Claude 认证架构师 — 基础认证

学习指南（基于官方考试指南）

简介

Claude 认证架构师 — 基础认证证明专家能够在实施基于 Claude 的实际解决方案时做出合理的权衡决策。考试评估 Claude Code、Claude Agent SDK、Claude API 和模型上下文协议（MCP）的基础知识——这些是使用 Claude 构建生产应用的核心技术。

考题基于真实的行业场景：构建客户支持代理系统、设计多代理研究管道、将 Claude Code 集成到 CI/CD、创建开发者生产力工具，以及从非结构化文档中提取结构化数据。

目标候选人

理想候选人是**解决方案架构师**，负责设计和交付基于 Claude 的生产应用。您应具备至少 6 个月的实践经验，涵盖：

- **Claude Agent SDK** — 多代理编排、委派子代理、工具集成、生命周期钩子
- **Claude Code** — CLAUDE.md、MCP 服务器、Agent 技能、规划模式
- **模型上下文协议（MCP）** — 用于后端集成的工具和资源
- **提示工程** — JSON 模式、少样本示例、数据提取模板
- **上下文窗口** — 处理长文档、多代理上下文传递
- **CI/CD 管道** — 自动代码审查、测试生成
- **升级与可靠性** — 错误处理、人在回路

考试格式

参数	值
题型	单选题（4 选 1）
评分	100–1000 分制，及格分数 **720 分**
猜测惩罚	无（请回答每道题！）
场景	从 8 个可能场景中随机抽取 4 个

考试内容：5 个领域

领域	权重
1. 代理架构与编排	**27%**
2. 工具设计与 MCP 集成	**18%**
3. Claude Code 配置与工作流	**20%**
4. 提示工程与结构化输出	**20%**
5. 上下文管理与可靠性	**15%**

考试场景

场景 1：客户支持代理

使用 Claude Agent SDK 构建一个处理退货、账单纠纷和账户问题的代理。该代理使用 MCP 工具（`get_customer`、`lookup_order`、`process_refund`、`escalate_to_human`）。目标是 80% 以上的首次联系解决率和适当的升级处理。

场景 2：使用 Claude Code 生成代码

使用 Claude Code 加速开发：代码生成、重构、调试、文档编写。需要将其与自定义斜杠命令和 CLAUDE.md 配置集成，并了解何时使用规划模式。

场景 3：多代理研究系统

协调代理将任务委派给专业子代理：网络研究、文档分析、综合和报告生成。系统必须生成带有引用的完整报告。

场景 4：开发者生产力工具

代理帮助工程师探索陌生代码库、生成样板代码并自动化日常任务。使用内置工具（Read、Write、Bash、Grep、Glob）和 MCP 服务器。

场景 5：用于持续集成的 Claude Code

将 Claude Code 集成到 CI/CD 管道中，用于自动代码审查、测试生成和拉取请求反馈。提示必须设计为最小化误报。

场景 6：结构化数据提取

系统从非结构化文档中提取信息，用 JSON 模式验证输出，并保持高精度。必须正确处理边缘情况。

场景 7：对话式 AI 架构模式

您设计多轮对话系统，涵盖上下文窗口管理、指令在多轮中的持久性、记忆策略、安全执行的工具设计，以及处理模糊或矛盾的用户输入。

场景 8：Agentic AI 工具 *（内容缺失 — 欢迎贡献！）*

考生报告在考试中遇到了这一场景，但本指南尚未涵盖相关内容。如果您在真实考试中遇到过此场景的题目，请在 GitHub Issues 中分享，以便我们添加完整内容。您的贡献将帮助所有备考的人。

官方文档

资源	URL
Claude API — Messages	https://platform.claude.com/docs/en/api/messages
Claude API — Tool Use	https://platform.claude.com/docs/en/build-with-claude/tool-use
Claude API — Message Batches	https://platform.claude.com/docs/en/build-with-claude/message-batches
Claude Agent SDK — 概述	https://platform.claude.com/docs/en/agent-sdk/overview
Claude Agent SDK — Hooks	https://platform.claude.com/docs/en/agent-sdk/hooks
Claude Agent SDK — Subagents	https://platform.claude.com/docs/en/agent-sdk/subagents
Claude Agent SDK — Sessions	https://platform.claude.com/docs/en/agent-sdk/sessions
模型上下文协议（MCP）	https://modelcontextprotocol.io/
MCP — Tools	https://modelcontextprotocol.io/docs/concepts/tools
MCP — Resources	https://modelcontextprotocol.io/docs/concepts/resources
MCP — Servers	https://modelcontextprotocol.io/docs/concepts/servers
Claude Code — 文档	https://code.claude.com/docs/en/overview
Claude Code — CLAUDE.md 与内存	https://code.claude.com/docs/en/memory
Claude Code — 技能（含斜杠命令）	https://code.claude.com/docs/en/skills
Claude Code — Hooks	https://code.claude.com/docs/en/hooks
Claude Code — 子代理	https://code.claude.com/docs/en/sub-agents
Claude Code — MCP 集成	https://code.claude.com/docs/en/mcp
Claude Code — GitHub Actions CI/CD	https://code.claude.com/docs/en/github-actions
Claude Code — GitLab CI/CD	https://code.claude.com/docs/en/gitlab-ci-cd
Claude Code — 无头模式（非交互式）	https://code.claude.com/docs/en/headless

提示工程指南	https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview
扩展思维	https://platform.claude.com/docs/en/build-with-claude/extended-thinking
Anthropic Cookbook（代码示例）	https://github.com/anthropics/anthropic-cookbook

---

第一部分：理论基础

本部分涵盖成功通过考试所需的全部理论知识。内容按技术和概念组织，而非按考试领域划分——这有助于您对每个主题建立更深入的理解。

第 1 章：Claude API — 模型交互基础

> 文档：Messages API | 提示工程

1.1 API 请求结构

Claude API 遵循请求-响应模型。对 Claude Messages API 的每个请求包含：

```
{
  "model": "claude-sonnet-4-6",
  "max_tokens": 1024,
  "system": "你是一个有帮助的助手。",
  "messages": [
    {"role": "user", "content": "你好！"},
    {"role": "assistant", "content": "你好！"},
    {"role": "user", "content": "你好吗？"}
  ],
  "tools": [...],
  "tool_choice": {"type": "auto"}
}
```

关键字段：

- `model` — 模型选择（`claude-opus-4-6`、`claude-sonnet-4-6`、`claude-haiku-4-5`）
- `max_tokens` — 响应中的最大令牌数
- `system` — 系统提示（定义模型行为）
- `messages` — 对话历史（**必须发送完整历史**以保持连贯性）
- `tools` — 可用工具的定义
- `tool_choice` — 工具选择策略

1.2 消息角色

`messages` 数组使用两种对话角色,外加一种指令性角色:

- `user` — 用户消息,包括工具结果(以 `user` 角色消息中的 `tool_result` 内容块形式发送,而不是作为单独的 `tool` 角色)
- `assistant` — 模型响应(发送历史时包含),包括工具使用请求(`tool_use` 内容块)
- `system` — 可以通过顶层的 `system` 字段设置(从第一轮开始生效),也可以在 `messages` 中以 `{"role": "system", ...}` 内联方式添加(从该点开始生效,受特定放置规则约束 — 见下文)

工具结果不是以 `"tool"` 角色的消息发送的。它们是以 `user` 角色的消息发送的,其内容包含一个 `tool_result` 内容块:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "tool_use_id": "toolu_01...",
      "content": "..."
    }
  ]
}
```

`system` 也可以直接作为角色出现在 `messages` 数组中,而不仅仅通过顶层的 `system` 参数。这是为了在不使顶层 `system` 字段的缓存前缀失效的情况下,在对话中途添加指令。它有特定的放置规则:

- 必须紧跟在一个 `user` 轮次(包括含有 `tool_result` 块的轮次)之后,或紧跟在以服务器端工具使用结尾的 `assistant` 轮次之后。
- 必须位于某个 `assistant` 轮次之前,或作为数组的最后一个元素。
- 不能位于 `tool_use` 块与其对应的 `tool_result` 之间 — 这样做会返回 400 错误。
- 较晚出现的 `system` 消息(包括对话中途插入的)对于后续轮次,优先于较早的消息以及顶层的 `system` 字段。

至关重要： 每次 API 请求都必须发送**完整的对话历史**。模型不会在请求之间保持状态——每次调用都是独立的。

1.3 响应中的 `stop_reason` 字段

Claude API 响应包含 `stop_reason`，表示模型停止生成的原因：

值	描述	操作
`"end_turn"`	模型完成了响应	向用户显示结果
`"tool_use"`	模型想要调用工具	执行工具并返回结果
`"max_tokens"`	达到令牌限制	响应被截断；可能需要增加限制

`"stop_sequence"`	遇到停止序列	根据应用程序逻辑处理

对于代理系统，`"tool_use"` 和 `"end_turn"` 最为重要——它们控制代理循环。

1.4 系统提示

系统提示是定义上下文和行为规则的特殊指令。它：

- 不是 `messages` 数组的一部分；通过 `system` 字段单独传递
- 优先于用户消息
- 一次加载，在整个对话中有效
- 用于定义角色、约束和输出格式

考试重点： 系统提示的措辞可能创建意外的工具关联。例如，"始终验证客户"之类的指令可能导致模型过度使用 `get_customer`，即使在不必要的时候。

1.5 上下文窗口

上下文窗口是模型一次可以处理的文本总量（以令牌为单位）。它包括：

- 系统提示
- 完整的消息历史
- 工具定义
- 工具结果

主要上下文窗口问题：

1. **中间丢失效应：** 模型可靠地处理长输入开头和结尾的信息，但可能遗漏中间的细节。缓解方法：将关键信息放在开头或结尾附近。
2. **工具结果积累：** 每次工具调用都会将输出添加到上下文中。如果工具返回 40 多个字段但只需要 5 个，大部分上下文就被浪费了。
3. **渐进式摘要：** 压缩历史时，数值、百分比和日期往往丢失，变成模糊的表达（"大约"、"差不多"、"几个"）。

第 2 章：工具与 `tool_use`

> 文档：Tool Use

2.1 什么是 `tool_use`

`tool_use` 是允许 Claude 调用外部函数的机制。模型不直接运行代码——它生成结构化的工具调用请求；您的代码执行它并返回结果。

2.2 工具定义

每个工具使用 JSON 模式定义：

```
{
  "name": "get_customer",
  "description": "通过邮箱或 ID 查找客户。返回客户档案，包括姓名、邮箱、订单历史和账户状态。使用此工具之前请先于 lookup_order 前验证客户身份。接受邮箱（格式：user@domain.com）或数字 customer_id。",
  "input_schema": {
    "type": "object",
    "properties": {
      "email": {"type": "string", "description": "客户邮箱"},
      "customer_id": {"type": "integer", "description": "数字客户 ID"}
    },
    "required": []
  }
}
```

工具描述的关键要素：

1. **描述是主要选择机制。** LLM 根据工具描述选择工具。最简描述（"获取客户信息"）在工具重叠时会导致错误。
2. **描述中应包含：**
   - 工具的功能和返回内容
   - 输入格式和示例值
   - 边缘情况和约束
   - 何时使用此工具而非类似替代品
3. **避免** 跨工具使用相同或重叠的描述。如果 `analyze_content` 和 `analyze_document` 描述几乎相同，模型会混淆它们。
4. **内置工具与 MCP 工具：** 代理可能更偏好内置工具（Read、Grep）而非功能类似的 MCP 工具。为防止这种情况，需强化 MCP 工具描述——突出内置工具无法提供的具体优势、独特数据或上下文。

2.3 `tool_choice` 参数

`tool_choice` 控制模型如何选择工具：

值	行为	使用时机
`{"type": "auto"}`	模型决定是否调用工具或用文本回答	大多数情况的默认值
`{"type": "any"}`	模型**必须**调用某个工具	需要保证结构化输出时
`{"type": "tool", "name": "extract_metadata"}`	模型**必须**调用特定工具	需要强制第一步/执行顺序时

重要场景：

- `tool_choice: "any"` + 多个提取工具 → 模型选择最佳工具，但仍能获得结构化输出
- 强制选择 → 需要保证特定首个操作（例如，在丰富之前执行 `extract_metadata`）

2.4 结构化输出的 JSON 模式

使用带 JSON 模式的 `tool_use` 是从 Claude 获取结构化输出的**最可靠**方式。它：

- 保证语法上有效的 JSON（无缺失括号，无尾随逗号）
- 强制执行所需结构（必填字段存在）
- **不**保证语义正确性（值仍可能是错误的）

模式设计 — 关键原则：

```
{
  "type": "object",
  "properties": {
    "category": {
      "type": "string",
      "enum": ["bug", "feature", "docs", "unclear", "other"]
    },
    "category_detail": {
      "type": ["string", "null"],
      "description": "category = 'other' 或 'unclear' 时的详情"
    },
    "severity": {
      "type": "string",
      "enum": ["critical", "high", "medium", "low"]
    },
    "confidence": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "optional_field": {
      "type": ["string", "null"],
      "description": "如果在源中未找到信息则为 Null"
    }
  },
  "required": ["category", "severity"]
}
```

模式设计规则：

1. **必填与可选：** 仅当信息始终可用时才将字段标记为必填。必填字段会促使模型在数据缺失时捏造值。
2. **可空字段：** 对可能缺失的信息使用 `"type": ["string", "null"]`。模型可以返回 `null` 而非产生幻觉。
3. **含 `"other"` 的枚举：** 添加 `"other"` + 详情字符串，避免丢失预定义类别之外的数据。
4. **枚举 `"unclear"`：** 用于模型无法自信选择类别的情况——诚实的 `"unclear"` 优于错误的类别。

2.5 语法错误与语义错误

错误类型	示例	缓解措施
语法	无效 JSON，字段类型错误	使用 JSON 模式的 `tool_use`（可消除）
语义	合计不符，值在错误字段中，幻觉	验证检查，带反馈的重试，自我纠正

第 3 章：Claude Agent SDK — 构建代理系统

> 文档：Agent SDK | Hooks | Subagents | Sessions

3.1 什么是代理循环

代理循环是自主任务执行的核心模式。模型不只是回答——它执行一系列操作：

```
1. 向 Claude 发送带工具的请求
2. 接收响应
3. 检查 stop_reason：
   - "tool_use" -> 执行工具，将结果追加到历史，返回步骤 1
   - "end_turn" -> 任务完成，向用户显示结果
4. 重复直到完成
```

这是模型驱动的方法： Claude 根据上下文和先前的工具结果决定下一个要调用的工具。这与动作顺序固定的硬编码决策树不同。

反模式（避免）：

- 解析助手文本以检测完成（"任务已完成"）
- 使用任意迭代限制（例如 `max_iterations=5`）作为主要停止条件
- 检查助手是否产生文本内容作为完成信号

正确方法： 唯一可靠的完成信号是 `stop_reason == "end_turn"`。

3.2 `AgentDefinition` 配置

`AgentDefinition` 是 Claude Agent SDK 中的代理配置对象：

```
agent = AgentDefinition(
    name="customer_support",
    description="处理客户的退货和订单问题",
    system_prompt="你是一个客户支持代理...",
    allowed_tools=["get_customer", "lookup_order", "process_refund",
"escalate_to_human"],
    # 对于协调代理：
    # allowed_tools=["Task", "get_customer", ...]
)
```

关键参数：

- `name` / `description` — 代理的标识和描述
- `system_prompt` — 带指令的系统提示
- `allowed_tools` — 允许的工具列表（最小权限原则）

3.3 轮辐式：协调代理与子代理

多代理架构通常构建为轮辐式拓扑：

```
          协调代理
        /     |      \
  子代理1   子代理2   子代理3
  （搜索）   （分析）   （综合）
```

协调代理负责：

- 将任务分解为子任务
- 决定需要哪些子代理（动态选择）
- 将工作委派给子代理
- 聚合和验证结果
- 处理错误和重试
- 向用户传达结果

关键原则：子代理有隔离的上下文。

- 子代理**不会**自动继承协调代理的对话历史
- 所有必需的上下文必须在子代理提示中**显式传递**
- 子代理不在调用之间共享内存
- 所有通信通过协调代理流动（用于可观察性和错误控制）

3.4 用于生成子代理的 `Task` 工具

子代理通过 `Task` 工具生成：

```
# 协调代理的 allowedTools 必须包含 "Task"
coordinator_agent = AgentDefinition(
    allowed_tools=["Task", "get_customer"]
)
```

显式上下文传递是强制性的：

```
# 不好：子代理没有上下文
Task: "分析文档"

# 好：提示中包含完整上下文
Task: "分析以下文档。
文档：[完整文档文本]
先前搜索结果：[网络搜索结果]
输出格式要求：[模式]"
```

并行生成： 协调代理可以在一个响应中调用多个 `Task`——子代理并行运行：

```
# 一个协调代理响应包含：
Task 1: "搜索关于 X 的文章"
Task 2: "分析文档 Y"
Task 3: "搜索关于 Z 的文章"
# 三个同时运行
```

3.5 Agent SDK 中的钩子

钩子允许在代理生命周期的特定点进行拦截和转换。

PostToolUse 在工具结果提供给模型之前拦截它：

```python
# 示例：规范化来自不同 MCP 工具的日期格式
@hook("PostToolUse")
def normalize_dates(tool_result):
    # 将 Unix 时间戳转换为 ISO 8601
    # 将 "Mar 5, 2025" 转换为 "2025-03-05"
    return normalized_result
```

出站调用拦截钩子阻止违反策略的操作：

```python
# 示例：阻止超过 500 美元的退款
@hook("PreToolUse")
def enforce_refund_limit(tool_call):
    if tool_call.name == "process_refund" and tool_call.args.amount > 500:
        return redirect_to_escalation(tool_call)
```

关键区别：钩子与提示指令

属性	钩子	提示指令
保证	**确定性**（100%）	**概率性**（>90%，非 100%）
使用时机	关键业务规则、财务操作、合规性	一般偏好、建议、格式
示例	阻止 > $500 的退款	"尝试先解决再升级"

规则： 当失败有财务、法律或安全后果时——使用钩子，而非提示。

第 4 章：模型上下文协议（MCP）

> 文档：MCP | 工具 | 资源 | 服务器

4.1 什么是 MCP

模型上下文协议（MCP）是一种用于将外部系统连接到 Claude 的开放协议。MCP 定义了三种主要资源类型：

1. **工具** — 代理可以调用以执行操作的函数（CRUD 操作、API 调用、命令执行）
2. **资源** — 代理可以读取以获取上下文的数据（文档、数据库模式、内容目录）
3. **提示** — 用于常见任务的预定义提示模板

4.2 MCP 服务器

MCP 服务器是实现 MCP 协议并提供工具/资源的进程。当您连接到 MCP 服务器时：

- 所有工具都会自动被发现
- 来自所有已连接服务器的工具同时可用
- 工具描述决定模型将如何使用它们

4.3 配置 MCP 服务器

项目配置（`.mcp.json`） — 供团队使用：

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_TOKEN": "${GITHUB_TOKEN}"
      }
    },
    "jira": {
      "command": "npx",
      "args": ["-y", "mcp-server-jira"],
      "env": {
        "JIRA_TOKEN": "${JIRA_TOKEN}"
      }
    }
  }
}
```

要点：

- `.mcp.json` 存储在项目根目录并通过版本控制进行管理
- 环境变量（`${GITHUB_TOKEN}`）用于存储密钥——令牌本身不会被提交
- 所有项目贡献者均可使用

