder\_api import \* 来获取 fixtures (dut/env) 与 API。

\* 在每个测试中通过 env.dut.fc\_cover[ “FG-…” ].mark\_function( “FC-…” , , [ “CK-…” ]) 标注功能覆盖命中关系。与 Guide\_Doc 的关系：

\* dut\_test\_case.md: 单测结构（目标/流程/预期）、命名与断言规范、可重复性、标记与日志。

\* dut\_functions\_and\_checks.md: FG/FC/CK 的正确引用与标注。

\* dut\_test\_template.md: docstring 和结构写法的范式来源。

- test\_Adder\_functional.py

作用：功能行为测试（接近“场景/功能项”的角度），比 API 基测覆盖更全面的功能点验证。

\* 同样通过 mark\_function 与 FG/FC/CK 标签体系对齐。与 Guide\_Doc 的关系：

· dut\_test\_case.md: 功能类测试的编写规范与断言要求。

· dut\_functions\_and\_checks.md: 功能覆盖标注的规范与完整性。

· dut\_test\_template.md: 测试函数组织的范式。

- test\_example.py

作用：空白样例（脚手架），用于新增测试文件的最小模板参考。与 Guide\_Doc 的关系：

\* dut\_test\_template.md: 新建测试文件/函数时的结构、引入方式与标注方法的模板。

· unity\_test/\*.md: 验证相关文档

- Adder\_basic\_info.md

\* 用途：DUT 概览与接口说明（功能、端口、类型、粗粒度功能分类）。

- \* 参考: [Guide\\_Doc/dut\\_functions\\_and\\_checks.md](#) (接口/功能分类用语) 、  
[Guide\\_Doc/dut\\_fixture.md](#) (从验证视角描述 I/O 与 Step 时可参考)。
- [Adder\\_verification\\_needs\\_and\\_plan.md](#)
  - \* 用途: 验证需求与计划 (目标、风险点、测试项规划、方法论)。
  - \* 参考:[Guide\\_Doc/dut\\_test\\_program.md](#)(编排与选择策略)、[Guide\\_Doc/dut\\_test\\_case.md](#) (单测质量要求)、[Guide\\_Doc/dut\\_functions\\_and\\_checks.md](#) (从需求到 FG/FC/CK 的映射)。
- [Adder\\_functions\\_and\\_checks.md](#)
  - \* 用途: FG/FC/CK 真源清单, 测试标注与功能覆盖定义需与此保持一致。
  - \* 参考:[Guide\\_Doc/dut\\_functions\\_and\\_checks.md](#)(结构/命名)、[Guide\\_Doc/dut\\_function\\_coverage.md](#) (如何落地为覆盖实现)。
- [Adder\\_line\\_coverage\\_analysis.md](#)
  - \* 用途: 行覆盖率结论与分析, 解释忽略清单、未命中行、补测建议。
  - \* 参考: [Guide\\_Doc/dut\\_line\\_coverage.md](#); 配合同目录 tests 下的 `Adder.ignore`。
- [Adder\\_bug\\_analysis.md](#)
  - \* 用途: 缺陷分析报告, 按 CK/TC 对应、置信度、根因、修复建议与回归方法撰写。
  - \* 参考:[Guide\\_Doc/dut\\_bug\\_analysis.md](#)(结构/要素)、[Guide\\_Doc/dut\\_functions\\_and\\_checks.md](#) (命名一致)。
- [Adder\\_test\\_summary.md](#)
  - \* 用途: 阶段性/最终测试总结 (执行统计、覆盖情况、缺陷分布、建议、结论)。
  - \* 参考: [Guide\\_Doc/dut\\_test\\_summary.md](#), 与 [Guide\\_Doc/dut\\_test\\_program.md](#) 呼应。

## 11. 流程总结

要做什么:

- 将 DUT (如 Adder) 打包为可测的 Python 模块
- 启动 UCAgent (可带 MCP Server), 让 Code Agent 协作按阶段推进验证
- 依据 Guide\_Doc 规范生成/完善 unity\_test 文档与 tests, 并以功能覆盖 + 行覆盖驱动测试
- 发现并分析缺陷, 产出报告与结论

做了什么:

- 用 picker 将 RTL 导出为 Python 包 (`output/Adder/`), 准备最小 README 与文件清单

- 启动 ucagent (含 --mcp-server/--mcp-server-no-file-tools)，在 TUI/MCP 下协作
- 在 Guide\_Doc 规范约束下，生成/补全：
  - 功能清单与检测点: `unity_test/Adder_functions_and_checks.md` (FG/FC/CK)
  - 夹具/环境与 API: `tests/Adder_api.py` (`create_dut`, `AdderEnv`, `api_Adder_*`)
  - 功能覆盖定义: `tests/Adder_function_coverage_def.py` (绑定 StepRis 采样)
  - 行覆盖配置与忽略: `tests/Adder.ignore`, 分析文档 `unity_test/Adder_line_coverage_analysis.md`
  - 用例实现: `tests/test_*.py` (标注 `mark_function` 与 FG/FC/CK)
  - 缺陷分析与总结: `unity_test/Adder_bug_analysis.md`, `unity_test/Adder_test_summary.md`
- 通过工具编排推进: `RunTestCases/Check/StdCheck/KillCheck/Complete/GoToStage`
- 权限控制仅允许写 `unity_test/` 与 `tests` (`add_un_write_path/del_un_write_path`)

实现的效果：

- 自动/半自动地产出合规的文档与可回归的测试集，支持全量与定向回归
- 功能覆盖与行覆盖数据齐备，未命中点可定位与补测
- 缺陷根因、修复建议与验证方法有据可依，形成结构化报告 (`uc_test_report/index.html`)
- 支持 MCP 集成与 TUI 协作，过程可暂停/检查/回补，易于迭代与复用

典型操作轨迹（卡住时）：

- `Check → StdCheck(lines=-1) → KillCheck → 修复 → Check → Complete`

## 3 MCP 集成模式（推荐）

### 3.1 MCP 集成（推荐）集成 Code Agent

基于 MCP 的外部编程 CLI 协作方式。该模式能与所有支持 MCP-Server 调用的 LLM 客户端进行协同验证，例如：Cherry Studio、Claude Code、Gemini-CLI、VS Code Copilot、Qwen-Code 等。平常使用是直接使用 `make` 命令的，要看详细命令可参考快速开始，也可以直接查看项目根目录的 `Makefile` 文件。

- 准备 RTL 和对应的 SPEC 文档放入 `examples/{dut}` 文件夹。`{dut}` 是模块的名称，比如 `Adder`，如果是 `Adder`，目录则为 `examples/Adder`。
- 打包 RTL，将文档放入工作目录并且启动 MCP server: `make mcp_{dut}`，`{dut}` 为对应的模块。此处如果使用的 `Adder`，则命令为 `make mcp_Adder`
- 在支持 MCP client 的应用中配置 JSON:

```
{
  "mcpServers": {
    "unitytest": {
      "httpUrl": "http://localhost:5000/mcp",
      "timeout": 10000
    }
  }
}
```

- 启动应用：此处使用的 Qwen Code，在 `UCAgent/output` 启动 `qwen`，然后输入提示词。
- 输入提示词：> 请通过工具 `RoleInfo` 获取你的角色信息和基本指导，然后完成任务。工具 `ReadTextFile` 读取文件。你需要在当前工作目录进行文件操作，不要超出该目录。

## 4 直接使用模式

### 4.1 直接使用

基于本地 CLI 和大模型的使用方式。需要准备好 OpenAI 兼容的 API 和嵌入模型 API。

#### 4.1.1 使用环境变量配置（推荐）

配置文件内容：

```
# OpenAI 兼容的 API 配置
openai:
  model_name: "$(OPENAI_MODEL: Qwen/Qwen3-Coder-30B-A3B-Instruct)" # 模型名称
  openai_api_key: "$(OPENAI_API_KEY: YOUR_API_KEY)" # API 密钥
  openai_api_base: "$(OPENAI_API_BASE: http://10.156.154.242:8000/v1)" # API 基础 URL
# 向量嵌入模型配置
# 用于文档搜索和记忆功能，不需要可通过 --no-embed-tools 关闭
embed:
  model_name: "$(EMBED_MODEL: Qwen/Qwen3-Embedding-0.6B)" # 嵌入模型名称
  openai_api_key: "$(EMBED_OPENAI_API_KEY: YOUR_API_KEY)" # 嵌入模型 API 密钥
  openai_api_base: "$(EMBED_OPENAI_API_BASE: http://10.156.154.242:8001/v1)" # 嵌入模型 API URL
```

```
dims: 4096 # 嵌入维度
```

UCAgent 的配置文件支持 Bash 风格的环境变量占位: `$(VAR: default)`。加载时会用当前环境变量 VAR 的值替换; 若未设置, 则使用 `default`。

- 例如在内置配置 `vagent/setting.yaml` 中:
  - `openai.model_name: "$(OPENAI_MODEL: <your_chat_model_name>)"`
  - `openai.openai_api_key: "$(OPENAI_API_KEY: [your_api_key])"`
  - `openai.openai_api_base: "$(OPENAI_API_BASE: http://<your_chat_model_url>/v1)"`
  - `embed.model_name: "$(EMBED_MODEL: <your_embedding_model_name>)"`
  - 也支持其他提供商: `model_type` 可选 `openai`、`anthropic`、`google_genai` (详见 `vagent/setting.yaml`)。

你可以仅通过导出环境变量完成模型与端点切换, 而无需改动配置文件。

示例: 设置聊天模型与端点

```
# 指定聊天模型 (OpenAI 兼容)
export OPENAI_MODEL='Qwen/Qwen3-Coder-30B-A3B-Instruct'

# 指定 API Key 与 Base (按你的服务商填写)
export OPENAI_API_KEY='你的 API 密钥'
export OPENAI_API_BASE='https://你的-openai-兼容端点/v1'

# 可选: 嵌入模型 (若使用检索/记忆等功能)
export EMBED_MODEL='text-embedding-3-large'
export EMBED_OPENAI_API_KEY="$OPENAI_API_KEY"
export EMBED_OPENAI_API_BASE="$OPENAI_API_BASE"
```

然后按前述命令启动 UCAgent 即可。若要长期生效, 可将上述 `export` 追加到你的默认 shell 启动文件 (例如 bash: `~/.bashrc`, zsh: `~/.zshrc`, fish: `~/.config/fish/config.fish`), 保存后重新打开终端或手动加载。

#### 4.1.2 使用 `config.yaml` 来配置

- 在项目根目录创建并编辑 `config.yaml` 文件, 配置 AI 模型和嵌入模型:

```
# OpenAI 兼容的 API 配置
openai:
```

```

openai_api_base: <your_openai_api_base_url> # API 基础 URL
model_name: <your_model_name> # 模型名称, 如 gpt-4o-mini
openai_api_key: <your_openai_api_key> # API 密钥

# 向量嵌入模型配置
# 用于文档搜索和记忆功能, 不需要可通过 --no-embed-tools 关闭
embed:
  model_name: <your_embed_model_name> # 嵌入模型名称
  openai_api_base: <your_openai_api_base_url> # 嵌入模型 API URL
  openai_api_key: <your_api_key> # 嵌入模型 API 密钥
  dims: <your_embed_model_dims> # 嵌入维度, 如 1536

