Claude Certified Architect — Foundations Certification

학습 가이드(공식 시험 가이드 기반)

소개

Claude Certified Architect — Foundations 인증은 실무에서 Claude 기반 솔루션을 설계하고 구현할 때 필요한 trade-off를 판단할 수 있는지 평가합니다. 시험 범위는 Claude Code, Claude Agent SDK, Claude API, Model Context Protocol(MCP)처럼 Claude로 프로덕션 애플리케이션을 만들 때 자주 쓰는 핵심 기술입니다.

시험 문제는 고객 지원 Agentic 시스템 구축, multi-agent research pipeline 설계, Claude Code의 CI/CD 통합, 개발자 생산성 tool 생성, 비정형 문서에서 structured data extraction 등 실제 업무 시나리오를 기반으로 합니다.

대상 응시자

권장 응시자는 Claude 기반 프로덕션 애플리케이션을 설계하고 출시해 본 **솔루션 아키텍트**입니다. 아래 영역에서 최소 6개월 이상 실무 경험이 있으면 가장 적합합니다.

- **Claude Agent SDK** — multi-agent orchestration, subagent 위임, tool 통합, 수명 주기 hook
- **Claude Code** — CLAUDE.md, MCP 서버, Agent Skills, 계획 모드
- **Model Context Protocol(MCP)** — backend 통합을 위한 tool과 resource
- **Prompt Engineering** — JSON schema, few-shot 예시, 데이터 extraction template
- **context window** — 긴 문서 처리, multi-agent context 전달
- **CI/CD pipeline** — 자동 코드 리뷰, 테스트 생성
- **escalation과 안정성** — 오류 처리, human-in-the-loop

시험 형식

항목	값
문제 유형	객관식(4개 중 정답 1개)
채점	100-1000점 척도, 합격 점수 **720점**
오답 감점	없음(모든 문제에 답하세요!)
시나리오	가능한 8개 중 4개(무작위 선택)

시험 내용: 5개 도메인

도메인	비중
1. Agent architecture와 orchestration	**27%**
2. tool 설계와 MCP 통합	**18%**
3. Claude Code 구성과 workflow	**20%**
4. Prompt Engineering과 structured output	**20%**
5. context 관리와 안정성	**15%**

시험 시나리오

시나리오 1: 고객 지원 Agent

Claude Agent SDK를 사용해 반품, 청구 분쟁, 계정 문제를 처리하는 Agent를 구축합니다. Agent는 MCP tool(`get_customer`, `lookup_order`, `process_refund`, `escalate_to_human`)를 사용합니다. 목표는 적절한 escalation과 함께 80% 이상의 첫 문의 해결률을 달성하는 것입니다.

시나리오 2: Claude Code를 사용한 코드 생성

Claude Code를 사용해 코드 생성, 리팩터링, 디버깅, 문서화를 가속합니다. 이를 사용자 지정 슬래시 명령과 CLAUDE.md 구성에 통합하고, 언제 계획 모드를 사용해야 하는지 이해해야 합니다.

시나리오 3: Multi-Agent Research System

Coordinator Agent가 웹 research, 문서 분석, 종합, 보고서 생성에 특화된 subagent에게 작업을 위임합니다. 시스템은 citation이 포함된 완전한 보고서를 생성해야 합니다.

시나리오 4: 개발자 생산성 tool

Agent가 엔지니어가 낯선 코드베이스를 탐색하고, boilerplate 코드를 생성하며, 반복 작업을 자동화하도록 돕습니다. built-in tool(Read, Write, Bash, Grep, Glob)와 MCP 서버를 사용합니다.

시나리오 5: 지속적 통합을 위한 Claude Code

자동 코드 리뷰, 테스트 생성, 풀 리퀘스트 피드백을 위해 Claude Code를 CI/CD pipeline에 통합합니다. prompt는 false positive를 최소화하도록 설계해야 합니다.

시나리오 6: structured data extraction

시스템이 비정형 문서에서 정보를 extraction하고, JSON schema로 출력을 validation하며, 높은 accuracy를 유지합니다. 엣지 케이스를 올바르게 처리해야 합니다.

시나리오 7: Conversational AI Architecture Patterns

context window 관리, 턴 간 지시사항 지속성, memory 전략, 안전한 실행을 위한 tool 설계, 모호하거나 충돌하는 사용자 입력 처리 등을 포함하는 멀티턴 대화 시스템을 설계합니다.

시나리오 8: Agentic AI tool *(내용 누락 — 채워 넣을 수 있게 도와주세요!)*

이 시나리오는 응시자 제보로 확인됐지만, 아직 이 가이드에는 정리되어 있지 않습니다. 실제 시험에서 관련 문제를 봤다면 GitHub Issues에 공유해 주세요. 내용을 보강하면 이 시험을 준비하는 다른 사람들에게도 도움이 됩니다.

공식 문서

resource	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 — Overview	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
Model Context Protocol (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 — Documentation	https://code.claude.com/docs/en/overview
Claude Code — CLAUDE.md and Memory	https://code.claude.com/docs/en/memory
Claude Code — Skills (incl. slash commands)	https://code.claude.com/docs/en/skills
Claude Code — Hooks	https://code.claude.com/docs/en/hooks
Claude Code — Sub-agents	https://code.claude.com/docs/en/sub-agents
Claude Code — MCP Integration	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 — Headless (non-interactive mode)	https://code.claude.com/docs/en/headless
Prompt Engineering Guide	https://platform.claude.com/docs/en/build-with-claude/prompt-engineering/overview
Extended Thinking	https://platform.claude.com/docs/en/build-with-claude/extended-thinking
Anthropic Cookbook (code examples)	https://github.com/anthropics/antropic-cookbook

---

PART I: 이론 기초

이 파트에서는 시험에 필요한 이론을 기술과 개념 중심으로 정리합니다. 시험 도메인 순서 그대로 나열하기보다, 실제로 이해해야 할 주제별로 묶어 두었습니다. 이렇게 보면 각 개념이 어디에 쓰이는지 더 쉽게 연결할 수 있습니다.

1장: Claude API — 모델 상호작용의 기초

> 문서: Messages API | Prompt Engineering

1.1 API 요청 구조

Claude API는 요청을 보내고 응답을 받는 방식으로 동작합니다. Messages API 요청의 기본 구성은 다음과 같습니다.

```
{
  "model": "claude-sonnet-4-6",
  "max_tokens": 1024,
  "system": "You are a helpful assistant.",
  "messages": [
    {"role": "user", "content": "Hi!"},
    {"role": "assistant", "content": "Hello!"},
    {"role": "user", "content": "How are you?"}
  ],
  "tools": [...],
  "tool_choice": {"type": "auto"}
}
```

핵심 필드:

- `model` — 모델 선택(`claude-opus-4-6`, `claude-sonnet-4-6`, `claude-haiku-4-5`)
- `max_tokens` — 응답의 최대 token 수

- `system` — system prompt(모델 동작 정의)
- `messages` — 대화 기록(일관성을 유지하려면 **전체 기록을 보내야 함**)
- `tools` — 사용 가능한 tool 정의
- `tool_choice` — tool 선택 전략

1.2 메시지 역할

`messages` 배열에는 두 가지 대화 역할과 한 가지 지시 역할이 있습니다.

- `user` — 사용자 메시지. tool 결과도 포함됩니다(별도의 `tool` 역할이 아니라 `user` 역할 메시지 안의 `tool_result` 콘텐츠 블록으로 전송됨)
- `assistant` — 모델 응답(기록을 보낼 때 포함), tool 사용 요청(`tool_use` 콘텐츠 블록)도 포함됨
- `system` — 최상위 `system` 필드로 설정하거나(첫 번째 turn부터 적용), `messages` 안에 `{"role": "system", ...}`로 인라인 삽입할 수 있습니다(그 시점부터 적용되며, 아래의 배치 규칙을 따릅니다)

tool 결과는 `"tool"` 역할 메시지로 전송되지 않습니다. `user` 역할 메시지로 전송되며, 그 content에 `tool_result` 콘텐츠 블록이 포함됩니다.

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

`system`은 최상위 `system` 파라미터뿐 아니라 `messages` 배열 안에 역할로 직접 나타날 수도 있습니다. 이는 최상위 `system` 필드의 캐시된 prefix를 무효화하지 않으면서 대화 중간에 지시를 추가하기 위한 것입니다. 여기에는 다음과 같은 배치 규칙이 있습니다.

- `user` turn(`tool_result` 블록이 있는 turn 포함) 또는 서버 tool 사용으로 끝나는 `assistant` turn 바로 뒤에 와야 합니다.
- `assistant` turn 앞에 오거나 배열의 마지막 항목이어야 합니다.
- `tool_use` 블록과 그에 대응하는 `tool_result` 사이에 위치할 수 없습니다 — 그렇게 하면 400 오류가 반환됩니다.
- 이후에 오는 `system` 메시지(대화 중간에 삽입된 것 포함)는 이전 메시지 및 이후 turn에 대한 최상위 `system` 필드보다 우선합니다.

중요: API를 호출할 때마다 **전체 대화 기록**을 함께 보내야 합니다. Claude는 요청 사이의 상태를 서버에 저장하지 않으므로, 각 호출은 독립적으로 처리됩니다.

1.3 응답의 `stop_reason` 필드

Claude API 응답에는 모델이 왜 응답 생성을 멈췄는지 알려 주는 `stop_reason`이 포함됩니다.

값	설명	조치
`"end_turn"`	모델이 응답을 완료함	결과를 사용자에게 표시
`"tool_use"`	모델이 tool 호출을 원함	tool을 실행하고 결과 반환
`"max_tokens"`	token 한도 도달	응답이 잘렸으므로 한도를 늘려야 할 수 있음
`"stop_sequence"`	중지 시퀀스를 만남	애플리케이션 로직에 따라 처리

Agentic 시스템에서는 특히 `"tool_use"`와 `"end_turn"`이 중요합니다. Agent 루프를 계속 돌릴지, 최종 응답으로 종료할지를 이 값으로 판단합니다.

1.4 system prompt

system prompt는 모델에게 역할, 제약, 동작 방식을 알려 주는 상위 지시사항입니다. 특징은 다음과 같습니다.

- `messages` 배열의 일부가 아니며 `system` 필드에 별도로 전달됨
- 사용자 메시지보다 우선순위가 높음
- 한 번 로드되어 대화 전체에 적용됨
- 역할, 제약, 출력 형식을 정의하는 데 사용됨

시험에서 중요한 점: system prompt의 표현이 특정 tool 사용을 과하게 유도할 수 있습니다. 예를 들어 "항상 고객을 확인하라"라고 쓰면, 실제로 필요하지 않은 상황에서도 모델이 `get_customer`를 호출할 가능성이 높아집니다.

1.5 context window

context window는 모델이 한 번에 볼 수 있는 전체 입력 크기입니다. token 기준으로 계산되며, 여기에 포함되는 항목은 다음과 같습니다.

- system prompt
- 전체 메시지 기록
- tool 정의
- tool 결과

주요 context window 문제:

1. **중간 손실 효과:** 긴 입력에서는 시작과 끝의 정보가 비교적 잘 반영되지만, 중간에 있는 세부사항은 놓칠 수 있습니다. 중요한 정보는 입력의 앞이나 뒤쪽에 모아 두는 편이 좋습니다.
2. **tool 결과 누적:** tool을 호출할 때마다 결과가 context에 추가됩니다. 예를 들어 tool이 40개 필드를 반환하지만 실제로 필요한 필드가 5개뿐이라면, 나머지는 context를 낭비하게 됩니다.
3. **점진적 summary:** 대화 기록을 압축하다 보면 숫자, 백분율, 날짜 같은 정확한 값이 빠지거나 "대략", "거의", "몇 개"처럼 흐려질 수 있습니다.

2장: tool과 `tool_use`

> 문서: Tool Use

2.1 `tool_use`란 무엇인가

`tool_use`는 Claude가 외부 함수를 호출하도록 요청하는 메커니즘입니다. 모델이 직접 코드를 실행하는 것은 아닙니다. Claude는 구조화된 tool 호출 요청을 만들고, 애플리케이션 코드가 그 요청을 실행한 뒤 결과를 다시 Claude에 전달합니다.

2.2 tool 정의

각 tool은 JSON schema를 사용해 정의됩니다.

```
{
  "name": "get_customer",
  "description": "Finds a customer by email or ID. Returns the customer profile,
including name, email, order history, and account status. Use this tool BEFORE
lookup_order to verify the customer's identity. Accepts an email (format:
user@domain.com) or a numeric customer_id.",
  "input_schema": {
    "type": "object",
    "properties": {
      "email": {"type": "string", "description": "Customer email"},
      "customer_id": {"type": "integer", "description": "Numeric customer ID"}
    },
    "required": []
  }
}
```

tool 설명에서 중요한 점:

1. **tool 선택은 대부분 설명에 좌우됩니다.** LLM은 tool 설명을 읽고 어떤 tool을 쓸지 고릅니다. "고객 정보를 검색함"처럼 짧은 설명만 있으면, 비슷한 tool이 있을 때 쉽게 혼동합니다.
2. **설명에 포함할 내용:**
   - tool이 수행하는 일과 반환하는 내용
   - 입력 형식과 예시 값
   - 엣지 케이스와 제약
   - 이 tool을 유사한 대안 대신 사용해야 하는 시점
3. tool 간 설명이 같거나 많이 겹치지 않게 하세요. `analyze_content`와 `analyze_document`의 설명이 거의 같으면 모델은 둘을 구분하기 어렵습니다.
4. **built-in tool vs MCP tool:** Agent는 비슷한 기능이라면 MCP tool보다 built-in tool(Read, Grep)을 먼저 고를 수 있습니다. MCP tool을 쓰게 하려면 built-in tool로는 얻을 수 없는 고유 데이터, 추가 context, 구체적인 장점

을 설명에 명시해야 합니다.

2.3 `tool_choice` 매개변수

`tool_choice`는 모델이 tool을 어떻게 사용할지 제어하는 옵션입니다.

값	동작	사용 시점
`{"type": "auto"}`	모델이 tool을 호출할지, 텍스트로 답할지 직접 결정	대부분의 경우 기본값
`{"type": "any"}`	모델이 **반드시** 어떤 tool이든 호출해야 함	structured output이 꼭 필요할 때
`{"type": "tool", "name": "extract_metadata"}`	모델이 **반드시** 지정된 tool을 호출해야 함	첫 단계나 실행 순서를 강제해야 할 때

중요 시나리오:

- `tool_choice: "any"` + 여러 extraction tool → 모델이 가장 적합한 tool을 선택하지만, 여전히 structured output을 얻을 수 있음
- 강제 선택 → 특정 첫 동작을 보장해야 할 때(예: 보강 전에 `extract_metadata` 실행)

2.4 structured output을 위한 JSON schema

Claude에서 structured output을 안정적으로 받으려면 JSON schema와 `tool_use`를 함께 쓰는 방식이 가장 신뢰할 만합니다. 이 방식은 다음을 보장하거나 제한합니다.

- JSON 문법 오류를 크게 줄임(누락된 중괄호, trailing comma 등)
- 필수 필드와 구조를 강제함
- 단, 값의 의미적 정확성까지 보장하지는 않음

schema 설계 — 핵심 원칙:

```
{
  "type": "object",
  "properties": {
    "category": {
      "type": "string",
      "enum": ["bug", "feature", "docs", "unclear", "other"]
    },
    "category_detail": {
      "type": ["string", "null"],
      "description": "Details if category = 'other' or 'unclear'"
    },
    "severity": {
      "type": "string",
      "enum": ["critical", "high", "medium", "low"]
    },
    "confidence": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "optional_field": {
      "type": ["string", "null"],
      "description": "Null if the information was not found in the source"
    }
  },
  "required": ["category", "severity"]
}
```

schema 설계 규칙:

1. **필수 vs 선택:** 정보가 항상 존재할 때만 필수 필드로 지정합니다. 원본에 없는 정보를 필수로 만들면 모델이 값을 억지로 채울 수 있습니다.
2. **Nullable 필드:** 없을 수 있는 정보에는 `"type": ["string", "null"]`을 사용합니다. 이렇게 하면 모델이 환각으로 값을 만들기보다 `null`을 반환할 수 있습니다.
3. **`"other"`가 있는 enum:** 미리 정의한 범주에 맞지 않는 데이터를 버리지 않도록 `"other"`와 세부 설명 필드를 함께 둡니다.
4. **Enum `"unclear"`:** 모델이 확실히 분류할 수 없는 경우를 위해 `"unclear"`를 둡니다. 틀린 범주를 자신 있게 고르는 것보다 정직하게 불확실성을 표시하는 편이 낫습니다.

2.5 구문 오류 vs 의미 오류

오류 유형	예	완화 방법
구문	유효하지 않은 JSON, 잘못된 필드 타입	JSON schema를 사용한 `tool_use`(제거 가능)
의미	합계가 맞지 않음, 값이 잘못된 필드에 있음, 환각	validation 검사, 피드백을 포함한 retry, 자기 교정

3장: Claude Agent SDK — Agentic 시스템 구축

> 문서: Agent SDK | Hooks | Subagents | Sessions

3.1 Agentic 루프란 무엇인가

Agentic 루프는 모델이 답변만 생성하는 대신, 필요한 tool을 호출하면서 작업을 진행하게 만드는 기본 패턴입니다.

```
1. tool과 함께 Claude에 요청 전송
2. 응답 수신
3. stop_reason 확인:
   - "tool_use" -> tool 실행, 결과를 기록에 추가, 1단계로 돌아감
   - "end_turn" -> 작업 완료, 결과를 사용자에게 표시
4. 완료될 때까지 반복
```

모델 주도 방식입니다: Claude는 현재 context와 이전 tool 결과를 보고 다음에 어떤 tool을 호출할지 결정합니다. 미리 정해 둔 순서대로만 움직이는 하드코딩된 의사결정 트리와는 다릅니다.

안티패턴(피해야 함):

- 완료 여부를 감지하기 위해 assistant 텍스트를 parsing함("Task completed")
- 임의의 반복 한도(예: `max_iterations=5`)를 기본 중지 조건으로 사용함
- assistant가 텍스트 콘텐츠를 생성했는지를 완료 신호로 확인함

올바른 접근: 유일하게 신뢰할 수 있는 완료 신호는 `stop_reason == "end_turn"`입니다.

3.2 `AgentDefinition` 구성

`AgentDefinition`은 Claude Agent SDK의 Agent 구성 객체입니다.

```python
agent = AgentDefinition(
    name="customer_support",
    description="Handles customer requests for returns and order issues",
    system_prompt="You are a customer support agent...",
    allowed_tools=["get_customer", "lookup_order", "process_refund",
"escalate_to_human"],
    # For a coordinator:
    # allowed_tools=["Task", "get_customer", ...]
)
```

핵심 매개변수:

- `name` / `description` — Agent의 식별자와 설명
- `system_prompt` — 지시사항을 담은 system prompt
- `allowed_tools` — 허용된 tool 목록(최소 권한 원칙)

3.3 허브 앤 스포크: Coordinator와 subagent

multi-agent architecture는 보통 허브 앤 스포크 구조로 설계합니다.

```
        Coordinator
       /     |      \
  Subagent1  Subagent2  Subagent3
   (search)   (analysis)   (synthesis)
```

Coordinator의 책임:

- 작업을 하위 작업으로 분해
- 필요한 subagent 결정(동적 선택)
- subagent에게 작업 위임
- 결과 집계와 validation
- 오류와 retry 처리
- 사용자에게 결과 전달

핵심 원칙: subagent의 context는 격리됩니다.

- subagent는 Coordinator의 대화 기록을 자동으로 상속하지 않음
- 필요한 모든 context는 subagent prompt에 **명시적으로 전달**해야 함
- subagent는 호출 간에 memory를 공유하지 않음
- 관찰 가능성과 오류 제어를 위해 모든 통신은 Coordinator를 통해 흐르게 함

3.4 subagent 생성을 위한 `Task` tool

subagent는 `Task` tool을 통해 생성됩니다.

```python
# The coordinator's allowedTools must include "Task"
coordinator_agent = AgentDefinition(
    allowed_tools=["Task", "get_customer"]
)
```

명시적 context 전달은 필수입니다:

```
# Bad: the subagent has no context
Task: "Analyze the document"

# Good: full context in the prompt
Task: "Analyze the following document.
Document: [full document text]
Prior search results: [web search results]
Output format requirements: [schema]"
```

병렬 생성: Coordinator는 한 번의 응답에서 여러 `Task`를 호출할 수 있습니다. 이 경우 subagent들은 병렬로 실행됩니다.

```
# One coordinator response contains:
Task 1: "Search for articles about X"
Task 2: "Analyze document Y"
Task 3: "Search for articles about Z"
# All three run concurrently
```

3.5 Agent SDK의 hook

hook은 Agent 수명 주기의 특정 지점에서 호출을 가로채거나 결과를 변환하는 데 사용됩니다.

PostToolUse는 tool 결과가 모델에 제공되기 전에 이를 가로챕니다.

```python
# Example: normalize date formats from different MCP tools
@hook("PostToolUse")
def normalize_dates(tool_result):
    # Convert Unix timestamp -> ISO 8601
    # Convert "Mar 5, 2025" -> "2025-03-05"
    return normalized_result
```

발신 호출 가로채기 hook은 정책을 위반하는 동작을 차단합니다.

```python
# Example: block refunds above $500
@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)
```

핵심 차이: hook vs prompt 지시사항

속성	hook	prompt 지시사항
보장	**결정적**(100%)	**확률적**(>90%, 100%는 아님)
사용 시점	중요한 비즈니스 규칙, 금융 작업, compliance	일반 선호사항, 권장사항, 형식 지정
예	500달러 초과 환불 차단	"escalation 전에 해결을 시도하라"

규칙: 실패가 금융, 법률, 안전상 결과를 초래한다면 prompt가 아니라 hook을 사용하세요.

4장: Model Context Protocol(MCP)

> 문서: MCP | Tools | Resources | Servers

4.1 MCP란 무엇인가

Model Context Protocol(MCP)은 Claude가 외부 시스템과 연결될 수 있도록 만든 개방형 프로토콜입니다. MCP에서 다루는 기본 resource 유형은 세 가지입니다.

1. **tool** — Agent가 동작을 수행하기 위해 호출할 수 있는 함수(CRUD 작업, API 호출, 명령 실행)
2. **resource** — Agent가 context를 위해 읽을 수 있는 데이터(문서, 데이터베이스 schema, 콘텐츠 카탈로그)
3. **prompt** — 일반 작업을 위한 사전 정의 prompt template

4.2 MCP 서버

MCP 서버는 MCP 프로토콜을 구현하고 tool/resource를 제공하는 프로세스입니다. 서버를 연결하면 Claude는 다음과 같이 동작할 수 있습니다.

- 모든 tool이 자동으로 발견됨
- 연결된 모든 서버의 tool을 한 번에 사용할 수 있음
- tool 설명이 모델의 사용 방식을 결정함

4.3 MCP 서버 구성

프로젝트 구성(`.mcp.json`) — 팀 사용 목적:

```
{
  "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}"
      }
    }
  }
}
```

핵심 사항:

- `.mcp.json`은 프로젝트 루트에 저장되고 버전 관리됨
- 환경 변수(`${GITHUB_TOKEN}`)를 비밀값에 사용하며 token 자체는 커밋하지 않음
- 모든 프로젝트 기여자가 사용할 수 있음

사용자 구성(`~/.claude.json`) — 개인/실험용 서버:

- 사용자의 홈 디렉터리에 저장됨
- 버전 관리를 통해 공유되지 않음
- 개인 실험과 테스트에 적합함

서버 선택:

- 표준 통합(Jira, GitHub, Slack)에는 기존 커뮤니티 MCP 서버를 선호
- 고유한 팀별 workflow에만 자체 서버 구축

4.4 MCP의 `isError` 플래그

MCP tool에서 오류가 발생하면 응답에 `isError: true`를 포함합니다. 이 값은 Agent에게 "이번 tool 호출은 실패했다"는 신호를 줍니다.

structured error(좋음):

```
{
  "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
  }
}
```

일반 오류(안티패턴):

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

일반 오류만 반환하면 Agent가 다음 행동을 판단하기 어렵습니다. retry해야 하는지, 쿼리를 바꿔야 하는지, 아니면 escalation해야 하는지 알 수 없기 때문입니다.

4.5 MCP resource

resource는 Agent가 어떤 동작을 실행하지 않고도 context를 얻기 위해 읽을 수 있는 데이터입니다.

- 콘텐츠 카탈로그(예: 모든 프로젝트 작업 목록, 계층형 내비게이션)
- 데이터베이스 schema(데이터 구조 이해)
- 문서(API 참조, 내부 가이드)
- 이슈/작업 summary

resource의 장점: Agent가 어떤 데이터가 있는지 알아내기 위해 불필요한 탐색 tool 호출을 반복하지 않아도 됩니다. resource는 바로 참고할 수 있는 "지도" 역할을 합니다.

5장: Claude Code — 구성과 workflow

> 문서: Claude Code | Memory / CLAUDE.md | Skills | MCP | Hooks | Sub-agents | GitHub Actions | Headless

5.1 CLAUDE.md 계층

CLAUDE.md는 Claude Code가 따라야 할 지시사항을 담는 파일입니다. 지시사항은 세 계층으로 나뉩니다.

```
1. 사용자 수준: ~/.claude/CLAUDE.md
   - 해당 사용자에게만 적용
   - VCS를 통해 공유하지 않음
   - 개인 선호사항과 작업 스타일

2. 프로젝트 수준: .claude/CLAUDE.md 또는 루트 CLAUDE.md
   - 모든 프로젝트 기여자에게 적용
   - VCS로 관리
   - 코딩 표준, 테스트 표준, architecture 결정

3. 디렉터리 수준: 하위 디렉터리의 CLAUDE.md
   - 해당 디렉터리의 파일을 작업할 때 적용
   - 코드베이스의 해당 부분에 특화된 규칙
```

자주 하는 실수: 새 팀원이 프로젝트 지시사항을 적용받지 못하는 경우가 있습니다. 보통 지시사항을 프로젝트의 `.claude/CLAUDE.md`가 아니라 개인 홈 디렉터리의 `~/.claude/CLAUDE.md`에 넣었기 때문입니다.