用户配置（`~/.claude.json`） — 用于个人/实验性服务器：

- 存储在用户的主目录中
- 不通过版本控制共享
- 适合个人实验和测试

选择服务器：

- 对于标准集成（Jira、GitHub、Slack），优先使用现有的社区 MCP 服务器
- 仅为独特的、团队专属的工作流构建自己的服务器

4.4 MCP 中的 `isError` 标志

当 MCP 工具遇到错误时，它在响应中使用 `isError: true`。这向代理发出信号，表明调用失败。

结构化错误（推荐做法）：

```
{
  "isError": true,
  "content": {
    "errorCategory": "transient",
    "isRetryable": true,
    "message": "The service is temporarily unavailable. Timeout while calling the
orders API.",
    "attempted_query": "order_id=12345",
    "partial_results": null
  }
}
```

通用错误（反模式）：

```
{
  "isError": true,
  "content": "Operation failed"
}
```

通用错误不为代理提供任何决策信息——它应该重试、更改查询还是升级？

4.5 MCP 资源

资源是代理可以请求以获取上下文而无需执行操作的数据：

- 内容目录（例如，所有项目任务的列表、层级导航）
- 数据库模式（了解数据结构）
- 文档（API 参考、内部指南）
- 问题/任务摘要

资源优势： 代理无需进行探索性工具调用即可了解存在哪些数据。资源提供了一个即时的"地图"。

第 5 章：Claude Code — 配置与工作流

> 文档：Claude Code | 内存 / CLAUDE.md | 技能 | MCP | 钩子 | 子代理 | GitHub Actions | 无头模式

5.1 CLAUDE.md 层级结构

CLAUDE.md 是 Claude Code 的指令文件。它有三个层级：

```
1. 用户级：~/.claude/CLAUDE.md
   - 仅适用于该用户
   - 不通过版本控制共享
   - 个人偏好和工作风格

2. 项目级：.claude/CLAUDE.md 或根目录 CLAUDE.md
   - 适用于所有项目贡献者
   - 通过版本控制管理
   - 编码标准、测试标准、架构决策

3. 目录级：子目录中的 CLAUDE.md
   - 在处理该目录中的文件时适用
   - 特定于代码库该部分的约定
```

常见错误： 新团队成员未能收到项目指令，因为这些指令被放在了 `~/.claude/CLAUDE.md`（用户级）而非 `.claude/CLAUDE.md`（项目级）。

5.2 `@path` 语法（文件导入）

CLAUDE.md 可以使用 `@path` 引用外部文件，使配置模块化：

```
# 项目 CLAUDE.md

编码标准详见 @./standards/coding-style.md
测试要求详见 @./standards/testing-requirements.md
项目概述详见 @README.md，依赖项详见 @package.json
```

`@path` 规则：

- 在文件路径前紧跟 `@`（无空格）
- 支持相对路径和绝对路径
- 相对路径相对于包含导入的文件进行解析
- 最大导入嵌套深度为 5

这避免了重复，并使每个包只包含相关标准。

5.3 `.claude/rules/` 目录

`.claude/rules/` 是整体式 CLAUDE.md 的替代方案，用于按主题组织规则：

```
.claude/rules/
  testing.md          -- 测试约定
  api-conventions.md  -- API 约定
  deployment.md       -- 部署规则
  react-patterns.md   -- React 模式
```

关键特性：带 `paths` 的 YAML 前置元数据用于条件加载：

```
---
paths: ["src/api/**/*"]
---

对于 API 文件，使用带有显式错误处理的 async/await。
每个端点必须返回标准响应包装器。
```

```
---
paths: ["**/*.test.tsx", "**/*.test.ts"]
---

测试必须使用 describe/it 块。
使用数据工厂而非硬编码。
不要模拟数据库——使用测试数据库。
```

工作原理：

- 规则**仅**在 Claude Code 编辑与 `paths` 模式匹配的文件时加载
- 这节省了上下文和令牌——不相关的规则不会被加载
- Glob 模式让您可以按文件类型应用约定，而与位置无关（非常适合分散在整个代码库中的测试）

何时使用带 `paths` 的 `.claude/rules/` 与目录级 CLAUDE.md：

- 带 `paths` 的 `.claude/rules/` — 当约定适用于分散在多个目录中的文件时（测试、迁移）
- 目录级 CLAUDE.md — 当约定与特定目录绑定且在其他地方不需要时

5.4 自定义斜杠命令与技能

> **注意：** 在当前 Claude Code 版本中，自定义命令（`.claude/commands/`）与技能（`.claude/skills/`）已统一。两种格式都会创建 `/名称` 命令。考试指南引用了 `.claude/commands/` — 该格式仍受支持。

斜杠命令是通过 `/名称` 调用的可复用提示模板：

`.claude/commands/` 格式（旧版，仍受支持）：

```
.claude/commands/
  review.md        -- /review -- 标准代码审查
  test-gen.md      -- /test-gen -- 测试生成
```

`.claude/skills/` 格式（当前）：

```
.claude/skills/
  review/SKILL.md  -- /review -- 带前置元数据配置
  test-gen/SKILL.md
```

项目命令（`.claude/commands/` 或 `.claude/skills/`）：

- 存储在版本控制中，克隆仓库时对所有人可用
- 确保团队内工作流的一致性

用户命令（`~/.claude/commands/` 或 `~/.claude/skills/`）：

- 不通过版本控制共享的个人命令
- 用于个人工作流

5.5 技能 — `.claude/skills/`

技能是通过 SKILL.md 前置元数据配置的高级命令：

```
---
context: fork
allowed-tools: ["Read", "Grep", "Glob"]
argument-hint: "要分析的目录路径"
---

分析指定目录中的代码结构。
输出关于依赖关系和架构模式的报告。
```

前置元数据参数：

参数	说明
`context: fork`	在隔离的子代理中运行技能。详细输出不会污染主会话
`allowed-tools`	限制可用的工具（安全性——例如，如果未被允许，技能无法删除文件）
`argument-hint`	在不带参数调用时提示输入参数的提示语

何时使用技能与 CLAUDE.md：

- **技能** — 按需调用以完成特定任务（审查、分析、生成）
- **CLAUDE.md** — 始终加载的通用标准和约定

个人技能（`~/.claude/skills/`）：

- 以不同名称创建个人变体，不影响队友

5.6 规划模式与直接执行

规划模式：

- 模型仅进行调查和规划；不进行更改
- 使用 Read、Grep、Glob 探索代码库
- 生成供用户审批的实施计划
- 安全探索，无副作用

何时使用规划模式：

- 大规模更改（数十个文件）
- 存在多种可行方案（微服务：如何定义边界？）
- 架构决策（选择哪个框架？什么结构？）
- 不熟悉的代码库（更改前必须先理解）
- 影响 45 个以上文件的库迁移

何时使用直接执行：

- 有明确堆栈跟踪的单文件修复
- 添加一个验证检查
- 已充分理解、无歧义的更改

组合方法：

1. 使用规划模式进行调查和设计
2. 用户审批计划
3. 使用直接执行实施已审批的计划

探索子代理 — 用于探索代码库的专用子代理：

- 将详细输出与主上下文隔离
- 仅返回摘要
- 防止多阶段任务中上下文窗口耗尽

5.7 `/compact` 命令

`/compact` 是用于压缩上下文的内置命令：

- 汇总先前的历史记录以释放上下文窗口
- 用于上下文被大量工具输出填满的长时间调查会话
- 风险：精确的数值、日期和具体细节可能在汇总过程中丢失

5.8 `/memory` 命令

`/memory` 是用于管理会话间内存的内置命令：

- 打开 `CLAUDE.md` 文件进行编辑，允许您保存笔记、偏好设置和上下文
- 信息在会话间持久保存，并在启动时自动加载
- 适用于存储项目约定、用户偏好、常用命令和当前工作上下文
- 无需在每个会话中重复说明相同指令的替代方案

5.9 用于 CI/CD 的 Claude Code CLI

`-p`（或 `--print`）标志：

```
claude -p "Analyze this pull request for security issues"
```

- 非交互式模式：处理提示，打印到标准输出，然后退出
- 不等待用户输入
- 在 CI/CD 管道中运行 Claude 的唯一正确方式

用于 CI 的结构化输出：

```
claude -p "Review this PR" --output-format json --json-schema
'{"type":"object",...}'
```

- `--output-format json` — 以 JSON 格式输出
- `--json-schema` — 根据模式验证输出
- 结果可被解析以自动发布内联 PR 评论

会话上下文隔离： 生成代码的同一 Claude 会话在审查该代码时通常效果较差（模型保留了其推理上下文，不太可能质疑自己的决策）。使用独立实例进行审查。

防止重复评论： 在新提交后重新审查时，将之前的审查结果包含在上下文中，并指示 Claude 仅报告新问题或未解决的问题。

5.10 `fork_session` 与会话管理

`--resume <session-name>` 恢复命名会话：

```
claude --resume investigation-auth-bug
```

- 继续之前保存上下文的对话
- 适用于跨多个会话的长期调查
- 风险：如果自上次会话以来文件已更改，工具结果可能已过时

`fork_session` 从共享上下文创建独立分支：

```
代码库调查
         |
    fork_session
    /           \
方法 A：         方法 B：
Redux           Context API
```

- 两个分支继承分支点之前的上下文
- 之后，它们独立分叉
- 适用于比较方法或测试策略

何时启动新会话而非恢复：

- 工具结果已过时（文件已更改）
- 已过很长时间，上下文已降级
- 以"以下是我们发现的简短摘要：..."重新开始，比用旧工具数据恢复更可靠

第6章：提示工程——高级技巧

> 文档：提示工程 | Anthropic Cookbook

6.1 少样本提示

少样本提示是在提示中包含2–4个输入/输出示例，以演示预期行为。

为什么少样本比文字描述更有效：

- "更精确"之类的模糊指令可以有多种解读方式
- 示例能明确展示预期格式和决策逻辑
- 模型将模式泛化到新情况（而不仅仅是重复示例）

少样本示例的类型及使用时机：

1. **针对模糊场景的示例：**

```
请求："我的订单坏了"
动作：调用 get_customer -> lookup_order -> check status。
理由："坏了"可能指商品损坏；需要订单详情。

请求："帮我找个经理"
动作：立即调用 escalate_to_human。
理由：客户明确要求人工服务。不要尝试自主解决。
```

2. **针对输出格式的示例：**

```
发现示例：
{
  "location": "src/auth/login.ts:42",
  "issue": "用户名参数中存在SQL注入",
  "severity": "critical",
  "suggested_fix": "使用参数化查询"
}
```

3. **区分可接受代码与问题代码的示例：**

```
// 可接受（不标记）：
const items = data.filter(x => x.active);

// 问题（标记）：
const items = data.filter(x => x.active == true); // 使用严格相等 ===
```

4. **从不同文档格式中提取的示例：**

```
带内联引用的文档：
"如研究所示（Smith, 2023），比率为42%。"
-> {"value": "42%", "source": "Smith, 2023", "type": "inline_citation"}

带参考书目引用的文档：
"比率为42%。[1]"
-> {"value": "42%", "source": "reference_1", "type": "bibliography"}
```

5. **非正式计量的示例：**

```
文本："大约两把米"
-> {"amount": "~100g", "original_text": "两把", "precision": "approximate"}

文本："一撮盐"
-> {"amount": "~1g", "original_text": "一撮", "precision": "approximate"}
```

少样本对于提取非正式和非标准计量单位尤其有效，因为这些单位种类繁多，纯粹基于规则的指令难以覆盖。

提示中的格式规范化规则： 当使用严格的JSON模式进行结构化输出时，在提示中添加规范化规则：

```
规范化：
- 日期：始终使用ISO 8601（YYYY-MM-DD）；"昨天" -> 计算绝对日期
- 货币：数字金额+货币代码；"五块钱" -> {"amount": 5, "currency": "CNY"}
- 百分比：小数形式；"一半" -> 0.5
```

这可以防止JSON语法有效但值不一致的语义错误。

6.2 明确标准与模糊指令

差（模糊）：

```
检查代码注释的准确性。
保守一点—只报告高置信度的发现。
```

好（明确标准）：

```
仅在以下情况下将注释标记为有问题：
1. 注释描述的行为与实际代码行为相矛盾
2. 注释引用了不存在的函数或变量
3. TODO/FIXME注释所指的bug已在代码中修复

不要标记：
- 仅在风格上过时的注释
- 措辞轻微不准确的注释
- 缺失的注释（这是单独的类别）
```

用示例定义严重性标准：

```
严重：用户运行时失败
  示例：处理付款时出现NullPointerException

高：安全漏洞
  示例：SQL注入、XSS、缺少授权检查

中：无即时影响的逻辑bug
  示例：排序错误、差一错误

低：代码质量
  示例：重复代码、小数据量的次优算法
```

6.3 提示链

提示链将复杂任务分解为一系列聚焦的步骤：

```
步骤1：分析 auth.ts（仅本地问题）
       -> 输出：auth.ts中的问题列表

步骤2：分析 database.ts（仅本地问题）
       -> 输出：database.ts中的问题列表

步骤3：集成检查（跨文件依赖）
       -> 输出：模块边界处的问题
```

为什么重要：

- 避免**注意力稀释**——当模型同时接收太多文件时，可能会遗漏某些文件中的bug，同时对其他文件只进行浅层评论
- 确保每个文件的分析质量一致
- 允许单独分析跨文件交互

何时使用提示链与动态分解：

- **提示链** — 可预测、可重复的任务（代码审查、文件迁移）
- **动态分解** — 开放式调查，子任务在执行过程中才变得清晰

6.4 "访谈"模式

在实施解决方案之前，Claude提出澄清问题：

```
Claude："在为API实现缓存之前，有几个问题：
1. 您偏好哪种缓存失效策略—TTL还是基于事件？
2. 当缓存不可用时，过时数据是否可以接受？
3. 缓存应该是按用户还是全局的？
4. 预期缓存的数据量是多少？"
```

何时有用：

- 陌生领域（金融科技、医疗、法律系统）

- 有非显而易见影响的任务（缓存策略、故障模式）
- 存在多种可行方案，最佳选择取决于上下文

6.5 验证与带反馈的重试

当提取的数据验证失败时：

```
步骤1：从文档中提取数据
步骤2：验证（Pydantic、JSON Schema、业务规则）
步骤3：如果有错误—带上下文重试：
  - 原始文档
  - 之前（错误的）提取结果
  - 具体错误："字段'total' = 150，但 sum(line_items) = 145。请重新检查值。"
```

重试有效的情况：

- 格式错误（日期格式不正确）
- 结构错误（字段放置位置错误）
- 算术不一致（模型可以重新检查）

重试无效的情况：

- 源文档中不存在该信息
- 所需上下文是外部的（数据在另一个未提供的文档中）

Pydantic作为验证工具： Pydantic是一个用于基于模式的数据验证的Python库。对于考试，关键点是：

- **结构验证：** 在从Claude接收JSON后，在代码中检查类型、必填项、枚举约束
- **语义验证：** 自定义验证器强制执行业务逻辑（条目总和等于合计；start_date < end_date）
- **验证-重试循环：** 在Pydantic验证失败时，构建错误消息并带错误上下文重新提示Claude
- **JSON Schema生成：** Pydantic模型可以为`tool_use`生成JSON Schema，提供单一事实来源

6.6 自我纠正

检测内部矛盾的模式：

```json
{
  "stated_total": "$150.00",
  "calculated_total": "$145.00",
  "conflict_detected": true,
  "line_items": [
    {"name": "Widget A", "price": 75.00},
    {"name": "Widget B", "price": 70.00}
  ]
}
```

模型同时提取声明值和计算值——如果两者不同，`conflict_detected`允许您处理差异。

第7章：消息批处理API

> 文档：消息批处理

7.1 概述

消息批处理API允许您提交批量请求进行异步处理：

属性	值
节省	与同步调用相比节省**50%**
处理窗口	最长**24小时**（无延迟SLA保证）
多轮工具调用	**不支持**（一个请求=一个响应）
关联	`custom_id`字段用于关联请求和响应

7.2 何时使用批处理API与同步API

任务	API	原因
合并前PR检查	**同步**	开发者在等待；24小时不可接受
隔夜技术债务报告	**批处理**	早上需要结果；节省50%
每周安全审计	**批处理**	不紧急；节省50%
交互式代码审查	**同步**	需要立即响应
处理10,000个文档	**批处理**	批量处理；节省显著

7.3 使用 `custom_id`

```
{
  "custom_id": "doc-invoice-2024-001",
  "params": {
    "model": "claude-sonnet-4-6",
    "max_tokens": 1024,
    "messages": [{"role": "user", "content": "从以下内容提取数据：..."}]
  }
}
```

`custom_id`允许您：

- 将结果关联到原始文档

- 在失败时，仅重新提交失败的文档
- 避免重新处理成功的文档

7.4 处理批处理中的失败

1. 提交100个文档的批次
2. 95个成功；5个失败（超出上下文限制）
3. 通过`custom_id`识别失败项
4. 修改策略（例如，将长文档分割成块）
5. 仅重新提交5个失败的文档

7.5 SLA规划

如果您需要在30小时内得到结果，而批处理API最长需要24小时：

- 提交窗口：30 - 24 = **6小时**
- 批次必须在截止时间前至少24小时提交
- 对于频繁提交，分割为4小时窗口

第8章：任务分解策略

8.1 固定管道（提示链）

每个步骤都预先定义：

```
文档 -> 元数据提取 -> 数据提取 -> 验证 -> 富化 -> 最终输出
```

何时使用：

- 任务结构可预测（评审始终遵循相同模板）
- 所有步骤预先已知
- 需要稳定性和可重复性

8.2 动态自适应分解

子任务根据中间结果生成：

```
1. "为遗留代码库添加测试"
2. -> 首先：映射结构（Glob、Grep）
3. -> 发现：3个模块无测试，2个覆盖率不完整
4. -> 优先级：从支付模块开始（高风险）
5. -> 工作中：发现依赖外部API
6. -> 适应：在编写测试前为外部API添加mock
```

何时使用：

- 开放式调查任务
- 全部范围事先未知
- 每个步骤依赖前一步骤的结果

8.3 多轮代码审查

对于包含10个以上文件的拉取请求：

```
第1轮（逐文件）：分析 auth.ts -> 列出本地问题
第1轮（逐文件）：分析 database.ts -> 列出本地问题
第1轮（逐文件）：分析 routes.ts -> 列出本地问题
...
第2轮（集成）：分析文件间关系
  -> 跨文件问题：类型不一致、循环依赖
```

为什么对14个文件进行单轮分析效果差：

- 注意力稀释：对某些文件深度分析，对其他文件浅层分析
- 评论不一致：某个模式在一个文件中被标记，在另一个文件中被批准
- 遗漏bug：由于认知过载，明显错误被跳过

第9章：升级与人工参与

9.1 何时升级到人工

升级触发条件（明确规则）：

情况	动作
客户明确要求"帮我找个经理"	立即升级；不要尝试解决
政策未涵盖该请求	升级（例如，政策未规定竞争对手价格匹配）
代理无法取得进展	在合理次数尝试后升级
财务操作超过阈值	升级（最好通过钩子而非提示强制执行）
搜索客户时出现多个匹配	请求额外标识符；不要猜测

不可靠的触发条件：

不可靠方法	失败原因
情感分析	客户情绪与案例复杂度不相关
模型自评置信度（1–10）	模型可能自信地犯错；校准效果差

自动分类器	过度工程化；可能需要您没有的训练数据

9.2 升级模式

立即升级：

```
客户："我想和经理谈"
代理：[立即调用 escalate_to_human]
不应该："我可以帮您解决问题，让我..."
```

尝试解决后升级：

```
客户："我的冰箱购买两天后坏了"
代理：[检查订单，提供保修更换]
如果客户不满意 -> 升级
```

细致的升级（承认 → 解决 → 重申时升级）：

```
客户："太过分了，我对质量非常不满意！"
代理：[承认不满] "我理解您的不满。"
        [提供解决方案] "我可以提供更换或退款。"
客户："不，我要和人说话！"
代理：[客户再次坚持 -> 立即升级]
```