```

#### 4.1.3 开始使用

- 第一步和 MCP 模式相同, 准备 RTL 和对应的 SPEC 文档放入 examples/{dut} 文件夹。{dut} 是模块的名称, 如果是 Adder, 目录则为 examples/Adder。
- 第二步开始就不同了, 打包 RTL, 将文档放入工作目录并启动 UCAgent TUI: make test\_{dut}, {dut} 为对应的模块。若使用 Adder, 命令为 make test\_Adder (可在 Makefile 查看全部目标)。该命令会:
  - 将 examples/{dut} 下文件拷贝到 output/{dut} (含.v/.sv/.md/.py 等)
  - 执行 python3 ucagent.py output/ {dut} --config config.yaml -s -hm --tui -l
  - 启动带 TUI 的 UCAgent, 并自动进入任务循环 (loop)

提示: 验证产物默认写入 output/unity\_test/, 若需更改可通过 CLI 的 --output 参数指定目录名。

#### 4.1.4 直接用 CLI 启动 (不经 Makefile)

- 未安装命令时 (项目内运行):
  - python3 ucagent.py output/ Adder --config config.yaml -s -hm --tui -l
- 安装为命令后:
  - ucagent output/ Adder --config config.yaml -s -hm --tui -l

参数对齐 vagent/cli.py:

- workspace: 工作区目录 (此处为 output/)
- dut: DUT 名称 (工作区子目录名, 如 Adder)

- 常用可选项：
  - `--tui` 启动终端界面
  - `-l/--loop --loop-msg "..."` 启动后立即进入循环并注入提示
  - `-s/--stream-output` 实时输出
  - `-hm/--human` 进入人工干预模式（在阶段间可暂停）
  - `--no-embed-tools` 如不需要检索/记忆工具
  - `--skip/--unskip` 跳过/取消跳过阶段（可多次传入）

#### 4.1.5 常用 TUI 命令速查（直接使用模式）

- 列出工具: `tool_list`
- 阶段检查: `tool_invoke Check timeout=0`
- 查看日志: `tool_invoke StdCheck lines=-1` (-1 表示所有行)
- 终止检查: `tool_invoke KillCheck`
- 阶段完成: `tool_invoke Complete timeout=0`
- 运行用例：
  - 全量: `tool_invoke RunTestCases target=''` `timeout=0`
  - 单测函数: `tool_invoke RunTestCases target='tests/test_checker.py::test_run'` `timeout=120` `return_line_coverage=True`
  - 过滤: `tool_invoke RunTestCases target=' -k add or mul'`
- 阶段跳转: `tool_invoke GoToStage index=2` (索引从 0 开始)
- 继续执行: `loop` 继续修复 ALU754 的未命中分支并重试用例

建议的最小可写权限（只允许生成验证产物处可写）：

- 仅允许 `unity_test/` 与 `unity_test/tests/` 可写：
  - `add_un_write_path *`
  - `del_un_write_path unity_test`
  - `del_un_write_path unity_test/tests`

#### 4.1.6 常见问题与提示

- 检查卡住/无输出：
  - 先 `tool_invoke StdCheck lines=-1` 查看全部日志；必要时 `tool_invoke KillCheck`；修复后重试 `tool_invoke Check`。
- 没找到工具名：
  - 先执行 `tool_list` 确认可用工具；若缺失，检查是否在 TUI 模式、是否禁用了嵌入工具（通常无关）。

- 产物位置：
  - 默认在 `workspace/output_dir`, 即本页示例为 `output/unity_test/`。

#### 4.1.7 相关文档

- 人机协同的完整流程与示例，见人机协同验证
- MCP 集成（如 gemini-cli / qwen code），见 MCP 集成模式
- TUI 界面与操作详解，见 TUI

## 5 人机协同验证

UCAgent 支持在验证过程中进行人机协同，允许用户暂停 AI 执行，人工干预验证过程，然后继续 AI 执行。这种模式适用于需要精细控制或复杂决策的场景。

### 协同流程：

#### 1. 暂停 AI 执行：

- 在直接接入 LLM 模式下：按 `Ctrl+C` 暂停。
- 在 Code Agent 协同模式下：根据 Agent 的暂停方式（如 Gemini-cli 使用 `Esc`）暂停。

#### 2. 人工干预：

- 手动编辑文件、测试用例或配置。
- 使用交互命令进行调试或调整。

#### 3. 阶段控制：

- 使用 `tool_invoke Check` 检查当前阶段状态。
- 使用 `tool_invoke Complete` 标记阶段完成并进入下一阶段。

#### 4. 继续执行：

- 使用 `loop [prompt]` 命令继续 AI 执行，并可提供额外的提示信息。
- 在 Code Agent 模式下，通过 Agent 的控制台输入提示。

#### 5. 权限管理：

- 可使用 `add_un_write_path`, `del_un_write_path` 等命令设置文件写权限，控制 AI 是否可以编辑特定文件。
- 适用于直接接入 LLM 或强制使用 UCAgent 文件工具。

## 6 参数说明

### 6.1 参数与选项

UCAgent 的使用方式为：

```
ucagent <workspace> <dut_name> {参数与选项}
```

#### 6.1.1 输入

- workspace: 工作目录：
  - workspace/: 待测设计 (DUT), 即由 picker 导出的 DUT 对应的 Python 包, 例如: Adder
  - workspace//README.md: 以自然语言描述的该 DUT 验证需求与目标
  - workspace//\*.md: 其他参考文件
  - workspace//\*.v/sv/scala: 源文件, 用于进行 bug 分析
  - 其他与验证相关的文件 (例如: 提供的测试实例、需求说明等)
- dut\_name: 待测设计的名称, 即, 例如: Adder

#### 6.1.2 输出

- workspace: 工作目录：
  - workspace/Guide\_Doc: 验证过程中所遵循的各项要求与指导文档
  - workspace/uc\_test\_report: 生成的 Toffee-test 测试报告
  - workspace/unity\_test/tests: 自动生成的测试用例
  - workspace//\*.md: 生成的各类文档, 包括 Bug 分析、检查点记录、验证计划、验证结论等

对输出的详细解释可以参考快速开始的结果分析

#### 6.1.3 位置参数

参数	必填	说明	示例
workspace	是	运行代理的工作目录	./output
dut	是	DUT 名称 (工作目录下的子目录名)	Adder

### 6.1.4 执行与交互

选项	简写	取值/类型	默认值	说明
-stream-output	-s	flag	关闭	流式输出到控制台
-human	-hm	flag	关闭	启动时进入人工输入/断点模式
-interaction-mode	-im	standard/enhanced/advanced	standard	交互模式；enhanced 含规划与记忆管理，advanced 含自适应策略
-tui		flag	关闭	启用终端 TUI 界面
-loop	-l	flag	关闭	启动后立即进入主循环（可配合-loop-msg），适用于直接使用模式
-loop-msg		str	空	进入循环时注入的首条消息
-seed		int	随机	随机种子（未指定则自动随机）
-sys-tips		str	空	覆盖系统提示词

### 6.1.5 配置与模板

选项	简写	取值/类型	默认值	说明
-config		path	无	配置文件路，如-config config.yaml 径
-template-dir		path	无	自定义模板目录
-template-overwrite		flag	否	渲染模板到 workspace 时允许覆盖已存在内容
-output		dir	unity_tes	输出目录名
-override		A.B.C=VALUE[,X.Y=VAL2,...]		以“点号路径 = 值”覆盖配置；字符串需引号，其它按 Python 字面量解析
-gen-instruct-file	-gif	file	无	在 workspace 下生成外部 Agent 的引导文件（存在则覆盖）
-guid-doc-path		path	无	使用自定义 Guide_Doc 目录（默认使用内置拷贝）

### 6.1.6 计划与 ToDo

选项	简写	取值/类型	默认值	说明
-force-todo	-fp	flag	否	在 standard 模式下也启用 ToDo 工具，并在每轮提示中附带 ToDo 信息
-use-todo-tools	-utt	flag	否	启用 ToDo 相关工具（不限于 standard 模式）

### 6.1.7 ToDo 工具概览与示例

给模型规划的，小模型关闭，大模型自行打开

说明：ToDo 工具是用于提升模型规划能力的工具，用户可以利用它来自定义模型的 ToDo 列表。目前该功能对模型能力要求较高，默认处于关闭状态。

启用条件：任意模式下使用 `--use-todo-tools`；或在 standard 模式用 `--force-todo` 强制启用并在每轮提示中附带 ToDo 信息。

约定与限制：步骤索引为 1-based；steps 数量需在 2~20；notes 与每个 step 文本长度  $\leq 100$ ；超限会拒绝并返回错误字符串。

#### 工具总览

工具类	调用名	主要功能	参数	返回	关键约束/行为
CreateToDo	CreateToDo	新建当前 ToDo（覆盖旧 ToDo）	task_description: str; steps: List[str]	成功提示 + 摘要字符串	校验步数与长度；成功后写入并返回摘要
CompleteToDo	CompleteToDo	标记步骤为完成，可附加备注	completed_steps: List[int]=[]; notes: str= “”	成功提示 + (完成数) + 摘要	仅未完成步骤生效；无 ToDo 时提示先创建；索引越界忽略
UndoToDo	UndoToDo	撤销步骤完成状态，可附加备注	steps: List[int]=[]; notes: str= “”	成功提示 + (撤销数) + 摘要	仅已完成步骤生效；无 ToDo 时提示先创建；索引越界忽略
ResetToDo	ResetToDo	重置/清空当前 ToDo	无	重置成功提示	清空步骤与备注，随后可重新创建
GetToDoState	GetToDoState	获取当前 ToDo 摘要	无	摘要字符串 / 无 ToDo 提示	只读，不修改状态
ToDoState	ToDoState	获取状态短语（看板/状态栏）	无	状态描述字符串	动态显示：无 ToDo/已完成/进度统计等

调用示例（以 MCP/内部工具调用为例，参数为 JSON 格式）：

```
{  
    "tool": "CreateToDo",  
    "args": {  
        "task_description": " 为 Adder 核心功能完成验证闭环",  
        "steps": [  
            " 阅读 README 与规格，整理功能点",  
            " 定义检查点与通过标准",  
            " 生成首批单元测试",  
            " 运行并修复失败用例",  
            " 补齐覆盖率并输出报告"  
        ]  
    }  
}
```

```
{  
    "tool": "CompleteToDoSteps",  
    "args": { "completed_steps": [1, 2], "notes": " 初始问题排查完成，准备补充  
    ↳ 用例" }  
}
```

```
{ "tool": "UndoToDoSteps", "args": { "steps": [2], "notes": " 第二步需要微调检  
    ↳ 查点" } }
```

```
{ "tool": "ResetToDo", "args": {} }
```

```
{ "tool": "GetToDoSummary", "args": {} }
```

```
{ "tool": "ToDoState", "args": {} }
```

### 6.1.8 外部与嵌入工具

选项	简写	取值/类型	默认值	说明
-ex-tools		name1[,name2 …]	无	逗号分隔的外部工具类名列表（如： SqThink）

选项	简写	取值/类型	默认值	说明
-no-embed-tools		flag	否	禁用内置的检索/记忆类嵌入工具

### 6.1.9 日志

选项	简写	取值/类型	默认值	说明
-log		flag	否	启用日志
-log-file		path	自动	日志输出文件 (未指定则使用默认)
-msg-file		path	自动	消息日志文件 (未指定则使用默认)