5.2 `@path` 문법(파일 가져오기)

CLAUDE.md에서는 `@path`로 다른 파일을 가져올 수 있습니다. 이 기능을 쓰면 하나의 큰 파일 대신 주제별 파일로 지시사항을 나눌 수 있습니다.

```
# Project CLAUDE.md

Coding standards are described in @./standards/coding-style.md
Test requirements are in @./standards/testing-requirements.md
Project overview is in @README.md and dependencies are in @package.json
```

`@path` 규칙:

- 파일 경로 바로 앞에 `@`를 사용합니다(공백 없음).
- 상대 경로와 절대 경로를 지원합니다.
- 상대 경로는 import를 포함하는 파일을 기준으로 해석됩니다.

- 최대 import 중첩 깊이는 5입니다.

이렇게 하면 중복을 줄이고, 각 패키지에는 필요한 표준만 포함시킬 수 있습니다.

5.3 `.claude/rules/` 디렉터리

`.claude/rules/`는 하나의 CLAUDE.md에 모든 규칙을 넣는 대신, 주제별 규칙 파일로 나누는 방식입니다.

```
.claude/rules/
  testing.md          -- testing conventions
  api-conventions.md  -- API conventions
  deployment.md       -- deployment rules
  react-patterns.md   -- React patterns
```

핵심 기능: 조건부 로딩을 위한 `paths`가 있는 YAML frontmatter

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

For API files, use async/await with explicit error handling.
Each endpoint must return a standard response wrapper.
```

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

Tests must use describe/it blocks.
Use data factories instead of hardcoding.
Do not mock the database—use a test database.
```

작동 방식:

- Claude Code가 `paths` 패턴과 일치하는 파일을 편집할 때만 해당 규칙이 **로드됨**
- 관련 없는 규칙은 로드하지 않으므로 context와 token을 절약할 수 있음
- Glob 패턴을 쓰면 파일 위치와 관계없이 유형별 규칙을 적용할 수 있음(예: 코드베이스 곳곳에 흩어진 테스트 파일)

`paths`가 있는 `.claude/rules/`와 디렉터리 수준 CLAUDE.md의 사용 기준:

- `paths`가 있는 `.claude/rules/` — 규칙이 여러 디렉터리에 흩어진 파일에 적용될 때(테스트, migration)
- 디렉터리 수준 CLAUDE.md — 규칙이 특정 디렉터리에 묶여 있고 다른 곳에는 필요하지 않을 때

5.4 사용자 지정 슬래시 명령과 Skill

> **참고:** 현재 Claude Code 버전에서는 사용자 지정 명령(`.claude/commands/`)이 Skill(`.claude/skills/`)과 통합되어 있습니다. 두 형식 모두 `/name` 명령을 만듭니다. 시험 가이드

> 는 `.claude/commands/`를 참조하며, 이 형식은 여전히 지원됩니다.

슬래시 명령은 `/name` 형태로 호출하는 재사용 가능한 prompt template입니다.

`.claude/commands/` 형식(레거시, 지원됨):

```
.claude/commands/
  review.md        -- /review -- standard code review
  test-gen.md      -- /test-gen -- test generation
```

`.claude/skills/` 형식(현재):

```
.claude/skills/
  review/SKILL.md  -- /review -- with frontmatter configuration
  test-gen/SKILL.md
```

프로젝트 명령(`.claude/commands/` 또는 `.claude/skills/`):

- VCS에 저장되며 저장소를 클론한 모든 사람이 사용할 수 있음
- 팀 전체의 일관된 workflow를 보장함

사용자 명령(`~/.claude/commands/` 또는 `~/.claude/skills/`):

- VCS로 공유되지 않는 개인 명령
- 개인 workflow용

5.5 Skill — `.claude/skills/`

Skill은 `SKILL.md`의 frontmatter로 동작을 설정하는 고급 명령 형식입니다.

```
---
context: fork
allowed-tools: ["Read", "Grep", "Glob"]
argument-hint: "Path to the directory to analyze"
---

Analyze the code structure in the specified directory.
Output a report on dependencies and architectural patterns.
```

Frontmatter 매개변수:

매개변수	설명
`context: fork`	격리된 subagent에서 Skill을 실행합니다. 상세 출력이 메인 session context를 불필요하게 채우지 않습니다
`allowed-tools`	사용할 수 있는 tool을 제한합니다(보안. 예: 허용되지 않으면 Skill이 파일을 삭제할 수 없음)

`argument-hint`	매개변수 없이 호출할 때 인수를 요청하는 힌트

Skill과 CLAUDE.md의 사용 기준:

- **Skill** — 특정 작업(리뷰, 분석, 생성)을 위한 on-demand 호출
- **CLAUDE.md** — 항상 로드되는 일반 표준과 규칙

개인 Skill(`~/.claude/skills/`):

- 팀원에게 영향을 주지 않도록 다른 이름으로 개인 변형을 만듭니다.

5.6 계획 모드 vs 직접 실행

계획 모드:

- 모델은 조사와 계획만 수행하고 파일은 변경하지 않음
- Read, Grep, Glob으로 코드베이스를 살펴봄
- 사용자가 승인할 수 있는 구현 계획을 작성함
- 부작용 없이 안전하게 탐색할 수 있음

계획 모드를 사용할 때:

- 큰 변경(수십 개 파일)
- 여러 접근 방식이 가능한 경우(마이크로서비스: 경계를 어떻게 정의할 것인가?)
- architecture 결정(어떤 프레임워크? 어떤 구조?)
- 낯선 코드베이스(변경 전에 이해해야 함)
- 45개 이상의 파일에 영향을 주는 라이브러리 migration

직접 실행을 사용할 때:

- 명확한 스택 트레이스가 있는 단일 파일 수정
- validation 체크 하나 추가
- 잘 이해되고 모호하지 않은 변경

결합 접근 방식:

1. 조사와 설계를 위해 계획 모드 사용
2. 사용자가 계획 승인
3. 승인된 계획을 구현하기 위해 직접 실행

탐색 subagent — 코드베이스 탐색에 특화된 subagent:

- 상세한 탐색 출력을 메인 context에서 분리
- 메인 Agent에는 summary만 반환
- 다단계 작업에서 context window가 빨리 소진되는 문제를 줄임

5.7 `/compact` 명령

`/compact`는 context를 압축하는 내장 명령입니다.

- 이전 기록을 summary해서 context window를 확보함
- 긴 조사 session에서 상세 tool 출력이 context를 많이 차지할 때 사용
- 주의: summary 과정에서 정확한 숫자, 날짜, 세부사항이 손실될 수 있음

5.8 `/memory` 명령

`/memory`는 session 간에 유지할 memory를 관리하는 내장 명령입니다.

- `CLAUDE.md` 파일을 열어 노트, 선호사항, context를 저장할 수 있음
- 저장된 정보는 다음 session에서도 유지되고 시작 시 자동으로 로드됨
- 프로젝트 규칙, 사용자 선호사항, 자주 쓰는 명령, 현재 작업 context를 기록할 때 유용함
- 매 session마다 같은 지시사항을 반복 설명하지 않아도 됨

5.9 CI/CD를 위한 Claude Code CLI

`-p` (또는 `--print`) 플래그:

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

- 비대화형 모드: prompt를 처리하고 stdout에 출력한 뒤 종료
- 사용자 입력을 기다리지 않음
- CI/CD pipeline에서 Claude를 실행하는 유일하게 올바른 방법

CI용 structured output:

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

- `--output-format json` — JSON으로 출력
- `--json-schema` — schema에 대해 출력 validation
- 결과를 parsing해 PR 인라인 댓글을 자동 게시할 수 있음

session context 격리: 코드를 생성한 것과 같은 Claude session에서 다시 리뷰까지 맡기면 리뷰 품질이 떨어질 수 있습니다. 모델이 자신의 이전 추론 context를 그대로 갖고 있어, 스스로 내린 결정을 덜 비판적으로 볼 수 있기 때문입니다. 리뷰에는 별도의 독립 instance를 사용하는 편이 좋습니다.

중복 댓글 방지: 새 커밋 이후 재리뷰할 때는 이전 리뷰 결과를 context에 포함하고, Claude에게 새 이슈나 미해결 이슈만 보고하도록 지시합니다.

5.10 `fork_session`과 session 관리

`--resume <session-name>`은 이름이 지정된 session을 재개합니다.

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

- 저장된 context로 이전 대화를 계속함
- 여러 session에 걸친 긴 조사에 유용함
- 위험: 이전 session 이후 파일이 변경되었다면 tool 결과가 오래되었을 수 있음

`fork_session`은 공유 context에서 독립 브랜치를 만듭니다.

```
Codebase investigation
         |
    fork_session
    /           \
Approach A:      Approach B:
Redux            Context API
```

- 두 포크는 분기점까지의 context를 상속함
- 이후에는 독립적으로 갈라짐
- 접근 방식 비교나 전략 테스트에 유용함

재개 대신 새 session을 시작할 때:

- tool 결과가 오래됨(파일 변경)
- 시간이 많이 지나 context가 저하됨
- 오래된 tool 데이터로 재개하기보다 "우리가 발견한 내용의 짧은 summary는 다음과 같습니다: ..."로 다시 시작하는 편이 나음

6장: Prompt Engineering — 고급 기법

> 문서: Prompt Engineering | Anthropic Cookbook

6.1 Few-shot prompting

Few-shot prompting은 prompt에 입력/출력 예시를 몇 개 넣어, 모델이 기대 동작을 따라 하도록 유도하는 방식입니다. 보통 2-4개 정도의 예시를 사용합니다.

Few-shot이 텍스트 설명보다 효과적인 이유:

- "더 정확하게 하라" 같은 모호한 지시는 해석이 갈릴 수 있음
- 예시는 원하는 출력 형식과 판단 기준을 구체적으로 보여줌
- 모델은 예시를 그대로 복사하는 것이 아니라, 그 패턴을 새 사례에 적용함

Few-shot 예시 유형과 사용 시점:

1. **모호한 시나리오를 위한 예시:**

```
Request: "My order is broken"
Action: Call get_customer -> lookup_order -> check status.
Rationale: “broken” may mean a damaged item; you need order details.

Request: "Get me a manager"
Action: Immediately call escalate_to_human.
Rationale: The customer explicitly requests a human. Do not attempt to solve
autonomously.
```

2. **출력 형식을 위한 예시:**

```
Finding example:
{
  "location": "src/auth/login.ts:42",
  "issue": "SQL injection in the username parameter",
  "severity": "critical",
  "suggested_fix": "Use a parameterized query"
}
```

3. **허용 가능한 코드와 문제 코드를 구분하는 예시:**

```
// Acceptable (do not flag):
const items = data.filter(x => x.active);

// Problem (flag):
const items = data.filter(x => x.active == true); // Use strict equality ===
```

4. **서로 다른 문서 형식에서 extraction하는 예시:**

```
Document with inline citations:
"As shown in the study (Smith, 2023), the rate is 42%."
-> {"value": "42%", "source": "Smith, 2023", "type": "inline_citation"}

Document with bibliography references:
"The rate is 42%. [1]"
-> {"value": "42%", "source": "reference_1", "type": "bibliography"}
```

5. **비공식 측정값을 위한 예시:**

```
Text: "about two handfuls of rice"
-> {"amount": "~100g", "original_text": "two handfuls", "precision": "approximate"}

Text: "a pinch of salt"
-> {"amount": "~1g", "original_text": "a pinch", "precision": "approximate"}
```

비공식적이거나 표준화되지 않은 측정 단위를 extraction해야 할 때는 Few-shot이 특히 효과적입니다. 이런 경우에는 규칙만 나열하는 것보다 예시로 판단 방식을 보여주는 편이 더 안정적입니다.

prompt의 형식 normalization 규칙: structured output에 엄격한 JSON schema를 사용할 때는 prompt에 normalization 규칙을 추가하세요.

```
Normalization:
- Dates: always ISO 8601 (YYYY-MM-DD); "yesterday" -> compute an absolute date
- Currency: numeric amount + currency code; "five bucks" -> {"amount": 5, "currency":
"USD"}
- Percentages: decimal fraction; "half" -> 0.5
```

이 규칙은 JSON 문법은 맞지만 값 형식이 제각각인 의미 오류를 줄여 줍니다.

6.2 명시적 기준 vs 모호한 지시사항

나쁨(모호함):

```
Check code comments for accuracy.
Be conservative—report only high-confidence findings.
```

좋음(명시적 기준):

```
Flag a comment as problematic ONLY if:
1. The comment describes behavior that CONTRADICTS the actual code behavior
2. The comment references a non-existent function or variable
3. A TODO/FIXME comment refers to a bug that has already been fixed in code

Do NOT flag:
- Comments that are merely stylistically outdated
- Comments with minor wording inaccuracies
- Missing comments (that is a separate category)
```

예시와 함께 severity 기준 정의:

```
CRITICAL: Runtime failure for users
  Example: NullPointerException while processing a payment

HIGH: Security vulnerability
  Example: SQL injection, XSS, missing authorization checks

MEDIUM: Logic bug without immediate impact
  Example: Wrong sorting, off-by-one error

LOW: Code quality
  Example: Duplication, suboptimal algorithm for small data
```

6.3 prompt chaining

prompt chaining은 복잡한 작업을 작고 집중된 단계로 나누어 순서대로 처리하는 방식입니다.

```
Step 1: Analyze auth.ts (local issues only)
       -> Output: list of issues in auth.ts

Step 2: Analyze database.ts (local issues only)
       -> Output: list of issues in database.ts

Step 3: Integration pass (cross-file dependencies)
       -> Output: issues at module boundaries
```

중요한 이유:

- **주의 분산**을 줄일 수 있음. 파일을 너무 많이 한 번에 주면 모델이 일부 파일은 깊게 보고, 다른 파일은 얕게 볼 수 있음
- 파일별 분석 품질을 더 일정하게 유지할 수 있음
- 파일 간 상호작용은 별도 단계에서 따로 검토할 수 있음

prompt chaining과 동적 분해의 사용 기준:

- **prompt chaining** — 예측 가능하고 반복 가능한 작업(코드 리뷰, 파일 migration)
- **동적 분해** — 실행 중에야 하위 작업이 명확해지는 개방형 조사

6.4 "인터뷰" 패턴

구현에 들어가기 전에 Claude가 먼저 clarification 질문을 던지는 패턴입니다.

```
Claude: "Before implementing caching for the API, a few questions:
1. Which cache invalidation strategy do you prefer—TTL or event-based?
2. Is stale data acceptable when the cache is unavailable?
3. Should caching be per-user or global?
4. What is the expected data volume to cache?"
```

유용한 경우:

- 핀테크, 헬스케어, 법률 시스템처럼 도메인 지식이 중요한 작업
- cache 전략이나 실패 모드처럼 요구사항의 함의가 명확하지 않은 작업
- 가능한 접근이 여러 개이고, 최선의 선택이 context에 따라 달라지는 작업

6.5 validation과 피드백 기반 retry

extraction한 데이터가 validation에 실패하면 다음 흐름으로 다시 시도합니다.

```
Step 1: Extract data from the document
Step 2: Validate (Pydantic, JSON Schema, business rules)
Step 3: If there's an error—retry with context:
  - The original document
  - The previous (incorrect) extraction
  - The specific error: "Field 'total' = 150, but sum(line_items) = 145. Re-check
values."
```

retry가 효과적인 경우:

- 날짜 형식처럼 고칠 수 있는 형식 오류
- 필드가 잘못된 위치에 들어간 구조 오류
- 합계가 맞지 않는 산술 불일치처럼 모델이 다시 확인할 수 있는 오류

retry가 도움이 되지 않는 경우:

- 정보가 원본 문서에 없음
- 필요한 context가 외부에 있음(제공되지 않은 다른 문서에 데이터가 있음)

validation tool로서의 Pydantic: Pydantic은 schema 기반 데이터 validation을 위한 Python 라이브러리입니다. 시험에서 핵심 사항은 다음과 같습니다.

- **구조 validation:** Claude로부터 JSON을 받은 뒤 코드에서 타입, 필수 여부, enum 제약을 확인
- **의미 validation:** 사용자 지정 validator로 비즈니스 로직 강제(항목 합계가 총액과 같음, start_date < end_date)
- **validation-retry 루프:** Pydantic validation 실패 시 오류 메시지를 구성하고 오류 context와 함께 Claude에 다시 prompt
- **JSON Schema 생성:** Pydantic 모델은 `tool_use`용 JSON Schema를 생성할 수 있어 단일 진실 공급원을 제공

6.6 자기 교정

내부 모순을 감지하는 패턴:

```
{
  "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}
  ]
}
```

모델은 문서에 적힌 값과 직접 계산한 값을 모두 extraction합니다. 두 값이 다르면 `conflict_detected`로 불일치를 표시하고 후속 처리에서 다룰 수 있습니다.

7장: Message Batches API

> 문서: Message Batches

7.1 개요

Message Batches API를 사용하면 여러 요청을 batch로 제출해 비동기로 처리할 수 있습니다.

속성	값
비용 절감	동기 호출 대비 **50%**
처리 window	최대 **24시간**(지연 시간 SLA 보장 없음)
멀티턴 tool 호출	**지원하지 않음**(요청 1개 = 응답 1개)
상관관계	요청과 응답을 연결하는 `custom_id` 필드

7.2 Batch API와 동기 API 사용 기준

작업	API	이유
병합 전 PR 검사	**동기**	개발자가 기다리고 있으며 24시간은 허용 불가
야간 기술 부채 보고서	**Batch**	아침까지 결과가 필요하며 50% 비용 절감
주간 보안 감사	**Batch**	긴급하지 않으며 50% 비용 절감
대화형 코드 리뷰	**동기**	즉각 응답 필요
문서 10,000개 처리	**Batch**	대량 처리이며 절감 효과가 큼

7.3 `custom_id` 사용

```
{
  "custom_id": "doc-invoice-2024-001",
  "params": {
    "model": "claude-sonnet-4-6",
    "max_tokens": 1024,
    "messages": [{"role": "user", "content": "Extract data from: ..."}]
  }
}
```

`custom_id`를 사용하면 다음이 가능합니다.

- 결과를 원본 문서와 연결
- 실패 시 실패한 문서만 다시 제출
- 성공한 문서의 재처리 방지

7.4 batch 실패 처리

1. 문서 100개의 batch를 제출
2. 95개 성공, 5개 실패(context 한도 초과)
3. `custom_id`로 실패 식별
4. 전략 수정(예: 긴 문서를 청크로 분할)

5. 실패한 5개 문서만 다시 제출

7.5 SLA 계획

30시간 안에 결과가 필요하고 Batch API 처리에 최대 24시간이 걸릴 수 있다면:

- 제출 window는 30 - 24 = **6시간**
- 즉, 마감 최소 24시간 전에는 batch를 제출해야 함
- 자주 제출한다면 4시간 window 단위로 나누는 방식이 안전함

8장: 작업 분해 전략

8.1 고정 pipeline(prompt chaining)

단계가 미리 정해져 있는 방식입니다.

```
Document -> Metadata extraction -> Data extraction -> Validation -> Enrichment -> Final output
```

사용 시점:

- 작업 구조가 예측 가능함(리뷰가 항상 같은 template을 따름)
- 모든 단계가 사전에 알려져 있음
- 안정성과 재현성이 필요함

8.2 동적 적응형 분해

실행 중에 나온 중간 결과를 보고 다음 하위 작업을 정합니다.

```
1. "Add tests for a legacy codebase"
2. -> First: map the structure (Glob, Grep)
3. -> Found: 3 modules with no tests, 2 with partial coverage
4. -> Prioritize: start with the payments module (high risk)
5. -> During work: discovered a dependency on an external API
6. -> Adapt: add a mock for the external API before writing tests
```

사용 시점:

- 개방형 조사 작업
- 전체 범위를 사전에 알 수 없음
- 각 단계가 이전 단계의 결과에 의존함

8.3 다중 패스 코드 리뷰

10개 이상의 파일이 있는 풀 리퀘스트의 경우:

```
Pass 1 (per-file): Analyze auth.ts -> list local issues
Pass 1 (per-file): Analyze database.ts -> list local issues
Pass 1 (per-file): Analyze routes.ts -> list local issues
...
Pass 2 (integration): Analyze relationships between files
  -> Cross-file issues: inconsistent types, circular dependencies
```

14개 파일을 한 번에 단일 패스로 보는 것이 좋지 않은 이유:

- 주의가 분산되어 일부 파일은 깊게 보고 다른 파일은 얕게 볼 수 있음
- 같은 패턴을 어떤 파일에서는 문제로 보고, 다른 파일에서는 놓치는 식의 불일치가 생김
- 인지 부하가 커져 명백한 버그도 건너뛸 수 있음

9장: escalation과 Human-in-the-Loop

9.1 사람에게 escalation해야 할 때

escalation 트리거(명확한 규칙):

상황	조치
고객이 명시적으로 "관리자를 연결해 달라"고 요청	즉시 escalation, 해결 시도하지 않음
정책이 요청을 다루지 않음	escalation(예: 정책이 경쟁사 가격 매칭을 언급하지 않는 경우)
Agent가 진행하지 못함	합리적인 시도 횟수 후 escalation
임계값을 초과하는 금융 작업	escalation(prompt가 아니라 hook으로 강제하는 것이 바람직)
고객 검색 시 여러 일치 항목	추가 식별자를 요청하고 추측하지 않음

신뢰할 수 없는 트리거:

신뢰할 수 없는 방법	실패 이유
감정 분석	고객의 기분은 케이스 복잡도와 상관관계가 없음
모델의 자체 confidence 평가(1-10)	모델은 확신 있게 틀릴 수 있으며 calibration이 좋지 않음
자동 classifier	과도한 엔지니어링이며 보유하지 않은 학습 데이터가 필요할 수 있음

9.2 escalation 패턴

즉시 escalation:

```
Customer: "I want to speak to a manager"
Agent: [immediately calls escalate_to_human]
NOT: "I can help with your issue, let me..."
```

해결 시도 후 escalation:

```
Customer: "My refrigerator broke two days after purchase"
Agent: [checks the order, offers a warranty replacement]
If the customer is not satisfied -> escalate
```

미묘한 escalation(인정 -> 해결 -> 반복 요청 시 escalation):

```
Customer: "This is outrageous, I'm very unhappy with the quality!"
Agent: [acknowledges frustration] "I understand your frustration."
       [offers resolution] "I can offer a replacement or a refund."
Customer: "No, I want to talk to someone!"
Agent: [customer insists again -> immediate escalation]
```

핵심 원칙은 먼저 고객의 감정을 인정하고, 그다음 구체적인 해결책을 제안하는 것입니다. 고객이 사람과 이야기하고 싶다는 의사를 반복해서 밝힐 때 escalation합니다. 불만을 한 번 표현했다는 이유만으로 바로 escalation하지는 않습니다. 그것은 관리자 연결 요청과 다릅니다.

정책 공백에 대한 escalation:

```
Customer: "Competitor X has this item 30% cheaper—give me a discount"
Policy: covers price adjustments only on your own site
Agent: [escalates — policy does not cover competitor price matching]
```

9.3 구조화된 handoff 프로토콜

escalation 시 Agent는 사람에게 구조화된 summary를 전달해야 합니다.

```
{
  "customer_id": "CUST-12345",
  "customer_name": "Ivan Petrov",
  "issue_summary": "Refund request for a damaged item",
  "order_id": "ORD-67890",
  "root_cause": "Item arrived damaged; photos attached",
  "actions_taken": [
    "Verified customer via get_customer",
    "Confirmed order via lookup_order",
    "Offered a standard replacement — customer insists on a refund"
  ],
  "refund_amount": "$89.99",
  "recommended_action": "Approve a full refund",
  "escalation_reason": "Customer requested to speak with a manager"
}
```

사람 운영자는 전체 대화를 보지 못하고 이 summary만 볼 수 있습니다. 따라서 summary만 읽어도 상황을 이해하고 바로 이어받을 수 있어야 합니다.

9.4 confidence calibration과 사람의 감독

데이터 extraction 시스템의 경우:

1. **필드별 confidence 점수:** 모델이 extraction 필드별 confidence 점수를 출력
2. **calibration:** 라벨링된 validation 세트를 사용해 임계값 조정
3. **routing:**
   - 높은 confidence + 안정적인 accuracy -> 자동 처리
   - 낮은 confidence 또는 모호한 source -> 사람 리뷰

계층화 무작위 sampling:

- confidence 높은 extraction이라도 정기적으로 샘플 감사
- 전체 97% accuracy는 특정 문서 유형에서 40% 오류를 숨길 수 있음
- 전체뿐 아니라 문서 유형별, 필드별 accuracy를 분석

10장: multi-agent 시스템의 오류 처리

10.1 오류 범주

범주	예시	retry 가능	Agent 조치
일시적	타임아웃, 503, 네트워크 실패	예	지수 백오프로 retry
validation	잘못된 입력 형식, 필수 필드 누락	아니요(입력 수정)	요청을 수정하고 retry
비즈니스	정책 위반, 임계값 초과	아니요	사용자에게 설명하고 대안 제안
권한	접근 거부	아니요	escalation

10.2 오류 처리 안티패턴

안티패턴	문제	올바른 접근
일반 상태 "검색 불가"	Coordinator가 복구 방법을 결정할 수 없음	오류 유형, 쿼리, 부분 결과, 대안 반환
조용한 억제(빈 결과 = 성공)	Coordinator는 일치 항목이 없다고 생각하지만 실제로는 실패임	"결과 없음"과 "검색 실패"를 구분
한 실패로 전체 workflow 중단	모든 부분 결과를 잃음	부분 결과로 계속 진행하고 공백을 주석 처리

subagent 내부 무한 retry	지연과 resource 낭비	로컬 복구(1-2회 retry) 후 Coordinator로 전파

10.3 구조화된 subagent 오류

```
{
  "status": "partial_failure",
  "failure_type": "timeout",
  "attempted_query": "AI impact on music industry 2024",
  "partial_results": [
    {"title": "AI Music Generation Report", "url": "...", "relevance": 0.8}
  ],
  "alternative_approaches": [
    "Try a narrower query: 'AI music composition tools'",
    "Use an alternative data source"
  ],
  "coverage_impact": "Not covered: AI impact on music production"
}
```

이 구조는 Coordinator가 다음 결정을 내리는 데 필요한 정보를 줍니다.