关键原则：先承认情绪，然后提出具体解决方案，只有当客户再次表达想要人工服务时才升级。不要在第一次表达不满时就升级（这与请求经理不同）。

因政策缺口升级：

```
客户："竞争对手X的这件商品便宜30%—给我打折"
政策：仅涵盖本网站的价格调整
代理：[升级—政策未涵盖竞争对手价格匹配]
```

9.3 结构化交接协议

升级时，代理应向人工传递结构化摘要：

```
{
  "customer_id": "CUST-12345",
  "customer_name": "张三",
  "issue_summary": "损坏商品的退款请求",
  "order_id": "ORD-67890",
  "root_cause": "商品到货时损坏；已附照片",
  "actions_taken": [
    "通过 get_customer 验证客户",
    "通过 lookup_order 确认订单",
    "提供标准更换——客户坚持要求退款"
  ],
  "refund_amount": "$89.99",
  "recommended_action": "批准全额退款",
  "escalation_reason": "客户要求与经理交谈"
}
```

人工操作员无法访问完整的对话记录——他们只看到这份摘要。因此摘要必须完整且自包含。

9.4 置信度校准与人工监督

对于数据提取系统：

1. **字段级置信度评分：** 模型为每个提取字段输出置信度评分
2. **校准：** 使用标记的验证集来调整阈值
3. **路由：**
   - 高置信度+稳定准确率 -> 自动处理
   - 低置信度或来源模糊 -> 人工审查

分层随机抽样：

- 即使对于高置信度提取，也要定期审计样本
- 整体97%的准确率可能掩盖某种文档类型40%的错误
- 按文档类型和字段分析准确率，而不仅仅是整体分析

第10章：多代理系统中的错误处理

10.1 错误类别

类别	示例	可重试	代理动作
瞬态	超时、503、网络故障	是	使用指数退避重试
验证	输入格式无效、缺少必填字段	否（修复输入）	修改请求后重试
业务	违反政策、超过阈值	否	向用户解释；提出替代方案

权限	拒绝访问	否	升级

10.2 错误处理反模式

反模式	问题	正确方法
通用状态"搜索不可用"	协调代理无法决定如何恢复	返回错误类型、查询、部分结果、替代方案
静默抑制（空结果=成功）	协调代理认为没有匹配，但实际是失败	区分"无结果"与"搜索失败"
单个失败时中止整个工作流	丢失所有部分结果	继续使用部分结果；注释缺口
子代理内无限重试	延迟和资源浪费	本地恢复（1–2次重试），然后传递给协调代理

10.3 结构化子代理错误

```
{
  "status": "partial_failure",
  "failure_type": "timeout",
  "attempted_query": "AI对音乐行业的影响2024",
  "partial_results": [
    {"title": "AI音乐生成报告", "url": "...", "relevance": 0.8}
  ],
  "alternative_approaches": [
    "尝试更窄的查询：'AI音乐创作工具'",
    "使用替代数据源"
  ],
  "coverage_impact": "未覆盖：AI对音乐制作的影响"
}
```

这为协调代理提供了决策所需的信息：

- 用修改后的查询重试？
- 使用部分结果？
- 委托给不同的子代理？
- 继续而不包含此部分并注释缺口？

10.4 最终综合中的覆盖注释

```
## 报告：AI对创意产业的影响

### 视觉艺术（完整覆盖）
[研究结果]

### 音乐（部分覆盖——搜索代理超时）
[部分结果]
⚠️ 注意：由于搜索代理超时，本节覆盖范围有限。

### 文学（完整覆盖）
[研究结果]
```

第11章：生产系统中的上下文管理

11.1 将关键事实提取到独立块中

与其依赖对话历史（摘要过程中会退化），不如将关键事实提取到结构化块中：

```
=== 案例事实（每次出现新事实时更新）===
客户ID：CUST-12345
订单ID：ORD-67890
订单日期：2025-01-15
订单金额：$89.99
问题：交货时商品损坏
客户要求：全额退款
状态：待经理审批
===
```

无论历史记录如何被摘要，都应在每个提示词中包含此块。

11.2 裁剪工具返回结果

如果 `lookup_order` 返回40+个字段，但当前任务只需要其中5个：

```
# PostToolUse钩子：只保留相关字段
@hook("PostToolUse", tool="lookup_order")
def trim_order_fields(result):
    return {
        "order_id": result["order_id"],
        "status": result["status"],
        "total": result["total"],
        "items": result["items"],
        "return_eligible": result["return_eligible"]
    }
```

这能节省上下文空间并减少噪音。

11.3 位置感知的输入

在编写输入时，应考虑"中间迷失"效应，将关键信息放在适当位置：

```
[关键发现——置于顶部]
发现3个严重漏洞...

[详细结果——置于中间]
=== 文件 auth.ts ===
...
=== 文件 database.ts ===
...

[行动项——置于末尾]
优先级：在合并前修复 auth.ts 中的漏洞。
```

11.4 草稿文件

在长时间调查中，代理可以将关键发现写入草稿文件：

```
# investigation-scratchpad.md
## 关键发现
- src/payments/processor.ts 中的 PaymentProcessor 继承自 BaseProcessor
- refund() 在3处被调用：OrderController、AdminPanel、CronJob
- 外部 PaymentGateway API 的速率限制为每分钟100次请求
- 迁移#47 添加了 refund_reason（NOT NULL）——2024-12-01
```

当上下文退化（或在新会话中），代理可以查阅草稿文件，而无需重新运行发现流程。

11.5 委托子代理以保护上下文

```
主代理："调查支付模块的依赖关系"
  -> 子代理（探索）：读取15个文件，追踪导入
  -> 返回："支付依赖于 AuthService、OrderModel 和外部 PaymentGateway API"

主代理：在上下文中只保留一行，而非15个文件
```

独立上下文层： 在多代理系统中，每个子代理在有限的上下文预算内运行——它只接收完成任务所需的信息。协调者充当独立的上下文层：汇总子代理输出、存储全局状态并分配上下文。这可以防止"上下文泄漏"——即一个代理用与其他代理无关的信息消耗掉整个窗口。

为子代理设置受限上下文预算：

- 发送最小化上下文：一个具体任务+必要数据
- 指示子代理返回结构化结果，而非原始数据转储
- 使用 `allowedTools` 限制子代理的工具集——工具越少意味着干扰越少，上下文成本越低

11.6 结构化状态持久化（用于崩溃恢复）

每个代理将其状态导出到已知位置：

```
// agent-state/web-search-agent.json
{
  "status": "completed",
  "queries_executed": ["AI music 2024", "AI music composition"],
  "results_count": 12,
  "key_findings": [...],
  "coverage": ["music composition", "music production"],
  "gaps": ["music distribution", "music licensing"]
}
```

协调者在恢复时加载清单：

```
// agent-state/manifest.json
{
  "web-search": "completed",
  "doc-analysis": "in_progress",
  "synthesis": "not_started"
}
```

第12章：保存溯源信息

12.1 归因丢失问题

在汇总多个来源的结果时，"声明→来源"的关联可能会丢失：

```
差：" AI音乐市场估值为32亿美元。"（无来源，无年份。）

好：
{
  "claim": "AI音乐市场估值为32亿美元。",
  "source_url": "https://example.com/report",
  "source_name": "2024全球AI音乐报告",
  "publication_date": "2024-06-15",
  "confidence": 0.9
}
```

12.2 处理冲突数据

当两个来源提供不同的数值时：

```
{
  "claim": "AI生成音乐在流媒体平台的占比",
  "values": [
    {
      "value": "12%",
      "source": "Spotify 2024年度报告",
      "date": "2024-03",
      "methodology": "自动分类"
    },
    {
      "value": "8%",
      "source": "音乐行业协会调查",
      "date": "2024-07",
      "methodology": "500家厂牌调查"
    }
  ],
  "conflict_detected": true,
  "possible_explanation": "方法论和时间段不同"
}
```

不要随意选择其中一个数值。保留两者并注明归因，让协调者来做决定。

12.3 包含日期以便正确解读

没有日期，时间差异可能被误解为矛盾：

```
差："来源A显示10%，来源B显示15%。存在矛盾。"
好："来源A（2023）显示10%，来源B（2024）显示15%。可能是一年增长了5%。"
```

12.4 按内容类型渲染

不要将所有内容强制放入一种格式：

- 财务数据 -> 表格
- 新闻和分析 -> 散文
- 技术发现 -> 结构化列表
- 时间序列 -> 按时间顺序排列

第13章：Claude Code内置工具

13.1 工具选择参考

任务	工具	示例
按名称/模式查找文件	**Glob**	`**/*.test.tsx`，`src/components/**/*.ts`
在文件内搜索	**Grep**	函数名、错误信息、导入
完整读取文件	**Read**	加载文件进行分析
写入新文件	**Write**	从头创建文件
精确编辑现有文件	**Edit**	通过唯一文本匹配替换特定片段
运行Shell命令	**Bash**	git、npm、运行测试、构建

13.2 渐进式调查策略

不要一次性读取所有文件。渐进式建立理解：

```
1. Grep：查找入口点（函数定义、导出）
2. Read：读取找到的文件
3. Grep：查找用法（导入、调用）
4. Read：读取消费者文件
5. 重复直到获得完整图景
```

13.3 备用方案：用Read+Write替代Edit

当Edit因文本匹配不唯一而失败时：

1. Read——加载完整文件内容
2. 以编程方式修改内容
3. Write——写入更新后的版本

第二部分：考试领域注释

领域1：代理架构与编排（27%）

1.1 为自主任务执行设计代理循环

关键知识：

- 代理循环生命周期：发送Claude请求，检查 `stop_reason`（`"tool_use"` vs `"end_turn"`），执行工具，返回结果进入下一次迭代
- 工具结果被追加到对话历史中，以便模型决定下一步行动
- 模型驱动的决策（Claude选择下一个工具）vs 硬编码决策树

关键技能：

- 流程控制：当 `stop_reason = "tool_use"` 时继续循环，在 `"end_turn"` 时停止
- 在迭代之间将工具结果追加到上下文中
- 需避免的反模式：解析助手文本以判断完成，将任意迭代次数限制作为主要停止机制

1.2 编排多代理系统（协调者-子代理）

关键知识：

- 中心辐射架构：协调者负责所有代理间通信、错误处理和路由
- 子代理在隔离的上下文中运行——它们不会自动继承协调者的历史记录
- 协调者职责：任务分解、委派、结果汇总、动态选择子代理
- 协调者分解过于狭窄的风险

关键技能：

- 在子代理之间分配研究覆盖范围以最小化重复
- 实现迭代细化循环（协调者评估综合结果并重新路由任务）
- 通过协调者路由所有通信以保证可观测性

1.3 配置子代理调用、上下文传递和生成

关键知识：

- `Task` 工具生成子代理；协调者的 `allowedTools` 必须包含 `"Task"`
- 子代理上下文必须在提示词中明确包含；子代理不继承父上下文
- `AgentDefinition` 配置：描述、系统提示词、工具约束
- 通过 `fork_session` 进行会话管理以探索替代方案

关键技能：

- 在子代理提示词中包含先前代理的完整输出
- 使用结构化格式在传递上下文时分离数据和元数据
- 通过在协调者的单个轮次中进行多次 `Task` 调用来生成并行子代理
- 用目标和质量标准而非逐步指令来编写协调者提示词

1.4 实现带有强制执行和交接模式的多步骤工作流

关键知识：

- **程序化强制执行**（钩子、前提条件）和**提示词指导**在工作流排序上的区别
- 当需要确定性保证时（例如，金融操作前的身份验证），仅靠提示词是不够的
- 升级时的结构化交接协议（客户ID、原因、推荐操作）

关键技能：

- 在先前步骤完成之前阻止下游调用的程序化前提条件（例如，在 `get_customer` 返回经过验证的ID之前阻止 `process_refund`）
- 将多方面的客户请求分解为独立项
- 升级到人工时产生结构化摘要

1.5 用于拦截工具调用和规范化数据的代理SDK钩子

关键知识：

- 钩子模式（例如 `PostToolUse`）在模型消费工具结果之前拦截它们
- 拦截出站调用的钩子以强制执行合规规则（例如，阻止超过阈值的退款）
- 钩子提供**确定性保证**，而提示词指令提供**概率性合规**

关键技能：

- `PostToolUse` 钩子用于规范化数据格式（Unix时间戳、ISO 8601、数字状态码）
- 拦截钩子以阻止违反策略的操作并重定向到升级
- 当业务规则需要保证合规时，选择钩子而非提示词

1.6 复杂工作流的任务分解策略

关键知识：

- **固定管道**（提示词链）vs 基于中间结果的**动态自适应分解**
- 提示词链：顺序步骤（分别分析每个文件，然后运行集成检查）
- 根据发现的内容生成子任务的自适应调查计划

关键技能：

- 对可预测的多方面审查使用提示词链；对开放式调查使用动态分解
- 将大型代码审查拆分为每文件分析加上独立的跨文件集成检查
- 分解开放式任务：先映射结构，再构建优先计划

1.7 会话状态、恢复和分叉

关键知识：

- `--resume <session-name>` 用于继续命名会话
- `fork_session` 从共享上下文创建独立的调查分支
- 恢复会话时告知代理文件变更的重要性
- 包含结构化摘要的新会话比恢复过时结果的会话更可靠

关键技能：

- 使用 `--resume` 继续命名调查会话
- 使用 `fork_session` 并行比较不同方法
- 在恢复（上下文仍为最新）和开始新会话（结果过时）之间做出选择

领域2：工具设计与MCP集成（18%）

2.1 设计具有清晰描述的工具接口

关键知识：

- 工具描述是LLM选择工具的**主要机制**；最简描述会导致选择不可靠
- 包含输入格式、示例查询、边缘情况和适用边界的重要性
- 模糊或重叠的描述会导致路由错误
- 系统提示词的措辞可能与工具产生意外关联

关键技能：

- 编写能清楚区分每个工具与类似替代品的描述
- 重命名工具以消除功能重叠（例如，`analyze_content` -> `extract_web_results`）

- 将通用工具拆分为具有清晰输入/输出合约的专用工具

2.2 为MCP工具实现结构化错误响应

关键知识：

- MCP工具响应中的 `isError` 标志
- **瞬时错误**（超时）、**验证错误**（错误输入）、**业务错误**（策略违规）和**访问/权限错误**之间的区别
- 通用错误（"操作失败"）阻止正确的恢复决策
- 可重试和不可重试错误之间的区别

关键技能：

- 返回结构化元数据，如 `errorCategory`（瞬时/验证/权限）、`isRetryable` 和人类可读的消息
- 对业务规则违规使用 `retryable: false`，并提供清晰的面向用户的解释
- 在子代理内对瞬时失败进行本地恢复；仅传播无法解决的错误
- 区分访问失败（重试决策）和有效的空结果（无匹配）

2.3 跨代理分配工具并配置 `tool_choice`

关键知识：

- 每个代理工具过多（例如18个而非4-5个）**会降低**工具选择的可靠性
- 拥有超出其专业范围工具的代理往往会误用这些工具
- 作用域工具访问：仅角色相关工具加上有限的跨角色实用工具集
- `tool_choice`：`"auto"`、`"any"` 和强制工具选择（`{"type": "tool", "name": "..."}`）

关键技能：

- 将每个子代理的工具集限制在其角色相关的范围内
- 用受约束的替代品替换通用工具（例如，`fetch_url` -> `load_document`）
- 使用 `tool_choice: "any"` 保证工具调用而非文本回答
- 强制特定工具以确保执行顺序

2.4 将MCP服务器集成到Claude Code和代理工作流中

关键知识：

- MCP服务器作用域：项目（`.mcp.json`）用于团队 vs 用户（`~/.claude.json`）用于实验
- `.mcp.json` 中的环境变量替换（例如，`${GITHUB_TOKEN}`）用于密钥管理
- 所有连接的MCP服务器的工具在连接时被发现，同时可用
- MCP资源作为"内容目录"（任务摘要、数据库模式）以减少探索性工具调用

关键技能：

- 在项目 `.mcp.json` 中使用基于环境变量的令牌配置共享MCP服务器
- 在 `~/.claude.json` 中保存个人/实验性服务器
- 对标准集成优先使用社区MCP服务器而非自定义服务器

2.5 选择和应用内置工具（Read、Write、Edit、Bash、Grep、Glob）

关键知识：

- **Grep**：在文件内容中搜索（函数名、错误消息、导入）
- **Glob**：按名称/扩展名模式查找文件
- **Read/Write**：全文件操作；**Edit**：通过唯一文本匹配进行精确修改
- 如果Edit因非唯一匹配而失败，则退回到Read + Write

关键技能：

- 使用Grep进行内容搜索，使用Glob按模式进行文件发现
- 渐进式建立理解：Grep入口点，然后Read追踪流程
- 通过包装模块追踪函数用法

领域3：Claude Code配置与工作流（20%）

3.1 配置具有层次结构、作用域和模块化组织的CLAUDE.md

关键知识：

- CLAUDE.md层次结构：用户（`~/.claude/CLAUDE.md`）、项目（`.claude/CLAUDE.md` 或根目录 `CLAUDE.md`）和目录级（子目录中的CLAUDE.md）
- 用户级设置仅适用于一个用户，不通过版本控制系统共享
- 引用外部文件的 `@path` 语法（例如，`@./standards/coding-style.md`）用于模块化CLAUDE.md
- 用于主题聚焦规则文件的 `.claude/rules/` 目录，而非单一的CLAUDE.md

关键技能：

- 诊断层次结构问题（新团队成员因指令位于用户级而非项目级而遗漏了指令）
- 使用 `@path`（例如，`@./standards/testing.md`）在每个包的CLAUDE.md中有选择地包含标准
- 将大型CLAUDE.md拆分为多个 `.claude/rules/` 文件（testing.md、api-conventions.md、deployment.md）

3.2 创建和配置自定义斜杠命令与技能

关键知识：

- `.claude/commands/` 中的**项目命令**（通过版本控制系统共享）vs `~/.claude/commands/` 中的**用户命令**
- `.claude/skills/` 中的技能，带有 `SKILL.md` 前置内容：`context: fork`、`allowed-tools`、`argument-hint`
- `context: fork` 在隔离的子代理上下文中运行技能，不污染主会话
- 个人技能变体可以以不同名称存储在 `~/.claude/skills/` 中

关键技能：

- 在 `.claude/commands/` 中存储项目斜杠命令，以便整个团队都能使用
- 使用 `context: fork` 隔离具有详细输出的技能
- 使用 `allowed-tools` 限制技能可以使用的工具
- 使用 `argument-hint` 提示开发者输入必要参数

3.3 使用路径特定规则进行条件约定加载

关键知识：

- `.claude/rules/` 文件可以包含YAML前置内容中的 `paths`，以根据glob模式激活规则
- 路径作用域规则**仅**在编辑匹配文件时加载，节省上下文和令牌
- 当约定跨越多个目录时（例如测试），基于glob的路径规则比目录级CLAUDE.md更优

关键技能：

- 创建带有 `paths: ["terraform/**/*"]` 的 `.claude/rules/` 文件，仅在处理匹配文件时加载
- 使用glob模式（`**/*.test.tsx`）按文件类型应用约定，不受位置限制
- 当约定跨越代码库时，优先使用路径特定规则而非目录级CLAUDE.md

3.4 决定何时使用规划模式vs直接执行

关键知识：

- **规划模式**：用于变更量大、有多种可行方案和架构决策的复杂任务
- **直接执行**：用于简单、易于理解的变更（例如，添加单个验证）
- 规划模式可在进行修改前安全探索代码库
- 探索子代理隔离详细的发现输出