### 6.1.10 MCP Server

选项	简写	取值/类型	默认值	说明
-mcp-server		flag	否	启动 MCP Server (含文件工具)
-mcp-server-no-file-tools		flag	否	启动 MCP Server (无文件操作工具)
-mcp-server-host		host	127.0.0.1	Server 监听地址
-mcp-server-port		int	5000	Server 端口

### 6.1.11 阶段控制与安全

选项	简写	取值/类型	默认值	说明
-force-stage-index		int	0	强制从指定阶段索引开始
-skip		int (可多次)	[]	跳过指定阶段索引, 可重复提供
-unskip		int (可多次)	[]	取消跳过指定阶段索引, 可重复提供
-no-write / -nw		path1 path2 ...	无	限制写入目标列表; 必须位于 workspace 内且存在

### 6.1.12 版本与检查

选项	简写	取值/类型	默认值	说明
-check		flag	否	检查默认配置、语言目录、模板与 Guide_Doc 是否存在后退出
-version		flag		输出版本并退出

### 6.1.13 示例

```
python3 ucagent.py ./output Adder \
\
-s \
-hm \
-im enhanced \
--tui \
-l \
--loop-msg 'start verification' \
--seed 12345 \
--sys-tips '按规范完成 Adder 的验证' \
\
--config config.yaml \
--template-dir ./templates \
--template-overwrite \
--output unity_test \
--override 'conversation_summary.max_tokens=16384,' \
    'conversation_summary.max_summary_tokens=2048,' \
    'conversa-
        ↳  tion_summary.use_uc_mode=True,lang="zh",openai.model_name="gpt-
        ↳  4o-mini"' \
--gen-instruct-file GEMINI.md \
--guid-doc-path ./output/Guide_Doc \
\
--use-todo-tools \
\
--ex-tools 'SqThink,AnotherTool' \
```

```
--no-embed-tools \
\
--log \
--log-file ./output/ucagent.log \
--msg-file ./output/ucagent.msg \
\
--mcp-server-no-file-tools \
--mcp-server-host 127.0.0.1 \
--mcp-server-port 5000 \
\
--force-stage-index 2 \
--skip 5 --skip 7 \
--unskip 6 \
--nw ./output/Adder ./output/unity_test
```

- 位置参数
  - ./output: workspace 工作目录
  - Adder: dut 子目录名
- 执行与交互
  - -s: 流式输出
  - -hm: 启动即人工可介入
  - -im enhanced: 交互模式为增强（含规划与记忆）
  - -tui: 启用 TUI
  - -l: 启动后立即进入循环
  - -loop/-loop-msg: 进入循环注入首条消息
  - -seed 12345: 固定随机种子
  - -sys-tips: 自定义系统提示
- 配置与模板
  - -config config.yaml: 从 config.yaml 加载项目配置
  - -template-dir ./templates: 指定模板目录为 ./templates
  - -template-overwrite: 渲染模板时允许覆盖
  - -output unity\_test: 输出目录名 unity\_test
  - -override ‘...’ : 覆盖配置键值（点号路径 = 值，多项用逗号分隔；字符串需内层引号，整体用单引号包裹以保留引号），示例里设置了会话摘要上限、启用裁剪、文档语言为“中文”、模型名为 gpt-4o-mini
  - -gif/-gen-instruct-file GEMINI.md: 在 <workspace>/GEMINI.md 下生成外部协作引导文件

- -guid-doc-path ./output/Guide\_Doc: 自定义 Guide\_Doc 目录为 ./output/Guide\_Doc
- 计划与 ToDo
  - -use-todo-tools: 启用 ToDo 工具及强制附带 ToDo 信息
- 外部与嵌入工具
  - -ex-tools ‘SqThink,AnotherTool’: 启用外部工具 SqThink,AnotherTool
  - -no-embed-tools: 禁用内置嵌入检索/记忆工具
- 日志
  - -log: 开启日志文件
  - -log-file ./output/ucagent.log: 指定日志输出文件为 ./output/ucagent.log
  - -msg-file ./output/ucagent.msg: 指定消息日志文件为 ./output/ucagent.msg
- MCP Server
  - -mcp-server-no-file-tools: 启动 MCP (无文件操作工具)
  - -mcp-server-host: Server 监听地址为 127.0.0.1
  - -mcp-server-port: Server 监听端口为 5000
- 阶段控制与安全
  - -force-stage-index 2: 从阶段索引 2 开始
  - -skip 5 -skip 7: 跳过阶段 5 和阶段 7
  - -unskip 7: 取消跳过阶段 7
  - -nw ./output/Adder ./output/unity\_test: 限制仅 ./output/Adder 和 ./output/unity\_test 路径可写
- 说明
  - -check 与 -version 会直接退出, 未与运行组合使用
  - -mcp-server 与 -mcp-server-no-file-tools 二选一; 此处选了后者带路径参数 (如 -template-dir/-guid-doc-path/-nw 的路径) 需实际存在, 否则会报错
  - -override 字符串值务必带引号, 并整体用单引号包住以避免 shell 吃掉引号 (示例写法已处理)

## 7 TUI

### 7.1 TUI (界面与操作)

UCAgent 自带基于 urwid 的终端界面 (TUI), 用于在本地交互式观察任务进度、消息流与控制台输出, 并直接输入命令 (如进入/退出循环、切换模式、执行调试命令等)。



图 6: tui 界面组成

### 7.1.1 界面组成

- Mission 面板 (左侧)
  - 阶段列表：显示当前任务的阶段（索引、标题、失败数、耗时）。颜色含义：
    - \* 绿色：已完成阶段
    - \* 红色：当前进行阶段
    - \* 黄色：被跳过的阶段（显示“skipped”）
  - Changed Files：近期修改文件（含修改时间与相对时间，如“3m ago”）。较新的文件以绿色显示。
  - Tools Call：工具调用状态与计数。忙碌中的工具会以黄色高亮（如 SqThink(2)）。
  - Deamon Commands：后台运行的 demo 命令列表（带开始时间与已运行时长）。
- Status 面板 (右上)
  - 显示 API 与代理状态摘要，以及当前面板尺寸参数（便于调节布局时参考）。
- Messages 面板 (右上中)
  - 实时消息流（模型回复、工具输出、系统提示）。
  - 支持焦点与滚动控制，标题会显示“当前/总计”的消息定位。例如：Messages (123/456)。

- Console (底部)
  - Output: 命令与系统输出区域，支持分页浏览。
  - Input: 命令输入行（默认提示符“(UnityChip)”）。提供历史、补全、忙碌提示等。

提示：界面每秒自动刷新一次（不影响输入）。当消息或输出过长时，会进入分页或手动滚动模式。

### 7.1.2 操作与快捷键

- Enter: 执行当前输入命令；若输入为空会重复上一次命令；输入 q/Q/exit/quit 退出 TUI。
- Esc:
  - 若正在浏览 Messages 的历史，退出滚动并返回末尾；
  - 若 Output 正在分页查看，退出分页；
  - 否则聚焦到底部输入框。
- Tab: 命令补全；再次按 Tab 可分批显示更多可选项。
- Shift+Right: 清空 Console Output。
- Shift+Up / Shift+Down: 在 Messages 中向上/向下移动焦点（浏览历史）。
- Ctrl+Up / Ctrl+Down: 增/减 Console 输出区域高度。
- Ctrl+Left / Ctrl+Right: 减/增 Mission 面板宽度。
- Shift+Up / Shift+Down (另一路径): 调整 Status 面板高度（最小 3，最大 100）。
- Up / Down:
  - 若 Output 在分页模式，Up/Down 用于翻页；
  - 否则用于命令历史导航（将历史命令放入输入行，可编辑后回车执行）。

分页模式提示：当 Output 进入分页浏览时，底部标题会提示“Up/Down: scroll, Esc: exit”，Esc 退出分页并返回输入状态。

### 7.1.3 命令与用法

- 普通命令：直接输入并回车，例如 loop、tui、help 等（由内部调试器处理）。
- 历史命令：在输入行为空时按 Enter，将重复执行上一条命令。
- 清屏：输入 clear 并回车，仅清空 Output（不影响消息记录）。
- 演示/后台命令：命令末尾添加 & 将在后台运行，完成后会在 Output 区域提示结束；当前后台命令可通过 list\_demo\_cmds 查看。
- 直接执行系统/危险命令：以! 前缀执行（例如!loop），该模式执行后优先滚动到最新输出。
- 列出后台命令：list\_demo\_cmds 显示正在运行的 demo 命令列表与开始时间。

#### 7.1.3.1 消息配置 (message\_config)

- 作用：在运行中查看/调整消息裁剪策略，控制历史保留与 LLM 输入 token 上限。
- 命令：
  - message\_config 查看当前配置
  - message\_config 设置配置项
- 可配置项：
  - max\_keep\_msgs: 保留的历史消息条数（影响会话记忆窗口）
  - max\_token: 进入模型前的消息裁剪 token 上限（影响开销/截断）
- 示例：
  - message\_config
  - message\_config max\_keep\_msgs 8
  - message\_config max\_token 4096

## 其他说明

- 自动补全：支持命令名与部分参数的补全；候选项过多时分批显示，可多次按 Tab 查看剩余项。
- 忙碌提示：命令执行期间，输入框标题会轮转显示 (wait.), (wait..), (wait…)，表示正在处理。
- 消息焦点：当未手动滚动时，消息焦点自动跟随最新消息；进入手动滚动后，会保持当前位置，直至按 Esc 或滚动至末尾。
- 错误容错：若某些 UI 操作异常（如终端不支持某些控制序列），TUI 会尽量回退到安全状态继续运行。

# 8 FAQ

## 8.1 FAQ

- 模型切换：在 config.yaml 改 openai.model\_name
- 验证过程中出现错误怎么办：使用 Ctrl+C 进入交互模式，通过 status 查看当前状态，使用 help 获取调试命令。
- Check 失败：先 ReadTextFile 阅读 reference\_files；再按返回信息修复，循环 RunTestCases → Check
- 自定义阶段：修改 vagent/lang/zh/config/default.yaml 的 stage；或用 --override 临时覆盖
- 添加工具：vagent/tools/ 下新建类，继承 UCTool，运行时 --ex-tools YourTool
- MCP 连接失败：检查端口/防火墙，改 --mcp-server-port；无嵌入可加 --no-embed-tools
- 只读保护：通过 --no-write/--nw 指定路径限制写入（必须位于 workspace 内）

### 8.1.1 为什么快速启动找不到 config.yaml/定制流程时找不到 config.yaml?

- 使用 pip 安装后并没有 `config.yaml` 那个文件，所以在快速启动的启动 MCP Server 没有加 `--config config.yaml` 这个选项。
- 可以通过在工作目录添加 `config.yaml` 文件并且加上 `--config config.yaml` 参数来启动；也可以使用克隆仓库来使用 UCAgent 的方式来解决。

### 8.1.2 运行中如何调整消息窗口与 token 上限？

- 在 TUI 输入: `message_config` 查看当前配置；
- 设置: `message_config max_keep_msgs 8` 或 `message_config max_token 4096`；
- 作用范围：影响会话历史裁剪与送入 LLM 的最大 token 上限（通过 Summarization/Trim 节点生效）。

### 8.1.3 文档中的“CK bug”要改吗？

- 是。术语统一为“TC bug”。同时确保 bug 文档里的 `<TC-*>` 能匹配失败用例（文件/类/用例名）。

### 8.1.4 为什么找不到 WriteTextFile 工具？

- 该工具已移除。请改用 `EditTextFile`（支持 `overwrite/append/replace` 三种模式）或其他文件工具（`Copy/Move/Delete` 等）。