- 수정된 쿼리로 retry할 것인가?
- 부분 결과를 사용할 것인가?
- 다른 subagent에게 위임할 것인가?
- 이 섹션 없이 계속하고 공백을 주석 처리할 것인가?

10.4 최종 종합의 커버리지 주석

```
## Report: AI Impact on Creative Industries

### Visual Art (FULL COVERAGE)
[research results]

### Music (PARTIAL COVERAGE — search agent timeout)
[partial results]
⚠️ Note: coverage for this section is limited due to a timeout in the search agent.

### Literature (FULL COVERAGE)
[research results]
```

11장: 프로덕션 시스템의 context 관리

11.1 사실을 별도 블록으로 extraction

대화 기록은 summary 과정에서 손상될 수 있으므로, 중요한 사실은 별도의 structured 블록으로 extraction해 둡니다.

```
=== CASE FACTS (updated whenever a new fact appears) ===
Customer ID: CUST-12345
Order ID: ORD-67890
Order Date: 2025-01-15
Order Amount: $89.99
Issue: Damaged item on delivery
Customer Request: Full refund
Status: Pending manager approval
===
```

기록이 어떻게 summary되든 이 블록을 모든 prompt에 포함하세요.

11.2 tool 결과 다듬기

`lookup_order`가 40개 이상의 필드를 반환하지만 현재 작업에 필요한 것은 5개뿐이라면:

```
# PostToolUse hook: keep only relevant fields
@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"]
    }
```

이렇게 하면 context 사용량을 줄이고, 모델이 볼 필요 없는 노이즈도 줄일 수 있습니다.

11.3 위치 인식 입력

중간 손실 효과를 줄이려면 중요한 정보를 눈에 잘 띄는 위치에 모아 둡니다.

```
[KEY FINDINGS — at the top]
Found 3 critical vulnerabilities...

[DETAILED RESULTS — middle]
=== File auth.ts ===
...
=== File database.ts ===
...

[ACTION ITEMS — at the end]
Priority: fix auth.ts vulnerabilities before merge.
```

11.4 스크래치패드 파일

긴 조사에서 Agent는 핵심 발견 사항을 스크래치패드 파일에 쓸 수 있습니다.

```
# investigation-scratchpad.md
## Key findings
- PaymentProcessor in src/payments/processor.ts inherits from BaseProcessor
- refund() is called from 3 places: OrderController, AdminPanel, CronJob
- External PaymentGateway API has a rate limit of 100 req/min
- Migration #47 added refund_reason (NOT NULL) — 2024-12-01
```

context가 약해졌거나 새 session을 시작한 경우에도, Agent는 탐색을 처음부터 다시 하지 않고 스크래치패드를 참고할 수 있습니다.

11.5 context 보호를 위한 subagent 위임

```
Main agent: "Investigate dependencies of the payments module"
  -> Subagent (Explore): reads 15 files, traces imports
  -> Returns: "Payments depends on AuthService, OrderModel, and the external
PaymentGateway API"

Main agent: keeps one line in context instead of 15 files
```

분리된 context 계층: multi-agent 시스템에서는 각 subagent가 제한된 context 예산 안에서 동작합니다. 따라서 자신의 작업에 필요한 정보만 받아야 합니다. Coordinator는 별도의 context 계층처럼 동작하면서 subagent 출력 집계, 전역 상태 저장, context 배분을 담당합니다. 이렇게 설계하면 한 Agent가 다른 Agent와 무관한 정보로 context window를 낭비하는 "context 누수"를 줄일 수 있습니다.

subagent를 위한 제한된 context 예산:

- 최소 context 전송: 구체적 작업 + 필요한 데이터
- subagent가 원시 데이터 덤프가 아니라 structured result를 반환하도록 지시
- `allowedTools`로 subagent의 tool 집합 제한. tool이 적을수록 산만함과 context 비용이 줄어듦

11.6 구조화된 상태 지속성(크래시 복구용)

각 Agent는 자신의 상태를 정해진 위치에 기록합니다.

```
// 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"]
}
```

Coordinator는 재개 시 manifest를 로드합니다.

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

12장: source 보존

12.1 attribution 손실 문제

여러 source의 결과를 summary할 때 "주장 -> source" 연결이 손실될 수 있습니다.

```
Bad: "The AI music market is estimated at $3.2B." (No source, no year.)

Good:
{
  "claim": "The AI music market is estimated at $3.2B.",
  "source_url": "https://example.com/report",
  "source_name": "Global AI Music Report 2024",
  "publication_date": "2024-06-15",
  "confidence": 0.9
}
```

12.2 충돌하는 데이터 처리

두 source가 서로 다른 값을 제공할 때:

```
{
  "claim": "Share of AI-generated music on streaming platforms",
  "values": [
    {
      "value": "12%",
      "source": "Spotify Annual Report 2024",
      "date": "2024-03",
      "methodology": "Automated classification"
    },
    {
      "value": "8%",
      "source": "Music Industry Association Survey",
      "date": "2024-07",
      "methodology": "Survey of 500 labels"
    }
  ],
  "conflict_detected": true,
  "possible_explanation": "Difference in methodology and time period"
}
```

임의로 하나의 값만 고르지 마세요. 두 값을 모두 source와 함께 보존한 뒤, Coordinator가 조정하도록 해야 합니다.

12.3 올바른 해석을 위해 날짜 포함

날짜가 없으면 시간적 차이가 모순으로 잘못 해석될 수 있습니다.

```
Bad: "Source A says 10%, source B says 15%. Contradiction."
Good: "Source A (2023) says 10%, source B (2024) says 15%. Likely +5% growth over a year."
```

12.4 콘텐츠 유형별 렌더링

모든 것을 하나의 형식에 억지로 넣지 마세요.

- 금융 데이터 -> 표
- 뉴스와 분석 -> 산문
- 기술 발견 사항 -> structured 목록
- 시계열 -> 시간순 정렬

13장: Claude Code built-in tool

13.1 tool 선택 참조

작업	tool	예
이름/패턴으로 파일 찾기	**Glob**	`**/*.test.tsx`, `src/components/**/*.ts`
파일 안에서 검색	**Grep**	함수 이름, 오류 메시지, import
파일 전체 읽기	**Read**	분석을 위해 파일 로드
새 파일 쓰기	**Write**	처음부터 파일 생성
기존 파일을 정확히 편집	**Edit**	고유 텍스트 일치로 특정 스니펫 교체
셸 명령 실행	**Bash**	git, npm, 테스트 실행, 빌드

13.2 점진적 조사 전략

모든 파일을 한 번에 읽지 마세요. 이해를 점진적으로 구축합니다.

```
1. Grep: find entry points (function definition, export)
2. Read: read the found files
3. Grep: find usages (import, calls)
4. Read: read consumer files
5. Repeat until you have a complete picture
```

13.3 대안: Edit 대신 Read + Write

Edit이 고유하지 않은 텍스트 일치 때문에 실패할 때:

1. Read — 전체 파일 내용 로드
2. 내용을 프로그래밍 방식으로 수정
3. Write — 업데이트된 버전 쓰기

PART II: 시험 도메인 노트

도메인 1: Agent architecture와 orchestration(27%)

1.1 자율 작업 실행을 위한 Agentic 루프 설계

핵심 지식:

- Agent 루프 수명 주기: Claude 요청 전송, `stop_reason` 확인(`"tool_use"` vs `"end_turn"`), tool 실행, 다음 반복을 위해 결과 반환
- tool 결과는 대화 기록에 추가되어 모델이 다음 동작을 결정할 수 있음
- 모델 주도 의사결정(Claude가 다음 tool 선택) vs 하드코딩된 의사결정 트리

핵심 기술:

- 흐름 제어: `stop_reason = "tool_use"`이면 루프를 계속하고 `"end_turn"`이면 중지
- 반복 사이에 tool 결과를 context에 추가
- 피해야 할 안티패턴: 완료 판단을 위해 assistant 텍스트 parsing, 임의 반복 한도를 기본 중지 메커니즘으로 사용

1.2 multi-agent 시스템 orchestration(Coordinator-subagent)

핵심 지식:

- 허브 앤 스포크 architecture: Coordinator가 모든 Agent 간 통신, 오류 처리, routing을 소유
- subagent는 격리된 context로 동작하며 Coordinator의 기록을 자동 상속하지 않음
- Coordinator 책임: 작업 분해, 위임, 결과 집계, subagent 동적 선택
- Coordinator가 지나치게 좁게 분해할 위험

핵심 기술:

- 중복을 최소화하도록 research 범위를 subagent 간에 분할
- 반복적 개선 루프 구현(Coordinator가 종합 결과를 평가하고 작업을 재routing)
- 관찰 가능성을 위해 모든 통신을 Coordinator를 통해 routing

1.3 subagent 호출, context 전달, 생성 구성

핵심 지식:

- `Task` tool이 subagent를 생성하며, Coordinator의 `allowedTools`에는 `"Task"`가 포함되어야 함
- subagent context는 prompt에 명시적으로 포함되어야 하며, subagent는 부모 context를 상속하지 않음
- `AgentDefinition` 구성: 설명, system prompt, tool 제약
- 대안 탐색을 위한 `fork_session` 기반 session 관리

핵심 기술:

- 이전 Agent의 전체 출력을 subagent prompt에 포함

- context 전달 시 데이터를 metadata와 분리하기 위해 structured 형식 사용
- 단일 Coordinator 턴에서 여러 `Task` 호출로 병렬 subagent 생성
- Coordinator prompt를 단계별 지시가 아니라 목표와 품질 기준 중심으로 작성

1.4 강제와 handoff 패턴을 갖춘 다단계 workflow 구현

핵심 지식:

- workflow 순서를 위한 **프로그래밍적 강제**(hook, 사전조건)와 **prompt 안내**의 차이
- 결정적 보장이 필요할 때(예: 금융 작업 전 신원 확인) prompt만으로는 부족함
- escalation 중 구조화된 handoff 프로토콜(고객 ID, 사유, 권장 조치)

핵심 기술:

- 이전 단계가 완료될 때까지 downstream 호출을 차단하는 코드 수준의 사전조건(예: `get_customer`가 validated ID를 반환할 때까지 `process_refund` 차단)
- 여러 측면을 가진 고객 요청을 별도 항목으로 분해
- 사람에게 escalation할 때 structured summary 작성

1.5 tool 호출 가로채기와 데이터 normalization을 위한 Agent SDK hook

핵심 지식:

- 모델이 tool 결과를 소비하기 전에 가로채는 hook 패턴(예: `PostToolUse`)
- compliance 규칙을 강제하기 위해 발신 호출을 가로채는 hook(예: 임계값 초과 환불 차단)
- hook은 **결정적 보장**을 제공하고, prompt 지시사항은 **확률적 준수**를 제공

핵심 기술:

- 데이터 형식 normalization을 위한 `PostToolUse` hook(Unix timestamp, ISO 8601, 숫자 상태 코드)
- 정책 위반 동작을 차단하고 escalation으로 리디렉션하는 가로채기 hook
- 비즈니스 규칙에 보장된 준수가 필요하면 prompt보다 hook 선택

1.6 복잡한 workflow를 위한 작업 분해 전략

핵심 지식:

- 중간 결과 기반 **고정 pipeline**(prompt chaining) vs **동적 적응형 분해**
- prompt chaining: 순차 단계(각 파일을 따로 분석한 뒤 통합 패스 실행)
- 발견 내용을 기반으로 하위 작업을 생성하는 적응형 조사 계획

핵심 기술:

- 예측 가능한 다면 리뷰에는 prompt chaining 사용, 개방형 조사에는 동적 분해 사용
- 큰 코드 리뷰를 파일별 분석과 별도 파일 간 통합 패스로 분할

- 개방형 작업 분해: 먼저 구조를 매핑한 뒤 우선순위 계획 수립

1.7 session 상태, 재개, 포킹

핵심 지식:

- 이름이 지정된 session을 계속하기 위한 `--resume <session-name>`
- 공유 context에서 독립 조사 브랜치를 만들기 위한 `fork_session`
- session 재개 시 파일 변경 사항을 Agent에 알리는 것의 중요성
- 오래된 결과로 재개하는 것보다 structured summary로 새 session을 시작하는 편이 더 신뢰할 수 있음

핵심 기술:

- 이름이 지정된 조사 session을 계속하기 위해 `--resume` 사용
- 접근 방식을 병렬로 비교하기 위해 `fork_session` 사용
- 재개(context가 여전히 최신)와 새 session 시작(결과가 오래됨) 중 선택

도메인 2: tool 설계와 MCP 통합(18%)

2.1 명확한 설명을 갖춘 tool interface 설계

핵심 지식:

- tool 설명은 LLM이 tool을 선택하는 데 사용하는 **기본 메커니즘**이며, 최소한의 설명은 불안정한 선택으로 이어짐
- 입력 형식, 예시 쿼리, 엣지 케이스, 적용 경계를 포함하는 것의 중요성
- 모호하거나 겹치는 설명은 잘못된 routing을 유발함
- system prompt 문구가 tool과 의도치 않은 연관을 만들 수 있음

핵심 기술:

- 각 tool을 유사한 대안과 명확히 구분하는 설명 작성
- 기능 겹침을 제거하기 위해 tool 이름 변경(예: `analyze_content` -> `extract_web_results`)
- 범용 tool을 명확한 입력/출력 계약을 가진 전문 tool로 분할

2.2 MCP tool의 structured error 응답 구현

핵심 지식:

- MCP tool 응답의 `isError` 플래그
- **일시적 오류**(타임아웃), **validation 오류**(잘못된 입력), **비즈니스 오류**(정책 위반), **접근/권한 오류**의 차이
- 일반 오류("Operation failed")는 올바른 복구 결정을 방해함
- retry 가능한 오류와 불가능한 오류의 차이

핵심 기술:

- `errorCategory`(transient/validation/permission), `isRetryable`, 사람이 읽을 수 있는 메시지 등 structured metadata 반환
- 비즈니스 규칙 위반에는 명확한 사용자 대상 설명과 함께 `retryable: false` 사용
- 일시적 실패는 subagent 내부에서 로컬 복구하고, 해결할 수 없는 오류만 전파
- 접근 실패(retry 결정 필요)와 유효한 빈 결과(일치 없음)를 구분

2.3 Agent 간 tool 할당과 `tool_choice` 구성

핵심 지식:

- Agent당 tool이 너무 많으면(예: 4-5개가 아니라 18개) tool 선택 신뢰성이 **낮아짐**
- 전문 범위 밖의 tool을 가진 Agent는 이를 오용하는 경향이 있음
- 범위 지정 tool 접근: 역할 관련 tool과 제한된 교차 역할 유틸리티만 허용
- `tool_choice`: `"auto"`, `"any"`, 강제 tool 선택(`{"type": "tool", "name": "..."}`)

핵심 기술:

- 각 subagent의 tool 집합을 역할과 관련된 것으로 제한
- 일반 tool을 제약된 대안으로 교체(예: `fetch_url` -> `load_document`)
- 텍스트 답변 대신 tool 호출을 보장하기 위해 `tool_choice: "any"` 사용
- 실행 순서를 보장하기 위해 특정 tool 강제

2.4 Claude Code와 Agent workflow에 MCP 서버 통합

핵심 지식:

- MCP 서버 범위: 팀용 프로젝트(`.mcp.json`) vs 실험용 사용자(`~/.claude.json`)
- 비밀값 관리를 위한 `.mcp.json`의 환경 변수 치환(예: `${GITHUB_TOKEN}`)
- 연결된 모든 MCP 서버의 tool은 연결 시 발견되며 동시에 사용 가능
- 탐색적 tool 호출을 줄이는 "콘텐츠 카탈로그"로서의 MCP resource(작업 summary, 데이터베이스 schema)

핵심 기술:

- env-var 기반 token으로 프로젝트 `.mcp.json`에 공유 MCP 서버 구성
- 개인/실험용 서버는 `~/.claude.json`에 유지
- 표준 통합에는 사용자 지정 서버보다 커뮤니티 MCP 서버 선호

2.5 built-in tool(Read, Write, Edit, Bash, Grep, Glob) 선택과 적용

핵심 지식:

- **Grep**: 파일 내용 안에서 검색(함수 이름, 오류 메시지, import)
- **Glob**: 이름/확장자 패턴으로 파일 찾기

- **Read/Write**: 전체 파일 작업, **Edit**: 고유 텍스트 일치를 통한 정확한 변경
- Edit이 고유하지 않은 일치 때문에 실패하면 Read + Write로 대체

핵심 기술:

- 내용 검색에는 Grep, 패턴 기반 파일 발견에는 Glob 사용
- 이해를 점진적으로 구축: 진입점을 Grep한 뒤 Read로 흐름 추적
- 래퍼 module을 통해 함수 사용처 추적

도메인 3: Claude Code 구성과 workflow(20%)

3.1 계층, 범위, module식 구성을 갖춘 CLAUDE.md 구성

핵심 지식:

- CLAUDE.md 계층: 사용자(`~/.claude/CLAUDE.md`), 프로젝트(`.claude/CLAUDE.md` 또는 루트 `CLAUDE.md`), 디렉터리 수준(하위 디렉터리의 CLAUDE.md)
- 사용자 수준 설정은 한 사용자에게만 적용되며 VCS를 통해 공유되지 않음
- CLAUDE.md를 module화하기 위한 외부 파일 참조용 `@path` 문법(예: `@./standards/coding-style.md`)
- 단일 CLAUDE.md 대신 주제 중심 규칙 파일을 위한 `.claude/rules/` 디렉터리

핵심 기술:

- 계층 문제 진단(새 팀원이 지시사항을 놓치는 이유가 프로젝트 수준이 아니라 사용자 수준에 있기 때문임)
- 각 패키지의 CLAUDE.md에 표준을 선택적으로 포함하기 위해 `@path`(예: `@./standards/testing.md`) 사용
- 큰 CLAUDE.md를 여러 `.claude/rules/` 파일(testing.md, api-conventions.md, deployment.md)로 분할

3.2 사용자 지정 슬래시 명령과 Skill 생성 및 구성

핵심 지식:

- `.claude/commands/`의 **프로젝트 명령**(VCS로 공유) vs `~/.claude/commands/`의 **사용자 명령**
- `.claude/skills/`의 Skill과 `SKILL.md` frontmatter: `context: fork`, `allowed-tools`, `argument-hint`
- `context: fork`는 Skill을 격리된 subagent context에서 실행해 메인 session context를 불필요하게 채우지 않음
- 개인 Skill 변형은 `~/.claude/skills/` 아래 다른 이름으로 둘 수 있음

핵심 기술:

- 팀 전체가 사용할 수 있도록 프로젝트 슬래시 명령을 `.claude/commands/`에 저장
- 상세 출력이 많은 Skill을 격리하기 위해 `context: fork` 사용
- Skill이 사용할 수 있는 tool을 제한하기 위해 `allowed-tools` 사용

- 개발자에게 필수 매개변수를 요청하기 위해 `argument-hint` 사용

3.3 조건부 규칙 로딩을 위한 경로별 규칙 사용

핵심 지식:

- `.claude/rules/` 파일은 glob 패턴을 기반으로 규칙을 활성화하기 위해 YAML frontmatter `paths`를 포함할 수 있음
- 경로 범위 규칙은 일치하는 파일을 편집할 때만 **로드**되어 context와 token을 절약함
- 규칙이 여러 디렉터리에 적용될 때(예: 테스트) glob 기반 경로 규칙이 디렉터리 수준 CLAUDE.md보다 나을 수 있음

핵심 기술:

- 일치하는 파일 작업 시에만 로드되도록 `paths: ["terraform/**/*"]`가 있는 `.claude/rules/` 파일 생성
- 위치와 관계없이 파일 유형별 규칙을 적용하기 위해 glob 패턴(`**/*.test.tsx`) 사용
- 규칙이 코드베이스 전반에 걸쳐 있을 때 디렉터리 수준 CLAUDE.md보다 경로별 규칙 선호

3.4 계획 모드와 직접 실행 사용 시점 결정

핵심 지식:

- **계획 모드**: 큰 변경, 여러 가능한 접근, architecture 결정이 있는 복잡한 작업용
- **직접 실행**: 단일 validation 추가처럼 단순하고 잘 이해된 변경용
- 계획 모드는 변경 전에 코드베이스를 안전하게 탐색할 수 있음
- Explore subagent는 상세 탐색 출력을 격리함

핵심 기술:

- architecture 결과가 있는 작업(마이크로서비스, 45개 이상 파일을 건드리는 migration)에 계획 모드 사용
- 명확한 스택 트레이스와 단일 파일이 있는 수정에는 직접 실행 사용
- 다단계 작업에서 context window 고갈을 막기 위해 Explore subagent 사용
- 접근 방식 결합: 발견에는 계획, 구현에는 실행

3.5 점진적 개선을 위한 반복적 정제

핵심 지식:

- 구체적인 입력/출력 예시는 기대사항을 전달하는 가장 효과적인 방법
- **테스트 주도 반복:** 먼저 테스트를 작성하고 실패를 기반으로 반복
- "인터뷰" 패턴: Claude가 질문을 통해 명확하지 않은 설계 고려사항을 드러냄
- 모든 이슈를 한 메시지에 제공할 때(상호 의존)와 순차적으로 제공할 때(독립)의 구분

핵심 기술:

- 변환 요구사항을 명확히 하기 위해 구체적 입력/출력 예시 2-3개 제공

- 구현 전에 기대 동작, 엣지 케이스, 성능 요구사항이 있는 테스트 세트 구축
- 인터뷰 패턴으로 설계 측면(cache 무효화, 실패 모드) 드러내기
- 엣지 케이스에 대한 샘플 입력과 기대 출력이 있는 구체적 테스트 케이스 제공

3.6 Claude Code를 CI/CD pipeline에 통합

핵심 지식:

- 자동화 pipeline의 비대화형 모드를 위한 `-p`(또는 `--print`) 플래그
- CI의 structured output을 위한 `--output-format json`과 `--json-schema`
- CLAUDE.md는 CI에서 트리거된 Claude Code에 프로젝트 context(테스트 표준, 리뷰 기준)를 제공
- **session context 격리:** 코드를 생성한 동일 session은 독립 instance보다 그 코드를 리뷰하는 데 효과가 떨어짐

핵심 기술:

- 대화형 입력에서 멈추지 않도록 CI에서 `-p`로 Claude Code 실행
- structured result(예: 인라인 PR 댓글)를 위해 `--output-format json` + `--json-schema` 사용
- 새 커밋 이후 다시 실행할 때 이전 리뷰 결과 포함(새 이슈/미수정 이슈만 보고)
- 테스트 생성 품질을 높이기 위해 CLAUDE.md에 테스트 표준과 사용 가능한 fixture 문서화
- 중복을 피하고 스타일을 일관되게 유지하기 위해 새 테스트 생성 시 기존 테스트 파일을 context에 포함

도메인 4: Prompt Engineering과 structured output(20%)

4.1 accuracy 향상을 위한 명시적 기준 prompt 설계

핵심 지식:

- 명시적 기준은 모호한 지시보다 효과적임(예: "코드와 모순될 때만 댓글 표시" vs "댓글 정확성 확인")
- "더 보수적으로 하라" 같은 일반 지침은 구체적인 범주형 기준보다 효과가 낮음
- false positive가 개발자 신뢰에 미치는 영향: 일부 범주의 높은 false positive 비율은 정확한 범주에 대한 confidence 약화시킴

핵심 기술:

- 리뷰 기준 정의: 보고할 것(버그, 보안) vs 무시할 것(사소한 스타일)
- false positive 비율이 높은 범주를 일시적으로 비활성화
- 각 수준별 코드 예시와 함께 명시적 severity 기준 정의

4.2 출력 일관성 향상을 위한 Few-shot prompting 사용

핵심 지식:

- Few-shot 예시는 일관된 형식의 실행 가능한 출력을 생성하는 가장 효과적인 방법
- Few-shot은 모호한 사례(tool 선택, 테스트 커버리지 공백) 처리를 보여줄 수 있음
- Few-shot은 모델이 기본값을 단순 반복하는 대신 새 패턴으로 일반화하도록 도움
- Few-shot은 extraction 작업에서 환각을 줄일 수 있음

핵심 기술:

- 근거와 함께 모호한 시나리오를 위한 목표 예시 2-4개 제공
- 출력 형식(위치, 이슈, severity, 제안 수정)을 보여주는 few-shot 예시 포함
- 허용 가능한 코드 패턴과 실제 이슈를 구분하는 예시 제공
- 서로 다른 구조의 문서에서 올바르게 extraction하는 예시 제공

4.3 `tool_use`와 JSON schema로 structured output 강제

핵심 지식:

- JSON Schema와 함께 `tool_use`를 사용하는 것은 schema 준수 출력을 보장하고 JSON 구문 오류를 제거하는 가장 신뢰할 수 있는 방법
- `tool_choice: "auto"`에서는 모델이 텍스트를 반환할 수 있고, `"any"`에서는 반드시 tool을 호출해야 하며, 강제 선택은 특정 tool을 선택함
- 엄격한 JSON Schema는 구문 오류를 제거하지만 의미 오류(합계 불일치, 값이 잘못된 필드에 있음)는 막지 못함
- schema 설계: 필수 vs 선택 필드, 확장성을 위한 "other" enum과 세부 문자열

핵심 기술:

- JSON Schema가 있는 extraction tool을 정의하고 `tool_use` 결과에서 데이터 parsing
- 여러 schema가 있을 때 structured output을 보장하기 위해 `tool_choice: "any"` 사용
- 특정 tool 호출 강제: `tool_choice: {"type": "tool", "name": "extract_metadata"}`
- 원본에 정보가 없을 수 있으면 값을 지어내지 않도록 필드를 선택/nullable로 설정
- 확장 가능한 범주화를 위해 `"unclear"`와 `"other"` 같은 enum 값과 세부 필드 사용

4.4 extraction 품질을 위한 validation, retry, 피드백 루프 구현

핵심 지식:

- 오류 피드백 기반 retry: retry prompt에 구체적 validation 오류를 포함해 수정을 유도
- 정보가 원본에 단순히 없을 때 retry는 효과가 없음
- 피드백 루프 설계: 발견을 트리거한 패턴(`detected_pattern`) 추적
- 의미 오류(합계가 맞지 않음) vs 구문 오류(`tool_use`로 처리)

핵심 기술:

- 원본 문서, 잘못된 extraction, 구체적 validation 오류를 포함한 후속 prompt
- retry가 효과 없는 시점 식별(필수 정보가 외부 문서에만 있음)
- false positive 분석을 위해 발견 사항에 `detected_pattern` 필드 포함
- 불일치 감지를 위해 `calculated_total`과 `stated_total`을 모두 extraction하는 자기 교정 설계

4.5 효율적인 batch 처리 전략 설계

핵심 지식:

- Message Batches API: 50% 비용 절감, 최대 24시간 처리 window, 지연 시간 SLA 보장 없음
- batch 처리는 비차단 작업(야간 보고서, 감사)에 적합하고 차단 작업(병합 전 검사)에는 부적합함
- Batch API는 단일 요청 내 멀티턴 tool 호출을 지원하지 않음
- `custom_id` 필드는 batch 내 요청/응답을 연결함

핵심 기술:

- 차단 검사는 동기 API 사용, 야간/주간 워크로드는 Batch API 사용
- SLA 요구에 따라 batch 제출 주기 계획(예: 24시간 처리로 30시간 보장을 위해 4시간 window)
- `custom_id`로 식별된 실패 문서만 다시 제출하여 실패 처리
- 대규모 처리 전에 샘플로 prompt 반복 개선

4.6 멀티 instance와 다중 패스 리뷰 architecture 설계

핵심 지식:

- 자기 리뷰의 한계: 모델은 자신의 추론 context를 유지해 스스로의 결정을 덜 비판할 가능성이 큼
- 생성 context가 없는 독립 리뷰 instance가 미묘한 이슈를 더 잘 찾음
- 다중 패스 리뷰: 주의 분산을 피하기 위해 파일별 로컬 분석과 파일 간 통합 패스 수행

핵심 기술:

- 생성 context 없이 변경 사항을 리뷰하도록 두 번째 독립 Claude instance 사용
- 다중 파일 리뷰를 파일별 패스와 파일 간 데이터 흐름 분석용 통합 패스로 분할
- calibration된 방식으로 리뷰를 routing하기 위해 자체 평가 confidence가 있는 validation 패스 사용

도메인 5: context 관리와 안정성(15%)

5.1 중요한 정보를 보존하기 위한 대화 context 관리

핵심 지식:

- 점진적 summary의 위험: 숫자 값, 백분율, 날짜가 모호한 summary로 압축됨

- 중간 손실 효과: 모델은 긴 입력의 시작과 끝은 안정적으로 처리하지만 중간의 발견 사항을 놓칠 수 있음
- tool 출력이 관련성에 비해 과도하게 context에 누적될 수 있음(필요한 것은 5개인데 40개 이상의 필드)
- 후속 API 요청에서 전체 대화 기록을 보내는 것의 중요성

핵심 기술:

- 거래 관련 사실을 summary된 기록 밖의 지속적 "case facts" 블록으로 extraction
- 상세한 tool 출력을 관련 필드로 축소
- 명시적 섹션 제목과 함께 집계 데이터의 시작 부분에 핵심 발견 batch
- subagent가 structured output에 metadata(날짜, source)를 포함하도록 요구

5.2 효과적인 escalation 패턴과 모호성 해결 설계

핵심 지식:

- 적절한 escalation 트리거: 사람에 대한 명시적 요청, 정책 공백/예외, 진행 불가
- 즉시 escalation(명시적 요청) vs 해결 시도(Agent 범위 내)
- 감정 분석과 모델 confidence 자체 평가는 케이스 복잡도에 대한 신뢰할 수 없는 대리 지표임
- 여러 고객 일치 항목은 heuristic 추측이 아니라 추가 식별자 요청이 필요함

핵심 기술:

- system prompt에 few-shot 예시와 함께 명시적 escalation 기준 작성
- 사람에 대한 명시적 요청은 추가 조사 없이 즉시 실행
- 특정 요청에 대해 정책이 모호하거나 침묵할 때 escalation
- tool 결과에 여러 일치 항목이 있으면 추가 식별자 요청

5.3 multi-agent 시스템의 오류 전파 전략 구현

핵심 지식:

- structured error context(실패 유형, 쿼리, 부분 결과, 대안)는 Coordinator가 더 나은 복구 결정을 내리는 데 필요함
- 접근 실패(타임아웃은 retry 결정 필요)와 유효한 빈 결과(일치 없음)를 구분
- 일반 오류 상태("search unavailable")는 Coordinator로부터 유용한 context를 숨김
- 조용한 억제나 단일 실패로 전체 workflow 중단은 모두 안티패턴

핵심 기술:

- structured error context 반환: 실패 유형, 시도한 내용, 부분 결과, 가능한 대안
- 접근 실패와 유효한 빈 결과 구분
- 일시적 실패는 subagent에서 로컬 복구하고, 복구 불가능한 오류만 부분 결과와 함께 전파
- 종합 결과에 커버리지 주석 추가: 근거가 충분한 부분과 공백이 남은 부분

5.4 큰 코드베이스 조사 시 효율적인 context 관리

핵심 지식:

- 긴 session의 context 저하: 모델이 특정 클래스 대신 "일반적 패턴"을 언급하며 불안정한 답을 내기 시작함
- 스크래치패드 파일은 context 경계 전반에서 핵심 발견을 보존함
- subagent에게 위임하면 상세 탐색 출력을 격리할 수 있음
- 구조화된 상태를 지속하면 크래시 후 복구할 수 있음

핵심 기술:

- 메인 Agent는 고수준 조정을 맡기고 구체적 질문에는 subagent 생성
- 핵심 발견을 저장하고 나중에 참조하기 위해 스크래치패드 파일 사용
- 다음 단계 subagent를 만들기 전에 핵심 발견 summary를 작성
- 긴 조사 중 context 사용량을 줄이기 위해 `/compact` 사용

5.5 사람의 감독과 confidence calibration을 갖춘 workflow 설계

핵심 지식:

- 전체 지표(예: 전체 accuracy 97%)는 특정 문서 유형이나 필드의 낮은 성능을 가릴 수 있음
- 계층화 무작위 sampling은 높은 confidence extraction의 오류율을 측정함
- 라벨링된 validation 세트를 사용한 필드 수준 confidence calibration
- 자동화 전에 문서 유형과 필드 세그먼트별 accuracy validation

핵심 기술:

- 새로운 오류 패턴 감지를 위해 계층화 무작위 sampling 구현
- 성능을 안정적으로 validation을 위해 문서 유형별, 필드별 accuracy 분석
- 필드별 confidence 점수를 출력하고 라벨링 데이터로 리뷰 임계값 calibration
- 낮은 confidence 또는 모호한 source의 extraction은 사람 리뷰로 routing

5.6 다중 source 종합에서 source 보존과 불확실성 처리

핵심 지식:

- "주장 -> source" 매핑을 보존하지 않으면 summary 중 attribution이 손실됨
- structured 매핑은 집계 중에도 보존되어야 함
- 충돌하는 통계는 임의로 한 값을 선택하지 말고 source와 함께 충돌을 주석 처리
- 시간적 차이를 모순으로 잘못 읽지 않도록 발행/수집 날짜 포함

핵심 기술:

- subagent가 "주장 -> source" 매핑(URL, 문서명, citation)을 출력하도록 요구
- 안정적인 발견과 논쟁적 발견을 분리하도록 보고서를 structured 형식으로 구성
- 충돌하는 값을 주석과 함께 보존하고 조정을 위해 Coordinator에 전달

- 올바른 시간적 해석을 위해 발행 날짜 포함
- 콘텐츠 유형별 렌더링: 금융 데이터는 표, 뉴스는 산문, 기술 발견은 structured 목록

해설이 포함된 시험 문제 예시

문제 1(시나리오: 고객 지원 Agent)

상황: 데이터에 따르면 12%의 경우 Agent가 `get_customer`를 건너뛰고 고객 이름만으로 `lookup_order`를 호출하여 잘못된 환불이 발생합니다.

가장 효과적인 변경 사항은 무엇인가요?

- A) `get_customer`에서 ID를 얻을 때까지 `lookup_order`와 `process_refund`를 차단하는 코드 수준의 사전조건 추가 **[정답]**
- B) system prompt 개선
- C) Few-shot 예시 추가
- D) routing classifier 구현

A인 이유: 중요한 비즈니스 로직에 특정 tool 순서가 필요할 때 소프트웨어는 prompt 기반 접근(B, C)이 제공할 수 없는 **결정적 보장**을 제공합니다. D는 tool 순서가 아니라 가용성 문제를 다룹니다.

문제 2(시나리오: 고객 지원 Agent)

상황: Agent가 주문 관련 질문에 대해 `lookup_order` 대신 `get_customer`를 자주 호출합니다. tool 설명은 최소한이며 서로 비슷합니다.

첫 단계는 무엇인가요?

- A) Few-shot 예시
- B) 각 tool 설명을 입력 형식, 예시, 경계와 함께 확장 **[정답]**
- C) routing 계층 추가
- D) tool 병합

B인 이유: tool 설명은 모델의 기본 선택 메커니즘입니다. 이는 가장 적은 노력으로 가장 큰 효과를 내는 수정입니다. A는 근본 원인을 해결하지 못하고 token만 더 사용합니다. C는 과도한 엔지니어링입니다. D는 효과에 비해 변경 범위가 너무 큽니다.

문제 3(시나리오: 고객 지원 Agent)

상황: Agent의 이슈 해결률은 목표 80%에 비해 55%에 그칩니다. 단순한 케이스를 escalation하고 복잡한 정책 예외는 자율적으로 처리하려고 합니다.

calibration을 어떻게 개선하나요?

- A) Few-shot 예시와 함께 명시적 escalation 기준 추가 **[정답]**
- B) 자체 평가 confidence(1-10)와 자동 escalation
- C) 과거 데이터로 학습한 별도 classifier
- D) 감정 분석

A인 이유: 불명확한 결정 경계라는 근본 원인을 직접 해결합니다. B는 신뢰할 수 없습니다(모델은 확신 있게 틀릴 수 있음). C는 과도한 엔지니어링입니다. D는 다른 문제를 해결하려는 접근입니다. 고객의 기분과 케이스 복잡도는 같은 개념이 아닙니다.

문제 4(시나리오: Claude Code를 사용한 코드 생성)

상황: 저장소를 클론한 전체 팀이 사용할 수 있는 표준 코드 리뷰용 사용자 지정 `/review` 명령이 필요합니다.

명령 파일을 어디에 만들어야 하나요?

- A) 프로젝트 저장소의 `.claude/commands/` **[정답]**
- B) `~/.claude/commands/`
- C) 루트 `CLAUDE.md`
- D) `.claude/config.json`

A인 이유: `.claude/commands/`에 저장된 프로젝트 명령은 버전 관리되며 모두에게 자동으로 제공됩니다. B는 개인 명령용입니다. C는 명령 정의가 아니라 지시사항용입니다. D는 존재하지 않습니다.

문제 5(시나리오: Claude Code를 사용한 코드 생성)

상황: 모놀리스를 마이크로서비스로 재구성해야 합니다(수십 개 파일, 서비스 경계 결정).

어떤 접근 방식을 사용해야 하나요?

- A) 계획 모드: 코드베이스 탐색, 의존성 이해, 접근 방식 설계 **[정답]**
- B) 점진적 직접 실행
- C) 상세한 사전 지시와 함께 직접 실행
- D) 직접 실행하다 어려워지면 계획으로 전환

A인 이유: 계획 모드는 큰 변경, 여러 가능한 접근, architecture 결정에 맞게 설계되었습니다. B는 비용이 큰 재작업으로 이어질 위험이 있습니다. C는 이미 구조를 알고 있다고 가정합니다. D는 문제가 생긴 뒤에야 대응하는 방식입니다.

문제 6(시나리오: Claude Code를 사용한 코드 생성)

상황: 코드베이스 영역마다 규칙이 다릅니다(React, API, 데이터베이스). 테스트는 코드와 함께 위치합니다. 규칙이 자동으로 적용되기를 원합니다.

어떤 접근 방식을 사용해야 하나요?

- A) YAML frontmatter와 glob 패턴이 있는 `.claude/rules/` 파일 **[정답]**
- B) 모든 것을 루트 CLAUDE.md에 넣기
- C) `.claude/skills/`의 Skill
- D) 모든 디렉터리에 CLAUDE.md batch

A인 이유: glob 패턴(예: `**/*.test.tsx`)이 있는 `.claude/rules/`는 파일 경로를 기반으로 규칙을 자동 적용할 수 있어 코드베이스 전반에 흩어진 테스트에 적합합니다. B는 모델이 알아서 판단하기를 기대하는 방식입니다. C는 수동으로 실행해야 하는 on-demand 방식입니다. D는 관련 파일이 여러 디렉터리에 있을 때 잘 작동하지 않습니다.

문제 7(시나리오: Multi-Agent Research System)

상황: 시스템이 "AI가 창작 산업에 미치는 영향"을 조사하지만 보고서는 시각 예술만 다룹니다. Coordinator는 주제를 "디지털 아트의 AI", "그래픽 디자인의 AI", "사진의 AI"로 분해했습니다.

원인은 무엇인가요?

- A) 종합 Agent가 공백을 감지하지 못함
- B) Coordinator가 작업을 너무 좁게 분해함 **[정답]**
- C) 웹 검색 Agent가 충분히 철저히 검색하지 않음
- D) 문서 분석 Agent가 비시각 자료를 필터링함

B인 이유: 로그는 Coordinator가 "창작 산업"을 시각 하위 주제로만 분해해 음악, 문학, 영화를 완전히 놓쳤음을 보여줍니다. subagent들은 주어진 작업을 제대로 수행했습니다. 문제는 처음부터 작업 범위가 잘못 나뉜 데 있습니다.

문제 8(시나리오: Multi-Agent Research System)

상황: 웹 검색 subagent가 복잡한 주제를 조사하다 타임아웃됩니다. 오류 정보가 Coordinator로 전달되는 방식을 설계해야 합니다.

복구 판단에 가장 도움이 되는 오류 전파 접근 방식은 무엇인가요?

- A) 실패 유형, 쿼리, 부분 결과, 대안을 포함한 structured error context를 Coordinator에 반환 **[정답]**
- B) subagent 내부에서 지수 백오프 자동 retry를 구현한 뒤 일반적인 "search unavailable" 상태 반환
- C) subagent 내부에서 타임아웃을 잡고 빈 결과 집합을 성공으로 표시해 반환
- D) 최상위 핸들러에 타임아웃 예외를 전파해 전체 workflow 종료

A인 이유: structured error context는 Coordinator가 수정된 쿼리로 retry할지, 대안 접근을 시도할지, 부분 결과로 계속할지 결정하는 데 필요한 정보를 줍니다. B는 일반 상태만 반환해 필요한 context를 숨깁니다. C는 실패를 성공으로 위장합니다. D는 전체 workflow를 불필요하게 중단합니다.

문제 9(시나리오: Multi-Agent Research System)

상황: 종합 Agent는 결과를 병합하면서 특정 주장을 validation해야 하는 경우가 많습니다. 현재 validation이 필요하면 종합 Agent가 제어를 Coordinator에게 돌려주고, Coordinator가 웹 검색 Agent를 호출한 뒤 새 결과로 종합을 다시 실행합니다. 그 결과 작업마다 왕복 호출이 2-3회 늘어나고 지연 시간이 40% 증가합니다. 평가 결과 이러한 확인의 85%는 단순 사실 확인(날짜, 이름, 통계)이고 15%는 더 깊은 조사가 필요합니다.

안정성을 유지하면서 overhead를 줄이려면 어떻게 해야 하나요?

- A) 종합 Agent에 단순 사실 확인용으로 범위를 제한한 `verify_fact` tool을 제공하고, 복잡한 validation은 계속 Coordinator를 통해 routing **[정답]**
- B) 모든 validation 필요 사항을 batch로 모아 마지막에 Coordinator에게 반환
- C) 종합 Agent에 모든 웹 검색 tool에 대한 전체 접근 권한 제공
- D) 각 source 주변의 추가 context를 사전에 cache

A인 이유: 이는 최소 권한 원칙에 맞는 설계입니다. 종합 Agent는 85%의 일반 사례(단순 사실 확인)에 필요한 것만 받고, 복잡한 조사를 위해서는 Coordinator 경유 경로를 유지합니다. B는 뒤 단계가 앞 단계 결과를 기다려야 하는 의존성을 만듭니다(나중 종합 단계가 앞서 validated 사실에 의존할 수 있음). C는 책임 경계를 무너뜨립니다. D는 언제 필요할지 안정적으로 예측하기 어려운 caching에 기대게 됩니다.

문제 10(시나리오: CI를 위한 Claude Code)

상황: pipeline이 `claude "Analyze this pull request for security issues"`를 실행하지만 대화형 입력을 기다리며 멈춥니다.

올바른 접근 방식은 무엇인가요?

- A) `-p` 플래그 사용: `claude -p "Analyze this pull request for security issues"` **[정답]**
- B) `CLAUDE_HEADLESS=true` 설정
- C) stdin을 `/dev/null`로 리디렉션
- D) `--batch` 사용

A인 이유: `-p`(또는 `--print`)는 Claude Code를 비대화형 모드로 실행하는 문서화된 실행 방식입니다. prompt를 처리하고 결과를 stdout에 출력한 뒤 종료합니다. 다른 옵션은 존재하지 않는 기능이거나 Unix 우회 방법입니다.

문제 11(시나리오: CI를 위한 Claude Code)

상황: 팀은 자동 분석의 API 비용을 줄이고 싶어 합니다. 현재 Claude는 실시간으로 두 workflow를 처리합니다. (1) 개발자가 PR을 병합하기 전에 반드시 완료되어야 하는 차단형 병합 전 검사, (2) 아침 검토를 위해 밤새 생성되는 기술 부채 보고서. 관리자는 50% 절감을 위해 둘 다 Message Batches API로 옮기자고 제안합니다.

이 제안을 어떻게 평가해야 하나요?

- A) 기술 부채 보고서에만 batch 처리를 사용하고, 병합 전 검사는 실시간 호출 유지 **[정답]**
- B) 두 workflow 모두 batch 처리로 옮기고 완료를 polling
- C) batch 결과의 순서 문제를 피하기 위해 둘 다 실시간 호출 유지

- D) 둘 다 batch 처리로 옮기고 batch가 너무 오래 걸리면 실시간으로 fallback

A인 이유: Message Batches API는 50%를 절감하지만 처리 시간이 최대 24시간이고 지연 시간 SLA가 보장되지 않습니다. 따라서 개발자가 기다리는 차단형 병합 전 검사에는 부적합하지만, 기술 부채 보고서처럼 밤새 처리해도 되는 작업에는 적합합니다.

문제 12(시나리오: 다중 파일 코드 리뷰)

상황: 풀 리퀘스트가 재고 추적 module의 14개 파일을 변경합니다. 모든 파일에 대한 단일 패스 리뷰는 일부 파일에는 자세한 댓글을 달지만 다른 파일은 피상적으로 다루고, 명백한 버그를 놓치며, 동일한 코드 패턴을 한 파일에서는 문제로 표시하지만 다른 파일에서는 승인하는 모순된 피드백을 생성합니다.

리뷰를 어떻게 재구성해야 하나요?

- A) 집중된 패스로 분할: 각 파일을 개별적으로 로컬 이슈에 대해 분석한 뒤 파일 간 데이터 흐름에 대한 별도 통합 패스 실행 **[정답]**
- B) 개발자가 큰 PR을 3-4개 파일 단위 제출로 나누도록 요구
- C) 더 큰 context window가 있는 상위 모델로 전환해 14개 파일을 한 번에 리뷰
- D) 전체 PR 리뷰 패스를 세 번 독립적으로 실행하고 최소 두 번 발견된 이슈만 보고

A인 이유: 집중 패스는 많은 파일을 한 번에 처리할 때 발생하는 주의 분산이라는 근본 원인을 직접 해결합니다. 파일별 분석은 일관된 깊이를 보장하고, 별도 통합 패스는 파일 간 이슈를 잡습니다. B는 시스템을 개선하기보다 개발자에게 부담을 넘기는 방식입니다. C는 문제를 잘못 짚은 선택입니다. context가 커진다고 attention 품질 문제가 자동으로 해결되지는 않습니다. D는 탐지가 일관되지 않다는 전제에서 다수결을 쓰므로 실제 버그까지 걸러낼 수 있습니다.

연습 테스트

> 4개 시나리오에 걸친 60개 문제입니다. 형식과 난이도는 실제 시험과 일치합니다.
>
> 또는 시험과 유사한 HTML 파일에서 이 문제들을 연습할 수 있습니다: Practical Test (KO)

시나리오: Multi-Agent Research System

문제 1(시나리오: Multi-Agent Research System)

상황: 문서 분석 Agent가 핵심 지표에 대해 두 개의 신뢰할 수 있는 source가 서로 정면으로 충돌하는 통계가 있음을 발견했습니다. 정부 보고서는 40% 성장을 제시하고, 산업 분석은 12%를 제시합니다. 두 source 모두 신뢰할 만해 보이며, 이 차이는 research 결론에 큰 영향을 줄 수 있습니다. 문서 분석 Agent는 이 상황을 가장 효과적으로 어떻게 처리해야 하나요?

가장 효과적인 접근 방법은 무엇인가요?

- A) confidence heuristic을 적용해 가장 맞을 가능성이 큰 숫자를 선택하고, 그 값으로 분석을 마치며 불일치를 언급하는 각주를 추가한다.
- B) 두 숫자를 충돌로 표시하지 않고 분석 출력에 모두 포함해, 종합 Agent가 더 넓은 context를 기준으로 사용할 값을 결정하게 한다.
- C) 분석을 중단하고 즉시 Coordinator에게 escalation해, 계속하기 전에 어느 source가 더 권위 있는지 결정해 달라고 요청한다.
- D) 두 숫자를 모두 포함해 분석을 마무리하되 source attribution과 함께 충돌을 명시적으로 주석 처리하고, 종합 단계로 넘기기 전에 Coordinator가 데이터를 조정할 방법을 결정하게 한다. **[정답]**

D인 이유: 이 방식은 책임 분리를 유지합니다. 분석 Agent는 작업을 멈추지 않고 핵심 분석을 끝내고, 충돌하는 두 값을 명확한 attribution과 함께 보존하며, 더 넓은 context를 가진 Coordinator에게 조정 책임을 넘깁니다.

문제 2(시나리오: Multi-Agent Research System)

상황: 웹 검색 Agent와 문서 분석 Agent가 작업을 완료하고 결과를 Coordinator에게 반환했습니다. 통합 research 보고서를 만들기 위한 다음 단계는 무엇인가요?

가장 적절한 다음 단계는 무엇인가요?

- A) 각 Agent가 Coordinator를 우회해 보고서 작성 Agent에게 직접 결과를 보낸다.
- B) 문서 분석 Agent가 웹 검색 결과를 요청해 내부적으로 병합한다.
- C) Coordinator가 두 결과 집합을 종합 Agent에게 전달해 통합한다. **[정답]**
- D) Coordinator가 두 Agent의 원시 출력을 연결해 최종 결과로 반환한다.

C인 이유: Coordinator-subagent architecture에서 Coordinator는 두 결과 집합을 중앙에서 통합할 종합 Agent에게 전달하여 제어권을 유지하고 품질 높은 병합을 수행할 수 있습니다.

문제 3(시나리오: Multi-Agent Research System)

상황: 문서 분석 subagent가 PDF 파일 처리에서 자주 실패합니다. 일부 파일에는 parsing 예외를 유발하는 손상된 섹션이 있고, 일부는 암호로 보호되어 있으며, 때로는 parsing 라이브러리가 큰 파일에서 멈춥니다. 현재는 예외가 발생하면 즉시 subagent를 종료하고 Coordinator에게 오류를 반환하며, Coordinator가 retry, 건너뛰기, 전체 작업 실패 여부를 결정해야 합니다. 이로 인해 일상적 오류 처리에 Coordinator가 과도하게 관여합니다. 가장 효과적인 architecture 개선은 무엇인가요?

가장 효과적인 개선은 무엇인가요?

