实验四：软件分析

0. 实验对象与产物位置

本次实验以“Python 美食点单项目”为对象，分别在**注入前**与**注入后**两份代码上进行分析对比。按照实验要求，额外在项目中**植入若干缺陷**并评估各工具的 TP/FP 表现。

- **注入前代码**：项目根目录（未注入缺陷）
- **注入后代码**：`exp4_injected/`（在副本中注入 8 处缺陷，尽量放在不常走路径，避免影响正常主流程）

本实验已将各工具的原始输出与人工标注整理到 `flaw_report/`（并遵循作业要求的 `true_positive.json` / `false_positive.json` 结构，每个工具最多挑选 10 条进行核对与标注）。

- **Pylint/Bandit（目录名沿用示例中的 CppCheck 作为“静态分析工具”文件夹）**：`flaw_report/CppCheck/`
  - 原始报告：`flaw_report/CppCheck/pylint.json`、`flaw_report/CppCheck/bandit.json`
  - 标注：`flaw_report/CppCheck/true_positive.json`、`flaw_report/CppCheck/false_positive.json`
- **LLM（ChatGPT）**：`flaw_report/ChatGPT/`
  - 结果记录：`flaw_report/ChatGPT/record.txt`
  - 标注：`flaw_report/ChatGPT/true_positive.json`、`flaw_report/ChatGPT/false_positive.json`
- **Semgrep（作为“形式化工具不可用时的替代静态分析工具”）**：`flaw_report/Semgrep/`
  - 原始报告：`flaw_report/Semgrep/semgrep_after_registry.json`
  - 标注：`flaw_report/Semgrep/true_positive.json`、`flaw_report/Semgrep/false_positive.json`

> 注：Pylint/Bandit 的“注入后”原始输出在：`exp4_injected/static_analysis_reports_exp4/`
> Semgrep 的“注入前/后（Registry packs）”原始输出分别在：
>
> - 注入前：`semgrep_reports_registry/semgrep_before_registry.json`
> - 注入后：`exp4_injected/semgrep_reports_registry/semgrep_after_registry.json`

---

1. 三种分析手段的检测过程

1.1 Pylint（偏代码质量/可维护性）

- **选择原因**：Pylint 对 Python 工程的“潜在 bug/可维护性风险”覆盖比较广，尤其是异常处理、默认参数、未使用变量等问题，适合当作代码质量基线。
- **运行方式**：使用 `.pylintrc` 控制输出格式与忽略目录；在 `exp4_injected/` 中运行：
  `python -m pylint app scripts tests --output-format=json > static_analysis_reports_exp4/pylint.json`
- **结果概况**：

- 注入前：287 条（`static_analysis_reports/pylint.json`）
- 注入后：292 条（`exp4_injected/static_analysis_reports_exp4/pylint.json`）
- 变化：**+5 条**

- **观察**：Pylint 的报告中有很大比例是 convention（如空白/导入顺序）。因此在后续 TP/FP 评估时，我主要从 warning 中挑选更像“真实缺陷”的条目进行核对与标注，避免被大量风格项淹没。

1.2 Bandit（偏安全基线/危险模式）

- **选择原因**：Bandit 更偏“安全模式匹配”，报告量通常更小但更聚焦，适合作为最低成本的安全基线扫描。
- **运行方式**：使用 `bandit.yaml` 排除无关目录；在 `exp4_injected/` 中运行：
  `python -m bandit -r app scripts -c bandit.yaml -f json -o static_analysis_reports_exp4/bandit.json`
- **结果概况**：
  - 注入前：1 条（`static_analysis_reports/bandit.json`）
  - 注入后：4 条（`exp4_injected/static_analysis_reports_exp4/bandit.json`）
  - 变化：**+3 条**

1.3 Semgrep（规则可扩展，作为替代工具）

- **选择原因**：由于 Python 项目使用形式化工具（如 ESBMC）在工程规模与依赖上成本较高，按实验要求改用“另一款静态分析工具”作为替代；Semgrep 的优势在于可通过规则包快速扩展覆盖面，适合做横向对比。
- **运行方式**：
  - `--config auto`：注入前/后均 0 findings（对本次注入缺陷覆盖不足）
  - 改用 Registry packs（并禁用 gitignore 影响）：
    `--config p/security-audit --config p/python --config p/bandit --no-git-ignore --metrics off`
- **结果（Registry packs）**：
  - 注入前：0 findings（`semgrep_reports_registry/semgrep_before_registry.json`）
  - 注入后：1 finding（`exp4_injected/semgrep_reports_registry/semgrep_after_registry.json`）
  - 命中规则：`gitlab.bandit.B101`（在非测试代码中使用 `assert`）

1.4 LLM（ChatGPT：语义审计）

```
你是 Python 项目的“安全与可靠性审计员”，项目名为“美食点单程序”。我会分批提供源文件内容（包含行号或可定位的片段）。你需要以“找出并解释我刻意植入的缺陷”为首要目标，并在此基础上最多额外补充 0–2 条你高度确信的真实缺陷（总计 ≤ 10 条）。

## 核心目标
1) 寻找“高可信”真实缺陷；如果证据不足就不要硬找。
2) 忽略纯风格问题：空格/换行/import 顺序/命名/行长等一律不报。
```