## 9 工作流

整体采用“按阶段渐进推进”的方式，每个阶段都有明确目标、产出与通过标准；完成后用工具 Check 验证并用 Complete 进入下一阶段。若阶段包含子阶段，需按顺序逐一完成子阶段并各自通过 Check。

- 顶层阶段总数：11（见 `vagent/lang/zh/config/default.yaml`）
- 推进原则：未通过的阶段不可跳转；可用工具 CurrentTips 获取当前阶段详细指导；需要回补时可用 GotoStage 回到指定阶段；命令行也可用 `-skip/-unskip` 控制阶段索引。

### 9.1 整体流程概览（11 个阶段）

目前的流程包含：

1. 需求分析与验证规划 → 2) {DUT} 功能理解 → 3) 功能规格分析与测试点定义 → 4) 测试平台基础架构设计 → 5) 功能覆盖率模型实现 → 6) 基础 API 实现 → 7) 基础 API 功能测试 → 8) 测试框架脚手架 → 9) 全面验证执行与缺陷分析 → 10) 代码行覆盖率分析与提升（默认跳过，可启用）→ 11) 验证审查与总结

以实际的工作流为准，下图仅供参考。

说明：以下路径中的默认为工作目录下的输出目录名（默认 unity\_test）。例如文档输出到 <workspace>/unity\_test/。

---

### 阶段 1：需求分析与验证规划

- 目标：理解任务、明确验证范围与策略。
- 怎么做：
  - 阅读 {DUT}/README.md，梳理“需要测哪些功能/输入输出/边界与风险”。
  - 形成可执行的验证计划与目标清单。
- 产出：<OUT>/《{DUT}\_verification\_needs\_and\_plan.md》（中文撰写）。
- 通过标准：文档存在、结构规范（自动检查 markdown\_file\_check）。
- 检查器：
  - UnityChipCheckerMarkdownFormat
    - \* 作用：校验 Markdown 文件存在与格式，禁止把换行写成字面量 \n。
    - \* 参数：
      - markdown\_file\_list (str | List[str]): 待检查的 MD 文件路径或路径列表。示例：  
    {OUT}/《{DUT}\_verification\_needs\_and\_plan.md》
      - no\_line\_break (bool): 是否禁止把换行写成字面量 \n； true 表示禁止。

### 阶段 2：{DUT} 功能理解

- 目标：掌握 DUT 的接口与基本信息，明确是组合/时序电路。
- 怎么做：
  - 阅读 {DUT}/README.md 与 {DUT}/\_\_init\_\_.py。
  - 分析 IO 端口、时钟/复位需求与功能范围。
- 产出：<OUT>/《{DUT}\_basic\_info.md》。
- 通过标准：文档存在、格式规范（markdown\_file\_check）。
- 检查器：
  - UnityChipCheckerMarkdownFormat
    - \* 作用：校验 Markdown 文件存在与格式，禁止把换行写成字面量 \n。
    - \* 参数：
      - markdown\_file\_list (str | List[str]): 待检查的 MD 文件路径或路径列表。示例：

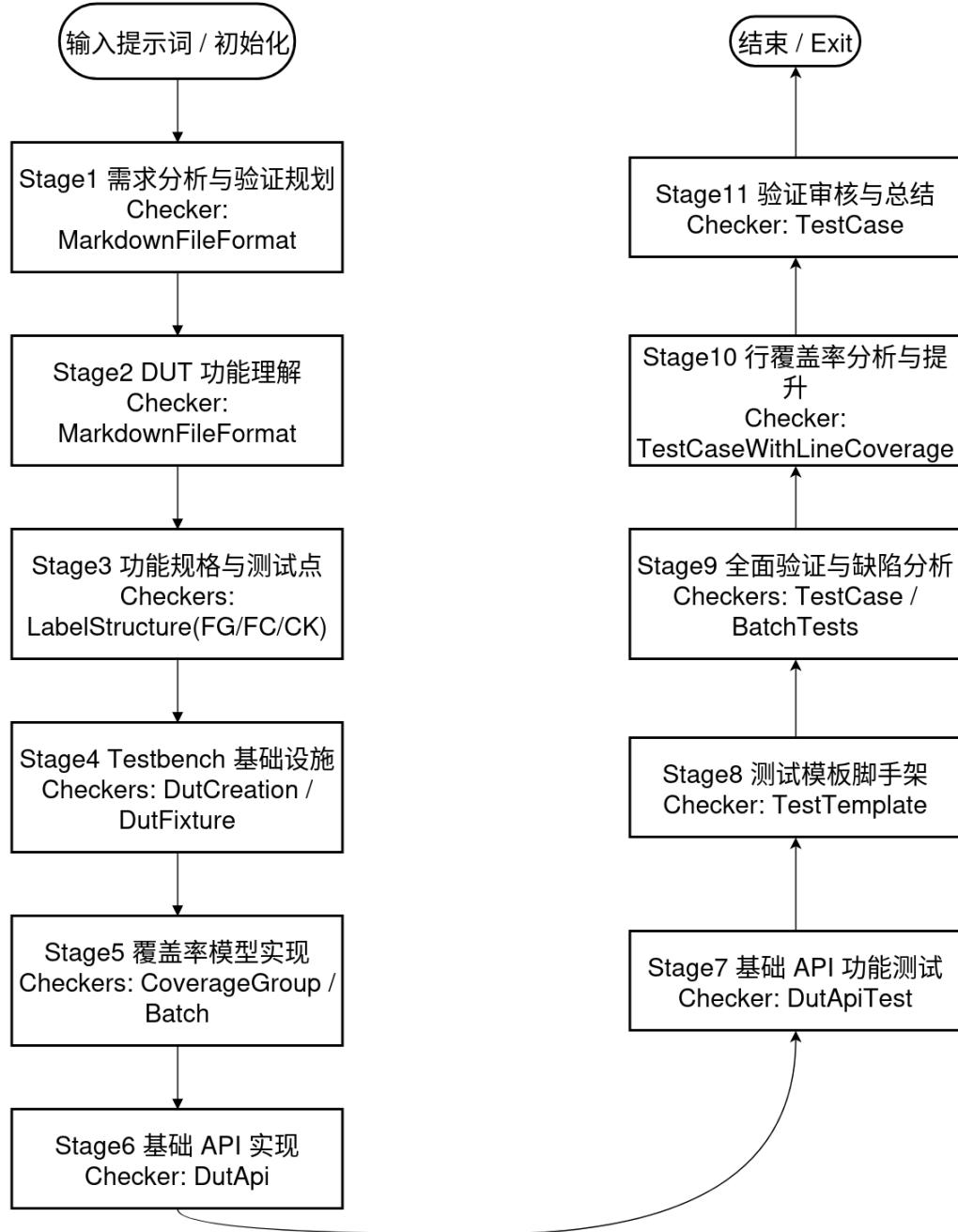


图 7: 工作流图

{OUT}/{DUT}\_basic\_info.md

- no\_line\_break (bool): 是否禁止把换行写成字面量 \n; true 表示禁止。

### 阶段 3: 功能规格分析与测试点定义 (含子阶段 FG/FC/CK)

- 目标: 把功能分组 (FG)、功能点 (FC) 和检测点 (CK) 结构化, 作为后续自动化的依据。
- 怎么做:
  - 阅读 {DUT}/\*.md 与已产出文档, 建立 {DUT}\_functions\_and\_checks.md 的 FG/FC/CK 结构。
  - 规范标签: <FG-组名>、<FC-功能名>、<CK-检测名>, 每个功能点至少 1 个检测点。
- 子阶段:
  - 3.1 功能分组与层次 (FG): 检查器 UnityChipCheckerLabelStructure(FG)
  - 3.2 功能点定义 (FC): 检查器 UnityChipCheckerLabelStructure(FC)
  - 3.3 检测点设计 (CK): 检查器 UnityChipCheckerLabelStructure(CK)
- 产出: <OUT>/ {DUT}\_functions\_and\_checks.md。
- 通过标准: 三类标签结构均通过对应检查。
- 对应检查器 (默认配置):
  - 3.1 UnityChipCheckerLabelStructure
    - \* 作用: 解析 {DUT}\_functions\_and\_checks.md 中的标签结构并校验层级与数量 (FG)。
    - \* 参数:
      - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/{DUT}\_functions\_and\_checks.md
      - leaf\_node ("FG" | "FC" | "CK"): 需要校验的叶子类型。示例: "FG"
      - min\_count (int, 默认 1): 该叶子类型的最小数量阈值。
      - must\_have\_prefix (str, 默认 "FG-API" ): FG 名称要求的前缀, 用于规范化分组命名。
    - 3.2 UnityChipCheckerLabelStructure
      - \* 作用: 解析文档并校验功能点定义 (FC)。
      - \* 参数:
        - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/{DUT}\_functions\_and\_checks.md
        - leaf\_node ("FG" | "FC" | "CK"): 需要校验的叶子类型。示例: "FC"
        - min\_count (int, 默认 1): 该叶子类型的最小数量阈值。
        - must\_have\_prefix (str, 默认 "FG-API" ): 所属 FG 的前缀规范, 用于一致性检查。
      - 3.3 UnityChipCheckerLabelStructure
        - \* 作用: 解析文档并校验检测点设计 (CK), 并缓存 CK 列表用于后续分批实现。
        - \* 参数:
          - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/{DUT}\_functions\_and\_checks.md
          - leaf\_node ("FG" | "FC" | "CK"): 需要校验的叶子类型。示例: "CK"
          - data\_key (str): 共享数据键名, 用于缓存 CK 列表 (供后续分批实现使用)。示例: "COVER\_GROUP\_DOC\_CK\_LIST"

- min\_count (int, 默认 1): 该叶子类型的最小数量阈值。
- must\_have\_prefix (str, 默认 “FG-API” ): 所属 FG 的前缀规范，用于一致性检查。

#### 阶段 4: 测试平台基础架构设计 (fixture/API 框架)

- 目标: 提供统一的 DUT 创建与测试生命周期管理能力。
- 怎么做:
  - 在 <OUT>/tests/{DUT}\_api.py 实现 create\_dut(); 时序电路配置时钟 (InitClock)，组合电路无需时钟。
  - 实现 pytest fixture dut，负责初始化/清理与可选的波形/行覆盖率开关。
- 产出: <OUT>/tests/{DUT}\_api.py (含注释与文档字符串)。
- 通过标准: DUT 创建与 fixture 检查通过 (UnityChipCheckerDutCreation / UnityChipCheckerDutFixture)。
- 子阶段检查器:
  - DUT 创建: UnityChipCheckerDutCreation
    - \* 作用: 校验 {DUT}\_api.py 中的 create\_dut(request) 是否实现规范 (签名、时钟/复位、覆盖率路径等约定)。
    - \* 参数:
      - target\_file (str): DUT API 与 fixture 所在文件路径。示例:{OUT}/tests/{DUT}\_api.py
  - dut fixture: UnityChipCheckerDutFixture
    - \* 作用: 校验 pytest fixture dut 的生命周期管理、yield/清理，以及覆盖率收集调用是否到位。
    - \* 参数:
      - target\_file (str): 包含 dut fixture 的文件路径。示例:{OUT}/tests/{DUT}\_api.py
  - env fixture: UnityChipCheckerEnvFixture
    - \* 作用: 校验 env\* 系列 fixture 的存在、数量与 Bundle 封装是否符合要求。
    - \* 参数:
      - target\_file (str): 包含 env\* 系列 fixture 的文件路径。示例:{OUT}/tests/{DUT}\_api.py
      - min\_env (int, 默认 1): 至少需要存在的 env\* fixture 数量。示例: 1
      - force\_bundle (bool, 当前未使用): 是否强制要求 Bundle 封装。