- A) 공유 큐를 통해 모든 실패를 모니터링하고 복구 동작을 결정하며, subagent에게 직접 재시작 명령을 보내는 전담 오류 처리 Agent를 만든다.
- B) subagent가 항상 부분 결과를 성공 상태로 반환하고, 오류 세부사항은 metadata에 넣도록 구성한다. Coordinator는 모든 응답을 성공으로 처리한다.
- C) Coordinator가 모든 문서를 subagent로 보내기 전에 validation해 실패를 유발할 수 있는 문서를 거부한다.
- D) subagent에서 일시적 실패에 대한 로컬 복구를 구현하고, 해결할 수 없는 오류만 시도한 단계와 부분 결과를 포함해 Coordinator에게 escalation한다. **[정답]**

D인 이유: 오류는 해결할 수 있는 가장 낮은 수준에서 처리합니다. 로컬 복구는 Coordinator의 부담을 줄이면서, 실제로 복구할 수 없는 이슈는 전체 context와 부분 진행 상황을 포함해 escalation합니다.

문제 4(시나리오: Multi-Agent Research System)

상황: "AI가 창작 산업에 미치는 영향"에 대해 시스템을 실행한 후, 모든 subagent가 성공적으로 완료되는 것을 확인했습니다. 웹 검색 Agent는 관련 기사를 찾고, 문서 분석 Agent는 이를 제대로 summary하며, 종합 Agent는 일관된 텍스트를 생성합니다. 하지만 최종 보고서는 시각 예술만 다루고 음악, 문학, 영화를 전혀 다루지 못합니다. Coordinator 로그에는 주제가 "디지털 아트의 AI", "그래픽 디자인의 AI", "사진의 AI"라는 세 하위 작업으로 분해된 것이 보입니다. 가장 가능성이 높은 근본 원인은 무엇인가요?

가장 가능성이 높은 근본 원인은 무엇인가요?

- A) 종합 Agent에 커버리지 공백을 감지하라는 지시가 부족하다.
- B) 문서 분석 Agent가 지나치게 엄격한 관련성 기준 때문에 비시각 자료를 필터링한다.
- C) Coordinator의 작업 분해가 너무 좁아 관련 영역 전체를 포괄하지 못하는 작업을 subagent에게 할당했다. **[정답]**
- D) 웹 검색 Agent의 쿼리가 불충분하며 더 많은 분야를 포괄하도록 넓혀야 한다.

C인 이유: Coordinator가 넓은 주제를 시각 예술 하위 작업으로만 분해해 음악, 문학, 영화를 완전히 놓쳤습니다. subagent는 할당된 작업을 올바르게 실행했으므로, 좁은 분해가 가장 직접적인 원인입니다.

문제 5(시나리오: Multi-Agent Research System)

상황: 웹 검색 subagent가 요청된 5개 source 범주 중 3개에 대해서만 결과를 반환합니다(경쟁사 사이트와 산업 보고서는 성공했지만 뉴스 아카이브와 소셜 피드는 타임아웃). 문서 분석 subagent는 제공된 모든 문서를 성공적으로 처리했습니다. 종합 subagent는 품질이 섞인 upstream 입력으로 summary를 생성해야 합니다. 가장 효과적인 오류 전파 전략은 무엇인가요?

가장 효과적인 오류 전파 전략은 무엇인가요?

- A) 성공한 source만 사용해 종합을 계속하고 어떤 데이터를 사용할 수 없었는지 언급하지 않고 출력을 생성한다.
- B) 종합 subagent가 Coordinator에게 오류를 반환해 불완전한 데이터 때문에 전체 retry나 작업 실패를 트리거한다.
- C) 종합 subagent가 시작 전에 Coordinator에게 더 긴 타임아웃으로 시간 초과 source를 retry하도록 요청한다.
- D) 사용할 수 없는 source로 인한 공백과 근거가 충분한 결론을 나타내는 커버리지 주석으로 종합 출력을 구조화한다. **[정답]**

D인 이유: 커버리지 주석은 투명성을 갖춘 graceful degradation을 구현합니다. 완료된 작업의 가치를 보존하면서 불확실성을 전파해 confidence를 근거 있게 판단할 수 있습니다.

문제 6(시나리오: Multi-Agent Research System)

상황: 문서 분석 subagent가 parsing할 수 없는 손상된 PDF 파일을 만났습니다. 시스템 오류 처리를 설계할 때 이 실패를 처리하는 가장 효과적인 방법은 무엇인가요?

가장 효과적인 접근 방법은 무엇인가요?

- A) Coordinator Agent에게 context를 담은 오류를 반환해 진행 방법을 결정할 수 있게 한다. **[정답]**
- B) workflow 중단을 피하기 위해 손상된 문서를 아무 알림 없이 건너뛰고 나머지 파일 처리를 계속한다.
- C) 실패를 보고하기 전에 지수 백오프로 문서 parsing을 세 번 자동 retry한다.
- D) 전체 research workflow를 종료하도록 예외를 전파한다.

A인 이유: context를 담은 오류를 Coordinator에게 반환하는 것이 가장 효과적입니다. Coordinator가 파일을 건너뛸지, 대체 parsing 방법을 시도할지, 사용자에게 알릴지 근거를 갖고 결정할 수 있으며 실패가 보이도록 유지합니다.

문제 7(시나리오: Multi-Agent Research System)

상황: 프로덕션 로그는 "업로드된 분기 보고서를 분석해줘" 같은 요청이 문서 분석 Agent 대신 웹 검색 Agent로 45% routing되는 지속적 패턴을 보여줍니다. tool 정의를 검토하니 웹 검색 Agent에는 "콘텐츠를 분석하고 핵심 정보를 extraction한다"라고 설명된 `analyze_content` tool이 있고, 문서 분석 Agent에는 "문서를 분석하고 핵심 정보를 extraction한다"라고 설명된 `analyze_document` tool이 있습니다. 잘못된 routing 문제를 어떻게 고쳐야 하나요?

잘못된 routing 문제를 어떻게 고쳐야 하나요?

- A) Coordinator가 위임을 결정하기 전에 사용자가 업로드 파일을 말하는지 웹 콘텐츠를 말하는지 감지하는 사전 routing classifier를 추가한다.
- B) 웹 검색 tool 이름을 `extract_web_results`로 바꾸고 설명을 "웹 검색과 URL에서 검색된 정보를 처리하고 반환한다"로 업데이트한다. **[정답]**
- C) Coordinator prompt에 올바른 routing을 보여주는 few-shot 예시를 추가한다: "사용자가 분기 보고서를 업로드 -> 문서 분석 Agent", "사용자가 웹 페이지를 질문 -> 웹 검색 Agent".
- D) 문서 분석 tool 설명에 "업로드된 PDF, Word 문서, 스프레드시트에 사용" 같은 사용 예시를 추가하고 웹 검색 tool은 그대로 둔다.

B인 이유: 웹 검색 tool 이름을 `extract_web_results`로 바꾸고 설명에 웹 검색과 URL을 명시하면 두 tool 이름과 설명 사이의 의미상 겹침을 제거해 근본 원인을 직접 해결합니다. 각 tool의 목적이 명확해져 Coordinator가 문서 분석과 웹 검색을 안정적으로 구분할 수 있습니다.

문제 8(시나리오: Multi-Agent Research System)

상황: 동료가 문서 분석 Agent가 Coordinator를 우회해 종합 Agent에게 결과를 직접 보내야 한다고 제안합니다. subagent 간 모든 통신에서 Coordinator를 중앙 허브로 유지하는 핵심 장점은 무엇인가요?

Coordinator를 중앙 허브로 유지하는 핵심 장점은 무엇인가요?

- A) Coordinator가 모든 상호작용을 관찰하고, 오류를 일관되게 처리하며, 각 subagent가 어떤 정보를 받을지 결정할 수 있다. **[정답]**
- B) Coordinator가 subagent에 대한 여러 요청을 batch 처리해 총 API 호출과 전체 지연 시간을 줄인다.
- C) Coordinator를 통한 routing은 직접 Agent 간 호출만으로는 어려운 자동 retry 로직을 제공한다.
- D) subagent는 격리된 memory를 사용하며, 직접 통신은 Coordinator만 수행할 수 있는 복잡한 직렬화를 요구한다.

A인 이유: Coordinator 패턴은 모든 상호작용에 대한 중앙 가시성, 시스템 전반의 일관된 오류 처리, 각 subagent가 받는 정보에 대한 세밀한 제어를 제공합니다. 이것이 hub-and-spoke 통신 구조의 주요 장점입니다.

문제 9(시나리오: Multi-Agent Research System)

상황: 웹 검색 subagent가 복잡한 주제를 조사하다 타임아웃됩니다. 이 실패 정보를 Coordinator에게 어떻게 돌려줄지 설계해야 합니다. 복구 판단에 가장 도움이 되는 오류 전파 접근 방식은 무엇인가요?

복구 판단에 가장 도움이 되는 오류 전파 접근 방식은 무엇인가요?

- A) 실패 유형, 실행된 쿼리, 부분 결과, 가능한 대안 접근법을 포함한 structured error context를 Coordinator에게 반환한다. **[정답]**
- B) subagent 내부에서 타임아웃을 잡고 빈 결과 집합을 성공으로 표시해 반환한다.
- C) subagent 내부에서 지수 백오프 자동 retry를 구현하고, retry를 모두 소진한 뒤 일반적인 "search unavailable" 상태만 반환한다.
- D) 타임아웃 예외를 최상위 핸들러에 직접 전파해 전체 research workflow를 종료한다.

A인 이유: 실패 유형, 실행된 쿼리, 부분 결과, 대안 접근법을 포함한 structured error context를 반환하면 Coordinator가 수정된 쿼리로 retry할지, 부분 결과로 계속할지 같은 복구 방향을 판단하는 데 필요한 정보를 얻습니다. 이는 조정 수준의 근거 있는 의사결정을 위해 필요한 context를 최대한 유지합니다.

문제 10(시나리오: Multi-Agent Research System)

상황: 시스템 설계에서 문서 분석 Agent가 URL로 문서를 다운로드할 수 있도록 범용 tool `fetch_url` 접근 권한을 부여했습니다. 프로덕션 로그는 이 Agent가 이제 임시 웹 검색을 수행하기 위해 검색 엔진 결과 페이지를 자주 다운로드한다는 것을 보여줍니다. 이 동작은 웹 검색 Agent를 통해 routing되어야 하며, 현재는 결과 불일치를 유발합니다. 가장 효과적인 수정안은 무엇인가요?

가장 효과적인 수정안은 무엇인가요?

- A) `fetch_url`을 URL이 문서 형식을 가리키는지 validation하는 `load_document` tool로 교체한다. **[정답]**
- B) 문서 분석 Agent에서 `fetch_url`을 제거하고 모든 URL 가져오기를 Coordinator를 통해 웹 검색 Agent로 routing한다.
- C) 알려진 검색 엔진 도메인에 대한 `fetch_url` 호출을 차단하고 다른 URL은 허용하는 필터링을 구현한다.
- D) 문서 분석 Agent prompt에 `fetch_url`은 검색이 아니라 문서 URL 다운로드에만 사용해야 한다는 지시를 추가한다.

A인 이유: 범용 `fetch_url`을 문서 형식 URL만 validation하는 전용 `load_document` tool로 바꾸면 interface 단계에서 기능 범위가 줄어듭니다. 최소 권한 원칙에 맞고, 원치 않는 검색 동작을 prompt 지시로 막는 대신 구조적

으로 차단합니다.

문제 11(시나리오: Multi-Agent Research System)

상황: 넓은 주제를 조사하는 동안 웹 검색 Agent와 문서 분석 Agent가 같은 하위 주제를 조사해 출력에 중복이 크게 발생합니다. token 사용량은 거의 두 배가 되지만 research 폭이나 깊이는 그만큼 늘지 않습니다. 이 문제를 가장 효과적으로 해결하려면 어떻게 해야 하나요?

이 문제를 가장 효과적으로 해결하려면 어떻게 해야 하나요?

- A) 두 Agent가 병렬로 끝나도록 허용한 뒤 Coordinator가 겹치는 결과를 중복 제거하고 종합 Agent에게 전달한다.
- B) Coordinator가 위임 전에 research 공간을 명시적으로 분할해 각 Agent에 서로 다른 하위 주제나 source 유형을 할당한다. **[정답]**
- C) Agent들이 현재 집중 영역을 기록해 다른 Agent가 실행 중 동적으로 중복을 피할 수 있도록 공유 상태 메커니즘을 구현한다.
- D) 웹 검색이 완료된 뒤 문서 분석을 실행하는 순차 실행으로 전환하고, 웹 검색 결과를 context로 사용해 중복을 피한다.

B인 이유: Coordinator가 위임 전에 research 공간을 명시적으로 분할하는 것이 가장 효과적입니다. 작업이 시작되기 전에 모호한 작업 경계라는 근본 원인을 해결하기 때문입니다. 병렬성을 보존하면서 중복 작업과 token 낭비를 방지합니다.

문제 12(시나리오: Multi-Agent Research System)

상황: research 중 웹 검색 subagent가 세 source 범주를 조회했으며 결과가 서로 다릅니다. 학술 데이터베이스는 관련 논문 15편을 반환하고, 산업 보고서는 "0 results"를 반환하며, 특허 데이터베이스는 "Connection timeout"을 반환합니다. Coordinator로의 오류 전파를 설계할 때 어떤 접근 방식이 복구 결정을 가장 잘 돕나요?

어떤 접근 방식이 복구 결정을 가장 잘 돕나요?

- A) 결과를 하나의 성공률 지표(예: "67% source coverage")로 집계하고 자세한 로그는 요청 시 제공한다.
- B) "timeout"과 "0 results"를 모두 Coordinator 개입이 필요한 실패로 보고한다.
- C) 일시적 실패를 내부적으로 retry하고 지속적인 오류만 보고한다.
- D) retry 결정이 필요한 접속 실패(timeout)와 성공한 쿼리를 나타내는 유효한 빈 결과("0 results")를 구분한다. **[정답]**

D인 이유: 타임아웃(접근 실패)과 "0 results"(유효한 빈 결과)는 의미적으로 다른 결과이며 서로 다른 대응이 필요합니다. 이를 구분하면 Coordinator가 특허 데이터베이스는 retry하고, 산업 보고서의 "0 results"는 유효하고 의미 있는 결과로 받아들일 수 있습니다.

문제 13(시나리오: Multi-Agent Research System)

상황: 프로덕션 모니터링에서 종합 품질이 일관되지 않음을 확인했습니다. 집계 결과가 약 75K token일 때, 종합 Agent는 첫 15K token(웹 검색 헤드라인/스니펫)과 마지막 10K token(문서 분석 결론)의 정보는 안정적으로 citation으로 뒷받침하지만, 중간 50K token의 중요한 발견은 research 질문에 직접 답하더라도 자주 놓칩니다. 집계 입력을 어떻게 재구성해야 하나요?

집계 입력을 어떻게 재구성해야 하나요?

- A) 모델의 안정적 처리 범위 안에 콘텐츠를 유지하기 위해 모든 subagent 출력을 집계 전에 20K token 이하로 summary한다.
- B) subagent 결과를 종합 Agent에 점진적으로 스트리밍해 웹 검색 결과를 먼저 완전히 처리한 뒤 문서 분석 결과를 추가한다.
- C) 집계 입력 시작 부분에 핵심 발견을 summary로 앞에 모으고, 세부 결과를 명시적 섹션 제목으로 구성해 탐색을 쉽게 한다. **[정답]**
- D) research 작업마다 어느 subagent 결과가 먼저 나오는지 번갈아 batch하는 회전을 구현해 두 source가 시간에 따라 동일하게 첫 위치를 얻도록 한다.

C인 이유: 핵심 발견 summary를 시작 부분에 두면 primacy 효과를 활용해 중요한 정보가 가장 안정적으로 처리되는 위치에 놓입니다. 전체에 명시적 섹션 제목을 추가하면 모델이 중간 입력 콘텐츠를 탐색하고 주의를 기울이는 데 도움이 되어 "lost in the middle" 현상을 직접 완화합니다.

문제 14(시나리오: Multi-Agent Research System)

상황: 테스트에서 웹 검색 Agent 출력(페이지 콘텐츠 포함 85K token)과 문서 분석 Agent 출력(사고 과정 포함 70K token)을 합치면 155K token이지만, 종합 Agent는 50K token 미만 입력에서 가장 잘 동작합니다. 가장 효과적인 해결책은 무엇인가요?

가장 효과적인 해결책은 무엇인가요?

- A) upstream Agent가 장황한 콘텐츠와 추론 대신 structured data(핵심 사실, citation, 관련성 점수)를 반환하도록 수정한다. **[정답]**
- B) 종합에 전달하기 전에 발견 사항을 압축하는 중간 summary Agent를 추가한다.
- C) 종합 Agent가 호출 사이에 상태를 유지하면서 발견 사항을 순차 batch로 처리하게 한다.
- D) 발견 사항을 vector 데이터베이스에 저장하고 종합 Agent에게 작업 중 조회할 검색 tool을 제공한다.

A인 이유: upstream Agent가 structured data를 반환하도록 수정하면 필수 정보를 보존하면서 token량을 source에서 줄여 근본 원인을 해결합니다. 종합 단계 개선에 기여하지 않는 분량이 큰 페이지 콘텐츠와 추론 흔적을 전달하는 것을 피합니다.

문제 15(시나리오: Multi-Agent Research System)

상황: 테스트에서 종합 Agent가 결과를 병합하면서 특정 주장을 validation해야 하는 경우가 많다는 것을 관찰했습니다. 현재 validation이 필요하면 종합 Agent가 제어를 Coordinator에게 반환하고, Coordinator가 웹 검색 Agent를 호출한 뒤 결과와 함께 종합을 다시 호출합니다. 이로 인해 작업당 루프가 2-3회 추가되고 지연 시간이 40% 증가

합니다. 평가 결과 이러한 validation의 85%는 단순 사실 확인(날짜, 이름, 통계)이고 15%는 더 깊은 research가 필요합니다. 시스템 안정성을 보존하면서 overhead를 가장 효과적으로 줄이는 접근 방식은 무엇인가요?

가장 효과적인 접근 방법은 무엇인가요?

- A) 종합 Agent에 모든 웹 검색 tool 접근 권한을 부여해 Coordinator 루프 없이 모든 validation 필요를 직접 처리하게 한다.
- B) 종합 Agent가 모든 validation 필요 사항을 모아 마지막에 batch로 Coordinator에게 반환하면, Coordinator가 이를 한 번에 웹 검색 Agent에게 보낸다.
- C) 웹 검색 Agent가 초기 research 중 종합에서 validation이 필요할 것을 예상해 각 source 주변의 추가 context를 사전에 cache한다.
- D) 단순 확인을 위한 제한 범위 `verify_fact` tool을 종합 Agent에게 제공하고, 복잡한 validation은 Coordinator를 통해 웹 검색 Agent로 routing한다. **[정답]**

D인 이유: 제한 범위 사실 validation tool은 종합 Agent가 단순 확인의 85%를 직접 처리하게 해 대부분의 추가 루프를 없애면서, 복잡한 validation 15%에는 Coordinator 위임 경로를 유지합니다. 이는 최소 권한 원칙을 지키면서 지연 시간을 크게 줄입니다.

시나리오: 지속적 통합을 위한 Claude Code

문제 16(시나리오: 지속적 통합을 위한 Claude Code)

상황: CI pipeline이 코드 리뷰를 위한 프로젝트 context를 제공하기 위해 CLAUDE.md를 사용하며, Claude Code CLI를 `--print` 모드로 실행합니다. 개발자들은 대체로 리뷰가 실질적이라고 느낍니다. 하지만 발견 사항을 workflow에 통합하기 어렵다고 보고합니다. Claude가 서술형 문단을 출력해 PR 댓글로 수동 복사해야 하기 때문입니다. 팀은 각 발견 사항을 코드의 관련 위치에 별도 인라인 PR 댓글로 자동 게시하고 싶어 하며, 이를 위해 파일 경로, 줄 번호, severity 수준, 제안 수정이 포함된 structured data가 필요합니다. 가장 효과적인 접근 방법은 무엇인가요?

가장 효과적인 접근 방법은 무엇인가요?

- A) CLAUDE.md에 "Output Format for Review" 섹션과 structured 발견 예시를 추가해 Claude가 프로젝트 context에서 기대 형식을 학습하게 한다.
- B) CLI 플래그 `--output-format json`과 `--json-schema`를 사용해 structured 발견을 강제한 뒤, 출력을 parsing해 GitHub API로 인라인 댓글을 게시한다. **[정답]**
- C) 리뷰 prompt에 각 발견 사항이 `[FILE:path] [LINE:n] [SEVERITY:level] ...` 같은 parsing 가능한 template을 따르도록 명시적 형식 지시를 포함한다.
- D) 서술형 리뷰 형식은 유지하되, Claude를 사용해 발견 사항의 structured JSON summary를 생성하는 summary 단계를 추가한다.

B인 이유: `--json-schema`와 함께 `--output-format json`을 사용하면 CLI 수준에서 structured output을 강제해, GitHub API로 인라인 PR 댓글을 게시하기 위해 안정적으로 parsing 가능한 필수 필드(파일 경로, 줄 번호, severity, 제안 수정)를 갖춘 올바른 JSON을 보장합니다. 이는 structured output에 맞게 설계된 내장 CLI 기능을 활용합니다.

문제 17(시나리오: 지속적 통합을 위한 Claude Code)

상황: 팀은 코드 제안 생성에 Claude Code를 사용하지만, 한 가지 패턴을 발견했습니다. 엣지 케이스를 깨뜨리는 성능 최적화, 예상치 않게 동작을 바꾸는 정리 작업 같은 명확하지 않은 이슈는 다른 팀원이 PR을 리뷰할 때만 잡힙니다. 생성 중 Claude의 추론을 보면 이러한 사례를 고려했지만 자신의 접근이 옳다고 결론 내렸습니다. 이 자기 점검 한계의 근본 원인을 직접 다루는 접근 방식은 무엇인가요?

근본 원인을 직접 다루는 접근 방식은 무엇인가요?

- A) 생성자의 추론에 접근하지 않는 두 번째 독립 Claude Code instance를 실행해 변경 사항을 리뷰한다. **[정답]**
- B) 제안 생성 전에 더 철저히 숙고할 수 있도록 생성 단계에 extended thinking 모드를 활성화한다.
- C) 최종 출력 전에 Claude가 자신의 제안을 비판하도록 생성 prompt에 명시적 자기 리뷰 지시를 추가한다.
- D) 생성 중 기대 동작을 더 잘 이해하도록 전체 테스트 파일과 문서를 prompt context에 포함한다.

A인 이유: 생성자의 추론에 접근하지 않는 두 번째 독립 Claude Code instance는 확증 편향을 피함으로써 근본 원인을 직접 해결합니다. 이 "fresh eyes" 관점은 작성자가 합리화한 이슈를 다른 리뷰어가 잡아내는 사람 동료 리뷰와 유사합니다.

문제 18(시나리오: 지속적 통합을 위한 Claude Code)

상황: 코드 리뷰 컴포넌트는 반복적입니다. Claude가 변경된 파일을 분석한 뒤 최종 피드백 전에 context 이해를 위해 관련 파일(import, base class, tests)을 tool 호출로 요청할 수 있습니다. 애플리케이션은 Claude가 파일 내용을 요청할 수 있는 tool을 정의합니다. Claude는 tool을 호출하고 결과를 받은 뒤 분석을 계속합니다. API 비용 절감을 위해 batch 처리를 평가하고 있습니다. 이 workflow에서 batch 처리를 고려할 때 주요 기술적 한계는 무엇인가요?

주요 기술적 한계는 무엇인가요?

- A) batch 처리는 출력을 입력 요청에 다시 매핑하는 상관관계 ID를 포함하지 않는다.
- B) 비동기 모델은 요청 중간에 tool을 실행하고 Claude가 분석을 계속할 수 있도록 결과를 반환할 수 없다. **[정답]**
- C) Batch API는 요청 매개변수의 tool 정의를 지원하지 않는다.
- D) 최대 24시간의 batch 처리 지연이 풀 리퀘스트 피드백에는 너무 느리지만, 그 외에는 workflow가 작동한다.

B인 이유: "fire-and-forget" 비동기 Batch API 모델에는 요청 중 tool 호출을 가로채고, tool을 실행한 후 Claude가 분석을 계속하도록 결과를 반환하는 메커니즘이 없습니다. 이는 단일 논리 상호작용 안에서 여러 tool 요청/응답 라운드가 필요한 반복적 tool 호출 workflow와 근본적으로 맞지 않습니다.

문제 19(시나리오: 지속적 통합을 위한 Claude Code)

상황: CI/CD 시스템은 세 가지 Claude 기반 분석을 실행합니다. (1) 완료 전까지 병합을 차단하는 모든 PR의 빠른 스타일 검사, (2) 전체 코드베이스에 대한 포괄적인 주간 보안 감사, (3) 최근 변경된 module에 대한 야간 테스트 케이스 생성. Message Batches API는 50% 절감을 제공하지만 처리에 최대 24시간이 걸릴 수 있습니다. 허용 가능한 개발자 경험을 유지하면서 API 비용을 최적화하려고 합니다. 각 작업과 API 접근 방식을 올바르게 매칭한 조합은 무엇인가요?

올바른 조합은 무엇인가요?

- A) 50% 절감을 최대화하기 위해 세 작업 모두 Message Batches API를 사용하고, pipeline이 batch 완료를 polling하도록 구성한다.
- B) PR 스타일 검사는 동기 호출을 사용하고, 주간 보안 감사와 야간 테스트 생성에는 Message Batches API를 사용한다. **[정답]**
- C) 세 작업 모두 동기 호출을 사용해 일관된 응답 시간을 유지하고, 워크로드 전반의 비용 절감은 prompt caching에 의존한다.
- D) PR 스타일 검사와 야간 테스트 생성은 동기 호출을 사용하고, 주간 보안 감사에만 Message Batches API를 사용한다.