证据与约束（必须遵守）
- 不能凭空猜测：每条问题必须直接引用我提供的代码片段作为证据；如果我没提供某段代码，你不能基于“应该有”去推断。
- 每条问题必须回答：缺陷如何触发、外部输入从哪里来（请求参数/表单/函数参数/环境变量等）、影响是什么、修复怎么做。
- 如果我未提供精确行号：`line` 填 `null`，并在 `detail` 里用“文件 + 函数/类名 + 关键代码片段”定位。
- 如果你需要更多上下文才能确定缺陷（例如变量来源、鉴权逻辑是否存在、SQL 是否参数化），请在 `detail` 中明确写出“缺少哪些上下文”和“需要我再提供哪些文件/函数片段”；但输出仍必须是 JSON。

优先关注的缺陷类型（从高到低）
1) 代码执行注入：`eval` / `exec` 使用外部输入（CWE-94）
2) 命令注入：`subprocess.*(shell=True)` 且拼接外部输入（CWE-78）
3) SQL 注入：拼接 SQL / 未参数化（CWE-89）
4) 鉴权/越权：缺少 admin 检查、可访问他人订单（CWE-285）
5) 输入未校验：数量/价格/ID 允许负数/超大/空值（CWE-20）
6) 异常吞掉：`except Exception: pass/continue` 导致状态不一致（CWE-703）
7) 资源泄漏：文件/连接未关闭，异常路径提前 return（CWE-772）
8) 不安全反序列化：`pickle.loads` 等处理外部输入（CWE-502）

工作步骤（你要按这个顺序思考与输出）
A. 对每个缺陷：解释
   - 触发条件：什么输入/什么路径会走到这里?
   - 输入来源：外部输入从哪来？（HTTP 参数/JSON body/Query/Form/Path/headers/环境变量/文件等）
   - 影响：可能导致什么后果（RCE/命令执行/数据泄露/越权/库存异常/订单错误/死锁/资源耗尽等）
   - CWE：选择最贴近的 CWE 编号
   - 修复：给出可落地的修复建议（参数化、移除 shell=True、鉴权校验、输入范围检查、with/try-finally、细化异常类型等）
B. 对所有问题排序：先 critical/high，再 medium/low。总数 ≤ 10。

输出格式（严格）
只输出一个 JSON 数组（不得有任何多余文字），每个元素结构如下：

```
{
  "id": "LLM-001",
  "name": "SQL injection via string concatenation",
  "type": "security|bug|logic|reliability",
  "severity": "critical|high|medium|low",
  "confidence": "high|medium|low",
  "file": "app/db.py",
  "line": 123,
  "at": "cursor.execute(f\"SELECT ... {user_id} ...\")",
  "detail": "说明缺陷如何触发，外部输入从哪里来（请求参数/表单/函数参数），并指出你依赖的证据代码片段。",
  "impact": "可能导致什么后果（RCE/命令执行/数据泄露/越权/库存异常/订单错误等）。",
  "cwe": "CWE-xx",
  "fix": "给出可落地修复方案（参数化/去掉shell=True/鉴权检查/输入校验/with/try-finally等）。"
}
```

字段要求：
- at：必须是你引用的“关键表达式/关键行”（从我提供的代码原样摘取，必要时可省略不关键部分用 `...`）。
- line：没有行号就填 null，并在 detail 用“函数名+附近代码”定位。

```
- confidence：如果缺少输入来源/调用路径/鉴权上下文等关键信息，必须降低为 medium/low，并在 detail 说明缺了什么上下文。 注意记录你的分析结果
```

- **方式**：对关键 API/service 做语义审计，重点关注鉴权/越权/身份绑定等规则工具不擅长的“语义层问题”，输出记录在 `record.txt`。
- **结果概况**：共 6 条，主要集中在鉴权/越权（IDOR/身份绑定/权限边界）等问题上。
- **说明**：LLM 的结论会强依赖“我提供给它的上下文是否足够”。优秀报告里也提到：如果代码片段不全或需求说明不清晰，LLM 很容易产生偏保守判断或误报。

---

2. TP/FP 评估（真实报告与误报）

2.1 判定标准

- **TP（True Positive）**：工具报告的问题在代码中确实存在，且能从代码直接证明风险/错误路径。
- **FP（False Positive）**：工具提示在本项目语境下不构成缺陷（例如纯风格、工程折中、或依赖上下文的保守猜测）。

标注文件已生成（每个工具最多抽取 10 条进行核对与记录）。

- Pylint/Bandit：`flaw_report/CppCheck/true_positive.json`、`flaw_report/CppCheck/false_positive.json`
- LLM：`flaw_report/ChatGPT/true_positive.json`、`flaw_report/ChatGPT/false_positive.json`
- Semgrep：`flaw_report/Semgrep/true_positive.json`、`flaw_report/Semgrep/false_positive.json`