#### 覆盖率路径规范 (重要):

- 在 create\_dut(request) 中, 必须通过 get\_coverage\_data\_path(request, new\_path=True) 获取新的行覆盖率文件路径，并传入 dut.SetCoverage(...).
- 在 dut fixture 的清理阶段, 必须通过 get\_coverage\_data\_path(request, new\_path=False) 获取已有路径，并调用 set\_line\_coverage(request, <path>, ignore=...) 写入统计。
- 若缺失上述调用，检查器会直接报错，并给出修复提示(含 tips\_of\_get\_coverage\_data\_path 示例)。

## 阶段 5：功能覆盖率模型实现

- 目标：将 FG/FC/CK 转为可统计的覆盖结构，支撑进度度量与回归。
- 怎么做：
  - 在 <OUT>/tests/{DUT}\_function\_coverage\_def.py 实现 get\_coverage\_groups(dut)。
  - 为每个 FG 建立 CovGroup；为 FC/CK 建 watch\_point 与检查函数（优先用 lambda，必要时普通函数）。
- 子阶段：
  - 5.1 覆盖组创建 (FG)
  - 5.2 覆盖点与检查实现(FC/CK)，支持“分批实现”提示(COMPLETED\_POINTS/TOTAL\_POINTS)。
- 产出：<OUT>/tests/{DUT}\_function\_coverage\_def.py。
- 通过标准：CoverageGroup 检查 (FG/FC/CK) 与批量实现检查通过。
- 子阶段检查器：
  - 5.1 UnityChipCheckerCoverageGroup
    - \* 作用：比对覆盖组定义与文档 FG 一致性。
    - \* 参数：
      - test\_dir (str): 测试目录根路径。示例: {OUT}/tests
      - cov\_file (str): 覆盖率模型定义文件路径。示例: {OUT}/tests/{DUT}\_function\_coverage\_def.py
      - doc\_file (str): 功能/检查点文档路径。示例: {OUT}/{DUT}\_functions\_and\_checks.md
      - check\_types (str | List[str]): 检查的类型集合。示例: "FG"
  - 5.2 UnityChipCheckerCoverageGroup
    - \* 作用：比对覆盖点/检查点实现与文档 FC/CK 一致性。
    - \* 参数：
      - test\_dir (str): 测试目录根路径。示例同上
      - cov\_file (str): 覆盖率模型定义文件路径。示例同上
      - doc\_file (str): 功能/检查点文档路径。示例同上
      - check\_types (List[str]): 检查类型集合。示例: ["FC", "CK"]
  - 5.2 (分批) UnityChipCheckerCoverageGroupBatchImplementation
    - \* 作用：按 CK 分批推进实现与对齐检查，维护进度 (TOTAL/COMPLETED)。
    - \* 参数：
      - test\_dir (str): 测试目录根路径。
      - cov\_file (str): 覆盖率模型定义文件路径。
      - doc\_file (str): 功能/检查点文档路径。
      - batch\_size (int, 默认 20): 每批实现与校验的 CK 数量上限。示例: 20
      - data\_key (str): 共享数据键名，用于读取 CK 列表。示例: "COVER\_GROUP\_DOC\_CK\_LIST"

## 阶段 6：基础 API 实现

- 目标：用 api\_{DUT}\_\* 前缀提供可复用的操作封装，隐藏底层信号细节。

- 怎么做:
  - 在 <OUT>/tests/{DUT}\_api.py 实现至少 1 个基础 API; 建议区分“底层功能 API”与“任务功能 API”。
  - 补充详细 docstring: 功能、参数、返回值、异常。
- 产出: <OUT>/tests/{DUT}\_api.py。
- 通过标准: UnityChipCheckerDutApi 通过 (前缀必须为 api\_{DUT}\_)。
- 检查器:
  - UnityChipCheckerDutApi
    - \* 作用: 扫描/校验 api\_{DUT}\_\* 函数的数量、命名、签名与 docstring 完整度。
    - \* 参数:
      - api\*prefix (str): API 前缀匹配表达式。建议: "api\*{DUT}\\_"
      - target\_file (str): API 定义所在文件。示例: {OUT}/tests/{DUT}\_api.py
      - min\_apis (int, 默认 1): 至少需要的 API 数量。

#### 阶段 7: 基础 API 功能正确性测试

- 目标: 为每个已实现 API 编写至少 1 个基础功能用例，并标注覆盖率。
- 怎么做:
  - 在 <OUT>/tests/test\_{DUT}\_api\_\*.py 新建测试；导入 from {DUT}\_api import \*。
  - 每个测试函数的第一行: dut.fc\_cover['FG-API'].mark\_function('FC-API-NAME', test\_func, ['CK-XXX'])。
  - 设计典型/边界/异常数据，断言预期输出。
  - 用工具 RunTestCases 执行与回归。
- 产出: <OUT>/tests/test\_{DUT}\_api\_\*.py 与缺陷记录 (若发现 bug)。
- 通过标准: UnityChipCheckerDutApiTest 通过 (覆盖、用例质量、文档记录齐备)。
- 检查器:
  - UnityChipCheckerDutApiTest
    - \* 作用: 运行 pytest 并检查每个 API 至少 1 个基础功能用例且正确覆盖标记；核对缺陷记录与文档一致。
    - \* 参数:
      - api\*prefix (str): API 前缀匹配表达式。建议: "api\*{DUT}\\_"
      - target\_file\_api (str): API 文件路径。示例: {OUT}/tests/{DUT}\_api.py
      - target\*file\_tests (str): 测试文件 Glob。示例: {OUT}/tests/test\*{DUT}\\_api\\*.py
      - doc\_func\_check (str): 功能/检查点文档。示例: {OUT}/{DUT}\_functions\_and\_checks.md
      - doc\_bug\_analysis (str): 缺陷分析文档。示例: {OUT}/{DUT}\_bug\_analysis.md
      - min\_tests (int, 默认 1): 单 API 最少测试用例数。
      - timeout (int, 默认 15): 单次测试运行超时 (秒)。

#### 阶段 8: 测试框架脚手架构建

- 目标：为尚未实现的功能点批量生成“占位”测试模板，确保覆盖版图完整。
- 怎么做：
  - 依据 `{DUT}_functions_and_checks.md`, 在 `<OUT>/tests/` 创建 `test_*.py`, 文件与用例命名语义化。
  - 每个函数首行标注覆盖率 mark; 补充 TODO 注释说明要测什么; 末尾添加 `assert False, 'Not implemented'` 防误通过。
- 产出：批量测试模板；覆盖率进度指标 (COVERED\_CKS/TOTAL\_CKS)。
- 通过标准：UnityChipCheckerTestTemplate 通过 (结构/标记/说明完整)。
- 检查器：
  - UnityChipCheckerTestTemplate
    - \* 作用：检查模板文件/用例结构、覆盖率标记、TODO 注释与防误通过断言；统计覆盖进度。
    - \* 参数：
      - `doc_func_check(str)`: 功能/检查点文档路径。示例: `{OUT}/{DUT}_functions_and_checks.md`
      - `test_dir(str)`: 测试目录根路径。示例: `{OUT}/tests`
      - `ignore_ck_prefix(str)`: 统计覆盖时忽略的 CK 前缀 (通常为基础 API 的用例)。示例: "test\_api\*{DUT}\\_"
      - `data_key(str)`: 共享数据键名, 用于生成/读取模板实现进度。示例: "TEST\_TEMPLATE\_IMP\_REPO"
      - `batch_size(int, 默认 20)`: 每批模板检查数量。
      - `min_tests(int, 默认 1)`: 最少要求的模板测试数。
      - `timeout(int, 默认 15)`: 测试运行超时 (秒)。

#### 阶段 9：全面验证执行与缺陷分析

- 目标：将模板填充为真实测试，系统发现并分析 DUT bug。
- 怎么做：
  - 在 `test_*.py` 填充逻辑，优先通过 API 调用，不直接操纵底层信号。
  - 设计充分数据并断言；用 RunTestCases 运行；对 Fail 进行基于源码的缺陷定位与记录。
- 子阶段：
  - 9.1 分批测试用例实现与对应缺陷分析 (COMPLETED\_CASES/TOTAL\_CASES)。
- 产出：成体系的测试集与 `/{DUT}_bug_analysis.md`。
- 通过标准：UnityChipCheckerTestCase (质量/覆盖/缺陷分析) 通过。
- 检查器：
  - 父阶段：UnityChipCheckerTestCase
    - \* 作用：运行整体测试并对照功能/缺陷文档检查质量、覆盖与记录一致性。
    - \* 参数：
      - `doc_func_check(str)`: 功能/检查点文档路径。示例: `{OUT}/{DUT}_functions_and_checks.md`
      - `doc_bug_analysis(str)`: 缺陷分析文档路径。示例: `{OUT}/{DUT}_bug_analysis.md`
      - `test_dir(str)`: 测试目录根路径。示例: `{OUT}/tests`
      - `min_tests(int, 默认 1)`: 最少要求的测试用例数量。

- timeout (int, 默认 15): 测试运行超时 (秒)。
- 子阶段 (分批实现): UnityChipCheckerBatchTestsImplementation
  - \* 作用: 分批将模板落地为真实用例并回归, 维护实现进度与报告。
  - \* 参数:
    - doc\_func\_check (str): 功能/检查点文档路径。
    - doc\_bug\_analysis (str): 缺陷分析文档路径。
    - test\_dir (str): 测试目录根路径。
    - ignore\*ck\_prefix (str): 统计覆盖时忽略的 CK 前缀。示例: "test\_api\*{DUT}\\_"
      - batch\_size (int, 默认 10): 每批转化并执行的用例数量。
      - data\_key (str): 共享数据键名 (必填), 用于保存分批实现进度。示例: "TEST\_TEMPLATE\_IMP\_REPORT"
      - pre\_report\_file (str): 历史进度报告路径。示例: {OUT}/{DUT}/.TEST\_TEMPLATE\_IMP\_REPORT
      - timeout (int, 默认 15): 测试运行超时 (秒)。

TC bug 标注规范与一致性 (与文档/报告强关联):

- 术语: 统一使用 “TC bug” (不再使用 “CK bug”)。
- 标注结构:<FG-\*>/<FC-\*>/<CK-\*>/<BG-NAME-XX>/<TC-test\_file.py::[ClassName]::test\_case>;  
其中 BG 的置信度 XX 为 0–100 的整数。
- 失败用例与文档关系:
  - 文档中出现的<TC-\*>必须能与测试报告中的失败用例一一对应 (文件名/类名/用例名匹配)。
  - 失败的测试用例必须标注其关联检查点 (CK), 否则会被判定为“未标记”。
  - 若存在失败用例未在 bug 文档中记录, 将被提示为“未文档化的失败用例”。

阶段 10: 代码行覆盖率分析与提升 (默认跳过, 可启用)