B인 이유: PR 스타일 검사는 개발자를 차단하며 동기 호출로 즉시 응답해야 합니다. 반면 주간 보안 감사와 야간 테스트 생성은 최대 24시간 batch window를 허용할 수 있는 마감이 비교적 유연한 예약 작업이므로 둘 다 50% 절감을 얻을 수 있습니다.

문제 20(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 리뷰가 실제 이슈를 찾지만 개발자들은 피드백이 실행 가능하지 않다고 보고합니다. 발견 사항에는 정확히 무엇을 바꿔야 하는지 지정하지 않은 채 "복잡한 티켓 routing 로직"이나 "잠재적 null pointer" 같은 문구가 포함됩니다. "항상 구체적 수정 제안을 포함하라" 같은 상세 지시를 추가해도 모델 출력은 여전히 어떤 때는 상세하지만 어떤 때는 모호해 일관성이 없습니다. 어떤 prompting 기법이 일관되게 실행 가능한 피드백을 가장 안정적으로 생성하나요?

가장 신뢰할 수 있는 prompting 기법은 무엇인가요?

- A) 피드백 형식의 각 부분(위치, 이슈, severity, 제안 수정)에 대해 더 명시적인 요구사항으로 지시를 더 다듬는다.
- B) 모델이 구체적 수정을 제안할 충분한 정보를 갖도록 더 많은 주변 코드베이스를 context window에 포함한다.
- C) 한 prompt는 이슈를 식별하고 두 번째 prompt는 수정을 생성하는 2패스 접근으로 전문화를 허용한다.
- D) 식별된 이슈, 코드 위치, 구체적 수정 제안 등 정확히 필요한 형식을 보여주는 few-shot 예시 3-4개를 추가한다. **[정답]**

D인 이유: Few-shot 예시는 지시만으로 출력이 가변적일 때 출력 형식을 일관되게 만드는 가장 효과적인 기법입니다. 원하는 구조(이슈, 위치, 구체적 수정)를 보여주는 예시 3-4개는 모델이 따를 구체적 패턴을 제공하며, 추상적인 지시만 쓰는 것보다 안정적입니다.

문제 21(시나리오: 지속적 통합을 위한 Claude Code)

상황: CI pipeline에는 두 가지 Claude 기반 코드 리뷰 모드가 있습니다. 완료될 때까지 PR 병합을 차단하는 병합 전 커밋 hook과, 밤새 실행되어 batch 완료를 polling하고 PR에 자세한 제안을 게시하는 "deep analysis"입니다. 50% 절감을 제공하지만 polling이 필요하고 최대 24시간 걸릴 수 있는 Message Batches API를 사용해 API 비용을 줄이고 싶습니다. 어떤 모드가 batch 처리를 사용해야 하나요?

어떤 모드가 batch 처리를 사용해야 하나요?

- A) 병합 전 커밋 hook만.
- B) deep analysis만. **[정답]**

- C) 두 모드 모두.
- D) 어느 모드도 아님.

B인 이유: Deep analysis는 이미 밤새 실행되고 지연을 허용하고 결과 게시 전에 polling하는 구조이므로 batch 처리에 이상적인 후보입니다. 이는 Message Batches API의 비동기 polling 기반 architecture와 잘 맞고 50% 절감을 제공합니다.

문제 22(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 리뷰가 댓글과 docstring을 분석합니다. 현재 prompt는 Claude에게 "댓글이 정확하고 최신인지 확인하라"고 지시합니다. 발견 사항은 TODO 표시나 단순 설명 같은 허용 가능한 패턴을 자주 문제로 표시하면서, 코드가 더 이상 구현하지 않는 동작을 설명하는 댓글은 놓칩니다. 이 일관되지 않은 분석의 근본 원인을 해결하려면 무엇을 바꿔야 하나요?

근본 원인을 해결하려면 무엇을 바꿔야 하나요?

- A) Claude가 최근 코드 변경보다 오래된 댓글을 식별할 수 있도록 `git blame` 데이터를 포함한다.
- B) 모델이 코드베이스의 유사 패턴을 인식하도록 오해를 부르는 댓글의 few-shot 예시를 추가한다.
- C) 노이즈를 줄이기 위해 분석 전에 TODO, FIXME, 설명형 댓글 패턴을 필터링한다.
- D) 명시적 기준을 지정한다. 댓글이 주장하는 동작이 코드의 실제 동작과 모순될 때만 표시한다. **[정답]**

D인 이유: 명시적 기준, 즉 댓글이 주장하는 동작이 실제 코드 동작과 모순될 때만 표시한다는 기준은 모호한 지시를 구체적인 판정 기준으로 바꿔 근본 원인을 해결합니다. 이는 허용 가능한 패턴에 대한 false positive과 실제로 잘못된 댓글을 놓치는 문제도 줄입니다.

문제 23(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 코드 리뷰 시스템이 일관되지 않은 severity 평가를 보입니다. null pointer 위험 같은 유사 이슈가 어떤 PR에서는 "critical"로, 다른 PR에서는 "medium"으로 평가됩니다. 개발자 설문에서도 신뢰가 떨어졌다는 결과가 나왔습니다. 많은 개발자가 "절반은 틀렸다"고 느끼며 발견 사항을 읽지 않고 무시하기 시작합니다. false positive 비율이 높은 범주는 정확한 범주에 대한 신뢰까지 훼손합니다. 개발자 신뢰를 회복하면서 시스템을 개선하는 가장 좋은 접근은 무엇인가요?

개발자 신뢰를 가장 잘 회복하는 접근은 무엇인가요?

- A) false positive 비율이 높은 범주(스타일, 명명, 문서화)를 일시적으로 비활성화하고, prompt를 개선하는 동안 정확도가 높은 범주만 유지한다. **[정답]**
- B) 모든 범주를 활성화하되 각 발견 사항에 confidence 점수를 표시해 개발자가 조사할 대상을 결정하게 한다.
- C) 모든 범주를 활성화한 채 다음 몇 주 동안 각 범주의 accuracy 개선을 위해 few-shot 예시를 추가한다.
- D) 전체 false positive 비율을 낮추기 위해 모든 범주에 균일한 엄격도 완화를 적용한다.

A인 이유: false positive 비율이 높은 범주를 일시적으로 비활성화하면 개발자가 모든 것을 무시하게 만드는 노이즈를 제거해 신뢰 하락을 즉시 멈출 수 있습니다. 동시에 보안과 정확성 같은 정확도가 높은 범주의 가치를 유지합니다. 또한 문제 범주를 다시 활성화하기 전에 prompt를 개선할 여지를 만듭니다.

문제 24(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 리뷰가 각 PR에 대해 테스트 케이스 제안을 생성합니다. 강의 완료 추적을 추가하는 PR을 리뷰할 때 Claude는 테스트 케이스 10개를 제안했지만, 개발자 피드백에 따르면 6개는 기존 테스트 스위트가 이미 다루는 중복 시나리오였습니다. 어떤 변경이 중복 제안을 가장 효과적으로 줄이나요?

가장 효과적인 변경 사항은 무엇인가요?

- A) Claude가 이미 다뤄진 시나리오를 판단할 수 있도록 기존 테스트 파일을 context에 포함한다. **[정답]**
- B) Claude가 가장 가치 있는 케이스를 먼저 우선순위화한다고 가정하고 제안 개수를 10개에서 5개로 줄인다.
- C) 성공 경로가 아니라 엣지 케이스와 오류 조건에만 집중하라는 지시를 추가한다.
- D) 설명이 키워드 겹침으로 기존 테스트 이름과 일치하는 제안을 필터링하는 후처리를 구현한다.

A인 이유: 기존 테스트 파일을 포함하면 중복의 근본 원인을 해결합니다. Claude는 어떤 테스트가 이미 존재하는지 알아야만 이미 커버된 시나리오 제안을 피할 수 있습니다. 이를 통해 Claude는 실제로 새롭고 가치 있는 테스트를 제안하는 데 필요한 정보를 얻습니다.

문제 25(시나리오: 지속적 통합을 위한 Claude Code)

상황: 초기 자동 리뷰가 12개 발견 사항을 식별한 뒤 개발자가 이슈를 해결하기 위해 새 커밋을 푸시합니다. 리뷰를 다시 실행하면 8개 발견 사항이 나오지만, 개발자들은 새 커밋에서 이미 수정된 코드에 대한 이전 댓글 5개가 중복된다고 보고합니다. 리뷰의 철저함은 유지하면서 이 중복 피드백을 제거하는 가장 효과적인 방법은 무엇인가요?

중복 피드백을 제거하는 가장 효과적인 방법은 무엇인가요?

- A) 중간 커밋 리뷰를 건너뛰고 PR 생성 시점과 최종 병합 전 상태에서만 리뷰를 실행한다.
- B) 게시 전에 파일 경로와 이슈 설명으로 이전 발견 사항과 일치하는 항목을 제거하는 후처리 필터를 추가한다.
- C) 리뷰 범위를 가장 최근 push에서 변경된 파일로 제한하고 이전 커밋의 파일은 제외한다.
- D) 이전 리뷰 발견 사항을 context에 포함하고 Claude에게 새 이슈나 아직 해결되지 않은 이슈만 보고하도록 지시한다. **[정답]**

D인 이유: 이전 리뷰 발견 사항을 context에 포함하면 Claude가 새 문제와 최근 커밋에서 이미 해결된 문제를 구분할 수 있습니다. 이렇게 하면 리뷰의 철저함은 유지하면서도 Claude가 새 이슈와 해결된 이슈를 구분해, 수정된 코드에 대한 중복 피드백을 피할 수 있습니다.

문제 26(시나리오: 지속적 통합을 위한 Claude Code)

상황: pipeline 스크립트가 `claude "Analyze this pull request for security issues"`를 실행하지만 작업이 끝나지 않고 멈춰 있습니다. 로그는 Claude Code가 대화형 입력을 기다리는 상태입니다. 자동화 pipeline에서 Claude Code를 실행하는 올바른 접근 방식은 무엇인가요?

올바른 접근 방식은 무엇인가요?

- A) `--batch` 플래그 추가: `claude --batch "Analyze this pull request for security issues"`.
- B) `-p` 플래그 추가: `claude -p "Analyze this pull request for security issues"`. **[정답]**

- C) stdin을 `/dev/null`에서 리디렉션: `claude "Analyze this pull request for security issues" < /dev/null`.
- D) 명령 실행 전에 환경 변수 `CLAUDE_HEADLESS=true` 설정.

B인 이유: `-p`(또는 `--print`) 플래그는 Claude Code를 비대화형으로 실행하는 문서화된 방법입니다. prompt를 처리하고 결과를 stdout에 출력한 뒤 사용자 입력을 기다리지 않고 종료합니다. CI/CD pipeline에 적합합니다.

문제 27(시나리오: 지속적 통합을 위한 Claude Code)

상황: 풀 리퀘스트가 재고 추적 module의 14개 파일을 변경합니다. 모든 파일을 함께 분석하는 단일 패스 리뷰는 일부 파일에는 자세한 피드백을 주지만 다른 파일에는 얕은 댓글을 달고, 명백한 버그를 놓치며, 한 파일에서는 패턴을 문제로 표시하지만 같은 PR의 동일 코드에서는 승인하는 모순된 피드백을 생성합니다. 리뷰를 어떻게 재구성해야 하나요?

리뷰를 어떻게 재구성해야 하나요?

- A) 전체 PR 리뷰 패스를 세 번 독립적으로 실행하고 세 번 중 최소 두 번 나타나는 이슈만 표시한다.
- B) 집중된 패스로 분할한다. 각 파일을 개별적으로 로컬 이슈에 대해 리뷰한 뒤, 파일 간 데이터 흐름을 검토하는 별도 통합 지향 패스를 실행한다. **[정답]**
- C) 자동 리뷰를 실행하기 전에 개발자에게 큰 PR을 3-4개 파일의 더 작은 제출로 나누도록 요구한다.
- D) 더 큰 context window가 있는 큰 모델로 전환해 14개 파일 모두에 한 번에 충분히 주의를 기울일 수 있게 한다.

B인 이유: 집중된 파일별 패스는 주의 분산이라는 근본 원인을 다루어 파일마다 비슷한 깊이로 리뷰할 수 있고 로컬 이슈도 더 안정적으로 찾을 수 있습니다. 이후 별도 통합 지향 패스가 의존성과 데이터 흐름 상호작용 같은 파일 간 문제를 다룹니다.

문제 28(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 코드 리뷰는 풀 리퀘스트당 평균 15개 발견 사항을 생성하며, 개발자는 40% false positive 비율을 보고합니다. 병목은 조사 시간입니다. 개발자는 수정할지 무시할지 결정하기 전에 각 발견 사항을 클릭해 Claude의 근거를 읽어야 합니다. CLAUDE.md에는 이미 허용 가능한 패턴에 대한 포괄적 규칙이 포함되어 있고, 이해관계자들은 개발자가 보기 전에 발견 사항을 필터링하는 모든 접근을 거부했습니다. 조사 시간을 가장 잘 줄이는 변경은 무엇인가요?

조사 시간을 가장 잘 줄이는 변경은 무엇인가요?

- A) Claude가 각 발견 사항에 근거와 confidence 추정치를 직접 포함하도록 요구한다. **[정답]**
- B) 발견 패턴을 분석하고 과거 false positive 시그니처와 일치하는 항목을 자동 억제하는 후처리기를 추가한다.
- C) 발견 사항을 "차단 이슈"와 "제안"으로 classification하고 수준별로 다른 리뷰 요구사항을 둔다.
- D) Claude가 높은 confidence 발견만 보여주도록 구성해 개발자가 보기 전에 불확실한 플래그를 필터링한다.

A인 이유: 각 발견 사항에 근거와 confidence를 직접 포함하면 개발자가 각 항목을 따로 열어보지 않고도 빠르게 triage할 수 있어 조사 시간이 줄어듭니다. 모든 발견 사항은 계속 보이므로 발견 사항을 숨기지 않는다는 제약도 지키면서 개발자 의사결정을 빠르게 합니다.

문제 29(시나리오: 지속적 통합을 위한 Claude Code)

상황: 자동 코드 리뷰 분석 결과 발견 범주별 false positive 비율 차이가 큽니다. 보안/정확성 발견은 false positive 8%, 성능 발견은 18%, 스타일/명명 발견은 52%, 문서화 발견은 48%입니다. 개발자 설문에서도 신뢰가 떨어졌다는 결과가 나왔습니다. 많은 개발자가 "절반은 틀렸다"고 느끼며 발견 사항을 읽지 않고 무시하기 시작합니다. false positive 비율이 높은 범주는 정확한 범주에 대한 신뢰까지 훼손합니다. 개발자 신뢰를 회복하면서 시스템을 개선하는 가장 좋은 접근은 무엇인가요?

개발자 신뢰를 가장 잘 회복하는 접근은 무엇인가요?

- A) false positive 비율이 높은 범주(스타일, 명명, 문서화)를 일시적으로 비활성화하고, prompt를 개선하는 동안 정확도가 높은 범주만 유지한다. **[정답]**
- B) 모든 범주를 활성화하되 각 발견 사항에 confidence 점수를 표시해 개발자가 조사할 대상을 결정하게 한다.
- C) 모든 범주를 활성화한 채 다음 몇 주 동안 각 범주의 accuracy 개선을 위해 few-shot 예시를 추가한다.
- D) 전체 false positive 비율을 낮추기 위해 모든 범주에 균일한 엄격도 완화를 적용한다.

A인 이유: false positive 비율이 높은 범주를 일시적으로 비활성화하면 개발자가 모든 것을 무시하게 만드는 노이즈를 제거해 신뢰 하락을 즉시 멈출 수 있습니다. 동시에 보안과 정확성 같은 정확도가 높은 범주의 가치를 유지합니다. 또한 문제 범주를 다시 활성화하기 전에 prompt를 개선할 여지를 만듭니다.

문제 30(시나리오: 지속적 통합을 위한 Claude Code)

상황: 팀은 자동 분석 API 비용을 줄이고 싶어 합니다. 현재 동기 Claude 호출은 두 workflow를 지원합니다. (1) 개발자가 병합하기 전에 반드시 완료되어야 하는 차단형 병합 전 검사, (2) 다음 날 아침에 검토할 수 있도록 밤새 생성되는 기술 부채 보고서. 관리자는 50% 절감을 위해 둘 다 Message Batches API로 옮기자고 제안합니다. 이 제안을 어떻게 평가해야 하나요?

이 제안을 어떻게 평가해야 하나요?

- A) 둘 다 batch 처리로 옮기고, batch가 너무 오래 걸리면 동기 호출로 fallback한다.
- B) 두 workflow 모두 batch 처리로 옮기고 완료 확인을 위해 상태를 polling한다.
- C) 기술 부채 보고서에만 batch 처리를 사용하고, 병합 전 검사는 동기 호출을 유지한다. **[정답]**
- D) batch 결과 순서 문제를 피하기 위해 두 workflow 모두 동기 호출을 유지한다.

C인 이유: Message Batches API 처리는 최대 24시간 걸릴 수 있고 지연 시간 SLA가 없어, 야간 기술 부채 보고서에는 허용 가능하지만 개발자가 기다리는 차단형 병합 전 검사에는 허용되지 않습니다. 핵심은 각 workflow의 지연 허용 범위에 맞는 API를 고르는 것입니다.

시나리오: Claude Code를 사용한 코드 생성

문제 31(시나리오: Claude Code를 사용한 코드 생성)

상황: Claude Code에 API 응답을 내부 normalization 형식으로 변환하는 함수를 구현해 달라고 요청했습니다. 두 번의 반복 후에도 출력 구조가 기대와 맞지 않습니다. 일부 필드는 다르게 중첩되고 타임스탬프 형식도 잘못되어 있

습니다. 요구사항은 산문으로 설명했지만 Claude가 매번 다르게 해석합니다.

다음 반복에 가장 효과적인 접근 방법은 무엇인가요?

- A) 기대 출력 구조를 설명하는 JSON schema를 작성하고 각 반복 후 Claude의 출력을 validation한다.
- B) 대표 API 응답에 대해 기대 변환을 보여주는 구체적인 입력-출력 예시 2-3개를 제공한다. **[정답]**
- C) 정확한 필드 매핑, 중첩 규칙, 타임스탬프 형식 문자열을 지정해 요구사항을 더 기술적으로 정확하게 다시 작성한다.
- D) 해석이 갈라지는 지점을 파악하기 위해 Claude에게 현재 요구사항 이해를 설명하게 한다.

B인 이유: 구체적인 입력-출력 예시는 Claude에게 정확히 기대되는 변환 결과를 보여줌으로써 글로만 설명했을 때 생기는 모호함을 줄입니다. 이는 필드 중첩과 타임스탬프 형식에 대한 명확한 패턴을 제공하므로, 산문 요구사항을 다르게 해석하는 문제를 줄입니다.

문제 32(시나리오: Claude Code를 사용한 코드 생성)

상황: 새 알림 채널로 Slack을 추가해야 합니다. 기존 코드베이스에는 email, SMS, push 채널에 대한 명확한 기존 패턴이 있습니다. 하지만 Slack API는 근본적으로 다른 통합 접근을 제공합니다. incoming webhook(단순, 단방향), bot token(전달 확인과 프로그래밍 제어 지원), Slack App(양방향 이벤트, 워크스페이스 승인 필요)입니다. 작업은 고급 기능(전달 추적 등) 요구 없이 "Slack 지원 추가"라고만 말합니다.

이 작업에 어떻게 접근해야 하나요?

- A) 기존 단방향 알림 패턴에 맞게 incoming webhook을 사용해 직접 실행 모드로 시작한다.
- B) 계획 모드로 전환해 통합 옵션과 architecture 함의를 탐색한 뒤 구현 전 권장안을 제시한다. **[정답]**
- C) 기존 패턴을 사용해 Slack 채널 클래스를 스캐폴딩하는 직접 실행 모드로 시작하고 통합 방식 결정은 미룬다.
- D) 전달 확인이 가능하도록 bot-token 접근 방식으로 직접 실행 모드를 시작한다.

B인 이유: Slack 통합에는 architecture에 미치는 영향이 서로 다른 여러 선택지가 있고 요구사항도 모호합니다. 계획 모드에서는 webhook, bot token, Slack App 사이의 trade-off를 먼저 평가하고 구현 전에 방향을 맞출 수 있습니다.

문제 33(시나리오: Claude Code를 사용한 코드 생성)

상황: CLAUDE.md 파일이 400줄 이상으로 커져 코딩 표준, 테스트 규칙, 상세 PR 리뷰 체크리스트, 배포 지시사항, 데이터베이스 migration 절차가 포함되어 있습니다. Claude가 코딩 표준과 테스트 규칙은 항상 따르되, PR 리뷰, 배포, migration 지침은 해당 작업을 할 때만 적용하기를 원합니다.

가장 효과적인 재구성 접근 방식은 무엇인가요?

- A) 모든 지침을 workflow 유형별 별도 Skills 파일로 옮기고 CLAUDE.md에는 간단한 프로젝트 설명만 남긴다.
- B) 모든 것을 CLAUDE.md에 유지하되 `@import` 문법을 사용해 범주별로 별도 관리 파일로 구성한다.
- C) CLAUDE.md를 `.claude/rules/` 아래 파일로 나누고 path-bound glob 패턴을 사용해 관련 파일 유형에만 각 규칙을 로드한다.
- D) 범용 표준은 CLAUDE.md에 유지하고, workflow별 지침(PR 리뷰, 배포, migration)은 트리거 키워드가 있는 Skills로 만든다. **[정답]**

D인 이유: CLAUDE.md 내용은 모든 session에 로드되어 코딩 표준과 테스트 규칙이 항상 적용되게 합니다. 반면 Skills는 Claude가 트리거 키워드를 감지할 때 on-demand로 호출되므로 PR 리뷰, 배포, migration 같은 workflow 별 지침을 넣기에 적합합니다.

문제 34(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀의 모놀리식 애플리케이션을 마이크로서비스로 재구성하는 작업을 맡았습니다. 이는 수십 개 파일에 걸친 변경에 영향을 주며 서비스 경계와 module 의존성에 대한 결정이 필요합니다.

어떤 접근 방식을 선택해야 하나요?

- A) 변경하기 전에 계획 모드로 전환해 코드베이스를 탐색하고, 의존성을 이해하며, 구현 접근 방식을 설계한다. **[정답]**
- B) 직접 실행 모드로 시작하고 구현 중 예상치 못한 복잡성을 만나면 그때 계획으로 전환한다.
- C) 직접 실행 모드로 시작해 점진적으로 변경하며 구현 과정에서 자연스러운 서비스 경계가 드러나게 한다.
- D) 각 서비스 구조를 지정하는 상세한 사전 지시와 함께 직접 실행을 사용한다.

A인 이유: 계획 모드는 모놀리스 분리 같은 복잡한 architecture 재구성에 적합한 전략입니다. 많은 파일에 걸친 비용이 큰 변경을 확정하기 전에 안전하게 탐색하고 경계를 근거 있게 판단할 수 있습니다.

문제 35(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀이 깊은 코드 분석(의존성 스캔, 테스트 커버리지 수, 코드 품질 지표)을 수행하는 `/analyze-codebase` Skill을 만들었습니다. 명령 실행 후 팀원들은 Claude가 session에서 덜 반응하고 원래 작업의 context를 잃는다고 보고합니다.

전체 분석 기능을 유지하면서 이를 가장 효과적으로 고치려면 어떻게 해야 하나요?

- A) Skill frontmatter에 `context: fork`를 추가해 격리된 subagent context에서 분석을 실행한다. **[정답]**
- B) 분석에 더 빠르고 저렴한 모델을 사용하기 위해 frontmatter에 `model: haiku`를 추가한다.
- C) Skill을 출력이 적은 세 개의 작은 Skill로 나눈다.
- D) 표시 전에 모든 결과를 짧은 summary로 압축하라는 지시를 Skill에 추가한다.

A인 이유: `context: fork`는 분석을 격리된 subagent context에서 실행해 큰 출력이 메인 session context window를 불필요하게 채우지 않게 하고 Claude가 원래 작업의 context를 잃지 않게 합니다. 전체 분석 기능을 보존하면서 메인 session의 반응성을 유지합니다.

문제 36(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀은 `.claude/skills/commit/SKILL.md`의 `/commit` Skill을 사용합니다. 한 개발자가 팀원에게 영향을 주지 않고 개인 workflow(다른 커밋 메시지 형식, 추가 검사)에 맞게 이를 사용자 지정하고 싶어 합니다.

무엇을 권장하나요?

- A) `~/.claude/skills/` 아래 `/my-commit` 같은 다른 이름으로 개인 버전을 만든다. **[정답]**

- B) 프로젝트 Skill frontmatter에 사용자 이름 기반 조건부 로직을 추가한다.
- C) 같은 이름으로 `~/.claude/skills/commit/SKILL.md`에 개인 버전을 만든다.
- D) 개인 Skill frontmatter에 `override: true`를 설정해 프로젝트 버전보다 우선하게 한다.

A인 이유: 개인 Skill은 같은 이름의 프로젝트 Skill보다 우선하므로, `commit`이라는 이름을 재사용하면 이 개발자에게만 팀의 Skill이 조용히 가려집니다. 팀이 `/commit`을 개선해도 업데이트를 받지 못하게 되고, 같은 명령으로 다른 Skill을 실행하고 있다는 사실을 스스로 기억해야 합니다. 개인 버전에 `/my-commit`이라는 이름을 붙이면 이런 충돌을 완전히 피할 수 있습니다. 개발자는 팀이 관리하는 `/commit`을 계속 사용하면서, 개인 workflow를 위한 별도의 명확히 구분된 Skill을 갖게 되어 둘을 혼동하거나 팀의 업데이트를 놓칠 위험이 없습니다.