关键技能：

- 对具有架构影响的任务使用规划模式（微服务、涉及45+文件的迁移）
- 对有清晰堆栈跟踪和单个文件的修复使用直接执行
- 使用探索子代理防止多阶段任务中的上下文窗口耗尽

- 结合两种方法：规划用于发现，执行用于实现

3.5 渐进式细化以持续改进

关键知识：

- 具体的输入/输出示例是传达期望最有效的方式
- **测试驱动迭代**：先写测试，然后根据失败进行迭代
- "访谈"模式：Claude提问以发现非显而易见的设计考量
- 何时在一条消息中提供所有问题（相互依赖）vs 顺序提供（独立）

关键技能：

- 提供2-3个具体的输入/输出示例来阐明转换需求
- 在实现之前构建包含预期行为、边缘情况和性能要求的测试集
- 使用访谈模式发现设计方面（缓存失效、故障模式）
- 为边缘情况提供具体测试用例和样本输入及预期输出

3.6 将Claude Code集成到CI/CD管道

关键知识：

- 用于自动化管道非交互模式的 `-p`（或 `--print`）标志
- CI中结构化输出的 `--output-format json` 和 `--json-schema`
- CLAUDE.md为CI触发的Claude Code提供项目上下文（测试标准、审查标准）
- **会话上下文隔离**：生成代码的同一会话在审查时比独立实例效果更差

关键技能：

- 在CI中使用 `-p` 运行Claude Code以避免挂起在交互式输入上
- 使用 `--output-format json` + `--json-schema` 获取结构化结果（例如，内联PR评论）
- 在新提交后重新运行时包含之前的审查结果（仅报告新问题/未修复问题）
- 在CLAUDE.md中记录测试标准和可用夹具以提高测试生成质量
- 在生成新测试时将现有测试文件包含在上下文中，以避免重复并保持风格一致

领域4：提示词工程与结构化输出（20%）

4.1 使用显式标准设计提示词以提高准确性

关键知识：

- 显式标准比模糊指令更有效（例如，"仅在注释与代码矛盾时标记"vs"检查注释准确性"）
- "更保守一点"这样的通用指导比具体的分类标准效果更差

- 误报对开发者信任的影响：某些类别的高误报率会破坏对准确类别的信任

关键技能：

- 定义审查标准：报告什么（错误、安全）vs 忽略什么（次要风格）
- 临时禁用误报率高的类别
- 为每个级别定义带有代码示例的显式严重性标准

4.2 使用少样本提示词提高输出一致性

关键知识：

- 少样本示例是产生格式一致、可操作输出的最有效方法
- 少样本可以演示处理模糊情况的方式（工具选择、测试覆盖缺口）
- 少样本帮助模型泛化到新模式，而不仅仅是重复默认行为
- 少样本可以减少提取任务中的幻觉

关键技能：

- 为模糊场景提供2-4个带有推理说明的针对性示例
- 包含演示输出格式的少样本示例（位置、问题、严重性、建议修复）
- 提供区分可接受代码模式与真实问题的示例
- 提供来自不同结构文档的正确提取示例

4.3 使用 `tool_use` 和JSON Schema强制结构化输出

关键知识：

- 带有JSON Schema的 `tool_use` 是保证模式合规输出并消除JSON语法错误的最可靠方式
- 使用 `tool_choice: "auto"` 模型可以返回文本；使用 `"any"` 时必须调用工具；强制选择会选择特定工具
- 严格的JSON Schema消除语法错误，但不能防止语义错误（总计不加；值在错误字段中）
- Schema设计：必填vs可选字段；带有"其他"加详情字符串的枚举以实现可扩展性

关键技能：

- 使用JSON Schema定义提取工具并从 `tool_use` 结果中解析数据
- 当存在多个Schema时，使用 `tool_choice: "any"` 保证结构化输出
- 强制特定工具调用：`tool_choice: {"type": "tool", "name": "extract_metadata"}`
- 当来源可能不包含信息时，将字段设为可选/可空，以避免伪造值
- 使用枚举值如 `"unclear"` 和 `"other"` 加上详细字段进行可扩展分类

4.4 为提取质量实现验证、重试和反馈循环

关键知识：

- 带错误反馈的重试：在重试提示词中包含具体的验证错误以指导修正
- 当信息根本不存在于来源中时，重试无效
- 反馈循环设计：追踪触发发现的模式（`detected_pattern`）
- 语义错误（总计不一致）vs 语法错误（由 `tool_use` 解决）

关键技能：

- 包含原始文档、不正确提取和具体验证错误的后续提示词
- 识别重试何时无效（所需信息只在外部文档中）
- 在发现中包含 `detected_pattern` 字段以分析误报
- 通过提取 `calculated_total` 和 `stated_total` 来设计自我纠正以检测差异

4.5 设计高效的批处理策略

关键知识：

- Message Batches API：节省50%，最长24小时处理窗口，无延迟SLA保证
- 批处理适用于非阻塞任务（隔夜报告、审计），不适用于阻塞任务（合并前检查）
- Batch API不支持单个请求中的多轮工具调用
- `custom_id` 字段在批次内关联请求/响应

关键技能：

- 对阻塞检查使用同步API；对隔夜/每周工作负载使用Batch API
- 根据SLA需求规划批次提交节奏（例如，对30小时保证使用4小时窗口，含24小时处理）
- 通过重新提交仅失败的文档（由 `custom_id` 标识）来处理失败
- 在大规模处理之前使用样本迭代优化提示词

4.6 设计多实例和多轮审查架构

关键知识：

- 自我审查的局限性：模型保留其推理上下文，不太可能挑战自己的决策
- 独立审查实例（没有生成上下文）更擅长发现细微问题
- 多轮审查：每文件本地分析加上跨文件集成检查，以避免注意力分散

关键技能：

- 使用第二个独立的Claude实例在没有生成上下文的情况下审查变更
- 将多文件审查拆分为每文件检查加上集成检查，用于跨文件数据流分析
- 使用带有自评置信度的验证检查以校准的方式路由审查

领域5：上下文管理与可靠性（15%）

5.1 管理对话上下文以保留关键信息

关键知识：

- 渐进式摘要的风险：数字值、百分比和日期在模糊摘要中被压缩
- 中间迷失效应：模型可靠地处理长输入的开头和结尾，但可能错过中间的发现
- 工具输出可能在上下文中积累，与相关性不成比例（需要5个字段时却有40+个字段）
- 在后续API请求中发送完整对话历史的重要性

关键技能：

- 将事务性事实提取到摘要历史之外的持久化"案例事实"块中
- 将详细的工具输出精简到相关字段
- 在汇总数据的开头放置关键发现，并使用明确的章节标题
- 要求子代理在结构化输出中包含元数据（日期、来源）

5.2 设计有效的升级模式并解决歧义

关键知识：

- 适当的升级触发器：明确请求人工干预、策略缺口/例外、无法取得进展
- 立即升级（明确请求）vs 尝试解决（在代理范围内）
- 情感分析和模型置信度自评是案例复杂性的不可靠代理
- 多个客户匹配需要请求额外标识符，而非启发式猜测

关键技能：

- 在系统提示词中使用少样本示例明确升级标准
- 立即执行人工干预的明确请求，不进行额外调查
- 当策略对特定请求模糊或未明确时升级
- 当工具结果包含多个匹配时请求额外标识符

5.3 在多代理系统中实现错误传播策略

关键知识：

- 结构化错误上下文（故障类型、查询、部分结果、替代方案）使协调者能够更智能地恢复
- 区分访问失败（超时需要重试决策）和有效的空结果（无匹配）
- 通用错误状态（"搜索不可用"）对协调者隐藏了有价值的上下文
- 静默抑制或在单个失败时中止整个工作流都是反模式

关键技能：

- 返回结构化错误上下文：故障类型、尝试内容、部分结果、可能的替代方案
- 区分访问失败和有效的空结果
- 在子代理内对瞬时失败进行本地恢复；仅传播无法恢复的错误以及部分结果
- 在综合中注释覆盖范围：哪些内容有充分支撑vs哪里存在缺口

5.4 在调查大型代码库时高效管理上下文

关键知识：

- 长会话中的上下文退化：模型开始产生不稳定的答案，并提及"典型模式"而非具体类
- 草稿文件跨上下文边界保留关键发现
- 委托给子代理可以隔离详细的发现输出
- 结构化状态持久化支持崩溃恢复

关键技能：

- 为具体问题生成子代理，同时在主代理中保持高层协调
- 使用草稿文件存储关键发现并在后续引用
- 在生成下一阶段子代理之前总结关键发现
- 在长时间调查中使用 `/compact` 减少上下文使用

5.5 设计具有人工监督和置信度校准的工作流

关键知识：

- 总体指标（例如，97%总体准确率）可能掩盖特定文档类型或字段上的差性能
- 分层随机抽样测量高置信度提取中的错误率
- 使用标注验证集进行字段级置信度校准
- 在自动化之前按文档类型和字段细分验证准确性

关键技能：

- 实现分层随机抽样以检测新的错误模式
- 按文档类型和字段分析准确性以验证稳定性能
- 输出字段级置信度分数并使用标注数据校准审查阈值
- 将低置信度或模糊来源的提取路由到人工审查

5.6 在多源综合中保存溯源信息并处理不确定性

关键知识：

- 在摘要期间未保留"声明→来源"映射时归因会丢失
- 在聚合期间必须保留结构化映射

- 通过用归因注释冲突来处理冲突统计数据，而非任意选择一个值
- 包含发布/收集日期以避免将时间差异误读为矛盾

关键技能：

- 要求子代理输出"声明→来源"映射（URL、文档名称、引用）
- 将报告结构化为将稳定发现与有争议发现分开
- 保留带有注释的冲突值，并传递给协调者进行调解
- 包含发布日期以便正确解读时间顺序
- 按类型渲染内容：财务数据用表格，新闻用散文，技术发现用结构化列表

含解析的考题示例

题目1（场景：客服代理）

情境： 数据显示，在12%的情况下，代理跳过 `get_customer` 而仅使用客户姓名调用 `lookup_order`，导致错误退款。

哪种变更最有效？

- A) 添加程序化前提条件，在从 `get_customer` 获得ID之前阻止 `lookup_order` 和 `process_refund` **[正确]**
- B) 改进系统提示词
- C) 添加少样本示例
- D) 实现路由分类器

为什么选A： 当关键业务逻辑需要特定工具顺序时，软件提供**确定性保证**，而基于提示词的方法（B、C）无法做到。D解决的是可用性问题，而非工具排序。

题目2（场景：客服代理）

情境： 代理对订单相关问题经常调用 `get_customer` 而非 `lookup_order`。工具描述最简化且相似。

第一步应该做什么？

- A) 少样本示例
- B) 用输入格式、示例和边界扩展每个工具的描述 **[正确]**
- C) 添加路由层
- D) 合并工具

为什么选B： 工具描述是模型的主要选择机制。这是成本最低、影响最大的修复。A增加了令牌但没有解决根本原因。C是过度设计。D需要付出比合理情况更多的努力。

题目3（场景：客服代理）

情境： 代理只解决了55%的问题，目标是80%。它升级了简单案例，同时试图自主处理复杂的策略例外。

如何改善校准？

- A) 添加带有少样本示例的显式升级标准 **[正确]**
- B) 自评置信度（1-10分）并自动升级
- C) 在历史数据上训练的单独分类器
- D) 情感分析

为什么选A： 它直接解决了根本原因——不明确的决策边界。B不可靠（模型可能自信地犯错）。C是过度设计。D解决的是不同的问题（情绪≠复杂性）。

题目4（场景：使用Claude Code生成代码）

情境： 你需要一个自定义 `/review` 命令用于标准代码审查，当团队成员克隆仓库时需要自动可用。

应该在哪里创建命令文件？

- A) 在项目仓库的 `.claude/commands/` 中 **[正确]**
- B) `~/.claude/commands/`
- C) 根目录 `CLAUDE.md`
- D) `.claude/config.json`

为什么选A： 存储在 `.claude/commands/` 中的项目命令受版本控制，对所有人自动可用。B用于个人命令。C用于指令，不用于命令定义。D不存在。

题目5（场景：使用Claude Code生成代码）

情境： 你需要将单体应用重构为微服务（涉及数十个文件、服务边界决策）。

应该使用什么方法？

- A) 规划模式：探索代码库、理解依赖关系、设计方案 **[正确]**
- B) 渐进式直接执行
- C) 带有详细预先指令的直接执行
- D) 直接执行，遇到困难时切换到规划

为什么选A： 规划模式专为大型变更、多种可行方案和架构决策而设计。B有高代价返工的风险。C假设你已经了解结构。D是被动反应式的。

题目6（场景：使用Claude Code生成代码）

情境： 代码库在不同区域有不同的约定（React、API、数据库）。测试与代码并存。你希望约定被自动应用。

应该使用什么方法？

- A) 带有YAML前置内容和glob模式的 `.claude/rules/` 文件 **[正确]**
- B) 将所有内容放入根目录CLAUDE.md
- C) `.claude/skills/` 中的技能
- D) 在每个目录中放置CLAUDE.md

为什么选A： 带有glob模式的 `.claude/rules/`（例如，`**/*.test.tsx`）可以基于文件路径自动应用约定——非常适合分散在代码库中的测试。B依赖模型推断。C是手动/按需的。D在相关文件分布于多个目录时效果不好。

题目7（场景：多代理研究系统）

情境： 系统研究"AI对创意产业的影响"，但报告只涵盖视觉艺术。协调者将主题分解为："数字艺术中的AI"、"平面设计中的AI"、"摄影中的AI"。

原因是什么？

- A) 综合代理没有检测到缺口
- B) 协调者分解任务过于狭窄 **[正确]**
- C) 网络搜索代理搜索不够彻底
- D) 文档分析代理过滤掉了非视觉来源

为什么选B： 日志显示协调者只将"创意产业"分解为视觉子主题，完全忽略了音乐、文学和电影。子代理执行正确——问题在于分配给它们的任务。

题目8（场景：多代理研究系统）

情境： 一个网络搜索子代理在研究复杂主题时超时。你需要设计如何将错误信息传回协调者。

哪种错误传播方式最能实现智能恢复？

- A) 向协调者返回结构化错误上下文：故障类型、查询、部分结果和替代方案 **[正确]**
- B) 在子代理内部使用指数退避实现自动重试，然后返回通用的"搜索不可用"状态
- C) 在子代理内部捕获超时并返回标记为成功的空结果集
- D) 将超时异常传播到终止整个工作流的顶层处理程序

为什么选A： 结构化错误上下文使协调者能够决定是否以修改后的查询重试、尝试替代方案，还是继续使用部分结果。B将上下文隐藏在通用状态后面。C将失败掩盖为成功。D不必要地中止整个工作流。

题目9（场景：多代理研究系统）

情境： 综合代理在合并结果时经常需要验证特定声明。目前，当需要验证时，综合代理将控制权交回协调者，协调者调用网络搜索代理，然后用新结果重新运行综合。这为每个任务增加了2-3次额外的往返，延迟增加了40%。你的评估显示85%的检查是简单的事实核查（日期、名称、统计数据），15%需要更深入的调查。

如何在保持可靠性的同时减少开销？

- A) 为综合代理提供有限的 `verify_fact` 工具用于简单检查，并继续通过协调者路由复杂验证 **[正确]**
- B) 将所有验证需求积累到批次中，最后统一返回给协调者
- C) 给综合代理完整访问所有网络搜索工具的权限
- D) 主动缓存每个来源周围的额外上下文

为什么选A： 这应用了最小权限原则：综合代理获得85%常见情况（简单事实检查）所需的工具，同时保留了协调者中介的路径用于复杂调查。B引入了阻塞依赖（后续综合步骤可能依赖先前验证的事实）。C破坏了职责分离。D依赖于无法可靠预测需求的投机性缓存。

题目10（场景：CI中的Claude Code）

情境： 管道运行 `claude "分析此拉取请求的安全问题"`，但挂起等待交互式输入。

正确的方法是什么？

- A) 使用 `-p` 标志：`claude -p "分析此拉取请求的安全问题"` **[正确]**
- B) 设置 `CLAUDE_HEADLESS=true`
- C) 从 `/dev/null` 重定向stdin
- D) 使用 `--batch`

为什么选A： `-p`（或 `--print`）是以非交互模式运行Claude Code的文档化方式。它处理提示词，打印到stdout并退出。其他选项要么是不存在的功能，要么是Unix变通方案。

题目11（场景：CI中的Claude Code）

情境： 团队希望降低自动化分析的API成本。Claude目前实时服务两个工作流：（1）阻塞性的合并前检查，必须在开发者合并PR之前完成；（2）每晚生成的技术债务报告，供早晨查阅。一位经理提议将两者都迁移到Message Batches API以节省50%。

你应该如何评估这个提议？

- A) 仅对技术债务报告使用批处理；保留合并前检查的实时调用 **[正确]**
- B) 将两个工作流都迁移到批处理，并轮询完成状态
- C) 两者都保留实时调用以避免批处理结果中的排序问题
- D) 将两者都迁移到批处理，如果批处理耗时过长则回退到实时

为什么选A： Message Batches API节省50%，但处理时间最长可达24小时，没有保证的延迟SLA。这使其不适合开发者等待的阻塞性合并前检查，但非常适合技术债务报告等隔夜批处理工作负载。

题目12（场景：多文件代码审查）

情境： 一个拉取请求更改了库存跟踪模块中的14个文件。对所有文件的单次审查产生了不一致的结果：一些文件有详细评论，另一些却只有表面性评论，遗漏了明显的错误，并且反馈矛盾（一个模式在一个文件中被标记为有问题，但在另一个文件中相同代码却被批准）。

应该如何重构审查？

- A) 拆分为聚焦检查：分别分析每个文件的本地问题，然后对跨文件数据流运行单独的集成检查 **[正确]**
- B) 要求开发者将大型PR拆分为3-4个文件的提交
- C) 切换到具有更大上下文窗口的高级模型，一次性审查所有14个文件
- D) 运行三次独立的完整PR审查，只报告至少在两次中发现的问题

为什么选A： 聚焦检查直接解决了根本原因——同时处理多个文件时的注意力分散。每文件分析确保深度一致，独立的集成检查捕获跨文件问题。B将负担转移给开发者而不改善系统。C是误解：更大的上下文不能修复注意力质量。D通过要求不一致检测之间的共识来抑制真实的错误。

模拟测试

> 涵盖4个场景的60道题。格式和难度与真实考试相匹配。
>
> 或者，您可以通过互动HTML文件练习这些题目: 模拟测试 (ZH)

场景：多代理研究系统

第1题（场景：多代理研究系统）

情境： 文档分析代理发现两个可信来源对一个关键指标的统计数据直接矛盾：政府报告显示40%的增长，而行业分析显示12%。两个来源都看起来可信，这种差异可能从根本上影响研究结论。文档分析代理应该如何最有效地处理这种情况？

哪种方法最有效？

- A) 应用可信度启发式选择最可能正确的数字，用该值完成分析，并添加脚注提及差异。
- B) 在分析输出中包含两个数字，不标记它们为冲突，让综合代理根据更广泛的上下文决定使用哪个。
- C) 停止分析并立即升级给协调者，在继续之前请它决定哪个来源更权威。
- D) 用两个数字完成分析，明确注释冲突和来源归因，在传递给综合之前让协调者决定如何调和数据。** [正确]**

为什么选D： 这种方法保持了职责分离：分析代理不阻塞地完成其核心工作，保留两个冲突值及清晰归因，并正确将调和工作传递给拥有更广泛上下文的协调者。

第2题（场景：多代理研究系统）

情境： 网络搜索代理和文档分析代理已完成任务并将结果返回给协调者。创建综合研究报告的下一步是什么？

哪个下一步最合适？

- A) 每个代理直接将结果发送给报告编写代理，绕过协调者。
- B) 文档分析代理请求网络搜索结果并在内部合并它们。
- C) 协调者将两组结果传递给综合代理进行统一整合。**[正确]**
- D) 协调者将两个代理的原始输出连接起来并作为最终结果返回。