- 目标: 回顾未覆盖代码行, 定向补齐。
- 怎么做:
  - 运行 Check 获取行覆盖率; 若未达标, 围绕未覆盖行增补测试并回归; 循环直至满足阈值。
- 产出: 行覆盖率报告与补充测试。
- 通过标准: UnityChipCheckerTestCaseWithLineCoverage 达标 (默认阈值 0.9, 可在配置中调整)。
- 说明: 该阶段在配置中标记 skip=true, 可用 --unskip 指定索引启用。
- 检查器:
  - UnityChipCheckerTestCaseWithLineCoverage
    - \* 作用: 在 TestCase 基础上统计行覆盖率并对比阈值。
    - \* 参数:
      - doc\_func\_check (str): 功能/检查点文档路径。示例: {OUT}/{DUT}\_functions\_and\_checks.md
      - doc\_bug\_analysis (str): 缺陷分析文档路径。示例: {OUT}/{DUT}\_bug\_analysis.md
      - test\_dir (str): 测试目录根路径。示例: {OUT}/tests

- cfg (dict | Config): 必填, 用于推导默认路径以及环境配置。
- min\_line\_coverage (float, 默认按配置, 未配置则 0.8): 最低行覆盖率阈值。
- coverage\_json (str, 可选): 行覆盖率 JSON 路径。默认: uc\_test\_report/line\_dat/code\_cov.json
- coverage\_analysis (str, 可选): 行覆盖率分析 MD 输出。默认: unity\_test/{DUT}\_line\_coverage.md
- coverage\_ignore (str, 可选): 忽略文件清单。默认: unity\_test/tests/{DUT}.ignore

## 阶段 11：验证审查与总结

- 目标: 沉淀成果、复盘流程、给出改进建议。
- 怎么做:
  - 完善 /{DUT}\_bug\_analysis.md 的缺陷条目 (基于源码分析)。
  - 汇总并撰写 /{DUT}\_test\_summary.md, 回看规划是否达成; 必要时用 GotoStage 回补。
- 产出: <OUT>/ {DUT}\_test\_summary.md 与最终结论。
- 通过标准: UnityChipCheckerTestCase 复核通过。
- 检查器:
  - UnityChipCheckerTestCase
    - \* 作用: 复核整体测试结果与文档一致性, 形成最终结论。
    - \* 参数: doc\_func\_check: "{OUT}/{DUT}\_functions\_and\_checks.md"; doc\_bug\_analysis: "{OUT}/{DUT}\_bug\_analysis.md"; test\_dir: "{OUT}/tests"。

## 提示与最佳实践

- 随时用工具: Detail/Status 查看 Mission 进度与当前阶段; CurrentTips 获取步骤级指导; Check/Complete 推进阶段。
- TUI 左侧 Mission 会显示阶段序号、跳过状态与失败计数; 可结合命令行 --skip/--unskip/-force-stage-index 控制推进。

## 9.2 定制工作流 (增删阶段/子阶段)

### 9.2.1 原理说明

- 工作流定义在语言配置 vagent/lang/zh/config/default.yaml 的顶层 stage: 列表。
- 配置加载顺序: setting.yaml → ~/ucagent/setting.yaml → 语言默认 (含 stage) → 项目根 config.yaml → CLI --override。
- 注意: 列表类型 (如 stage 列表) 在合并时是“整体覆盖”, 不是元素级合并; 因此要“增删改”阶段, 建议把默认的 stage 列表复制到你的项目 config.yaml, 在此基础上编辑。
- 临时不执行某阶段: 优先使用 CLI --skip 跳过该索引; 持久跳过可在你的 config.yaml 中把该阶段条目的 skip: true 写上 (同样需要提供完整的 stage 列表)。

### 9.2.2 增加阶段

- 需求：在“全面验证执行”之后新增一个“静态检查与 Lint 报告”阶段，要求生成<OUT>/**{DUT}\_lint\_report.md** 并做格式检查。
- 做法：在项目根 config.yaml 中提供完整的 stage：列表，并在合适位置插入如下条目（片段示例，仅展示新增项，实际需要放入你的完整 stage 列表里）。

```
stage:  
  # ... 前面的既有阶段...  
  - name: static_lint_and_style_check  
    desc: "静态分析与代码风格检查报告"  
    task:  
      - "目标：完成 DUT 的静态检查/Lint，并输出报告"  
      - "第 1 步：运行 lint 工具（按项目需要）"  
      - "第 2 步：将结论整理为 <OUT>/{DUT}_lint_report.md (中文)"  
      - "第 3 步：用 Check 校验报告是否存在且格式规范"  
    checker:  
      - name: markdown_file_check  
        class: "UnityChipCheckerMarkdownFileFormat"  
        args:  
          markdown_file_list: "{OUT}/{DUT}_lint_report.md" # MD 文件路  
        ↵ 径或列表  
          no_line_break: true # 禁止字面量 "\n" 作为换行  
        reference_files: []  
        output_files:  
          - "{OUT}/{DUT}_lint_report.md"  
        skip: false  
    # ... 后续既有阶段...
```

### 9.2.3 减少子阶段

- 场景：在“功能规格分析与测试点定义”中，临时不执行“功能点定义(FC)”子阶段。
- 推荐做法：运行时使用 CLI --skip 跳过该索引；若需长期配置，复制默认 stage：列表到你的 config.yaml，在父阶段 functional\_specification\_analysis 的 stage：子列表里移除对应子阶段条目，或为该子阶段加 skip: true。

子阶段移除（片段示例，仅展示父阶段结构与其子阶段列表）：

```

stage:
  - name: functional_specification_analysis
    desc: "功能规格分析与测试点定义"
    task:
      - "目标: 将芯片功能拆解成可测试的小块, 为后续测试做准备"
        # ... 省略父阶段任务...
    stage:
      - name: functional_grouping # 保留 FG 子阶段
        # ... 原有配置...
        # - name: function_point_definition # 原来的 FC 子阶段 (此行及其内
        → 容整体删除, 或在其中加 skip: true)
      - name: check_point_design # 保留 CK 子阶段
        # ... 原有配置...
        # ... 其他字段...

```

## 小贴士

- 仅需临时跳过: 用 `--skip`/`--unskip` 最快, 无需改配置文件。
- 需要永久增删: 复制默认 `stage:` 列表到项目 `config.yaml`, 编辑后提交到仓库; 注意列表是整体覆盖, 别只贴新增/删减的片段。
- 新增阶段的检查器可复用现有类 (如 `Markdown`/`Fixture`/`API`/`Coverage`/`TestCase` 等), 也可以扩展自定义检查器 (放在 `vagent/checkers/` 并以可导入路径填写到 `clss`)。

## 9.3 定制校验器 (checker)

### 原理说明

- 每个 (子) 阶段下的 `checker:` 是一个列表; 执行 `Check` 时会依次运行该列表里的所有检查器。
- 配置字段:
  - `name:` 该检查器在阶段内的标识 (便于阅读/日志)
  - `clss:` 检查器类名; 短名默认从 `vagent.checkers` 命名空间导入, 也可写完整模块路径 (如 `mypkg.mychk.MyChecker`)
  - `args:` 传给检查器构造函数的参数, 支持模板变量 (如 `{OUT}`、`{DUT}`)
  - `extra_args:` 可选, 部分检查器支持自定义提示/策略 (如 `fail_msg`、`batch_size`、`pre_report_file` 等)
- 解析与实例化: `vagent/stage/vstage.py` 会读取 `checker:`, 按 `clss/args` 生成实例; 运行期由 `ToolStdCheck/Check` 调用其 `do_check()`。

- 合并语义：配置合并时列表是“整体替换”，要在项目 config.yaml 修改某个阶段的 checker:，建议复制该阶段条目并完整替换其 checker: 列表。

### 9.3.1 增加 checker

在“功能规格分析与测试点定义”父阶段，新增“文档格式检查”，确保 {OUT}/{DUT}\_functions\_and\_checks.md 没有把换行写成字面量 \n。

```
# 片段示例：需要放入你的完整 stage 列表对应阶段中
- name: functional_specification_analysis
  desc: "功能规格分析与测试点定义"
  # ...existing fields...
  output_files:
    - "{OUT}/{DUT}_functions_and_checks.md"
  checker:
    - name: functions_and_checks_doc_format
      class: "UnityChipCheckerMarkdownFileFormat"
      args:
        markdown_file_list: "{OUT}/{DUT}_functions_and_checks.md" # 功能规格/检查点文档
        no_line_break: true # 禁止字面量 "\n"
  stage:
    # ... 子阶段 FG/FC/CK 原有配置...
```

(可扩展) 自定义检查器 (最小实现，放在 vagent/checkers/unity\_test.py)

很多场景下“增加的 checker”并非复用已有检查器，而是需要自己实现一个新的检查器。最小实现步骤：

1. 新建类并继承基类 vagent.checkers.base.Checker
2. 在 \_\_init\_\_ 里声明你需要的参数 (与 YAML args 对应)
3. 实现 do\_check(self, timeout=0, \*\*kw) -> tuple[bool, object]，返回 (是否通过, 结构化消息)
4. 如需读/写工作区文件，使用 self.get\_path(rel) 获取绝对路径；如需跨阶段共享数据，使用 self.smanager\_set\_value/get\_value
5. 若想用短名 class 引用，请在 vagent/checkers/\_\_init\_\_.py 导出该类 (或在 class 写完整模块路径)

最小代码骨架 (示例)：

```
# 文件: vagent/checkers/unity_test.py
from typing import Tuple
import os
from vagent.checkers.base import Checker

class UnityChipCheckerMyCustomCheck(Checker):
    def __init__(self, target_file: str, threshold: int = 1, **kw):
        self.target_file = target_file
        self.threshold = threshold

    def do_check(self, timeout=0, **kw) -> Tuple[bool, object]:
        """ 检查 target_file 是否存在并做简单规则校验。 """
        real = self.get_path(self.target_file)
        if not os.path.exists(real):
            return False, {"error": f"file '{self.target_file}' not found"}
        # TODO: 这里写你的具体校验逻辑, 例如统计/解析/比对等
        return True, {"message": "MyCustomCheck passed"}
```

在阶段 YAML 中引用 (与“增加一个 checker”一致):

```
checker:
  - name: my_custom_check
    class: "UnityChipCheckerMyCustomCheck" # 若未在 __init__.py 导出, 写完整
    ↳ 路径 mypkg.mychk.UnityChipCheckerMyCustomCheck
    args:
      target_file: "{OUT}/{DUT}_something.py"
      threshold: 2
    extra_args:
      fail_msg: " 未满足自定义阈值, 请完善实现或调低阈值。" # 可选: 通过
    ↳ extra_args 自定义默认失败提示
```

进阶提示 (按需):

- 长时任务/外部进程: 在运行子进程时调用 `self.set_check_process(p, timeout)`, 即可用工具 `KillCheck/StdCheck` 管理与查看进程输出。
- 模板渲染: 实现 `get_template_data()` 可将进度/统计渲染到阶段标题与任务文案中。
- 初始化钩子: 实现 `on_init()` 以在阶段开始时加载缓存/准备批任务 (与 Batch 系列 checker 一致)。

### 9.3.2 删除 checker

如临时不要“第 2 阶段基本信息文档格式检查”，将该阶段的 checker：置空或移除该项：

```
- name: dut_function_understanding
  desc: "{DUT} 功能理解"
  # ...existing fields...
  checker: [] # 删除原本的 markdown_file_check
```