2.2 植入缺陷清单与命中情况（注入后 exp4_injected）

注入点（文件:行）	缺陷类型	Pylint	Bandit	Semgrep（Registry）
`app/services/inventory_service.py:73`	吞异常导致部分成功/状态不一致（CWE-703）	✅	✅	❌
`app/services/exp4_experiment_service.py:17`	资源管理不当：`open` 未使用 `with`（CWE-772）	✅	❌	❌
`app/services/exp4_experiment_service.py:27`	计算错误/逻辑错误（CWE-682）	❌	❌	❌
`app/services/exp4_experiment_service.py:36`	吞异常返回默认值（CWE-703）	✅	✅	❌
`app/services/exp4_experiment_service.py:45`	输入校验缺失示例（CWE-20）	❌	❌	❌
`app/services/exp4_experiment_service.py:48`	可变默认参数（Python 常见陷阱）	✅	❌	❌
`app/services/exp4_experiment_service.py:60`	用 `assert` 做运行时校验（CWE-754）	❌	✅	✅
`app/main.py:65`	`open` 未关闭（CWE-772）	（未纳入标注）	❌	❌

> 注：这张表以“本次实际输出”为准，强调的是“实验可验证性”。其中未命中的点会在第 3 部分讨论原因与局限。

2.3 误报率与原因（基于本次标注集）

> 这里的误报率基于我抽样出来用于报告的标注集（全量报告里风格项过多，没有逐条核对意义）。这一点也符合实验要求“每个工具最多检查 10 条”。

- **Pylint/Bandit（`flaw_report/CppCheck/`）**
  - TP：5 条；FP：3 条
  - 误报率：3 / (5 + 3) = **37.5%**
  - 误报原因（个人观察）：
    - Pylint 的一些规则更像“工程建议”，在真实项目里可能是折中做法（例如为规避循环依赖或延迟加载而做的导入方式），容易被误判成问题。
    - Bandit 对部分容错写法偏保守（例如批处理里 `try/except/continue`），如果业务允许“尽力而为”，它更像提醒“需要日志/监控”，未必是必然缺陷。
- **Semgrep（`flaw_report/Semgrep/`，Registry packs）**
  - TP：1 条；FP：0 条；误报率：0%
  - 更显著的问题是**漏报**：只命中 1 个点。说明 Semgrep 的能力强依赖“规则选择/是否自定义规则”，在默认规则包不对味时会出现“看不见你注入的大部分缺陷”的情况（上限高，但配置成本也更高）。
- **LLM（`flaw_report/ChatGPT/`）**
  - TP：6 条；FP：0 条（本次没有为了凑数强行写 FP）
  - 需要如实说明的风险：LLM 很依赖上下文信息。如果项目存在全局鉴权中间件/统一校验逻辑，但没提供给模型，它可能把“路由里没校验”直接判成漏洞；这类偏差本质上是“输入信息不完整导致的保守判断”。

---

3. 整体评价与推荐组合

这次跑下来，四种方法大相径庭：

- **Pylint** 的覆盖面确实大，但对我这个项目来说，报告里大量都是 convention（空白、导入顺序、命名等）。如果不先做筛选，很容易把真正需要关注的 warning 淹没。它更适合当“代码质量基线”，我最终只把和 bug 风险相关的条目拿来做 TP/FP。
- **Bandit** 的输出很少，但相对集中在“安全/危险模式”。注入缺陷后它能稳定命中一部分点（比如 assert、异常处理相关），作为安全基线是够用的；缺点是它对业务逻辑类问题基本无能为力。
- **Semgrep** 这次最大的结论不是误报，而是**覆盖依赖规则包**：`auto` 基本扫不出东西；换成 Registry packs 后能命中 1 条，但对我这批“可靠性/逻辑/资源管理”注入缺陷仍然漏得多。它要想发挥上限，基本离不开“选对规则 + 必要时补自定义规则”（这点和优秀报告里强调的“工具要结合场景取舍”是一致的）。
- **LLM** 在“鉴权/越权/身份绑定”这种语义问题上确实更敏感，能补规则工具的短板。但它的结论强依赖我给它的上下文：如果我没把全局中间件、权限约定、路由层封装说明清楚，它会倾向于保守判定。所以我这里把 LLM 的输出当作“候选问题列表”，最后必须回到代码逐条核实再定 TP/FP。

推荐组合：

- **日常开发**：Pylint + Bandit
  - Pylint 用来兜底常见 bug 风险（主要看 warning 级别，convention 不作为重点）；Bandit 做安全基线，保证明显危险模式不漏。

- **上线前/做实验对比**：在上面基础上加 Semgrep
    - 直接用现成规则包先扫一遍，命中少就别硬吹，把“规则选择导致覆盖变化”如实写进报告；如果时间允许，补 1–2 条自定义规则针对项目特有的模式（收益会明显高于盲扫）。
- **语义类漏洞**：单独加一轮 LLM review（但输出必须人工复核）
    - 重点放在鉴权边界、IDOR、身份绑定、输入校验等，作为“补充发现渠道”，不要当成最终结论。