为什么选C： 在协调者-子代理架构中，协调者将两组结果集转发给综合代理进行集中整合，保持控制并确保高质量的合并。

第3题（场景：多代理研究系统）

情境： 文档分析子代理在处理PDF文件时经常失败：一些有损坏的部分会触发解析异常，另一些受密码保护，有时解析库在大文件上挂起。目前，任何异常都会立即终止子代理并向协调者返回错误，协调者必须决定是否重试、跳过或使整个任务失败。这导致协调者过度介入常规错误处理。哪种架构改进最有效？

哪种改进最有效？

- A) 创建专用的错误处理代理，通过共享队列监控所有失败，并直接向子代理发送重启命令来决定恢复操作。
- B) 配置子代理始终返回带成功状态的部分结果，将错误详情嵌入元数据；协调者将所有响应视为成功。
- C) 让协调者在将文档发送给子代理之前进行验证，拒绝可能导致失败的文档。
- D) 在子代理中对瞬时失败实现本地恢复，只向协调者升级无法解决的错误，包含已尝试的步骤和部分结果。**[正确]**

为什么选D： 在最低能够解决问题的层级处理错误。本地恢复减少了协调者的工作负担，同时仍然以完整上下文和部分进度升级真正无法恢复的问题。

第4题（场景：多代理研究系统）

情境： 在"AI对创意产业的影响"主题上运行系统后，你观察到每个子代理都成功完成：网络搜索代理找到了相关文章，文档分析代理正确总结了它们，综合代理产生了连贯的文本。然而，最终报告只涵盖视觉艺术，完全遗漏了音乐、文学和电影。在协调者日志中，你看到它将主题分解为三个子任务："数字艺术中的AI"、"平面设计中的AI"和"摄影中的AI"。最可能的根本原因是什么？

最可能的根本原因是什么？

- A) 综合代理缺少检测覆盖缺口的指令。
- B) 文档分析代理因过于严格的相关性标准而过滤掉了非视觉来源。
- C) 协调者的任务分解过于狭窄，给子代理分配的工作没有涵盖所有相关领域。**[正确]**
- D) 网络搜索代理的查询不足，应该扩大以覆盖更多行业。

为什么选C： 协调者将一个宽泛的主题只分解为视觉艺术子任务，完全遗漏了音乐、文学和电影。由于子代理正确执行了分配的任务，狭窄的分解是显而易见的根本原因。

第5题（场景：多代理研究系统）

情境： 网络搜索子代理仅返回5个请求来源类别中3个的结果（竞争对手网站和行业报告成功，但新闻档案和社交feed超时）。文档分析子代理成功处理了所有提供的文档。综合子代理必须从混合质量的上游输入中生成摘要。哪种错误传播策略最有效？

哪种错误传播策略最有效？

- A) 仅使用成功来源继续综合，生成输出而不提及哪些数据不可用。
- B) 综合子代理向协调者返回错误，由于数据不完整触发完整重试或任务失败。
- C) 综合子代理请求协调者在开始综合之前以更长的超时重试超时的来源。
- D) 用覆盖注释结构化综合输出，指示哪些结论有充分支撑，哪里因不可用来源存在缺口。**[正确]**

为什么选D： 覆盖注释实现了透明的优雅降级，保留了已完成工作的价值，同时传播不确定性以实现有知情的置信度决策。

第6题（场景：多代理研究系统）

情境： 文档分析子代理遇到了一个无法解析的损坏PDF文件。在设计系统的错误处理时，处理这种失败最有效的方式是什么？

哪种方法最有效？

- A) 向协调者代理返回带上下文的错误，使其能够决定如何继续。**[正确]**
- B) 静默跳过损坏的文档并继续处理剩余文件，以避免中断工作流。
- C) 在报告失败之前自动以指数退避重试解析文档三次。
- D) 抛出终止整个研究工作流的异常。

为什么选A： 向协调者返回带上下文的错误是最有效的方法，因为它让协调者能够做出明智的决定——跳过文件、尝试替代解析方法或通知用户——同时保持对失败的可见性。

第7题（场景：多代理研究系统）

情境： 生产日志显示持续的模式：像"分析上传的季度报告"这样的请求有45%的时间被路由到网络搜索代理，而非文档分析代理。查看工具定义，你发现网络搜索代理有一个工具 `analyze_content`，描述为"分

析内容并提取关键信息"，而文档分析代理有一个工具 `analyze_document`，描述为"分析文档并提取关键信息"。应该如何修复路由错误问题？

应该如何修复路由错误问题？

- A) 在协调者决定委派之前添加预路由分类器，检测用户是指上传的文件还是网络内容。
- B) 将网络搜索工具重命名为 `extract_web_results`，并将其描述更新为"处理并返回从网络搜索和URL检索到的信息"。**[正确]**
- C) 在协调者提示词中添加少样本示例，显示正确的路由："用户上传季度报告→文档分析代理"和"用户询问网页→网络搜索代理"。
- D) 用"用于上传的PDF、Word文档和电子表格"等使用示例扩展文档分析工具描述，保持网络搜索工具不变。

为什么选B： 将网络搜索工具重命名为 `extract_web_results`，并将其描述更新为明确引用网络搜索和URL，通过消除两个工具名称和描述之间的语义重叠直接消除了根本原因。这使每个工具的目的明确无误，使协调者能够可靠地区分文档分析和网络搜索。

第8题（场景：多代理研究系统）

情境： 一位同事提议文档分析代理应该直接将其结果发送给综合代理，绕过协调者。保持协调者作为所有子代理间通信的中央枢纽的主要优势是什么？

保持协调者作为中央枢纽的主要优势是什么？

- A) 协调者可以观察所有交互，统一处理错误，并决定每个子代理应该接收哪些信息。**[正确]**
- B) 协调者将多个对子代理的请求批量处理，减少总API调用次数和整体延迟。
- C) 通过协调者路由可以实现自动重试逻辑，而直接代理间调用无法支持。
- D) 子代理使用隔离内存，直接通信需要只有协调者才能执行的复杂序列化。

为什么选A： 协调者模式提供了对所有交互的集中可见性、系统范围内统一的错误处理，以及对每个子代理接收哪些信息的精细控制——这些是星型通信拓扑的主要优势。

第9题（场景：多代理研究系统）

情境： 网络搜索子代理在研究复杂主题时超时。你需要设计如何将有关此失败的信息返回给协调者。哪种错误传播方式最能实现智能恢复？

哪种错误传播方式最能实现智能恢复？

- A) 向协调者返回结构化错误上下文，包括故障类型、执行的查询、任何部分结果和潜在的替代方法。**[正确]**
- B) 在子代理内捕获超时并返回标记为成功的空结果集。
- C) 在子代理内实现自动指数退避重试，在耗尽重试后只返回通用的"搜索不可用"状态。
- D) 将超时异常直接传播到顶层处理程序，终止整个研究工作流。

为什么选A： 返回结构化错误上下文——包括故障类型、执行的查询、部分结果和替代方法——为协调者提供了做出智能恢复决策所需的一切（例如，以修改后的查询重试或继续使用部分结果）。它为明智的协调级

决策保留了最大上下文。

第10题（场景：多代理研究系统）

情境： 在系统设计中，你为文档分析代理提供了通用工具 `fetch_url`，使其可以通过URL下载文档。生产日志显示该代理现在频繁下载搜索引擎结果页面进行临时网络搜索——这种行为应该通过网络搜索代理路由——导致结果不一致。哪种修复最有效？

哪种修复最有效？

- A) 将 `fetch_url` 替换为验证URL指向文档格式的 `load_document` 工具。**[正确]**
- B) 从文档分析代理中移除 `fetch_url`，通过协调者将所有URL获取路由到网络搜索代理。
- C) 实现过滤，阻止对已知搜索引擎域的 `fetch_url` 调用，同时允许其他URL。
- D) 在文档分析代理提示词中添加指令，说明 `fetch_url` 只应用于下载文档URL，而非搜索。

为什么选A： 将通用工具替换为在接口级别验证URL格式的文档特定工具，通过在接口级别约束能力来修复根本原因。这遵循了最小权限原则，使不期望的搜索行为不可能发生，而不仅仅是受到阻止。

第11题（场景：多代理研究系统）

情境： 在研究广泛主题时，你观察到网络搜索代理和文档分析代理调查相同的子主题，导致其输出中出现大量重复。令牌使用量几乎翻倍，而研究广度或深度没有相应增加。解决这个问题最有效的方法是什么？

解决这个问题最有效的方法是什么？

- A) 让两个代理并行完成，然后让协调者在将结果传递给综合代理之前去重重叠结果。
- B) 协调者在委派之前明确划分研究空间，给每个代理分配不同的子主题或来源类型。**[正确]**
- C) 实现共享状态机制，代理记录其当前关注区域，以便其他代理在执行期间动态避免重复。
- D) 切换到顺序执行，文档分析在网络搜索完成后才运行，使用网络搜索结果作为上下文以避免重复。

为什么选B： 在委派之前让协调者明确划分研究空间最有效，因为它在任何工作开始之前就解决了根本原因——不清晰的任务边界。它在防止重复工作和浪费令牌的同时保留了并行性。

第12题（场景：多代理研究系统）

情境： 在研究过程中，网络搜索子代理对三个来源类别进行查询，结果各不相同：学术数据库返回15篇相关论文，行业报告返回"0个结果"，专利数据库返回"连接超时"。在设计向协调者的错误传播时，哪种方式能最好地实现恢复决策？

哪种方式能最好地实现恢复决策？

- A) 将结果汇总为单一成功率指标（例如，"67%来源覆盖率"），按需提供详细日志。
- B) 将"超时"和"0个结果"都报告为需要协调者干预的失败。
- C) 在内部重试瞬时失败，只报告持续性错误。

- D) 区分需要重试决策的访问失败（超时）和代表成功查询的有效空结果（"0个结果"）。**[正确]**

为什么选D： 超时（访问失败）和"0个结果"（有效空结果）是语义上不同的结果，需要不同的响应。区分它们允许协调者重试专利数据库，同时接受行业报告的"0个结果"作为有效的信息性发现。

第13题（场景：多代理研究系统）

情境： 生产监控显示综合质量不一致。当汇总结果约为75K令牌时，综合代理可靠地引用前15K令牌（网络搜索标题/摘要）和后10K令牌（文档分析结论）中的信息，但经常遗漏中间50K令牌中的关键发现——即使它们直接回答了研究问题。应该如何重构汇总输入？

应该如何重构汇总输入？

- A) 在汇总之前将所有子代理输出摘要到20K令牌以下，使内容保持在模型的可靠处理范围内。
- B) 将子代理结果增量流式传输给综合代理，先完整处理网络搜索结果，再添加文档分析结果。
- C) 在汇总输入的开头放置关键发现摘要，并用明确的章节标题组织详细结果以便导航。**[正确]**
- D) 实现轮换，在各研究任务中交替哪个子代理的结果首先出现，以确保两个来源随时间获得平等的顶部位置。

为什么选C： 在开头放置关键发现摘要利用了首要效应，使关键信息位于最可靠处理的位置。在整个过程中添加明确的章节标题帮助模型导航和关注中间输入内容，直接缓解了"中间迷失"现象。

第14题（场景：多代理研究系统）

情境： 在测试中，网络搜索代理（85K令牌，含页面内容）和文档分析代理（70K令牌，含思维链）的组合输出总计155K令牌，但综合代理在50K令牌以下的输入中表现最佳。哪种解决方案最有效？

哪种解决方案最有效？

- A) 修改上游代理以返回结构化数据（关键事实、引用、相关性分数）而非详细内容和推理。**[正确]**
- B) 添加中间摘要代理，在传递给综合之前压缩发现。
- C) 让综合代理分批顺序处理发现，在调用之间维护状态。
- D) 将发现存储在向量数据库中，给综合代理搜索工具以便在工作期间查询。

为什么选A： 修改上游代理以返回结构化数据通过在来源减少令牌量同时保留必要信息来修复根本原因。它避免了传递臃肿的页面内容和推理痕迹，这些内容会膨胀令牌而不改善综合步骤。

第15题（场景：多代理研究系统）

情境： 在测试中，你观察到综合代理在合并结果时经常需要验证特定声明。目前，当需要验证时，综合代理将控制权返回给协调者，协调者调用网络搜索代理，然后用结果重新调用综合。这为每个任务增加了2-3次额外循环，延迟增加了40%。你的评估显示85%的验证是简单事实核查（日期、名称、统计数据），15%需要更深入的研究。哪种方法在保持系统可靠性的同时最有效地减少开销？

哪种方法最有效？

- A) 给综合代理访问所有网络搜索工具的权限，使其可以直接处理任何验证需求，不需要协调者循环。
- B) 让综合代理积累所有验证需求，最后作为批次返回给协调者，协调者一次性将它们全部发送给网络搜索代理。
- C) 让网络搜索代理在初始研究期间主动缓存每个来源周围的额外上下文，预期综合需要验证。
- D) 给综合代理提供有限范围的 `verify_fact` 工具用于简单检查，同时通过协调者将复杂验证路由到网络搜索代理。**[正确]**

为什么选D： 有限范围的事实验证工具让综合代理直接处理85%的简单检查，消除了大多数循环，同时保留了协调者委派路径用于15%的复杂验证。这应用了最小权限同时显著降低了延迟。

场景：Claude Code用于持续集成

第16题（场景：Claude Code用于持续集成）

情境： 你的CI管道在 `--print` 模式下运行Claude Code CLI，使用CLAUDE.md提供代码审查的项目上下文，开发者普遍认为审查内容翔实。然而，他们报告将发现整合到工作流中很困难——Claude输出的叙述性段落必须手动复制到PR评论中。团队希望将每个发现自动发布为代码中相关位置的单独内联PR评论，这需要包含文件路径、行号、严重性级别和建议修复的结构化数据。哪种方法最有效？

哪种方法最有效？

- A) 在CLAUDE.md中添加"审查输出格式"部分，带有结构化发现示例，让Claude从项目上下文中学习预期格式。
- B) 使用CLI标志 `--output-format json` 和 `--json-schema` 强制结构化发现，然后解析输出通过GitHub API发布内联评论。**[正确]**
- C) 在审查提示词中包含明确的格式指令，要求每个发现遵循可解析的模板，如 `[FILE:path] [LINE:n] [SEVERITY:level] ...`。
- D) 保留叙述性审查格式，但添加摘要步骤，使用Claude生成发现的结构化JSON摘要。

为什么选B： 使用 `--output-format json` 和 `--json-schema` 在CLI级别强制结构化输出，保证具有所需字段（文件路径、行号、严重性、建议修复）的格式良好的JSON，可以可靠地解析并通过GitHub API发布为内联PR评论。它利用了专为结构化输出设计的内置CLI功能。

第17题（场景：Claude Code用于持续集成）

情境： 你的团队使用Claude Code生成代码建议，但你注意到一个模式：不明显的问题——破坏边缘情况的性能优化、意外改变行为的清理——只有当另一名团队成员审查PR时才会被发现。Claude在生成过程中的推理显示它考虑了这些情况但认为其方法是正确的。哪种方法直接解决了这种自检限制的根本原因？

哪种方法直接解决了根本原因？

- A) 运行Claude Code的第二个独立实例，在没有生成器推理的情况下审查变更。**[正确]**
- B) 为生成阶段启用扩展思维模式，在产生建议之前允许更彻底的审议。

- C) 在生成提示词中添加明确的自我审查指令，要求Claude在最终确定输出之前批评自己的建议。
- D) 在提示词上下文中包含完整的测试文件和文档，使Claude在生成期间更好地理解预期行为。

为什么选A： 没有生成器推理访问权限的第二个独立Claude Code实例通过避免确认偏差直接解决了根本原因。这种"新鲜视角"反映了人工同行评审，另一位审查者会发现作者合理化了的问题。

第18题（场景：Claude Code用于持续集成）

情境： 你的代码审查组件是迭代的：Claude分析修改的文件，然后可能通过工具调用请求相关文件（导入、基类、测试）以在提供最终反馈之前理解上下文。你的应用程序定义了一个让Claude请求文件内容的工具；Claude调用该工具，获取结果，并继续分析。你正在评估批处理以降低API成本。在考虑此工作流的批处理时，主要技术限制是什么？

主要技术限制是什么？

- A) 批处理不包含关联ID来将输出映射回输入请求。
- B) 异步模型无法在请求中途执行工具并返回结果以便Claude继续分析。**[正确]**
- C) Batch API不支持请求参数中的工具定义。
- D) 批处理最长24小时的延迟对于PR反馈来说太慢，尽管工作流在其他方面可以运行。

为什么选B： "即发即忘"的异步Batch API模型没有机制在请求中拦截工具调用、执行工具并返回结果供Claude继续分析。这与需要在单个逻辑交互中进行多轮工具请求/响应的迭代工具调用工作流从根本上不兼容。

第19题（场景：Claude Code用于持续集成）

情境： 你的CI/CD系统运行三种基于Claude的分析：（1）每个PR上的快速风格检查，在完成之前阻止合并；（2）每周对整个代码库进行全面安全审计；（3）最近修改模块的夜间测试用例生成。Message Batches API提供50%的节省，但处理时间最长可达24小时。你希望在保持可接受的开发者体验的同时优化API成本。哪种组合正确地将每个任务与API方式匹配？

哪种组合是正确的？

- A) 对三个任务都使用Message Batches API以最大化50%的节省，配置管道轮询批次完成情况。
- B) 对PR风格检查使用同步调用；对每周安全审计和夜间测试生成使用Message Batches API。**[正确]**
- C) 对三个任务都使用同步调用以保持一致的响应时间，依靠提示词缓存降低跨工作负载的成本。
- D) 对PR风格检查和夜间测试生成使用同步调用；只对每周安全审计使用Message Batches API。

为什么选B： PR风格检查阻止开发者，需要通过同步调用立即响应，而每周安全审计和夜间测试生成是有灵活截止期限的定时任务，可以容忍最多24小时的批处理窗口——两者都能获得50%的节省。

第20题（场景：Claude Code用于持续集成）

情境： 你的自动审查发现了真实问题，但开发者报告反馈不可操作。发现包含"复杂的工单路由逻辑"或"潜在的空指针"等短语，没有指定具体需要改变什么。当你添加"始终包含具体修复建议"等详细指令时，模型仍然产生不一致的输出——有时详细，有时模糊。哪种提示词技术最可靠地产生一致可操作的反馈？

哪种提示词技术最可靠？

- A) 进一步细化指令，对反馈格式的每个部分（位置、问题、严重性、建议修复）提出更明确的要求。
- B) 扩展上下文窗口以包含更多周边代码库，使模型有足够信息提出具体修复。
- C) 实现两步骤方法，一个提示词识别问题，另一个生成修复，允许专业化。
- D) 添加3-4个少样本示例，显示所需的确切格式：识别的问题、代码中的位置、具体修复建议。**[正确]**

为什么选D： 当指令单独产生可变结果时，少样本示例是实现一致输出格式最有效的技术。提供3-4个显示确切所需结构（问题、位置、具体修复）的示例，为模型提供了比抽象指令更可靠的具体模式。

第21题（场景：Claude Code用于持续集成）

情境： 你的CI管道包括两种基于Claude的代码审查模式：合并前提交钩子（在完成之前阻止PR合并）和"深度分析"（隔夜运行，轮询批次完成情况，并将详细建议发布到PR）。你想使用Message Batches API降低API成本，它提供50%的节省但需要轮询且最长可达24小时。哪种模式应该使用批处理？