문제 37(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀은 몇 달 동안 Claude Code를 사용해 왔습니다. 최근 세 명의 개발자는 Claude가 "항상 포괄적인 오류 처리를 포함하라"는 지침을 따른다고 보고하지만, 새로 합류한 네 번째 개발자는 Claude가 이를 따르지 않는다고 말합니다. 네 명 모두 같은 저장소에서 최신 코드를 사용합니다.

가장 가능성 높은 원인과 수정은 무엇인가요?

- A) 지침이 프로젝트 `.claude/CLAUDE.md`가 아니라 기존 개발자들의 사용자 수준 `~/.claude/CLAUDE.md` 파일에 있다. 모든 팀원이 받도록 지시를 프로젝트 수준 파일로 옮긴다. **[정답]**
- B) 새 개발자의 `~/.claude/CLAUDE.md`에 프로젝트 설정을 override하는 충돌 지시가 있으며, 충돌 섹션을 삭제해야 한다.
- C) Claude Code는 사용자별 선호를 시간이 지나며 학습하므로, 새 개발자는 Claude가 "기억"할 때까지 요구사항을 반복해야 한다.
- D) Claude Code는 처음 읽은 뒤 CLAUDE.md를 cache하며 기존 개발자들은 cache된 버전을 사용한다. 모두 Claude Code cache를 지워야 한다.

A인 이유: 지침이 프로젝트 수준 `.claude/CLAUDE.md`가 아니라 기존 개발자들의 사용자 수준 구성에만 추가되었다면 새 팀원은 이를 받지 못합니다. 프로젝트 수준 구성으로 옮기면 현재와 미래의 모든 팀원이 자동으로 해당 지침을 받습니다.

문제 38(시나리오: Claude Code를 사용한 코드 생성)

상황: 새 API endpoint를 생성할 때 context로 전체 endpoint 구현 예시 2-3개를 포함하면 일관성이 크게 향상된다는 것을 발견했습니다. 하지만 이 context는 새 endpoint를 만들 때만 유용하며, API 디렉터리에서 디버깅, 코드 리뷰, 기타 작업을 할 때는 유용하지 않습니다.

가장 효과적인 구성 접근 방식은 무엇인가요?

- A) endpoint 예시와 패턴 문서를 프로젝트 CLAUDE.md에 추가해 항상 사용할 수 있게 한다.
- B) 모든 생성 요청마다 코드를 prompt에 복사해 endpoint 예시를 수동 참조한다.
- C) API 디렉터리에서 작업할 때 활성화되는 `.claude/rules/api/`의 경로별 규칙에 endpoint 예시를 포함한다.
- D) endpoint 예시를 참조하고 패턴 준수 지시를 포함하는 Skill을 만들고, 슬래시 명령으로 on-demand 호출한다. **[정답]**

D인 이유: on-demand로 호출되는 Skill은 새 endpoint를 생성할 때만 예시 context를 로드하고 디버깅이나 리뷰 같은 무관한 작업 중에는 로드하지 않습니다. 이렇게 하면 메인 context는 깔끔하게 유지하고, 새 endpoint 생성에 필요한 예시 context만 필요한 순간에 로드할 수 있습니다.

문제 39(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀이 데이터베이스 migration 파일을 생성하는 `/migration` Skill을 만들었습니다. 이 Skill은 `$ARGUMENTS`로 migration 이름을 받습니다. 프로덕션에서 세 가지 문제가 관찰됩니다. (1) 개발자가 종종 인수 없이 Skill을 실행해 파일 이름이 부적절해짐, (2) Skill이 때때로 관련 없는 이전 대화의 데이터베이스 schema 세부사항을 사용함, (3) Skill이 넓은 tool 접근 권한을 가져 개발자가 실수로 파괴적 테스트 정리를 실행함.

세 문제를 모두 해결하는 구성 접근 방식은 무엇인가요?

- A) `$ARGUMENTS` 대신 위치 매개변수 `$1`과 `$2`를 사용해 특정 입력을 강제하고, context 제어를 위해 `@` 문법으로 명시적 schema 파일 참조를 포함하며, frontmatter description에 파괴적 작업 경고를 추가한다.
- B) frontmatter에 `argument-hint`를 추가해 필수 매개변수를 요청하고, `context: fork`로 실행을 격리하며, `allowed-tools`를 파일 쓰기 작업으로 제한한다. **[정답]**
- C) `/migration-create`와 `/migration-apply` Skill로 나누고, migration 이름이 누락되면 요청하는 validation 지시를 추가하며, 각각 다른 `allowed-tools` 범위를 사용한다.
- D) `$ARGUMENTS`가 유효한 이름인지 확인하는 validation 지시를 SKILL.md에 추가하고, 이전 대화 context를 무시하라는 prompt를 추가하며, 피해야 할 금지 작업을 나열한다.

B인 이유: 세 가지 별도 구성 기능으로 각 문제를 다룹니다. `argument-hint`는 인수 입력을 개선해 누락 인수를 줄이고, `context: fork`는 이전 대화의 context 누수를 방지하며, `allowed-tools`는 Skill을 안전한 파일 쓰기 작업으로 제한해 파괴적 동작을 막습니다.

문제 40(시나리오: Claude Code를 사용한 코드 생성)

상황: 코드베이스에는 서로 다른 코딩 규칙을 가진 영역이 있습니다. React 컴포넌트는 hooks가 있는 함수형 스타일을 사용하고, API handler는 특정 오류 처리와 함께 async/await를 사용하며, 데이터베이스 모델은 repository 패턴을 따릅니다. 테스트 파일은 테스트 대상 코드 옆에 코드베이스 전반에 분산되어 있습니다(예: `Button.tsx` 옆의 `Button.test.tsx`). 모든 테스트가 위치와 관계없이 같은 규칙을 따르기를 원합니다.

Claude가 코드 생성 시 올바른 규칙을 자동 적용하도록 보장하는 가장 지원되는 방법은 무엇인가요?

- A) 모든 규칙을 루트 CLAUDE.md에 영역별 제목 아래 넣고 Claude가 적용할 섹션을 추론하도록 의존한다.
- B) 각 코드 유형에 대해 `.claude/skills/`에 Skill을 만들고 각 SKILL.md에 규칙을 포함한다.
- C) 해당 영역의 규칙을 포함하는 별도 CLAUDE.md 파일을 각 하위 디렉터리에 둔다.
- D) 파일 경로를 기반으로 규칙을 조건부 적용하도록 glob 패턴을 지정하는 YAML frontmatter가 있는 규칙 파일을 `.claude/rules/` 아래 만든다. **[정답]**

D인 이유: `.claude/rules/` 파일과 YAML frontmatter, glob 패턴(예: `**/*.test.tsx`, `src/api/**/*.ts`)을 쓰면 디렉터리 구조와 관계없이 파일 경로에 맞는 규칙을 결정적으로 적용할 수 있습니다. 분산된 테스트 파일처럼 여러 디렉터리에 걸친 패턴에는 이 방식이 가장 적합합니다.

문제 41(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀의 표준 코드 리뷰 체크리스트를 실행하는 사용자 지정 슬래시 명령 `/review`를 만들고 싶습니다. 저장소를 클론하거나 업데이트하는 모든 개발자가 사용할 수 있어야 합니다.

명령 파일을 어디에 만들어야 하나요?

- A) 각 개발자 홈 디렉터리의 `~/.claude/commands/`.
- B) 프로젝트 저장소의 `.claude/commands/`. **[정답]**
- C) `.claude/config.json`의 commands 배열.
- D) 루트 프로젝트 CLAUDE.md.

B인 이유: 프로젝트 저장소 안의 `.claude/commands/` 아래에 사용자 지정 슬래시 명령을 두면 버전 관리되며 저장소를 클론하거나 업데이트하는 모든 개발자가 자동으로 사용할 수 있습니다. Claude Code에서 프로젝트 수준 사용자 지정 명령은 이 위치에 두는 것이 맞습니다.

문제 42(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀의 CLAUDE.md가 500줄을 넘어 TypeScript 규칙, 테스트 지침, API 패턴, 배포 절차가 섞여 있습니다. 개발자들은 올바른 섹션을 찾고 업데이트하기 어렵다고 느낍니다.

Claude Code가 프로젝트 수준 지시사항을 집중된 주제별 module로 구성하기 위해 지원하는 접근 방식은 무엇인가요?

- A) 파일 패턴을 CLAUDE.md 내부 특정 섹션에 매핑하는 `.claude/config.yaml`을 정의한다.
- B) `.claude/rules/`에 각 주제를 다루는 별도 Markdown 파일을 만든다(예: `testing.md`, `api-conventions.md`). **[정답]**
- C) 관련 하위 디렉터리의 README.md 파일로 지시사항을 나누면 Claude가 자동으로 지시사항으로 로드한다.
- D) 디렉터리 트리의 여러 수준에 CLAUDE.md라는 이름의 파일을 여러 개 만들어 각각 부모 지시사항을 override하게 한다.

B인 이유: Claude Code는 `.claude/rules/` 디렉터리를 지원하며, 여기에 주제별 지침(예: `testing.md`, `api-conventions.md`)을 위한 별도 Markdown 파일을 만들 수 있습니다. 이를 통해 팀은 큰 지시사항 세트를 집중되고 유지보수 가능한 module로 구성할 수 있습니다.

문제 43(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀이 구현 접근 방식을 선택하기 전에 brainstorming하고 평가하는 사용자 지정 Skill `/explore-alternatives`를 만들었습니다. 개발자들은 Skill 실행 후 후속 Claude 응답이 대안 논의의 영향을 받는다고 보고합니다. 때로는 거부된 접근을 언급하거나 실제 구현에 방해가 되는 탐색 context를 유지합니다.

이 Skill을 가장 효과적으로 어떻게 구성해야 하나요?

- A) Skill에서 `!` 접두사를 사용해 탐색 로직을 bash subprocess로 실행한다.
- B) Skill frontmatter에 `context: fork`를 추가한다. **[정답]**

- C) 탐색 context를 버려야 하는 경계를 표시하기 위해 `/explore-start`와 `/explore-end` 두 Skill로 나눈다.
- D) `.claude/skills/` 대신 `~/.claude/skills/`에 Skill을 만든다.

B인 이유: `context: fork`는 Skill을 격리된 subagent context에서 실행해 탐색 논의가 메인 대화 기록을 불필요하게 채우지 않게 합니다. 이를 통해 거부된 접근과 brainstorming context가 후속 구현 작업에 영향을 주는 것을 방지합니다.

문제 44(시나리오: Claude Code를 사용한 코드 생성)

상황: 팀은 Claude Code를 통해 PR 검색과 CI 상태 확인을 할 수 있도록 GitHub MCP 서버를 추가하고 싶어 합니다. 여섯 명의 개발자는 각자 개인 GitHub 액세스 token을 가지고 있습니다. 자격 증명을 버전 관리에 커밋하지 않으면서 팀 전체에 일관된 tool을 제공하고 싶습니다.

가장 효과적인 구성 접근 방식은 무엇인가요?

- A) 각 개발자가 `claude mcp add --scope user`로 사용자 범위에 서버를 추가하게 한다.
- B) `.env` 파일에서 token을 읽고 GitHub API 호출을 프록시하는 MCP 서버 래퍼를 만든 뒤, 그 래퍼를 프로젝트 `.mcp.json`에 추가한다.
- C) 인증에 환경 변수 치환(`${GITHUB_TOKEN}`)을 사용해 프로젝트 `.mcp.json`에 서버를 추가하고, 필요한 환경 변수를 프로젝트 README에 문서화한다. **[정답]**
- D) placeholder token으로 프로젝트 범위에 서버를 구성한 뒤, 개발자에게 로컬 구성에서 이를 override하라고 말한다.

C인 이유: 환경 변수 치환을 사용하는 프로젝트 `.mcp.json`은 관용적입니다. MCP 구성에 대한 버전 관리되는 단일 기준을 제공하면서 각 개발자가 환경 변수로 자격 증명을 제공할 수 있습니다. 변수를 문서화하면 비밀을 커밋하지 않고 온보딩도 쉬워집니다.

문제 45(시나리오: Claude Code를 사용한 코드 생성)

상황: 120개 파일 코드베이스 전반의 외부 API 호출 주변에 오류 처리 wrapper를 추가하고 있습니다. 작업은 세 단계입니다. (1) 모든 호출 지점과 패턴 발견, (2) 오류 처리 접근 방식 공동 설계, (3) wrapper를 일관되게 구현. 1단계에서 Claude는 수백 개 호출 지점과 context를 나열하는 큰 출력을 생성해, 발견이 끝나기 전에 context window가 빠르게 차오릅니다.

구현 일관성을 유지하면서 작업을 완료하는 가장 효과적인 접근 방법은 무엇인가요?

- A) 1단계에는 Explore subagent를 사용해 상세 발견 출력을 격리하고 summary를 반환하게 한 뒤, 2-3단계는 메인 대화에서 계속한다. **[정답]**
- B) 모든 단계를 메인 대화에서 수행하고 파일을 진행하면서 주기적으로 `/compact`를 사용해 context 사용량을 줄인다.
- C) `--continue`와 함께 headless 모드로 전환하고, 연속성을 유지하기 위해 batch 호출 사이에 명시적 context summary를 전달한다.
- D) 오류 처리 패턴을 CLAUDE.md에 정의한 뒤, 일관성을 위해 공유 memory 파일에 의존하며 여러 session에서 파일을 batch로 처리한다.

A인 이유: Explore subagent는 상세한 발견 출력을 별도 context에 격리하고 메인 대화에는 간결한 summary만 반환합니다. 이는 context가 특히 중요한 공동 설계와 일관된 구현 단계에 메인 context window를 아껴 둘 수 있습니다.

시나리오: 고객 지원 Agent

문제 46(시나리오: 고객 지원 Agent)

상황: 테스트 중 사용자가 주문 상태를 물을 때 `lookup_order`가 더 적절함에도 Agent가 `get_customer`를 자주 호출하는 것을 확인했습니다. 이 문제를 해결하기 위해 먼저 무엇을 확인해야 하나요?

먼저 무엇을 확인해야 하나요?

- A) 주문 관련 요청을 감지해 `lookup_order`로 직접 routing하는 전처리 classifier를 구현한다.
- B) 선택을 단순화하기 위해 Agent가 사용할 수 있는 tool 수를 줄인다.
- C) tool 선택을 개선하기 위해 가능한 모든 주문 요청 패턴을 다루는 few-shot 예시를 system prompt에 추가한다.
- D) tool 설명이 각 tool의 목적을 명확히 구분하는지 확인한다. **[정답]**

D인 이유: tool 설명은 모델이 어떤 tool을 호출할지 결정하는 데 사용하는 기본 입력입니다. Agent가 지속적으로 잘못된 tool을 선택할 때 첫 진단 단계는 tool 설명이 각 tool의 목적과 사용 경계를 명확히 분리하는지 확인하는 것입니다.

문제 47(시나리오: 고객 지원 Agent)

상황: Agent는 단일 이슈 요청(예: "주문 #1234 환불이 필요합니다")을 94% accuracy로 처리합니다. 하지만 고객이 한 메시지에 여러 이슈를 포함하면(예: "주문 #1234 환불이 필요하고 주문 #5678의 배송 주소도 업데이트하고 싶습니다") tool 선택 accuracy가 58%로 떨어집니다. Agent는 보통 하나의 이슈만 해결하거나 요청 간 매개변수를 섞습니다. 다중 이슈 요청의 안정성을 가장 효과적으로 개선하는 접근은 무엇인가요?

가장 효과적인 접근 방법은 무엇인가요?

- A) 별도 모델 호출을 사용해 다중 이슈 메시지를 별도 요청으로 분해하고, 각 요청을 독립적으로 처리한 뒤 결과를 병합하는 전처리 계층을 구현한다.
- B) 관련 tool을 더 적은 범용 tool로 결합한다.
- C) 다중 이슈 요청에 대한 올바른 추론과 tool 순서를 보여주는 few-shot 예시를 prompt에 추가한다. **[정답]**
- D) 불완전한 답변을 감지하고 누락된 이슈를 해결하도록 Agent에 자동으로 다시 prompt하는 응답 validation을 구현한다.

C인 이유: Agent가 단일 이슈에서는 이미 잘 수행하므로, 필요한 것은 여러 이슈를 분해하고 routing하며 매개변수를 분리해 유지하는 패턴에 대한 지침입니다. 이를 보여주는 few-shot 예시가 가장 효과적입니다.

문제 48(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그에 따르면 "주문 #1234 환불" 같은 단순 요청은 Agent가 3-4회 tool 호출로 91% 성공률을 보입니다. 하지만 "이중 청구되었고, 할인이 적용되지 않았으며, 취소하고 싶습니다" 같은 복잡한 요청은 평균 12회 이상의 tool 호출과 54% 성공률을 보입니다. Agent는 종종 이슈를 순차적으로 조사하고 각 이슈마다 중복으로 고객 데이터를 가져옵니다. 복잡한 요청 처리를 가장 효과적으로 개선하는 변경은 무엇인가요?

가장 효과적인 변경 사항은 무엇인가요?

- A) 단계 사이에 명시적 validation 체크포인트를 추가해, 각 이슈 해결 후 다음으로 넘어가기 전에 진행 상황을 기록하게 한다.
- B) `get_customer`, `lookup_order`, billing 관련 tool을 하나의 `investigate_issue` tool로 결합해 tool 수를 줄인다.
- C) 요청을 별도 이슈로 분해한 뒤 공유 고객 context를 사용해 각 이슈를 병렬 조사하고 최종 해결책을 종합한다. **[정답]**
- D) 다양한 다면적 billing 시나리오에 대한 이상적인 tool 호출 순서를 보여주는 few-shot 예시를 system prompt에 추가한다.

C인 이유: 별도 이슈로 분해하고 공유 고객 context로 병렬 조사하면 두 핵심 문제가 모두 해결됩니다. 이슈 간 공유 context를 재사용해 중복 데이터 조회를 없애고, 단일 해결책을 종합하기 전에 조사를 병렬화해 총 tool 호출 루프를 줄입니다.

문제 49(시나리오: 고객 지원 Agent)

상황: Agent의 첫 문의 해결률은 목표 80%보다 훨씬 낮은 55%입니다. 로그는 Agent가 단순한 케이스(사진 증거가 있는 손상 상품의 표준 교체)는 escalation하면서, 정책 예외가 필요한 복잡한 상황은 자율적으로 처리하려 한다는 것을 보여줍니다. escalation calibration을 개선하는 가장 효과적인 방법은 무엇인가요?

escalation calibration을 개선하는 가장 효과적인 방법은 무엇인가요?

- A) 각 응답 전에 Agent가 confidence를 1-10 척도로 자체 평가하게 하고, confidence가 임계값 아래로 떨어지면 자동으로 사람 리뷰로 routing한다.
- B) 메인 Agent가 처리를 시작하기 전에 어떤 요청이 escalation이 필요한지 예측하도록 과거 티켓으로 학습한 별도 classifier 모델을 배포한다.
- C) system prompt에 escalation할 때와 자율 해결할 때를 보여주는 few-shot 예시와 함께 명시적 escalation 기준을 추가한다. **[정답]**
- D) 고객 불만 수준을 판단하는 감정 분석을 구현하고 부정 감정 임계값을 넘으면 자동으로 escalation한다.

C인 이유: few-shot 예시가 포함된 명시적 escalation 기준은 단순 케이스와 복잡한 케이스 사이의 불명확한 결정 경계라는 근본 원인을 직접 해결합니다. 추가 인프라 없이 Agent에게 언제 escalation하고 언제 자율 해결할지 가르치는 가장 현실적이고 효과적인 첫 수정입니다.

문제 50(시나리오: 고객 지원 Agent)

상황: `get_customer`와 `lookup_order`를 호출한 후 Agent는 사용 가능한 시스템 데이터는 모두 조회했지만 여전히 판단이 애매한 상황입니다. 어떤 상황이 `escalate_to_human` 호출을 가장 정당화할 수 있나요?

어떤 상황이 escalation에 가장 정당한가요?

- A) 고객이 어제 배송되어 내일 도착 예정인 주문을 취소하려 한다. 고객이 패키지를 받은 후 마음을 바꿀 수 있으므로 escalation해야 한다.
- B) 고객은 주문을 받지 못했다고 주장하지만 추적 정보는 3일 전 고객 주소에서 배송 완료 및 서명 수령을 보여준다. 모순 증거를 제시하면 고객 관계를 해칠 수 있으므로 escalation해야 한다.
- C) 고객이 경쟁사 가격 매칭을 요청한다. 정책은 자사 사이트의 14일 내 가격 인하에 대한 가격 조정은 허용하지만 경쟁사 가격에 대해서는 언급하지 않는다. 정책 해석을 위해 escalation해야 한다. **[정답]**
- D) 고객 메시지에 billing 질문과 제품 반품이 모두 포함되어 있다. 사람이 한 상호작용에서 두 이슈를 조정할 수 있도록 escalation해야 한다.

C인 이유: 이는 실제 정책 공백입니다. 회사 규칙은 자사 사이트의 가격 인하는 다루지만 경쟁사 가격 매칭은 다루지 않습니다. Agent는 정책을 지어내면 안 되므로, 기존 규칙을 어떻게 해석하거나 확장할지는 사람에게 escalation해야 합니다.

문제 51(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그에 따르면 12%의 경우 Agent가 `get_customer`를 건너뛰고 고객이 제공한 이름만으로 `lookup_order`를 직접 호출해, 때로는 잘못된 계정을 식별하고 잘못된 환불을 유발합니다. 이 안정성 문제를 가장 효과적으로 해결하는 변경은 무엇인가요?

가장 효과적인 변경 사항은 무엇인가요?

- A) 고객이 자발적으로 주문 세부사항을 제공하더라도 Agent가 항상 `get_customer`를 먼저 호출하는 few-shot 예시를 추가한다.
- B) 각 요청을 분석하고 해당 요청 유형에 적절한 일부 tool만 활성화하는 routing classifier를 구현한다.
- C) `get_customer`가 validated 고객 식별자를 반환할 때까지 `lookup_order`와 `process_refund`를 차단하는 코드 수준의 사전조건을 추가한다. **[정답]**
- D) 주문 작업 전 `get_customer`를 통한 고객 validation이 필수라고 system prompt를 더 강하게 작성한다.

C인 이유: 코드 수준의 사전조건은 필요한 순서가 지켜진다는 결정적 보장을 제공합니다. LLM 동작과 관계없이 validation을 건너뛸 가능성을 제거하므로 가장 효과적인 접근입니다.

문제 52(시나리오: 고객 지원 Agent)

상황: 프로덕션 지표는 복잡한 billing 분쟁이나 다중 주문 반품을 해결할 때, 해결이 기술적으로는 맞더라도 고객 만족도 점수가 단순 케이스보다 15% 낮다는 것을 보여줍니다. 근본 원인 분석에 따르면 Agent는 정확한 해결책을 제공하지만 근거 설명이 일관되지 않습니다. 때로는 관련 정책 세부사항을 빠뜨리고, 때로는 일정 정보나 다음 단계를 놓칩니다. 구체적인 context 공백은 케이스마다 다릅니다. 사람의 감독을 추가하지 않고 솔루션 품질을 개선하고 싶습니다. 가장 효과적인 접근 방법은 무엇인가요?

가장 효과적인 접근 방법은 무엇인가요?

- A) Agent가 초안 응답의 완전성을 평가하는 자기 비판 단계를 추가한다. 고객 이슈를 해결하는지, 관련 context를 포함하는지, 후속 질문을 예상하는지 확인한다. **[정답]**
- B) 종료 전에 Agent가 "이것이 문제를 완전히 해결하나요?"라고 묻는 확인 단계를 추가해, 고객이 필요한 추가 정보를 요청할 수 있게 한다.
- C) 미리 정의한 복잡도 지표로 복잡한 케이스를 routing해 모델을 Haiku에서 Sonnet으로 업그레이드한다.
- D) 정책 context, 일정, 다음 단계를 포함하는 방법을 보여주는 다섯 가지 일반 복잡 케이스에 대한 완전한 설명 few-shot 예시를 system prompt에 구현한다.

A인 이유: 자기 비판 단계(evaluator-optimizer 패턴)는 정책 context, 일정, 다음 단계 같은 구체 기준에 대해 Agent가 자신의 초안을 평가하도록 강제해 설명 완성도가 들쭉날쭉한 문제를 직접 줄입니다. 이는 사람의 감독 없이 케이스별 공백을 잡아냅니다.

문제 53(시나리오: 고객 지원 Agent)

상황: 프로덕션 지표는 Agent가 해결당 평균 4회 이상의 API 루프를 사용함을 보여줍니다. 분석 결과 Claude는 처음부터 둘 다 필요한 경우에도 `get_customer`와 `lookup_order`를 별도 순차 턴으로 요청하는 경우가 많습니다. 루프 수를 가장 효과적으로 줄이려면 어떻게 해야 하나요?

루프를 줄이는 가장 효과적인 방법은 무엇인가요?

- A) 요청된 tool과 병렬로 필요할 가능성이 있는 tool을 자동 호출하고 Claude가 요청했는지와 관계없이 모든 결과를 반환하는 speculative execution을 구현한다.
- B) Claude가 계획할 공간을 더 많이 갖고 자연스럽게 tool 요청을 결합하도록 `max_tokens`를 늘린다.
- C) 일반 조회 조합을 단일 호출로 묶는 `get_customer_with_orders` 같은 복합 tool을 만든다.
- D) prompt에서 Claude에게 관련 tool 요청을 한 턴에 묶고 다음 API 호출 전에 모든 결과를 함께 반환하라고 지시한다. **[정답]**

D인 이유: Claude에게 관련 tool 요청을 단일 턴에 묶도록 prompt하면 한 번에 여러 tool을 요청하는 Claude의 기본 능력을 활용할 수 있습니다. 이는 최소한의 architecture 변경으로 순차 호출 패턴을 줄입니다.