### 9.3.3 修改 checker

把“行覆盖率检查”的阈值从 0.9 调整到 0.8，并自定义失败提示：

```
- name: line_coverage_analysis_and_improvement
  desc: "代码行覆盖率分析与提升 {COVERAGE_COMPLETE}"
  # ...existing fields...
  checker:
    - name: line_coverage_check
      class: "UnityChipCheckerTestCaseWithLineCoverage"
      args:
        doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"
        doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
        test_dir: "{OUT}/tests"
        min_line_coverage: 0.8 # 调低阈值
      extra_args:
        fail_msg: "未达到 80% 的行覆盖率，请补充针对未覆盖行的测试。"
```

可选：自定义检查器类

- 在 `vagent/checkers/` 新增类，继承 `vagent.checkers.base.Checker` 并实现 `do_check()`；
- 在 `vagent/checkers/_init_.py` 导出类后，可在 `class` 用短名；或直接写完整模块路径；
- `args` 中的字符串支持模板变量渲染；`extra_args` 可用于自定义提示文案（具体视检查器实现而定）。

### 9.3.4 常用 checker 参数（结构化）

以下参数均来自实际代码实现 (`vagent/checkers/unity_test.py`)，名称、默认值与类型与代码保持一致；示例片段可直接放入阶段 YAML 的 `checker[]`.`args`。

### 9.3.4.1 UnityChipCheckerMarkdownFileFormat

- 参数：
  - markdown\_file\_list (str | List[str]): 要检查的 Markdown 文件路径或路径列表。
  - no\_line\_break (bool, 默认 false): 是否禁止把换行写成字面量 \n; true 表示禁止。
- 示例：

```
args:  
  markdown_file_list: "{OUT}/{DUT}_basic_info.md"  
  no_line_break: true
```

### 9.3.4.2 UnityChipCheckerLabelStructure

- 参数：
  - doc\_file (str)
  - leaf\_node ( “FG” | “FC” | “CK” )
  - min\_count (int, 默认 1)
  - must\_have\_prefix (str, 默认 “FG-API” )
  - data\_key (str, 可选)
- 示例：

```
args:  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  leaf_node: "CK"  
  data_key: "COVER_GROUP_DOC_CK_LIST"
```

### 9.3.4.3 UnityChipCheckerDutCreation

- 参数：
  - target\_file (str)
- 示例：

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"
```

#### 9.3.4.4 UnityChipCheckerDutFixture

- 参数:

- `target_file` (str)

- 示例:

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"
```

#### 9.3.4.5 UnityChipCheckerEnvFixture

- 参数:

- `target_file` (str)
  - `min_env` (int, 默认 1)
  - `force_bundle` (bool, 当前未使用)

- 示例:

```
args:  
  target_file: "{OUT}/tests/{DUT}_api.py"  
  min_env: 1
```

#### 9.3.4.6 UnityChipCheckerDutApi

- 参数:

- `api_prefix` (str)
  - `target_file` (str)
  - `min_apis` (int, 默认 1)

- 示例:

```
args:  
  api_prefix: "api_{DUT}_"  
  target_file: "{OUT}/tests/{DUT}_api.py"  
  min_apis: 1
```

#### 9.3.4.7 UnityChipCheckerCoverageGroup

- 参数:

- `test_dir` (str)
- `cov_file` (str)
- `doc_file` (str)
- `check_types` (str|List[str])

- 示例:

```
args:  
  test_dir: "{OUT}/tests"  
  cov_file: "{OUT}/tests/{DUT}_function_coverage_def.py"  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  check_types: ["FG", "FC", "CK"]
```

### 9.3.4.8 UnityChipCheckerCoverageGroupBatchImplementation

- 参数:

- `test_dir` (str)
- `cov_file` (str)
- `doc_file` (str)
- `batch_size` (int)
- `data_key` (str)

- 示例:

```
args:  
  test_dir: "{OUT}/tests"  
  cov_file: "{OUT}/tests/{DUT}_function_coverage_def.py"  
  doc_file: "{OUT}/{DUT}_functions_and_checks.md"  
  batch_size: 20  
  data_key: "COVER_GROUP_DOC_CK_LIST"
```

### 9.3.4.9 UnityChipCheckerTestTemplate

- 基类参数: `doc_func_check` (str), `test_dir` (str), `min_tests` (int, 默认 1), `timeout` (int, 默认 15)
- 扩展参数 (extra\_args): `batch_size` (默认 20), `ignore_ck_prefix` (str), `data_key` (str)

- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  test_dir: "{OUT}/tests"  
  ignore_ck_prefix: "test_api_{DUT}_"  
  data_key: "TEST_TEMPLATE_IMP_REPORT"  
  batch_size: 20
```

#### 9.3.4.10 UnityChipCheckerDutApiTest

- 参数:

- api\_prefix (str)
- target\_file\_api (str)
- target\_file\_tests (str)
- doc\_func\_check (str)
- doc\_bug\_analysis (str)
- min\_tests (int, 默认 1)
- timeout (int, 默认 15)

- 示例:

```
args:  
  api_prefix: "api_{DUT}_"  
  target_file_api: "{OUT}/tests/{DUT}_api.py"  
  target_file_tests: "{OUT}/tests/test_{DUT}_api*.py"  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"
```

#### 9.3.4.11 UnityChipCheckerBatchTestsImplementation

- 基类参数: doc\_func\_check (str), test\_dir (str), doc\_bug\_analysis (str), ignore\_ck\_prefix (str), timeout (int, 默认 15)
- 进度参数: data\_key (str, 必填)
- 扩展参数 (extra\_args): batch\_size (默认 5), pre\_report\_file (str)
- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"  
  test_dir: "{OUT}/tests"  
  ignore_ck_prefix: "test_api_{DUT}_"  
  batch_size: 10  
  data_key: "TEST_TEMPLATE_IMP_REPORT"  
  pre_report_file: "{OUT}/{DUT}/.TEST_TEMPLATE_IMP_REPORT.json"
```

### 9.3.4.12 UnityChipCheckerTestCase

- 参数:

- doc\_func\_check (str)
- doc\_bug\_analysis (str)
- test\_dir (str)
- min\_tests (int, 默认 1)
- timeout (int, 默认 15)

- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"  
  test_dir: "{OUT}/tests"
```

### 9.3.4.13 UnityChipCheckerTestCaseWithLineCoverage

- 基础参数同 UnityChipCheckerTestCase
- 额外必需: cfg (dict|Config)
- 额外可选 (extra\_args):

- min\_line\_coverage (float, 默认按配置, 未配置则 0.8)
- coverage\_json (str, 默认 uc\_test\_report/line\_dat/code\_coverage.json)
- coverage\_analysis (str, 默认 unity\_test/{DUT}\_line\_coverage\_analysis.md)
- coverage\_ignore (str, 默认 unity\_test/tests/{DUT}.ignore)

- 示例:

```
args:  
  doc_func_check: "{OUT}/{DUT}_functions_and_checks.md"  
  doc_bug_analysis: "{OUT}/{DUT}_bug_analysis.md"  
  test_dir: "{OUT}/tests"  
  cfg: "<CONFIG_OBJECT_OR_DICT>"  
  min_line_coverage: 0.9
```

提示：上面的示例仅展示 args 片段；实际需置于阶段条目下的 checker[].args。

## 10 定制功能

### 10.1 添加工具与 MCP Server 工具

面向可修改本仓库代码的高级用户，以下说明如何：

- 添加一个新工具（供本地/Agent 内调用）
- 将工具暴露为 MCP Server 工具（供外部 IDE/客户端调用）
- 控制选择哪些工具被暴露与如何调用

涉及关键位置：

- `vagent/tools/uctool.py`: 工具基类 UCTool、`to_fastmcp` (LangChain Tool → FastMCP Tool)
- `vagent/util/functions.py`: `import_and_instance_tools` (按名称导入实例) 、`create_verify_mcps` (启动 FastMCP)
- `vagent/verify_agent.py`: 装配工具清单，`start_mcps` 组合并启动 Server
- `vagent/cli.py / vagent/verify_pdb.py`: 命令行与 TUI 内的 MCP 启动命令

#### 10.1.1 1) 工具体系与装配

- 工具基类 UCTool：
  - 继承 LangChain BaseTool，内置：call\_count 计数、call\_time\_out 超时、流式/阻塞提示、MCP Context 注入 (ctx.info)、防重入等。
  - 推荐自定义工具继承 UCTool，获得更好的 MCP 行为与调试体验。
- 运行期装配 (VerifyAgent 初始化)：
  - 基础工具：RoleInfo、ReadTextFile
  - 嵌入工具：参考检索与记忆（除非 `--no-embed-tools`）

- 文件工具：读/写/查找/路径等（可在 MCP 无文件工具模式下剔除）
- 阶段工具：由 StageManager 按工作流动态提供
- 外部工具：来自配置项 `ex_tools` 与 CLI `--ex-tools`（通过 `import_and_instance_tools` 零参实例化）
- 名称解析：
  - 短名：类/工厂函数需在 `vagent/tools/__init__.py` 导出（例如 `from .mytool import HelloTool`），即可在 `ex_tools` 写 `HelloTool`
  - 全路径：`mypkg.mytools.HelloTool / mypkg.mytools.Factory`

### 10.1.2 2) 添加一个新工具（本地/Agent 内）

规范要求：

- 唯一 name、清晰 description
- 使用 pydantic BaseModel 定义 args\_schema（MCP 转换依赖）
- 实现 \_run（同步）或 \_arun（异步）；继承 UCTool 可直接获得超时、流式与 ctx 注入

示例 1：同步工具（计数问候）

```
from pydantic import BaseModel, Field
from vagent.tools.uctool import UCTool

class HelloArgs(BaseModel):
    who: str = Field(..., description="要问候的人")

class HelloTool(UCTool):
    name: str = "Hello"
    description: str = "向指定对象问候，并统计调用次数"
    args_schema = HelloArgs

    def _run(self, who: str, run_manager=None) -> str:
        return f"Hello, {who}! (called {self.call_count+1} times)"
```

注册与使用：

- 临时：`--ex-tools mypkg.mytools.HelloTool`
- 持久：项目 `config.yaml`

```
ex_tools:
  - mypkg.mytools.HelloTool
```

(可选) 短名注册: 在 `vagent/tools/__init__.py` 导出 `HelloTool` 后, 可写 `--ex-tools HelloTool`。

示例 2: 异步流式工具 (`ctx.info` + 超时)

```
from pydantic import BaseModel, Field
from vagent.tools.uctool import UCTool
import asyncio

class ProgressArgs(BaseModel):
    steps: int = Field(5, ge=1, le=20, description="进度步数")

class ProgressTool(UCTool):
    name: str = "Progress"
    description: str = "演示流式输出与超时处理"
    args_schema = ProgressArgs

    async def _arun(self, steps: int, run_manager=None):
        for i in range(steps):
            self.put_alive_data(f"step {i+1}/{steps}") # 供阻塞提示/日志缓冲
            await asyncio.sleep(0.5)
        return "done"
```

说明: `UCTool.ainvoke` 会在 MCP 模式下注入 `ctx`, 并启动阻塞提示线程; 当 `sync_block_log_to_client=True` 时会周期性 `ctx.info` 推送日志, 超时后返回错误与缓冲日志。

### 10.1.3 3) 暴露为 MCP Server 工具

工具 → MCP 转换 (`vagent/tools/uctool.py::to_fastmcp`):

- 必须: `args_schema` 继承 `BaseModel`; 不支持“注入参数”签名。
- `UCTool` 子类会得到 `context_kwarg=“ctx”` 的 `FastMCP` 工具, 具备流式交互能力。

Server 端启动:

- `VerifyAgent.start_mcps` 组合工具: `tool_list_base + tool_list_task + tool_list_ext + [tool_list_file]`
- `vagent/util/functions.py::create_verify_mcps` 将工具序列转换为 `FastMCP` 工具并启动 `uvicorn (mcp.streamable_http_app())`。

如何选择暴露范围：

- CLI:
  - 启动（含文件工具）: `--mcp-server`
  - 启动（无文件工具）: `--mcp-server-no-file-tools`
  - 地址: `--mcp-server-host`, 端口: `--mcp-server-port`
- TUI 命令: `start_mcp_server [host] [port]` / `start_mcp_server_no_file_ops [host] [port]`

#### 10.1.4 4) 客户端调用流程

FastMCP Python 客户端（参考 `tests/test_mcps.py`）：