哪种模式应该使用批处理？

- A) 只有合并前提交钩子。
- B) 只有深度分析。**[正确]**
- C) 两种模式。
- D) 两种模式都不使用。

为什么选B： 深度分析是批处理的理想候选，因为它已经隔夜运行，容忍延迟，并在发布结果之前使用轮询模型——与Message Batches API的异步、基于轮询的架构相匹配，同时获得50%的节省。

第22题（场景：Claude Code用于持续集成）

情境： 你的自动审查分析注释和文档字符串。当前提示词指示Claude"检查注释是否准确和最新"。发现经常标记可接受的模式（TODO标记、简单描述），同时遗漏描述代码不再实现的行为的注释。哪种变更解决了这种不一致分析的根本原因？

哪种变更解决了根本原因？

- A) 包含 `git blame` 数据，使Claude能够识别早于近期代码变更的注释。
- B) 添加误导性注释的少样本示例，帮助模型识别代码库中的类似模式。
- C) 在分析之前过滤TODO、FIXME和描述性注释模式以减少噪音。
- D) 指定显式标准：仅当注释声明的行为与代码的实际行为矛盾时才标记。**[正确]**

为什么选D： 显式标准——仅在声明的行为与实际代码行为矛盾时标记注释——通过用问题的精确定义替换模糊指令直接解决了根本原因。这减少了对可接受模式的误报和对真正误导性注释的遗漏。

第23题（场景：Claude Code用于持续集成）

情境： 你的自动代码审查系统显示不一致的严重性评级——类似的空指针风险问题在某些PR中被评为"严重"，而在其他PR中只被评为"中等"。开发者调查显示信任度下降——许多人开始不阅读就忽视发现，因为"有一半是错的"。高误报类别侵蚀了对准确类别的信任。哪种方法在改善系统的同时最好地恢复开发者信任？

哪种方法最好地恢复开发者信任？

- A) 临时禁用高误报类别（风格、命名、文档），在改善提示词的同时只保留高精度类别。**[正确]**
- B) 保持所有类别启用，但为每个发现显示置信度分数，以便开发者决定调查什么。
- C) 保持所有类别启用，在接下来几周内为每个类别添加少样本示例以提高准确性。
- D) 在所有类别中均匀降低严格性以降低总体误报率。

为什么选A： 临时禁用高误报类别通过删除导致开发者忽视一切的嘈杂发现立即阻止信任侵蚀，同时保留来自安全和正确性等高精度类别的价值。它还为在重新启用之前改善有问题类别的提示词创造了空间。

第24题（场景：Claude Code用于持续集成）

情境： 你的自动审查为每个PR生成测试用例建议。审查一个添加课程完成跟踪的PR时，Claude建议了10个测试用例，但开发者反馈有6个重复了现有测试套件已覆盖的场景。哪种变更最有效地减少重复建议？

哪种变更最有效？

- A) 在上下文中包含现有测试文件，使Claude能够确定哪些场景已经被覆盖。**[正确]**
- B) 将请求的建议数量从10个减少到5个，假设Claude首先优先考虑最有价值的案例。
- C) 添加指令，指导Claude专门关注边缘情况和错误条件，而非成功路径。
- D) 实现后处理，通过关键词重叠过滤描述与现有测试名称匹配的建议。

为什么选A： 包含现有测试文件修复了重复的根本原因：Claude只有在知道已有哪些测试的情况下才能避免建议已覆盖的场景。这为Claude提供了提出真正新的、有价值测试所需的信息。

第25题（场景：Claude Code用于持续集成）

情境： 初始自动审查识别出12个发现后，开发者推送新提交以解决问题。重新运行审查产生8个发现，但开发者报告有5个重复了对新提交中已修复代码的先前评论。在保持彻底性的同时消除这种冗余反馈的最有效方法是什么？

消除冗余反馈最有效的方法是什么？

- A) 只在创建PR和最终合并前状态时运行审查，跳过中间提交。
- B) 添加后处理过滤器，在发布评论之前删除通过文件路径和问题描述匹配先前发现的内容。

- C) 将审查范围限制在最近推送中更改的文件，排除早期提交中的文件。
- D) 在上下文中包含先前的审查发现，指示Claude只报告新问题或仍未解决的问题。**[正确]**

为什么选D： 在上下文中包含先前的审查发现使Claude能够区分新问题和最近提交中已解决的问题。这在使用Claude的推理避免对已修复代码提供冗余反馈的同时保持了审查彻底性。

第26题（场景：Claude Code用于持续集成）

情境： 你的管道脚本运行 `claude "分析此拉取请求的安全问题"`，但作业无限期挂起。日志显示Claude Code在等待交互式输入。在自动化管道中运行Claude Code的正确方法是什么？

正确的方法是什么？

- A) 添加 `--batch` 标志：`claude --batch "分析此拉取请求的安全问题"`。
- B) 添加 `-p` 标志：`claude -p "分析此拉取请求的安全问题"`。**[正确]**
- C) 从 `/dev/null` 重定向stdin：`claude "分析此拉取请求的安全问题" < /dev/null`。
- D) 在运行命令之前设置环境变量 `CLAUDE_HEADLESS=true`。

为什么选B： `-p`（或 `--print`）标志是以非交互方式运行Claude Code的文档化方式。它处理提示词，将结果打印到stdout，并退出而不等待用户输入——非常适合CI/CD管道。

第27题（场景：Claude Code用于持续集成）

情境： 一个拉取请求更改了库存跟踪模块中的14个文件。对所有文件一起分析的单次审查产生了不一致的结果：对一些文件的详细反馈，对另一些的表面评论，遗漏了明显的错误，以及矛盾的反馈（一个模式在一个文件中被标记，但在同一PR的另一个文件中相同代码被批准）。应该如何重构审查？

应该如何重构审查？

- A) 运行三次独立的完整PR审查，只标记至少在三次中两次出现的问题。
- B) 拆分为聚焦检查：分别审查每个文件的本地问题，然后运行独立的集成导向检查来检查跨文件数据流。**[正确]**
- C) 要求开发者在运行自动审查之前将大型PR拆分为3-4个文件的较小提交。
- D) 切换到更大的具有更大上下文窗口的模型，使其能够在一次检查中对14个文件给予足够的注意。

为什么选B： 聚焦的每文件检查通过确保一致的深度和可靠的本地问题检测直接解决了根本原因——注意力分散。然后，独立的集成导向检查涵盖了跨文件的问题，如依赖关系和数据流交互。

第28题（场景：Claude Code用于持续集成）

情境： 你的自动代码审查每个PR平均有15个发现，开发者报告误报率为40%。瓶颈是调查时间：开发者必须点击每个发现阅读Claude的推理后才决定是否修复或忽视。你的CLAUDE.md已经包含了可接受模式的全面规则，利益相关者拒绝了任何在开发者看到之前过滤发现的方法。哪种变更最好地解决调查时间问题？

哪种变更最好地解决调查时间？

- A) 要求Claude在每个发现中直接包含其推理和置信度估计。**[正确]**
- B) 添加后处理器，分析发现模式，自动抑制匹配历史误报特征的发现。
- C) 将发现分类为"阻塞问题"vs"建议"，按级别设置不同的审查要求。
- D) 配置Claude只显示高置信度发现，在开发者看到之前过滤不确定的标记。

为什么选A： 在每个发现中直接包含推理和置信度通过让开发者快速分类而无需打开每个发现来减少调查时间。它满足"不过滤"约束，因为所有发现仍然可见，同时加速了开发者的决策。

第29题（场景：Claude Code用于持续集成）

情境： 对你的自动代码审查的分析显示，按发现类别的误报率存在较大差异：安全/正确性发现有8%的误报，性能发现18%，风格/命名发现52%，文档发现48%。开发者调查显示信任度下降——许多人开始不阅读就忽视发现，因为"有一半是错的"。高误报类别侵蚀了对准确类别的信任。哪种方法在改善系统的同时最好地恢复开发者信任？

哪种方法最好地恢复开发者信任？

- A) 临时禁用高误报类别（风格、命名、文档），在改善提示词的同时只保留高精度类别。**[正确]**
- B) 保持所有类别启用，但为每个发现显示置信度分数，以便开发者决定调查什么。
- C) 保持所有类别启用，在接下来几周内为每个类别添加少样本示例以提高准确性。
- D) 在所有类别中均匀降低严格性以降低总体误报率。

为什么选A： 临时禁用高误报类别通过删除导致开发者忽视一切的嘈杂发现立即阻止信任侵蚀，同时保留来自安全和正确性等高精度类别的价值。它还为在重新启用之前改善有问题类别的提示词创造了空间。

第30题（场景：Claude Code用于持续集成）

情境： 你的团队希望降低自动化分析的API成本。目前，同步Claude调用支持两个工作流：（1）阻塞性的合并前检查，必须在开发者合并之前完成；（2）每晚生成的技术债务报告，供次日早晨审阅。你的经理提议将两者都迁移到Message Batches API以节省50%。应该如何评估这个提议？

应该如何评估这个提议？

- A) 将两者都迁移到批处理，如果批次耗时过长则回退到同步调用。
- B) 将两个工作流都迁移到批处理，带状态轮询以验证完成情况。
- C) 仅对技术债务报告使用批处理；保留合并前检查的同步调用。**[正确]**
- D) 对两个工作流都保留同步调用以避免批处理结果排序问题。

为什么选C： Message Batches API处理最长可达24小时，没有延迟SLA，这对于隔夜技术债务报告是可接受的，但对于开发者等待的阻塞性合并前检查是不可接受的。这根据延迟要求将每个工作流与正确的API匹配。

场景：使用Claude Code生成代码

第31题（场景：使用Claude Code生成代码）

情境： 你要求Claude Code实现一个将API响应转换为内部标准化格式的函数。经过两次迭代，输出结构仍然不符合预期——一些字段的嵌套方式不同，时间戳格式不正确。你用散文描述了需求，但Claude每次都以不同方式解读。

下一次迭代哪种方法最有效？

- A) 编写描述预期输出结构的JSON Schema，并在每次迭代后验证Claude的输出。
- B) 提供2-3个具体的输入输出示例，显示代表性API响应的预期转换。**[正确]**
- C) 用更高的技术精度重写需求，指定确切的字段映射、嵌套规则和时间戳格式字符串。
- D) 要求Claude解释其对需求的当前理解，以确定解释在哪里产生分歧。

为什么选B： 具体的输入输出示例通过向Claude展示确切的预期转换结果消除了散文描述中固有的歧义。这直接解决了根本原因——对文本需求的误解——通过提供字段嵌套和时间戳格式的明确模式。

第32题（场景：使用Claude Code生成代码）

情境： 你需要添加Slack作为新的通知渠道。现有代码库对邮件、短信和推送渠道有清晰、成熟的模式。然而，Slack的API提供了根本不同的集成方式——传入webhooks（简单、单向）、bot tokens（支持投递确认和程序化控制）或Slack Apps（双向事件，需要工作区审批）。你的任务说"添加Slack支持"，没有指定集成方法或要求投递跟踪等高级功能。

应该如何处理这个任务？

- A) 以直接执行模式开始，使用传入webhooks匹配现有的单向通知模式。
- B) 切换到规划模式探索集成选项和架构影响，然后在实现之前提出建议。**[正确]**
- C) 以直接执行模式开始，使用现有模式搭建Slack渠道类，推迟集成方法决策。
- D) 以直接执行模式开始，使用bot-token方法确保可能的投递确认。

为什么选B： Slack集成有多种有效方法，架构影响显著不同，需求也不明确。规划模式让你在实现之前评估webhooks、bot tokens和Slack Apps之间的权衡并对方法达成一致。

第33题（场景：使用Claude Code生成代码）

情境： 你的CLAUDE.md文件已增长到400+行，混合了编码标准、测试约定、详细的PR审查清单、部署指令和数据库迁移程序。你希望Claude始终遵循编码标准和测试约定，但仅在执行这些任务时应用PR审查、部署和迁移指导。

哪种重构方式最有效？

- A) 将所有指导移入按工作流类型组织的独立技能文件，在CLAUDE.md中只留下简短的项目描述。
- B) 在CLAUDE.md中保留所有内容，但使用 `@import` 语法按类别整理到单独维护的文件中。
- C) 将CLAUDE.md拆分为 `.claude/rules/` 下带有路径绑定glob模式的文件，使每个规则只对相关文件类型加载。

- D) 在CLAUDE.md中保留通用标准，为工作流特定指导（PR审查、部署、迁移）创建带有触发关键词的技能。**[正确]**

为什么选D： CLAUDE.md内容在每次会话中加载，确保编码标准和测试约定始终适用，而技能在Claude检测到触发关键词时按需调用——非常适合PR审查、部署和迁移等工作流特定指导。

第34题（场景：使用Claude Code生成代码）

情境： 你的任务是将团队的单体应用重构为微服务。这涉及到跨数十个文件的变更，以及关于服务边界和模块依赖关系的决策。

应该选择哪种方式？

- A) 切换到规划模式探索代码库、理解依赖关系，并在进行变更之前设计实现方法。**[正确]**
- B) 以直接执行模式开始，只在实现过程中遇到意外复杂性时切换到规划。
- C) 以直接执行模式开始，进行渐进式变更，让实现揭示自然的服务边界。
- D) 使用指定每个服务结构的详细预先指令进行直接执行。

为什么选A： 规划模式是复杂架构重构（如拆分单体）的正确策略：它允许在提交到跨多个文件的潜在高代价变更之前进行安全探索和明智的边界决策。

第35题（场景：使用Claude Code生成代码）

情境： 你的团队创建了一个 `/analyze-codebase` 技能，执行深度代码分析——依赖扫描、测试覆盖率计数和代码质量指标。运行该命令后，团队成员报告Claude在会话中变得不那么响应，并失去了原始任务的上下文。

在保留完整分析能力的同时，如何最有效地解决这个问题？

- A) 在技能前置内容中添加 `context: fork`，以在隔离的子代理上下文中运行分析。**[正确]**
- B) 在前置内容中添加 `model: haiku`，使用更快、更便宜的模型进行分析。
- C) 将技能拆分为三个较小的技能，每个产生较少的输出。
- D) 在技能中添加指令，在显示之前将所有结果压缩为简短摘要。

为什么选A： `context: fork` 在隔离的子代理上下文中运行分析，使大量输出不污染主会话的上下文窗口，Claude不会失去原始任务的轨迹。它在保持完整分析能力的同时保持主会话的响应性。

第36题（场景：使用Claude Code生成代码）

情境： 你的团队在 `.claude/skills/commit/SKILL.md` 中使用 `/commit` 技能。一位开发者想为他们的个人工作流定制它（不同的提交消息格式、额外检查），而不影响队友。

你的建议是什么？

- A) 在 `~/.claude/skills/` 下以不同名称创建个人版本，例如 `/my-commit`。**[正确]**

- B) 在项目技能前置内容中基于用户名添加条件逻辑。
- C) 在 `~/.claude/skills/commit/SKILL.md` 中创建同名的个人版本。
- D) 在个人技能前置内容中设置 `override: true` 以优先于项目版本。

为什么选A： 个人技能优先于同名的项目技能，因此重复使用 `commit` 这个名称，会让团队的技能仅对这位开发者悄悄被遮蔽——每当团队改进 `/commit` 时，他都收不到更新，还得自己记住当前运行的是同名命令下的另一个技能。将个人版本命名为 `/my-commit` 可以完全避免这种冲突：开发者仍可继续使用团队维护的 `/commit`，同时拥有一个名称清晰、独立的个人工作流技能，不会把两者搞混，也不会错过团队的更新。

第37题（场景：使用Claude Code生成代码）

情境： 你的团队使用Claude Code已有几个月。最近，三位开发者报告Claude遵循"始终包含全面的错误处理"指导，但一位刚加入的第四位开发者说Claude不遵循它。所有四人都在同一个仓库工作，代码是最新的。

最可能的原因和解决方法是什么？

- A) 指导存在于原始开发者的用户级 `~/.claude/CLAUDE.md` 文件中，而非项目 `.claude/CLAUDE.md` 中。将指令移到项目级文件，使所有团队成员都能接收到它。**[正确]**
- B) 新开发者的 `~/.claude/CLAUDE.md` 包含覆盖项目设置的冲突指令；他们应该删除冲突的部分。
- C) Claude Code随时间学习每用户偏好；新开发者必须重复该要求直到Claude"记住"它。
- D) Claude Code在首次读取后缓存CLAUDE.md；原始开发者使用缓存版本。所有人应该清除Claude Code缓存。

为什么选A： 如果指导只添加到原始开发者的用户级配置而非项目级 `.claude/CLAUDE.md`，新团队成员将不会接收到它。将其移到项目级配置确保所有当前和未来的团队成员自动获得指导。

第38题（场景：使用Claude Code生成代码）

情境： 你发现在生成新API端点时，将2-3个完整的端点实现示例作为上下文包含显著提高了一致性。然而，这些上下文只在创建新端点时有用——不适用于调试、代码审查或API目录中的其他工作。

哪种配置方式最有效？

- A) 将端点示例和模式文档添加到项目CLAUDE.md，使其始终可用。
- B) 通过将代码复制到提示词中，在每次生成请求中手动引用端点示例。
- C) 在 `.claude/rules/api/` 中配置路径特定规则，包含端点示例并在API目录中工作时激活。
- D) 创建引用端点示例并包含模式遵循指令的技能，通过斜杠命令按需调用。**[正确]**

为什么选D： 按需调用的技能只在生成新端点时加载示例上下文，而非在调试或审查等不相关任务中。这在需要时保留高质量生成的同时保持主上下文清洁。

第39题（场景：使用Claude Code生成代码）

情境： 你的团队创建了一个 `/migration` 技能，通过 `$ARGUMENTS` 接受迁移名称生成数据库迁移文件。在生产中你观察到三个问题：（1）开发者经常在没有参数的情况下运行技能，导致命名不好的文件；（2）技能有时使用来自无关先前对话的数据库Schema详情；（3）一位开发者在技能拥有广泛工具访问权限时意外运行了破坏性测试清理。

哪种配置方式修复了所有三个问题？

- A) 使用位置参数 `$1` 和 `$2` 而非 `$ARGUMENTS` 来强制特定输入，通过 `@` 语法包含显式Schema文件引用以控制上下文，并添加关于破坏性操作的前置内容描述警告。
- B) 在前置内容中添加 `argument-hint` 以请求必要参数，使用 `context: fork` 隔离执行，并将 `allowed-tools` 限制为文件写入操作。**[正确]**
- C) 拆分为 `/migration-create` 和 `/migration-apply` 技能，如果缺少迁移名称则添加验证指令请求，并为每个使用不同的 `allowed-tools` 范围。
- D) 在技能SKILL.md中添加验证指令确保 `$ARGUMENTS` 是有效名称，添加提示词忽略先前对话上下文，并列出禁止操作以避免。

为什么选B： 这使用三个独立的配置功能来解决每个问题：`argument-hint` 改善参数输入并减少缺失参数，`context: fork` 防止来自先前对话的上下文泄漏，`allowed-tools` 将技能约束到安全的文件写入操作，防止破坏性操作。

第40题（场景：使用Claude Code生成代码）

情境： 你的代码库在不同区域有不同的编码约定：React组件使用带hooks的函数式风格，API处理程序使用带特定错误处理的async/await，数据库模型遵循仓库模式。测试文件分布在代码库中，与被测试的代码相邻（例如，`Button.test.tsx` 在 `Button.tsx` 旁边），你希望所有测试无论位置如何都遵循相同的约定。

确保Claude在生成代码时自动应用正确约定最受支持的方式是什么？

- A) 将所有约定放在根目录CLAUDE.md中按区域标题分组，依靠Claude推断哪个部分适用。
- B) 在 `.claude/skills/` 中为每种代码类型创建技能，在每个SKILL.md中嵌入约定。
- C) 在每个子目录中放置包含该区域约定的独立CLAUDE.md文件。
- D) 在 `.claude/rules/` 下创建带有YAML前置内容指定glob模式的规则文件，根据文件路径条件性地应用约定。**[正确]**

为什么选D： 带有YAML前置内容和glob模式的 `.claude/rules/` 文件（例如，`**/*.test.tsx`、`src/api/**/*.ts`）无论目录结构如何都能实现确定性的、基于路径的约定应用。这是处理分布式测试文件等跨领域模式最受支持的方式。

第41题（场景：使用Claude Code生成代码）

情境： 你想创建一个自定义斜杠命令 `/review`，运行团队的标准代码审查清单。当开发者克隆或更新仓库时，它应该对每位开发者可用。

应该在哪里创建命令文件？

- A) 在每个开发者主目录的 `~/.claude/commands/` 中。
- B) 在项目仓库的 `.claude/commands/` 下。**[正确]**
- C) 在 `.claude/config.json` 中作为命令数组。
- D) 在根项目CLAUDE.md中。

为什么选B： 在项目仓库内 `.claude/commands/` 下放置自定义斜杠命令确保它们受版本控制，并对每个克隆或更新仓库的开发者自动可用。这是Claude Code中项目级自定义命令的预期位置。

第42题（场景：使用Claude Code生成代码）

情境： 你的团队CLAUDE.md增长到500+行，混合了TypeScript约定、测试指导、API模式和部署程序。开发者觉得很难找到和更新正确的部分。

Claude Code支持什么方式将项目级指令整理为聚焦的主题模块？

- A) 定义将文件模式映射到CLAUDE.md内特定部分的 `.claude/config.yaml` 映射文件。
- B) 在 `.claude/rules/` 中创建独立的Markdown文件，每个覆盖一个主题（例如，`testing.md`、`api-conventions.md`）。**[正确]**
- C) 在相关子目录的README.md文件中拆分指令，Claude自动将其作为指令加载。
- D) 在目录树的不同级别创建多个名为CLAUDE.md的文件，每个覆盖父级指令。

为什么选B： Claude Code支持 `.claude/rules/` 目录，你可以在那里为主题指导（例如，`testing.md`、`api-conventions.md`）创建独立的Markdown文件，允许团队将大型指令集整理为聚焦的、可维护的模块。

第43题（场景：使用Claude Code生成代码）

情境： 你创建了一个自定义技能 `/explore-alternatives`，团队用它在选择方案之前集思广益和评估实现方法。开发者报告运行技能后，后续Claude响应受到替代方案讨论的影响——有时引用被拒绝的方法或保留干扰实际实现的探索上下文。

应该如何最有效地配置这个技能？

- A) 在技能中使用 `!` 前缀将探索逻辑作为bash子进程运行。
- B) 在技能前置内容中添加 `context: fork`。**[正确]**
- C) 拆分为两个技能——`/explore-start` 和 `/explore-end`——以标记何时应该丢弃探索上下文的边界。
- D) 在 `~/.claude/skills/` 而非 `.claude/skills/` 中创建技能。

为什么选B： `context: fork` 在隔离的子代理上下文中运行技能，使探索讨论不污染主对话历史。这防止了被拒绝的方法和集思广益上下文影响后续的实现工作。

第44题（场景：使用Claude Code生成代码）

情境： 你的团队想添加一个GitHub MCP服务器，通过Claude Code搜索PR和检查CI状态。六位开发者每人有自己的个人GitHub访问令牌。你希望在团队中保持一致的工具，不将凭证提交到版本控制。

哪种配置方式最有效？

- A) 让每位开发者通过 `claude mcp add --scope user` 在用户范围内添加服务器。
- B) 创建一个从 `.env` 文件读取令牌并代理GitHub API调用的MCP服务器包装器，然后将包装器添加到项目 `.mcp.json`。
- C) 使用环境变量替换（`${GITHUB_TOKEN}`）进行身份验证，将服务器添加到项目 `.mcp.json`，并在项目README中记录所需的环境变量。**[正确]**
- D) 在项目范围内用占位令牌配置服务器，然后告诉开发者在其本地配置中覆盖它。

为什么选C： 带有环境变量替换的项目 `.mcp.json` 是惯用方式：它提供了一个版本控制的MCP配置单一来源，同时让每位开发者通过环境变量提供凭证。在README中记录变量使入职变得容易，而不提交密钥。

第45题（场景：使用Claude Code生成代码）

情境： 你正在120个文件的代码库中添加外部API调用的错误处理包装器。工作分三个阶段：（1）发现所有调用点和模式；（2）协作设计错误处理方法；（3）一致地实现包装器。在第一阶段，Claude生成列出数百个带上下文的调用点的大量输出，在发现完成之前快速填满上下文窗口。

在保持实现一致性的同时完成任务哪种方式最有效？

- A) 使用探索子代理进行第一阶段，隔离详细的发现输出并返回摘要，然后在主对话中继续第2-3阶段。**[正确]**
- B) 在主对话中完成所有阶段，在处理文件时定期使用 `/compact` 减少上下文使用。
- C) 切换到带 `--continue` 的无头模式，在批处理调用之间传递明确的上下文摘要以保持连续性。
- D) 在CLAUDE.md中定义错误处理模式，然后跨多个会话批量处理文件，依靠共享记忆文件保持一致性。

为什么选A： 探索子代理将详细的发现输出隔离在独立上下文中，只向主对话返回简洁摘要。这为协作设计和一致实现阶段保留了主上下文窗口，这些阶段中保留的上下文最有价值。

场景：客服代理

第46题（场景：客服代理）

情境： 在测试中，你注意到代理在用户询问订单状态时经常调用 `get_customer`，即使 `lookup_order` 更合适。解决这个问题应该首先检查什么？

应该首先检查什么？

- A) 实现预处理分类器，检测与订单相关的请求并直接路由到 `lookup_order`。
- B) 减少代理可用的工具数量以简化选择。
- C) 在系统提示词中添加少样本示例，涵盖所有可能的订单请求模式以改善工具选择。
- D) 检查工具描述，确保它们清楚地区分每个工具的目的。**[正确]**

为什么选D： 工具描述是模型用来决定调用哪个工具的主要输入。当代理持续选择错误的工具时，第一个诊断步骤是验证工具描述是否清楚地分离了每个工具的目的和使用边界。

第47题（场景：客服代理）

情境： 你的代理处理单一问题请求的准确率为94%（例如，"我需要订单#1234的退款"）。但当客户在一条消息中包含多个问题时（例如，"我需要订单#1234的退款，还想更新订单#5678的配送地址"），工具选择准确率降至58%。代理通常只解决一个问题，或者将参数混合在不同请求中。哪种方法最有效地提高多问题请求的可靠性？

哪种方法最有效？

- A) 实现预处理层，使用单独的模型调用将多问题消息分解为单独请求，独立处理每个请求，然后合并结果。
- B) 将相关工具合并为较少的通用工具。
- C) 在提示词中添加少样本示例，演示多问题请求的正确推理和工具排序。**[正确]**
- D) 实现响应验证，检测不完整的答案并自动重新提示代理解决遗漏的问题。

为什么选C： 演示多问题请求的正确推理和工具排序的少样本示例最有效，因为代理在单个问题上已经表现良好——它需要的是关于分解和路由多个问题以及分离参数模式的指导。

第48题（场景：客服代理）

情境： 生产日志显示，对于"订单#1234的退款"等简单请求，你的代理以91%的成功率在3-4次工具调用中解决问题。但对于"我被重复收费，折扣没有应用，我想取消"等复杂请求，代理平均超过12次工具调用，成功率仅54%——通常顺序调查问题并为每个问题获取冗余客户数据。哪种变更最有效地改善复杂请求的处理？

哪种变更最有效？

- A) 在阶段之间添加显式验证检查点，要求代理在进入下一个问题之前记录每个解决的问题的进度。
- B) 通过将 `get_customer`、`lookup_order` 和与账单相关的工具合并为单个 `investigate_issue` 工具来减少工具数量。
- C) 将请求分解为独立问题，然后使用共享客户上下文并行调查每个问题，然后综合最终解决方案。**[正确]**
- D) 在系统提示词中添加少样本示例，演示各种多方面账单场景的理想工具调用序列。

为什么选C： 分解为独立问题并使用共享客户上下文并行调查同时修复了两个关键问题：它通过在问题间重用共享上下文消除了冗余数据检索，并通过在综合单一解决方案之前并行化调查减少了总工具调用循环。

第49题（场景：客服代理）

情境： 你的代理首次联系解决率为55%，远低于80%的目标。日志显示它升级了简单案例（有照片证明的损坏商品标准替换），同时试图自主处理需要策略例外的复杂情况。最有效地改善升级校准的方法是什么？

最有效地改善升级校准的方法是什么？

- A) 要求代理在每次响应前自评置信度（1-10分），当置信度低于阈值时自动路由到人工。
- B) 部署一个在历史工单上训练的单独分类器模型，在主代理开始处理之前预测哪些请求需要升级。
- C) 在系统提示词中添加显式升级标准，带有显示何时升级versus自主解决的少样本示例。**[正确]**
- D) 实现情感分析来确定客户沮丧程度，当负面情感超过阈值时自动升级。

为什么选C： 带有少样本示例的显式升级标准直接解决了根本原因——简单和复杂案例之间不清晰的决策边界。这是最适度、最有效的首次干预，教导代理何时升级和何时自主解决，不需要额外的基础设施。

第50题（场景：客服代理）

情境： 调用 `get_customer` 和 `lookup_order` 后，代理拥有所有可用的系统数据，但仍面临不确定性。哪种情况是调用 `escalate_to_human` 最合理的触发器？

哪种情况最合理地需要升级？

- A) 客户想取消昨天发货明天到达的订单。代理应该升级，因为客户收到包裹后可能会改变主意。
- B) 客户声称没有收到订单，但跟踪显示三天前已在其地址签收。代理应该升级，因为呈现矛盾证据可能损害客户关系。
- C) 客户要求竞争对手价格匹配。你的策略允许在14天内对你自己网站的降价进行价格调整，但对竞争对手价格只字未提。代理应该为策略解读升级。**[正确]**
- D) 客户消息同时包含账单问题和产品退货。代理应该升级，以便人工可以在一次交互中协调两个问题。

为什么选C： 这是真正的策略缺口：公司规则涵盖了你自己网站的降价，但没有涉及竞争对手价格匹配。代理不得自行创造策略，应该升级以便人工判断如何解读或扩展现有规则。

第51题（场景：客服代理）

情境： 生产日志显示，在12%的情况下，你的代理跳过 `get_customer`，仅使用客户提供的姓名直接调用 `lookup_order`，有时导致账户误识别和错误退款。哪种变更最有效地修复这个可靠性问题？

哪种变更最有效？

- A) 添加少样本示例，显示代理即使在客户自愿提供订单详情时也始终先调用 `get_customer`。
- B) 实现路由分类器，分析每个请求并只启用该请求类型适合的工具子集。
- C) 添加程序化前提条件，在 `get_customer` 返回经过验证的客户标识符之前阻止 `lookup_order` 和 `process_refund`。**[正确]**
- D) 加强系统提示词，说明在任何订单操作之前通过 `get_customer` 进行客户验证是必须的。

为什么选C： 程序化前提条件提供了确定性保证，确保遵循所需的排序。这是最有效的方法，因为它消除了跳过验证的可能性，无论LLM的行为如何。

第52题（场景：客服代理）

情境： 生产指标显示，在解决复杂账单纠纷或多订单退货时，客户满意度评分比简单案例低15%——即使解决方案在技术上是正确的。根本原因分析显示代理提供准确的解决方案，但不一致地解释推理：有时省略相关策略详情，有时遗漏时间线信息或后续步骤。具体的上下文缺口因情况而异。你想在不增加人工监督的情况下改善解决方案质量。哪种方法最有效？

哪种方法最有效？

- A) 添加自我批评阶段，代理评估草稿响应的完整性——确保它解决了客户的问题，包含相关上下文，并预测后续问题。**[正确]**
- B) 添加确认阶段，代理在关闭之前询问"这完全解决了你的问题吗？"，允许客户在需要时请求更多信息。
- C) 对于复杂案例将模型从Haiku升级到Sonnet，根据定义的复杂性指标进行路由。
- D) 在系统提示词中实现少样本示例，为五种常见复杂案例类型显示完整解释，演示如何包含策略上下文、时间线和后续步骤。

为什么选A： 自我批评阶段（评估者-优化器模式）通过强制代理在呈现之前根据具体标准（如策略上下文、时间线和后续步骤）评估自己的草稿直接解决了不一致的解释完整性。这捕获了特定案例的缺口，不需要人工监督。

第53题（场景：客服代理）

情境： 生产指标显示你的代理每次解决平均超过4次API循环。分析显示Claude经常在即使最初两者都需要时，也在单独的顺序轮次中请求 `get_customer` 和 `lookup_order`。减少循环次数最有效的方法是什么？

减少循环次数最有效的方法是什么？

- A) 实现投机执行，自动与任何请求的工具并行调用可能需要的工具，无论请求什么都返回所有结果。
- B) 增加 `max_tokens` 给Claude更多空间进行规划并自然地合并工具请求。
- C) 创建复合工具如 `get_customer_with_orders`，将常见的查找组合捆绑为单次调用。
- D) 在提示词中指示Claude将工具请求捆绑到一个轮次中，并在下一次API调用之前一起返回所有结果。**[正确]**

为什么选D： 提示Claude将相关工具请求捆绑到单个轮次中利用了其原生的一次请求多个工具的能力。它以最小的架构变更直接修复了顺序调用模式。

第54题（场景：客服代理）

情境： 生产日志显示一个模式：客户引用具体金额（例如，"我提到的15%折扣"），但代理以错误的值响应。调查显示这些细节是在20+轮之前提到的，被压缩为"讨论了促销定价"等模糊摘要。哪种修复最有效？

哪种修复最有效？

- A) 将摘要阈值从70%提高到85%，使对话在触发摘要之前有更多空间。
- B) 在外部存储中存储完整的对话历史，当代理检测到"如我所说"等引用时实现检索。
- C) 将事务性事实（金额、日期、订单号）提取到在摘要历史之外每个提示词中包含的持久化"案例事实"块中。**[正确]**
- D) 修改摘要提示词，明确保留所有数字、百分比、日期和客户陈述的期望原文。

为什么选C： 摘要本质上会失去精确细节。将事务性事实提取到摘要历史之外的结构化"案例事实"块中保留了关键信息，使其在每个提示词中都可靠地可用，不管已经摘要了多少轮。

第55题（场景：客服代理）

情境： 你的 `get_customer` 工具在按姓名搜索时返回所有匹配项。目前，当有多个结果时，Claude选择有最近订单的客户，但生产数据显示对于模糊匹配15%的时间选择了错误的账户。应该如何解决这个问题？

应该如何解决这个问题？

- A) 实现置信度评分系统，在置信度85%以上自主行动，低于阈值时请求澄清。
- B) 指示Claude在 `get_customer` 返回多个匹配时，在采取任何客户特定操作之前请求额外标识符（邮件、电话或订单号）。**[正确]**
- C) 修改 `get_customer` 基于排名算法只返回单个最可能匹配，消除歧义。
- D) 在提示词中添加少样本示例，演示模糊匹配的正确推理和工具排序。

为什么选B： 要求用户提供额外标识符是解决歧义最可靠的方式，因为用户对自己的身份有确定性知识。多一次对话轮次是消除因选择错误账户导致的15%错误率的小代价。

第56题（场景：客服代理）

情境： 生产日志显示一个一致的模式：当客户消息中包含"账户"这个词时（例如，"我想查看我昨天下的订单的账户"），代理78%的时间首先调用 `get_customer`。当客户以不含"账户"的方式措辞类似请求时（例如，"我想查看我昨天下的订单"），它93%的时间首先调用 `lookup_order`。工具描述清晰且没有歧义。造成这种差异最可能的根本原因是什么？

最可能的根本原因是什么？

- A) 系统提示词包含基于"账户"等术语引导行为的关键词敏感指令，创建了意外的工具选择模式。**[正确]**
- B) 模型的基础训练在"账户"术语和客户相关操作之间创建了关联，覆盖了工具描述。
- C) 模型需要更多关于多概念消息的训练数据，应该在同时包含账户和订单术语的示例上进行微调。
- D) 工具描述需要额外的负面示例，指定何时不使用每个工具，以防止这种关键词诱导的混淆。

为什么选A： 系统性的关键词驱动模式（78% vs 93%）强烈表明系统提示词中有明确的路由逻辑对"账户"这个词做出反应，并将代理引导向客户相关工具。由于工具描述已经清晰，这种差异指向提示词级别的指令创建了意外的行为引导。

第57题（场景：客服代理）

情境： 生产日志显示，当用户询问订单时（例如，"查看我的订单#12345"），代理经常调用 `get_customer` 而非调用 `lookup_order`。两个工具都有最简化的描述（"获取客户信息"/"获取订单详情"），并接受看起来相似的标识符格式。提高工具选择可靠性的最有效第一步是什么？

最有效的第一步是什么？

- A) 实现路由层，在每个轮次之前分析用户输入，并基于检测到的关键词和ID模式预选正确的工具。
- B) 将两个工具合并为单个 `lookup_entity`，接受任何标识符并在内部决定查询哪个后端。
- C) 在系统提示词中添加少样本示例，演示正确的工具选择模式，带有5-8个将与订单相关的查询路由到 `lookup_order` 的示例。
- D) 扩展每个工具的描述，包含输入格式、示例查询、边缘情况以及解释何时使用它vs类似工具的边界。**[正确]**

为什么选D： 用输入格式、示例查询、边缘情况和清晰边界扩展工具描述直接修复了根本原因——没有给LLM提供足够信息来区分类似工具的最简化描述。这是提高LLM用于工具选择的主要机制的低成本、高影响的第一步。

第58题（场景：客服代理）

情境： 你正在为支持代理实现代理循环。每次Claude API调用后，你必须决定是继续循环（运行请求的工具并再次调用Claude）还是停止（向客户呈现最终答案）。什么决定这个决策？

什么决定这个决策？

- A) 检查Claude响应中的 `stop_reason` 字段——如果是 `tool_use` 则继续，如果是 `end_turn` 则停止。**[正确]**
- B) 解析Claude的文本中"我完成了"或"我还能帮什么吗？"等短语——自然语言信号表示任务完成。
- C) 设置最大迭代计数（例如，10次调用），达到时停止，无论Claude是否表示需要更多工作。
- D) 检查响应是否包含助手文本内容——如果Claude生成了解释性文本，循环应该终止。

为什么选A： `stop_reason` 是Claude用于循环控制的显式结构化信号：`tool_use` 表示Claude想运行一个工具并接收返回的结果，而 `end_turn` 表示Claude已完成其响应，循环应该结束。

第59题（场景：客服代理）

情境： 生产日志显示代理误解了MCP工具的输出：`get_customer` 返回Unix时间戳，`lookup_order` 返回ISO 8601日期，数字状态码（1=待处理，2=已发货）。一些工具是你无法修改的第三方MCP服务器。哪种数据格式规范化方法最可维护？

哪种方法最可维护？

- A) 使用PostToolUse钩子拦截工具输出，在代理处理之前应用格式转换。**[正确]**
- B) 修改你控制的工具以返回人类可读的格式，并为第三方工具创建包装器。
- C) 创建一个 `normalize_data` 工具，代理在每次数据检索后调用以转换值。

- D) 在系统提示词中添加详细的格式文档，解释每个工具的数据约定。

为什么选A： PostToolUse钩子提供了一个集中的、确定性的点来拦截和规范化所有工具输出——包括第三方MCP服务器数据——在代理处理之前。它更可维护，因为转换存在于代码中并统一应用，而非依赖LLM解读。