```
from fastmcp import Client

client = Client("http://127.0.0.1:5000/mcp", timeout=10)
print(client.list_tools())
print(client.call_tool("Hello", {"who": "UCAgent"}))
```

IDE/Agent(Claude Code、Copilot、Qwen Code 等): 将 `httpUrl` 指向 `http://<host>:<port>/mcp`, 即可发现并调用工具。

#### 10.1.5 5) 生命周期、并发与超时

- 计数: UCTool 内置 `call_count`; 非 UCTool 工具由 `import_and_instance_tools` 包装计数。
- 并发保护: `is_in_streaming/is_alive_loop` 防止重入; 同一实例不允许并发执行。
- 超时: `call_time_out` (默认 20s) + 客户端 `timeout`; 阻塞时可用 `put_alive_data + sync_block_log_to_client=True` 推送心跳。

#### 10.1.6 6) 配置策略与最佳实践

- `ex_tools` 列表为“整体覆盖”，项目 `config.yaml` 需写出完整清单。
- 短名 vs 全路径: 短名更便捷, 全路径适用于私有包不修改本仓库时。
- 无参构造/工厂: 装配器直接调用`(...)(...)`, 复杂配置建议在工厂内部处理(读取环境/配置文件)。
- 文件写权限: MCP 无文件工具模式下不要暴露写类工具; 如需写入, 请在本地 Agent 内使用或显式允许写目录。

**10.1.6.1 通过环境变量注入外部工具 (EX\_TOOLS)** 配置文件支持 Bash 风格环境变量占位: `$(VAR: default)`。你可以让 `ex_tools` 从环境变量注入工具类列表 (支持模块全名或 `vagent.tools` 下的短名)。

1. 在项目的 `config.yaml` 或用户级 `~/.ucagent/setting.yaml` 中写入:

```
ex_tools: $(EX_TOOLS: [])
```

2. 用环境变量提供列表 (必须是可被 YAML 解析的数组字面量):

```
export EX_TOOLS='["SqThink", "HumanHelp"]'  
# 或使用完整类路径:  
# export  
↪ EX_TOOLS='["vagent.tools.extool.SqThink", "vagent.tools.human.HumanHelp"]'
```

3. 启动后本地对话与 MCP Server 中都会出现这些工具。短名需要在 `vagent/tools/__init__.py` 导出; 否则请使用完整模块路径。
4. 与 CLI 的 `--ex-tools` 选项是合并关系 (两边都会被装配)。

### 10.1.7 7) 常见问题排查

- 工具未出现在 MCP 列表: 未被装配 (`ex_tools` 未配置/未导出)、`args_schema` 非 `BaseModel`、`Server` 未按预期启动。
- 调用报“注入参数不支持”：工具定义包含 `LangChain` 的 `injected args`; 请改成显式 `args_schema` 参数。
- 超时: 调大 `call_time_out` 或客户端 `timeout`; 在长任务中输出进度维持心跳。
- 短名无效: 未在 `vagent/tools/__init__.py` 导出; 改用全路径或补导出。

## 11 工具列表

以下为当前仓库内内置工具 (UCTool 家族) 的概览, 按功能类别归纳: 名称 (调用名)、用途与参数说明 (字段: 类型—含义)。

提示:

- 带有“文件写”能力的工具仅在本地/允许写模式下可用; MCP 无文件工具模式不会暴露写类工具。
- 各工具均基于 `args_schema` 校验参数, MCP 客户端将根据 `schema` 生成参数表单。

## 11.1 基础/信息类

- RoleInfo (RoleInfo)
  - 用途: 返回当前代理的角色信息 (可在启动时自定义 role\_info)。
  - 参数: 无
- HumanHelp (HumanHelp)
  - 用途: 向人类请求帮助 (仅在确实卡住时使用)。
  - 参数:
    - \* message: str —求助信息

## 11.2 规划/ToDo 类

- CreateToDo
  - 用途: 创建 ToDo (覆盖旧 ToDo)。
  - 参数:
    - \* task\_description: str —任务描述
    - \* steps: List[str] —步骤 (1–20 步)
- CompleteToDoSteps
  - 用途: 将指定步骤标记为完成, 可附加备注。
  - 参数:
    - \* completed\_steps: List[int] —完成的步骤序号 (1-based)
    - \* notes: str —备注
- UndoToDoSteps
  - 用途: 撤销步骤完成状态, 可附加备注。
  - 参数:
    - \* steps: List[int] —撤销的步骤序号 (1-based)
    - \* notes: str —备注
- ResetToDo
  - 用途: 重置/清空当前 ToDo。
  - 参数: 无
- GetToDoSummary / ToDoState
  - 用途: 获取 ToDo 摘要 / 看板状态短语。
  - 参数: 无

### 11.3 记忆/检索类

- SemanticSearchInGuidDoc (SemanticSearchInGuidDoc)
  - 用途: 在 Guide\_Doc/项目文档中做语义检索, 返回最相关片段。
  - 参数:
    - \* query: str — 查询语句
    - \* limit: int — 返回条数 (1-100, 默认 3)
- MemoryPut
  - 用途: 按 scope 写入长时记忆。
  - 参数:
    - \* scope: str — 命名空间/范围 (如 general/task-specific)
    - \* data: str — 内容 (可为 JSON 文本)
- MemoryGet
  - 用途: 按 scope 检索记忆。
  - 参数:
    - \* scope: str — 命名空间/范围
    - \* query: str — 查询语句
    - \* limit: int — 返回条数 (1-100, 默认 3)

### 11.4 测试/执行类

- RunPyTest (RunPyTest)
  - 用途: 在指定目录/文件下运行 pytest, 支持返回 stdout/stderr。
  - 参数:
    - \* test\_dir\_or\_file: str — 测试目录或文件
    - \* pytest\_ex\_args: str — 额外 pytest 参数 (如 “-v --capture=no”)
    - \* return\_stdout: bool — 是否返回标准输出
    - \* return\_stderr: bool — 是否返回标准错误
    - \* timeout: int — 超时秒数 (默认 15)
- RunUnityChipTest (RunUnityChipTest)
  - 用途: 面向 UnityChip 项目封装的测试执行, 产生 toffee\_report.json 等结果。
  - 参数: 同 RunPyTest; 另含内部字段 (workspace/result\_dir/result\_json\_path)。

## 11.5 文件/路径/文本类

- SearchText (SearchText)
  - 用途：在工作区内按文本搜索，支持通配与正则。
  - 参数：
    - \* pattern: str — 搜索模式（明文/通配/正则）
    - \* directory: str — 相对目录（为空则全仓；填文件则仅搜该文件）
    - \* max\_match\_lines: int — 每个文件返回的最大匹配行数（默认 20）
    - \* max\_match\_files: int — 返回的最大文件数（默认 10）
    - \* use\_regex: bool — 是否使用正则
    - \* case\_sensitive: bool — 区分大小写
    - \* include\_line\_numbers: bool — 返回是否带行号
- FindFiles (FindFiles)
  - 用途：按通配符查找文件。
  - 参数：
    - \* pattern: str — 文件名模式 (fnmatch 通配)
    - \* directory: str — 相对目录（为空则全仓）
    - \* max\_match\_files: int — 返回最大文件数（默认 10）
- PathList (PathList)
  - 用途：列出目录结构（可限制深度）。
  - 参数：
    - \* path: str — 目录（相对 workspace）
    - \* depth: int — 深度 (-1 全部, 0 当前)
- ReadBinFile (ReadBinFile)
  - 用途：读取二进制文件（返回 [BIN\_DATA]）。
  - 参数：
    - \* path: str — 文件路径（相对 workspace）
    - \* start: int — 起始字节（默认 0）
    - \* end: int — 结束字节（默认 -1 表示 EOF）
- ReadTextFile (ReadTextFile)
  - 用途：读取文本文件（带行号，返回 [TXT\_DATA]）。
  - 参数：
    - \* path: str — 文件路径（相对 workspace）

- \* start: int —起始行 (1-based, 默认 1)
- \* count: int —行数 (-1 到文件末尾)
- EditTextFile (EditTextFile)
  - 用途: 编辑/创建文本文件, 模式: replace/overwrite/append.
  - 参数:
    - \* path: str —文件路径 (相对 workspace, 不存在则创建)
    - \* data: str —写入的文本 (None 表示清空)
    - \* mode: str —编辑模式 (replace/overwrite/append, 默认 replace)
    - \* start: int —replace 模式的起始行 (1-based)
    - \* count: int —replace 模式替换行数 (-1 到末尾, 0 插入)
    - \* preserve\_indent: bool —replace 时是否保留缩进
- CopyFile (CopyFile)
  - 用途: 复制文件; 可选覆盖。
  - 参数:
    - \* source\_path: str —源文件
    - \* dest\_path: str —目标文件
    - \* overwrite: bool —目标存在时是否覆盖
- MoveFile (MoveFile)
  - 用途: 移动/重命名文件; 可选覆盖。
  - 参数:
    - \* source\_path: str —源文件
    - \* dest\_path: str —目标文件
    - \* overwrite: bool —目标存在时是否覆盖
- DeleteFile (DeleteFile)
  - 用途: 删除文件。
  - 参数:
    - \* path: str —文件路径
- CreateDirectory (.CreateDirectory)
  - 用途: 创建目录 (递归)。
  - 参数:
    - \* path: str —目录路径
    - \* parents: bool —递归创建父目录
    - \* exist\_ok: bool —已存在是否忽略

- ReplaceStringInFile (ReplaceStringInFile)
  - 用途: 精确字符串替换 (强约束匹配; 可新建文件)。
  - 参数:
    - \* path: str — 目标文件
    - \* old\_string: str — 需要被替换的完整原文 (含上下文, 精确匹配)
    - \* new\_string: str — 新内容
- GetFileInfo (GetFileInfo)
  - 用途: 获取文件信息 (大小、修改时间、人类可读尺寸等)。
  - 参数:
    - \* path: str — 文件路径

## 11.6 扩展示例

- SimpleReflectionTool (SimpleReflectionTool)
  - 用途: 示例型“自我反思”工具 (来自 extool.py), 可作为扩展参考。
  - 参数:
    - \* message: str — 自我反思文本

备注:

- 工具调用超时默认 20s (具体工具可重写); 长任务请周期性输出进度避免超时。
- MCP 无文件工具模式下默认不暴露写类工具; 如需写入, 建议在本地 Agent 模式或按需限制可写目录。