第60题（场景：客服代理）

情境： 生产日志显示代理有时在 `lookup_order` 更合适时选择 `get_customer`，特别是对于"我需要帮助处理最近的购买"等模糊查询。你决定在系统提示词中添加少样本示例以改善工具选择。哪种方法最有效地解决问题？

哪种方法最有效？

- A) 在每个工具描述中添加明确的"何时使用"和"何时不使用"指导，涵盖模糊情况。
- B) 按工具分组添加示例——所有 `get_customer` 场景在一起，然后所有 `lookup_order` 场景。
- C) 添加4-6个针对模糊场景的示例，每个都有为什么选择一个工具而非可能的替代品的推理说明。**[正确]**
- D) 添加10-15个针对每个工具的典型场景的清晰、明确的请求示例，演示正确的工具选择。

为什么选C： 将少样本示例针对发生错误的特定模糊场景，并带有明确的为什么一个工具比替代品更优的推理说明，教导模型边缘情况所需的比较决策过程。这比通用示例或声明性规则更有效。

问题 61（场景：对话式 AI 架构模式）

情境： 您的 `remove_team_member` 工具使用 `dry_run: boolean` 参数在执行前预览影响。生产监控显示 Agent 通过直接使用 `dry_run=false` 调用绕过了预览步骤。您需要确保每次删除操作之前都有用户明确确认的预览。

哪种方法最可靠？

- A) 添加服务器端验证，仅在 60 秒内发生过参数相同的 `dry_run=true` 调用时才允许 `dry_run=false`。
- B) 将工具标注为需要确认，并配置编排层在转发调用给标注工具前向用户请求审批。
- C) 在工具描述中添加详细指令和少样本示例，要求 Agent 始终先用 `dry_run=true` 调用并等待用户确认后再调用。
- D) 替换为两个工具：`preview_remove_member` 返回影响详情和一次性确认令牌；`execute_remove_member` 需要该令牌，将执行与预览绑定。**【正确】**

为什么选 D： 令牌绑定方案使得在未经预览的情况下执行在架构上成为不可能——执行工具字面上需要只有预览工具才能生成的令牌。这是唯一在代码层面强制执行约束的方案，而不依赖 LLM 遵守指令 (C)、时间启发式 (A) 或编排基础设施 (B)。

问题 62（场景：对话式 AI 架构模式）

情境： 生产监控显示您的 `search_catalog` 工具 12% 的时间会失败：8% 是网络超时（重试后成功），4% 是查询语法错误（无论重试多少次都不会成功）。目前两种错误类型的返回方式相同，导致无效重试。

您应该如何修改工具的错误处理？

- A) 在系统提示词中添加少样本示例，演示如何区分网络错误和语法错误。
- B) 对所有错误统一应用指数退避重试逻辑。
- C) 在工具内部对网络超时实现自动重试并退避；对语法错误立即返回并附带参数验证详情。**【正确】**
- D) 对所有错误返回带有 `retryable` 布尔标志和错误类型详情的结构。

为什么选 C： 在工具层面处理瞬态错误的重试是正确的抽象边界——工具对错误类型有明确了解，可以实现确定性重试逻辑，而不依赖 Agent 解释标志 (D) 或遵循提示词指令 (A)。统一退避 (B) 在语法错误上浪费时间，而这些错误永远不会成功。

问题 63（场景：对话式 AI 架构模式）

情境： 在多轮讨论投资策略的过程中，用户说了"我的风险承受能力很低"，然后又说"我想最大化回报"。他们现在问："我应该投资什么？"

哪种方法最能确保建议与用户的真实优先级保持一致？

- A) 指出矛盾，请用户澄清哪个更重要。**【正确】**
- B) 分别为两种情景提供建议。
- C) 按最近表达的偏好继续。
- D) 在不解决冲突的情况下推荐均衡投资组合。

为什么选 A： 当用户偏好直接矛盾时，指出冲突并请求澄清是确保建议与用户真实意图保持一致的唯一方式。最大化回报和低风险承受能力是根本不兼容的目标，需要人工决策。

问题 64（场景：对话式 AI 架构模式）

情境： 用户在多轮对话中细化播放列表偏好。用户说"我喜欢爵士乐"后两条消息，Claude 问"您喜欢什么流派？"

最可能的原因是什么？

- A) Claude 需要向量数据库连接才能维持对话记忆。
- B) 模型的上下文窗口已超出。
- C) Claude API 需要 `session_id` 参数。
- D) 您的应用程序没有在 `messages` 数组中包含之前的消息。**【正确】**

为什么选 D： Claude 没有服务器端记忆——每次 API 调用都是无状态的。如果不在每个请求的 `messages` 数组中包含完整的对话历史，Claude 就无法知道之前轮次发生了什么。向量数据库 (A) 和 `session_id` (C) 不是 Claude 架构的组成部分；两条消息的交流不可能超出上下文窗口 (B)。

问题 65（场景：对话式 AI 架构模式）

情境： 经过 40 分钟的烹饪会话，对话达到 78,000 个令牌。历史记录包括过敏信息、食谱缩放、澄清的烹饪术语和一般讨论。您必须在保留重要信息的同时减少令牌。

哪种方法最好地平衡了保留与令牌减少？

- A) 总结整个对话历史。
- B) 只保留最近的 20,000 个令牌。
- C) 提取关键结构化数据（过敏信息、数量、偏好），总结一般讨论，逐字保留最近的交流。**【正确】**
- D) 将完整对话存储在外部，通过语义搜索检索相关部分。

为什么选 C： 混合方法以最低成本保留最高价值的信息。过敏信息和食谱数量等关键事实被提取到紧凑的结构化块中（防止摘要过程中的精度损失），一般讨论被摘要，最近的交流逐字保留以保持对话连贯性。选项 A 和 B 有丢失关键饮食信息的风险；D 对单次烹饪会话来说过于复杂。

问题 66（场景：对话式 AI 架构模式）

情境： 用户反映在长时间对话中，助手会失去对早期话题和偏好的追踪。您当前的实现只保留最后 25 对消息。

最有效的解决方案是什么？

- A) 混合方法：摘要较旧的消息，同时逐字保留最近的消息。**【正确】**
- B) 对完整对话历史进行向量相似性搜索。
- C) 将窗口增加到 50 对消息。
- D) 每轮摘要已删除的消息并将运行摘要添加到前面。

为什么选 A： 混合方法解决了问题的两个维度：保留精确的最近上下文（对对话连贯性至关重要），同时维护对早期偏好的压缩表示。增加窗口 (C) 只是推迟了同样的问题。向量搜索 (B) 可能会错过与当前查询语义不相似的重要上下文。

问题 67（场景：对话式 AI 架构模式）

情境： 用户反映当对话超过 50 轮时，延迟增加，成本上升。

主要原因是什么？

- A) 每个 API 请求都包含完整的对话历史。**【正确】**
- B) 模型生成越来越长的响应。
- C) 随着历史记录增长，数据库操作变慢。
- D) 模型构建需要更多处理的内部用户画像。

为什么选 A： Claude 的 API 完全无状态——每个请求必须在 `messages` 数组中包含完整的对话历史。随着对话增长，每个请求携带更多令牌，这直接增加了处理延迟和成本。模型不在调用之间维护任何内部状态

（D 为假）。

问题 68（场景：对话式 AI 架构模式）

情境： 经过三个月的每周会话，对话历史增长到 85,000 个令牌。当用户问"我们关于孤立主题得出了什么结论？"时，助手给出了通用答案，而不是引用之前的讨论。

最有效的方法是什么？

- A) 滚动窗口截断。
- B) 逐步摘要，捕获关键结论。
- C) 语义嵌入，检索相关交流。**【正确】**
- D) 添加结构化 XML 标签标记讨论结论。

为什么选 C： 对对话历史进行语义搜索是唯一能扩展到三个月讨论，同时能按需检索特定相关交流的方法。滚动窗口截断 (A) 会丢弃大部分历史。渐进摘要 (B) 将讨论压缩成抽象概念，丢失了用户询问的具体结论。

问题 69（场景：对话式 AI 架构模式）

情境： 在 QA 测试期间，Claude 在最初 10–15 轮遵循系统提示词指南，但后续响应出现偏差。对话仍在令牌限制内。

最佳解决方案是什么？

- A) 将行为指南移至第一条用户消息。
- B) 20 轮后开始新的对话。
- C) 在对话断点处插入强化指南的用户角色消息。**【正确】**
- D) 使用响应后验证来重新生成不合规的响应。

为什么选 C： 定期注入行为提醒直接对抗指令漂移，随着对话历史积累在定期间隔重新建立约束。将指南移至第一条用户消息 (A) 降低了其权威性。响应后验证 (D) 是纠正性的而非预防性的，会增加显著延迟。

问题 70（场景：对话式 AI 架构模式）

情境： 您的 AI 辅导工具有一个 2,800 个令牌的系统提示词，定义了教学方法和适配规则。12 轮后，助手开始忽略熟练程度级别。

最有效的修复方案是什么？

- A) 每 4–5 轮注入提醒。
- B) 用演示熟练度适配的少样本示例替换冗长规则。**【正确】**
- C) 将关键规则放在系统提示词末尾。
- D) 评估响应，如果难度级别不匹配则重新生成。

为什么选 B： 带有声明性规则的 2,800 令牌系统提示词容易漂移，因为抽象规则要求模型在每轮都进行推理。将冗长规则替换为演示正确熟练度适配的具体少样本示例，为模型提供了清晰的行为模式可供匹配——这比多轮抽象指令更可靠地被遵循。

问题 71（场景：对话式 AI 架构模式）

情境： 您的助手必须保持热情的语气、解释推理并提出澄清问题。这些行为指南应该在哪里定义？

这些行为指南应该在哪里定义？

- A) 添加到每条用户消息之前。
- B) 在系统提示词中。**【正确】**
- C) 在第一条助手消息中。
- D) 在环境变量中。

为什么选 B： 系统提示词专门设计用于在整个对话过程中应用的持久行为约束和指南。在每条用户消息前添加 (A) 是冗余的开销。第一条助手消息 (C) 不可靠，因为模型可能偏离自己之前的陈述。环境变量 (D) 对模型行为没有影响。

问题 72（场景：对话式 AI 架构模式）

情境： 用户报告响应开头重复，如"当然！"和"我很乐意帮助！"

最有效的方法是什么？

- A) 附加带有直接响应开头的部分助手消息。**【正确】**
- B) 降低温度设置。
- C) 后处理响应以删除问候语。
- D) 在系统提示词中添加指令以避免这些短语。

为什么选 A： 用直接答案的开头预填助手响应，在生成层面阻止问候模式——模型从预填内容继续，而不是生成新的开头短语。系统提示词指令 (D) 可以有所帮助，但不那么可靠。后处理 (C) 是一个脆弱的变通方案。温度 (B) 控制随机性，而不是特定短语模式。

问题 73（场景：对话式 AI 架构模式）

情境： 当用户正在聊天时，webhook 通知您的系统用户的包裹已发货。您希望助手在下一个响应中自然地融入这一信息。

最佳方法是什么？

- A) 将发货状态添加到系统提示词。
- B) 发送立即的合成用户消息。
- C) 强制助手每轮调用状态工具。
- D) 将状态更新作为前缀附加到下一条用户消息。**【正确】**

为什么选 D： 将状态更新作为前缀附加到下一条用户消息，在自然对话边界注入实时上下文，不会中断对话流程。修改系统提示词 (A) 需要重建会话。合成用户消息 (B) 可能破坏自然对话流程。每轮强制调用工具 (C) 在事件罕见时很浪费。

问题 74（场景：对话式 AI 架构模式）

情境： 用户经常发送"为派对预订一个场地"这样的请求。助手提出 4 个以上的澄清问题，导致 35% 的放弃率。

哪种方法最能改善这种权衡？

- A) 使用隐藏默认值继续。
- B) 在一条复合消息中提出所有澄清问题。
- C) 明确说明假设并继续，同时邀请更正。**【正确】**
- D) 使用结构化接收表单。

为什么选 C： 明确说明假设并继续，为用户提供即时有用的响应，同时保留他们纠正错误假设的能力。隐藏默认值 (A) 让用户不知道做了什么假设。复合问题列表 (B) 仍然需要用户的前期努力。结构化表单 (D) 增加了更多摩擦，而不是更少。

问题 75（场景：对话式 AI 架构模式）

情境： 您的助手使用承包商角色系统提示词。早期轮次遵循规则，但到第 7 轮，助手给出通用建议。对话长度仅 2,500 个令牌。

最可能的原因是什么？

- A) 系统提示词只建立初始行为。
- B) 随着轮次积累，模型注意力减弱。
- C) 积累的助手响应稀释了系统提示词的影响。**【正确】**
- D) 系统提示词只发送一次。

为什么选 C： 随着助手响应在对话历史中积累，反映系统提示词行为约束的文本比例相对于不断增长的助手生成内容而言在减少。即使在短令牌长度下，模型也越来越多地匹配自己之前的输出模式，而不是系统提示词指令，从而加剧了漂移。

问题 76（场景：对话式 AI 架构模式）

情境： 用户提出模糊请求，如"你能帮忙处理报告吗？"助手通过提问多个问题（哪份报告？需要什么帮助？截止日期是什么？）来回应，导致 40% 的放弃率。

最佳解决方案是什么？

- A) 做出合理假设，明确说明，并提供调整。**【正确】**
- B) 在响应之前用较小的模型对模糊性进行分类。

- C) 使用预定义的解释，不说明假设。
- D) 限制助手每轮只提一个澄清问题。

为什么选 A： 以合理的明确假设继续，完全消除了来回交流，同时让用户保持知情和掌控。无声的预定义解释 (C) 当响应与用户意图不符时会让用户困惑。每轮一个问题的限制 (D) 仍然需要来回多轮。较小的分类模型 (B) 增加了延迟和基础设施复杂性，而没有解决核心 UX 问题。

实践练习

练习1：带升级逻辑的多工具代理

目标： 设计一个带有工具集成、结构化错误处理和升级的代理循环。

步骤：

1. 定义3-4个带详细描述的MCP工具（包含两个类似工具以测试工具选择）
2. 实现一个检查 `stop_reason`（`"tool_use"` / `"end_turn"`）的代理循环
3. 添加结构化错误响应：`errorCategory`、`isRetryable`、描述
4. 实现一个拦截器钩子，阻止超过阈值的操作并路由到升级
5. 使用多方面请求进行测试

领域： 1（代理架构），2（工具和MCP），5（上下文和可靠性）

练习2：为团队开发配置Claude Code

目标： 配置CLAUDE.md、自定义命令、路径特定规则和MCP服务器。

步骤：

1. 创建带有通用标准的项目级CLAUDE.md
2. 为不同代码区域创建带YAML前置内容的 `.claude/rules/` 文件（`paths: ["src/api/**/*"]`、`paths: ["**/*.test.*"]`）
3. 在 `.claude/skills/` 下创建带有 `context: fork` 和 `allowed-tools` 的项目技能
4. 在 `.mcp.json` 中配置带有环境变量的MCP服务器 + 在 `~/.claude.json` 中配置个人覆盖
5. 在不同复杂度的任务上测试规划模式vs直接执行

领域： 3（Claude Code配置），2（工具和MCP）

练习3：结构化数据提取管道

目标： JSON Schemas、`tool_use` 用于结构化输出、验证/重试循环、批处理。

步骤：

1. 定义带JSON Schema的提取工具（必填/可选字段、带"其他"的枚举、可空字段）
2. 构建验证循环：出错时，用文档、错误提取和具体验证错误重试
3. 为不同结构文档添加少样本示例
4. 通过Message Batches API使用批处理：100个文档，通过 `custom_id` 处理失败
5. 路由到人工：字段级置信度分数、文档类型分析

领域： 4（提示词工程），5（上下文和可靠性）

练习4：设计和调试多代理研究管道

目标： 子代理编排、上下文传递、错误传播、带来源跟踪的综合。

步骤：

1. 一个协调者带2+子代理（`allowedTools` 包含 `"Task"`，上下文在提示词中明确传递）
2. 通过在单个响应中进行多次 `Task` 调用并行运行子代理
3. 要求结构化子代理输出：声明、引用、来源URL、发布日期
4. 模拟子代理超时：向协调者返回结构化错误上下文，并继续使用部分结果
5. 使用冲突数据测试：保留两个值及归因；分离已确认vs有争议的发现

领域： 1（代理架构），2（工具和MCP），5（上下文和可靠性）

附录：技术和概念

技术	关键方面
Claude Agent SDK	AgentDefinition、代理循环、`stop_reason`、钩子（PostToolUse）、通过Task生成子代理、`allowedTools`
模型上下文协议（MCP）	MCP服务器、工具、资源、`isError`、工具描述、`.mcp.json`、环境变量
Claude Code	CLAUDE.md层次结构、带glob模式的 `.claude/rules/`、`.claude/commands/`、带SKILL.md的 `.claude/skills/`、规划模式、`/compact`、`--resume`、`fork_session`
Claude Code CLI	非交互模式的 `-p` / `--print`、`--output-format json`、`--json-schema`
Claude API	带JSON Schema的 `tool_use`、`tool_choice`（"auto"/"any"/强制）、`stop_reason`、`max_tokens`、系统提示词
Message Batches API	50%节省、最长24小时窗口、`custom_id`、不支持多轮工具调用
JSON Schema	必填vs可选、可空字段、枚举类型、"其他"+详情、严格模式

Pydantic	Schema验证、语义错误、验证/重试循环
内置工具	Read、Write、Edit、Bash、Grep、Glob——目的和选择标准
少样本提示词	模糊情况的针对性示例、对新模式的泛化
提示词链	顺序分解为聚焦检查
上下文窗口	令牌预算、渐进摘要、"中间迷失"、草稿文件
会话管理	恢复、`fork_session`、命名会话、上下文隔离
置信度校准	字段级评分、标注集校准、分层抽样

考试范围外的主题

以下相邻主题**不会**出现在考试中：

- 微调Claude模型或训练自定义模型
- Claude API认证、计费或账户管理
- 特定编程语言或框架中的详细实现（超出工具/Schema配置所需）
- 部署或托管MCP服务器（基础设施、网络、容器编排）
- Claude的内部架构、训练过程或模型权重
- 宪法AI、RLHF或安全训练方法
- 嵌入模型或向量数据库实现细节
- 计算机使用（浏览器自动化、桌面交互）
- 图像分析能力（视觉）
- 流式API或服务器发送事件
- 速率限制、配额或详细的API成本计算
- OAuth、API密钥轮换或认证协议细节
- 云供应商特定配置（AWS、GCP、Azure）
- 性能基准或模型比较指标
- 提示词缓存实现细节（超出了解其存在）
- 令牌计数算法或分词细节

备考建议

1. **使用Claude Agent SDK构建代理** — 实现包含工具调用、错误处理和会话管理的完整代理循环。练习子代理和明确的上下文传递。
2. **为真实项目配置Claude Code** — 使用CLAUDE.md层次结构、`.claude/rules/` 中的路径特定规则、带 `context: fork` 和 `allowed-tools` 的技能，以及MCP服务器集成。

3. **设计和测试MCP工具** — 编写区分类似工具的描述，返回带类别和重试标志的结构化错误，并针对模糊用户请求进行测试。
4. **构建数据提取管道** — 使用带JSON Schema的 `tool_use`、验证/重试循环、可选/可空字段，以及通过Message Batches API进行批处理。
5. **练习提示词工程** — 为模糊场景添加少样本示例、明确的审查标准，以及大型代码审查的多轮架构。
6. **研究上下文管理模式** — 从详细输出中提取事实、使用草稿文件，以及将发现委托给子代理以处理上下文限制。
7. **理解升级和人在环中** — 何时升级（策略缺口、用户明确请求、无法取得进展）以及基于置信度的路由工作流。
8. **在真正考试前参加模拟考试** — 它使用相同的场景和格式。