문제 54(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그는 고객이 특정 금액을 언급하지만(예: "제가 말한 15% 할인") Agent가 잘못된 값으로 응답하는 패턴을 보여줍니다. 조사 결과 이러한 세부사항은 20턴 이상 전에 언급되었고 "프로모션 가격이 논의됨" 같은 모호한 summary로 압축되었습니다. 가장 효과적인 수정안은 무엇인가요?

가장 효과적인 수정안은 무엇인가요?

- A) summary가 트리거되기 전 대화에 더 많은 공간을 주도록 summary 임계값을 70%에서 85%로 높인다.
- B) 전체 대화 기록을 외부 저장소에 저장하고 Agent가 "앞서 말했듯이" 같은 참조를 감지하면 검색을 구현한다.
- C) 거래 관련 사실(금액, 날짜, 주문 번호)을 summary 기록 밖에 있는 지속적 "case facts" 블록으로 extraction 해 모든 prompt에 포함한다. **[정답]**
- D) 모든 숫자, 백분율, 날짜, 고객이 말한 기대사항을 원문 그대로 보존하도록 summary prompt를 수정한다.

C인 이유: summary는 본질적으로 정확한 세부사항을 잃습니다. 거래 관련 사실을 summary 기록 밖의 구조화된 "case facts" 블록으로 extraction하면 몇 턴이 summary되었는지와 관계없이 중요한 정보가 모든 prompt에서 안정적으로 사용 가능합니다.

문제 55(시나리오: 고객 지원 Agent)

상황: `get_customer` tool은 이름으로 검색할 때 모든 일치 항목을 반환합니다. 현재 여러 결과가 있으면 Claude가 가장 최근 주문이 있는 고객을 선택하지만, 프로덕션 데이터는 모호한 일치에서 이 방식이 15%의 경우 잘못된 계정을 선택함을 보여줍니다. 이를 어떻게 해결해야 하나요?

어떻게 해결해야 하나요?

- A) 85% 이상 confidence에서는 자율적으로 행동하고 그 아래에서는 clarification을 요청하는 confidence 점수 시스템을 구현한다.
- B) `get_customer`가 여러 일치 항목을 반환하면 고객별 동작을 하기 전에 추가 식별자(email, phone, order number)를 요청하도록 Claude에 지시한다. **[정답]**
- C) `get_customer`를 수정해 ranking 알고리즘 기반으로 가장 가능성 높은 단일 일치만 반환하게 하여 모호성을 제거한다.
- D) 모호한 일치에 대한 올바른 추론과 tool 순서를 보여주는 few-shot 예시를 prompt에 추가한다.

B인 이유: 추가 식별자를 사용자에게 요청하는 것이 모호성을 해결하는 가장 신뢰할 수 있는 방법입니다. 사용자는 자신의 신원에 대한 확정적 지식을 가지고 있기 때문입니다. 잘못된 계정을 선택해 생기는 15% 오류율을 제거하기 위해 대화 한 턴을 추가하는 것은 작은 비용입니다.

문제 56(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그는 일관된 패턴을 보여줍니다. 고객이 메시지에 "account"라는 단어를 포함하면(예: "I want to check my account for an order I made yesterday") Agent가 78%의 경우 `get_customer`를 먼저 호출합니다. 고객이 유사한 요청을 "account" 없이 표현하면(예: "I want to check an order I made yesterday") 93%의 경우 `lookup_order`를 먼저 호출합니다. tool 설명은 명확하고 모호하지 않습니다. 이 불일치의 가장 가능성이 높은 근본 원인은 무엇인가요?

가장 가능성이 높은 근본 원인은 무엇인가요?

- A) system prompt에 "account" 같은 용어를 기준으로 동작을 유도하는 키워드 민감 지시사항이 있어 의도치 않은 tool 선택 패턴을 만든다. **[정답]**
- B) 모델의 기본 학습이 "account" 용어와 고객 관련 작업 사이의 연관을 만들어 tool 설명을 덮어쓴다.
- C) 모델에는 다중 개념 메시지에 대한 더 많은 학습 데이터가 필요하며 account와 order 용어가 모두 포함된 예시로 fine-tuning해야 한다.
- D) 이 키워드 유발 혼동을 막기 위해 각 tool을 언제 사용하지 말아야 하는지 지정하는 부정 예시를 tool 설명에 추가해야 한다.

A인 이유: 체계적인 키워드 기반 패턴(78% vs 93%)은 system prompt의 명시적 routing 로직이 "account"라는 단어에 반응해 Agent를 고객 관련 tool로 유도하고 있음을 강하게 시사합니다. tool 설명이 이미 명확하므로, 불일치는 의도치 않은 행동 유도를 만드는 prompt 수준 지시를 가리킵니다.

문제 57(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그는 사용자가 주문에 대해 물을 때(예: "check my order #12345") `lookup_order` 대신 `get_customer`를 호출하는 경우가 많음을 보여줍니다. 두 tool은 최소한의 설명("Gets customer information" / "Gets order details")만 가지고 있고 유사해 보이는 식별자 형식을 받습니다. tool 선택 신뢰성을 개선하기 위한 가장 효과적인 첫 단계는 무엇인가요?

가장 효과적인 첫 단계는 무엇인가요?

- A) 각 턴 전에 사용자 입력을 분석하고 감지된 키워드와 ID 패턴을 기반으로 올바른 tool을 사전 선택하는 routing 계층을 구현한다.
- B) 두 tool을 모든 식별자를 받고 내부적으로 어떤 backend를 조회할지 결정하는 단일 `lookup_entity`로 결합한다.
- C) 주문 관련 쿼리를 `lookup_order`로 routing하는 5-8개 예시와 함께 올바른 tool 선택 패턴을 보여주는 few-shot 예시를 system prompt에 추가한다.
- D) 각 tool 설명을 입력 형식, 예시 쿼리, 엣지 케이스, 유사 tool 대비 사용 시점을 설명하는 경계까지 포함하도록 확장한다. **[정답]**

D인 이유: 입력 형식, 예시 쿼리, 엣지 케이스, 명확한 경계를 포함해 tool 설명을 확장하면 LLM이 유사한 tool을 구분할 만큼 충분한 정보를 받지 못하는 문제를 해결합니다. 이는 LLM이 tool 선택에 사용하는 기본 메커니즘을 개선하는 저노력, 고효과의 첫 단계입니다.

문제 58(시나리오: 고객 지원 Agent)

상황: 지원 Agent를 위한 Agent 루프를 구현하고 있습니다. 각 Claude API 호출 후 루프를 계속할지(요청된 tool을 실행하고 Claude를 다시 호출) 또는 중지할지(최종 답변을 고객에게 제시) 결정해야 합니다. 이 결정을 무엇을 기준으로 판단해야 하나요?

이 결정을 무엇을 기준으로 판단해야 하나요?

- A) Claude 응답의 `stop_reason` 필드를 확인한다. `tool_use`이면 계속하고 `end_turn`이면 중지한다. **[정답]**
- B) "I'm done" 또는 "Can I help with anything else?" 같은 문구를 Claude 텍스트에서 parsing한다. 자연어 신호가 작업 완료를 나타낸다.
- C) 최대 반복 횟수(예: 10회 호출)를 설정하고 Claude가 더 필요하다고 나타내는지와 관계없이 도달하면 중지한다.
- D) 응답에 assistant 텍스트 콘텐츠가 포함되어 있는지 확인한다. Claude가 설명 텍스트를 생성했다면 루프는 종료되어야 한다.

A인 이유: `stop_reason`은 루프 제어를 위한 Claude의 명시적 structured 신호입니다. `tool_use`는 Claude가 tool을 실행하고 결과를 돌려받고 싶다는 뜻이고, `end_turn`은 Claude가 응답을 완료했으며 루프가 끝나야 함을 뜻합니다.

문제 59(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그는 Agent가 MCP tool 출력을 잘못 해석한다는 것을 보여줍니다. `get_customer`의 Unix timestamp, `lookup_order`의 ISO 8601 날짜, 숫자 상태 코드(1=pending, 2=shipped)가 혼재합니다. 일부 tool은 수정할 수 없는 third-party MCP 서버입니다. 데이터 형식 normalization에 가장 유지보수하기 좋은 접근은 무엇인가요?

가장 유지보수하기 좋은 접근은 무엇인가요?

- A) PostToolUse hook을 사용해 Agent가 처리하기 전에 tool 출력을 가로채고 형식 변환을 적용한다. **[정답]**
- B) 제어 가능한 tool을 사람이 읽기 쉬운 형식을 반환하도록 수정하고 third-party tool용 wrapper를 만든다.
- C) 모든 데이터 조회 후 Agent가 값을 변환하기 위해 호출하는 `normalize_data` tool을 만든다.
- D) 각 tool의 데이터 규칙을 설명하는 상세 형식 문서를 system prompt에 추가한다.

A인 이유: PostToolUse hook은 third-party MCP 서버 데이터를 포함한 모든 tool 출력을 Agent가 처리하기 전에 가로채 normalization할 수 있는 중앙화되고 결정적인 지점을 제공합니다. 변환이 코드에 존재하고 균일하게 적용되므로 LLM 해석에 의존하는 것보다 유지보수하기 좋습니다.

문제 60(시나리오: 고객 지원 Agent)

상황: 프로덕션 로그는 특히 "최근 구매 건을 도와주세요" 같은 모호한 쿼리에서 Agent가 `lookup_order`가 더 적절한데도 때때로 `get_customer`를 선택함을 보여줍니다. tool 선택을 개선하기 위해 system prompt에 few-shot 예시를 추가하기로 했습니다. 어떤 접근이 문제를 가장 효과적으로 다루나요?

가장 효과적인 접근 방법은 무엇인가요?

- A) 모호한 사례를 포함해 각 tool 설명에 명시적 "use when"과 "don't use when" 지침을 추가한다.
- B) tool별로 예시를 그룹화한다. 모든 `get_customer` 시나리오를 모은 뒤 모든 `lookup_order` 시나리오를 모은다.
- C) 모호한 시나리오를 겨냥한 4-6개 예시를 추가하고, 각 예시에 왜 한 tool이 가능한 대안보다 선택되었는지 근거를 포함한다. **[정답]**
- D) 각 tool의 일반적 시나리오에 대한 올바른 tool 선택을 보여주는 명확하고 모호하지 않은 요청 예시 10-15개를 추가한다.

C인 이유: 오류가 발생하는 특정 모호한 시나리오를 겨냥하고, 왜 한 tool이 대안보다 나은지 명시적 근거를 포함한 few-shot 예시는 엣지 케이스에 필요한 비교 판단 과정을 모델에 가르칩니다. 일반 예시나 선언적 규칙보다 이 상황에 더 잘 맞습니다.

문제 61(시나리오: Conversational AI Architecture Patterns)

상황: `remove_team_member` tool은 실행 전 영향을 미리 보기 위해 `dry_run: boolean` 매개변수를 사용합니다. 프로덕션 모니터링은 Agent가 `dry_run=false`로 직접 호출해 미리보기 단계를 우회한다는 것을 보여줍니다. 모든 제거 작업이 사용자가 명시적으로 확인한 미리보기 이후에만 수행되도록 해야 합니다.

가장 신뢰할 수 있는 접근 방식은 무엇인가요?

- A) 동일한 매개변수의 `dry_run=true` 호출이 지난 60초 내에 있었을 때만 `dry_run=false`를 허용하는 서버 측 validation을 추가한다.
- B) tool에 confirmation 필요 표시를 하고, annotated tool 호출을 전달하기 전에 사용자 승인을 요청하도록 orchestration 계층을 구성한다.
- C) tool 설명에 Agent가 항상 먼저 `dry_run=true`로 호출하고 사용자 확인을 기다린 뒤 다시 호출해야 한다는 상세 지시와 few-shot 예시를 추가한다.
- D) 두 tool로 교체한다. `preview_remove_member`는 영향 세부사항과 1회용 confirmation token을 반환하고, `execute_remove_member`는 그 token을 요구해 실행을 미리보기에 바인딩한다. **[정답]**

D인 이유: 두 tool로 나누고 token-binding을 적용하면 사전 미리보기 없이 실행하는 경로가 architecture상 사라집니다. 실행 tool이 미리보기 tool에서만 발급되는 token을 반드시 요구하기 때문입니다. 따라서 LLM이 지시를 잘 따르는지(C), 시간 간격 heuristic이 맞는지(A), orchestration 계층이 제대로 동작하는지(B)에 기대지 않고 코드 수준에서 제약을 강제할 수 있습니다.

문제 62(시나리오: Conversational AI Architecture Patterns)

상황: 프로덕션 모니터링에 따르면 `search_catalog` tool이 12% 실패합니다. 8%는 retry하면 성공하는 네트워크 타임아웃이고, 4%는 retry해도 절대 성공하지 않는 쿼리 구문 오류입니다. 현재 두 오류 유형이 동일하게 반환되어 불필요한 retry를 유발합니다.

tool의 오류 처리를 어떻게 수정해야 하나요?

- A) 네트워크 오류와 구문 오류를 구분하는 방법을 보여주는 few-shot 예시를 system prompt에 추가한다.
- B) 모든 오류에 지수 백오프 retry 로직을 균일하게 적용한다.
- C) 네트워크 타임아웃에는 tool 내부에서 backoff 자동 retry를 구현하고, 구문 오류는 매개변수 validation 세부사항과 함께 즉시 반환한다. **[정답]**
- D) 모든 오류를 `retryable` boolean 플래그와 오류 유형 세부사항과 함께 반환한다.

C인 이유: 일시적 오류에 대한 retry를 tool 수준에서 처리하는 것이 올바른 추상화 경계입니다. tool은 오류 유형을 확정적으로 알고 있으며, Agent가 플래그(D)를 해석하거나 prompt 지시(A)를 따르는 것에 의존하지 않고 결정적 retry 로직을 구현할 수 있습니다. 균일한 backoff(B)는 절대 성공하지 않을 구문 오류에 시간을 낭비합니다.

문제 63(시나리오: Conversational AI Architecture Patterns)

상황: 투자 전략을 여러 턴에 걸쳐 논의하는 동안 사용자는 "위험 허용도가 매우 낮다"고 말했고, 나중에는 "수익을 극대화하고 싶다"고 말했습니다. 이제 "무엇에 투자해야 하나요?"라고 묻습니다.

추천이 사용자의 실제 우선순위에 맞도록 가장 잘 보장하는 접근은 무엇인가요?

- A) 모순을 드러내고 무엇이 더 중요한지 사용자에게 명확히 해 달라고 요청한다. **[정답]**
- B) 두 시나리오에 대해 별도 추천을 제공한다.
- C) 가장 최근에 말한 선호로 진행한다.
- D) 충돌을 다루지 않고 균형 잡힌 포트폴리오를 추천한다.

A인 이유: 사용자 선호가 직접 모순될 때 충돌을 드러내고 clarification을 요청하는 것이 추천이 사용자의 진짜 의도에 맞도록 보장하는 유일한 방법입니다. 다른 접근은 틀릴 수 있는 가정을 포함합니다. 수익 극대화와 낮은 위험 허

용도는 근본적으로 양립하기 어려운 목표이며 사람의 결정이 필요합니다.

문제 64(시나리오: Conversational AI Architecture Patterns)

상황: 사용자는 여러 대화 턴에 걸쳐 플레이리스트 선호를 다듬습니다. 사용자가 "재즈를 좋아해"라고 말한 두 메시지 뒤에 Claude가 "어떤 장르를 좋아하나요?"라고 묻습니다.

가장 가능성이 높은 원인은 무엇인가요?

- A) Claude는 대화 memory를 유지하려면 vector database 연결이 필요하다.
- B) 모델의 context window가 초과되었다.
- C) Claude API에는 `session_id` 매개변수가 필요하다.
- D) 애플리케이션이 이전 메시지를 `messages` 배열에 포함하지 않고 있다. **[정답]**

D인 이유: Claude에는 서버 측 memory가 없습니다. 모든 API 호출은 stateless입니다. 각 요청의 `messages` 배열에 전체 대화 기록을 포함하지 않으면 Claude는 이전 턴을 알 수 없습니다. Vector database(A)와 `session_id`(C)는 Claude architecture의 일부가 아니며, 두 메시지 교환에서 context window 초과(B)는 불가능합니다.

문제 65(시나리오: Conversational AI Architecture Patterns)

상황: 40분 요리 session 후 대화가 78,000token에 도달했습니다. 기록에는 알레르기, 레시피 배율 조정, clarification된 요리 용어, 일반 논의가 포함됩니다. 중요한 정보를 보존하면서 token을 줄여야 합니다.

보존과 token 감소의 균형을 가장 잘 맞추는 접근은 무엇인가요?

- A) 전체 대화 기록을 summary한다.
- B) 가장 최근 20,000token만 유지한다.
- C) 중요한 structured data(알레르기, 수량, 선호)를 extraction하고, 일반 논의는 summary로 압축하고, 최근 교환은 원문 그대로 유지한다. **[정답]**
- D) 전체 대화를 외부에 저장하고 semantic search로 관련 부분을 검색한다.

C인 이유: 하이브리드 방식은 가치가 큰 정보를 낮은 비용으로 유지합니다. 알레르기와 레시피 수량 같은 중요 사실은 압축된 structured 블록으로 extraction해 summary 중 발생하는 정밀도 손실을 방지하고, 일반 논의는 summary로 압축하고, 최근 교환은 대화 일관성을 위해 원문 그대로 유지합니다. A와 B는 중요한 식이 정보를 잃을 위험이 있고, D는 단일 요리 session에는 architecture상 과합니다.

문제 66(시나리오: Conversational AI Architecture Patterns)

상황: 사용자는 긴 대화 중 assistant가 이전 주제와 선호를 놓친다고 보고합니다. 현재 구현은 마지막 25개 메시지 쌍만 유지합니다.

가장 효과적인 해결책은 무엇인가요?

- A) 하이브리드 접근: 오래된 메시지는 summary하고 최근 메시지는 원문 그대로 유지한다. **[정답]**

- B) 전체 대화 기록에 대한 vector similarity search.
- C) window를 50개 메시지 쌍으로 늘린다.
- D) 매 턴 삭제된 메시지를 summary하고 running summary를 앞에 붙인다.

A인 이유: 하이브리드 방식은 문제의 두 차원을 모두 다룹니다. 대화 일관성에 중요한 정확한 최근 context를 유지하면서, 쌍이 삭제될 때 전체 손실을 막기 위해 이전 선호의 압축 표현을 유지합니다. window 확대(C)는 같은 문제를 지연할 뿐입니다. Vector search(B)는 현재 쿼리와 의미적으로 유사하지 않은 중요한 context를 놓칠 수 있습니다. 매 턴 전체 summary(D)은 overhead를 추가하고 summary 오류를 누적합니다.

문제 67(시나리오: Conversational AI Architecture Patterns)

상황: 사용자는 대화가 50턴을 넘으면 지연 시간이 증가하고 비용이 오른다고 보고합니다.

주요 원인은 무엇인가요?

- A) 전체 대화 기록이 각 API 요청에 포함된다. **[정답]**
- B) 모델이 점점 더 긴 응답을 생성한다.
- C) 기록이 커지면서 데이터베이스 작업이 느려진다.
- D) 모델이 내부 사용자 프로필을 구축해 더 많은 처리가 필요하다.

A인 이유: Claude API는 완전히 stateless입니다. 모든 요청은 `messages` 배열에 완전한 대화 기록을 포함해야 합니다. 대화가 커질수록 각 요청이 더 많은 token을 운반하므로 처리 지연과 비용이 직접 증가합니다. 모델은 호출 사이에 내부 상태를 유지하지 않으며(D는 거짓), 응답 길이가 대화 길이와 본질적으로 연결된 것은 아닙니다(B).

문제 68(시나리오: Conversational AI Architecture Patterns)

상황: 3개월간의 주간 session 후 대화 기록이 85,000token으로 커졌습니다. 사용자가 "고립이라는 주제에 대해 우리가 어떤 결론을 냈지?"라고 묻자 assistant는 이전 논의를 참조하지 않고 일반적인 답변을 합니다.

가장 효과적인 접근 방법은 무엇인가요?

- A) Rolling window truncation.
- B) 핵심 결론을 포착하는 점진적 summary.
- C) 관련 교환을 검색하는 semantic embeddings. **[정답]**
- D) 논의 결론을 표시하는 structured XML 태그 추가.

C인 이유: 대화 기록에 대한 semantic search는 3개월간의 논의로 확장되면서도 필요한 특정 관련 교환을 on-demand로 표면화할 수 있는 유일한 접근입니다. Rolling window(A)는 대부분의 기록을 버립니다. 점진적 summary(B)은 논의를 추상화로 압축해 사용자가 묻는 구체적 결론을 잃습니다. XML 태그(D)는 모든 과거 콘텐츠 재구성을 요구하며 이 규모의 검색 문제를 해결하지 못합니다.

문제 69(시나리오: Conversational AI Architecture Patterns)

상황: QA 테스트 중 Claude는 처음 10-15턴 동안 system prompt 지침을 따르지만 이후 응답은 벗어납니다. 대화는 여전히 token 한도 내에 있습니다.

가장 좋은 해결책은 무엇인가요?

- A) 행동 지침을 첫 번째 사용자 메시지로 옮긴다.
- B) 20턴 후 새 대화를 시작한다.
- C) 대화 분기점에 지침을 강화하는 user-role 메시지를 삽입한다. **[정답]**
- D) 응답 후 validation을 사용해 비준수 응답을 재생성한다.

C인 이유: 행동 reminder를 주기적으로 주입하면 대화 기록이 누적될 때 정기적으로 제약을 재확립해 instruction drift에 직접 대응합니다. 지침을 첫 사용자 메시지로 옮기면(A) 권위가 낮아집니다. 새 대화를 시작하면(B) context가 사라집니다. 응답 후 validation(D)은 예방이 아니라 교정이며 상당한 지연을 추가합니다.

문제 70(시나리오: Conversational AI Architecture Patterns)

상황: AI 튜터에는 교수 방법론과 적응 규칙을 정의하는 2,800token system prompt가 있습니다. 12턴 후 assistant가 숙련도 수준을 무시하기 시작합니다.

가장 효과적인 수정안은 무엇인가요?

- A) 4-5턴마다 reminder를 주입한다.
- B) 장황한 규칙을 숙련도 수준 적응을 보여주는 few-shot 예시로 교체한다. **[정답]**
- C) 중요한 규칙을 system prompt 끝에 batch한다.
- D) 응답을 평가하고 난이도 수준이 맞지 않으면 재생성한다.

B인 이유: 선언적 규칙이 있는 2,800token system prompt는 모델이 매 턴 이를 추론해야 하므로 drift에 취약합니다. 장황한 규칙을 올바른 숙련도 수준 적응을 보여주는 구체적인 few-shot 예시로 바꾸면 모델이 맞출 명확한 행동 패턴을 제공합니다. 이는 여러 턴에 걸쳐 추상 지시보다 더 안정적으로 따릅니다. Reminder 주입(A)은 도움이 되지만 증상을 다루고, 끝 batch(C)는 초기에는 도움이 되지만 턴 수준 drift에는 부족하며, 재생성(D)은 수정 비용이 큽니다.

문제 71(시나리오: Conversational AI Architecture Patterns)

상황: assistant는 열정적인 톤을 유지하고, 자신의 추론을 설명하며, clarification 질문을 해야 합니다. 이러한 행동 지침은 어디에 정의해야 하나요?

이 행동 지침은 어디에 정의해야 하나요?

- A) 각 사용자 메시지 앞에 붙인다.
- B) system prompt에 둔다. **[정답]**
- C) 첫 번째 assistant 메시지에 둔다.
- D) 환경 변수에 둔다.

B인 이유: system prompt는 전체 대화에 적용되는 지속적 행동 제약과 지침을 위해 특별히 설계되었습니다. 각 사용자 메시지에 붙이는 방식(A)은 불필요한 overhead를 만듭니다. 첫 번째 assistant 메시지(C)는 모델이 자신의 이전 진술에서 벗어날 수 있어 신뢰할 수 없습니다. 환경 변수(D)는 모델 동작에 영향을 주지 않습니다.

문제 72(시나리오: Conversational AI Architecture Patterns)

상황: 사용자는 "Certainly!"나 "I'd be happy to help!" 같은 반복적인 응답 시작을 보고합니다.

가장 효과적인 접근 방법은 무엇인가요?

- A) 직접적인 응답 시작으로 partial assistant message를 덧붙인다. **[정답]**
- B) temperature 설정을 낮춘다.
- C) 후처리로 인사말을 제거한다.
- D) system prompt에 해당 문구를 피하라는 지시를 추가한다.

A인 이유: assistant 응답을 직접 답변의 시작으로 prefill하면 생성 수준에서 인사 패턴을 방지합니다. 모델은 새 시작 문구를 생성하는 대신 prefill에서 이어서 생성합니다. system prompt 지시(D)는 도움이 될 수 있지만 모델이 여전히 변형을 생성할 수 있어 덜 신뢰할 수 있습니다. 후처리(C)는 취약한 우회입니다. Temperature(B)는 무작위성을 제어하지 특정 문구 패턴을 제어하지 않습니다.

문제 73(시나리오: Conversational AI Architecture Patterns)

상황: 사용자가 활발히 채팅하는 동안 webhook이 사용자의 패키지가 배송되었다고 시스템에 알립니다. assistant가 이를 다음 응답에 자연스럽게 반영하길 원합니다.

가장 좋은 접근은 무엇인가요?

- A) 배송 상태를 system prompt에 추가한다.
- B) 즉시 합성 사용자 메시지를 보낸다.
- C) 매 턴 상태 tool을 호출하도록 assistant를 강제한다.
- D) 다음 사용자 메시지의 prefix로 상태 업데이트를 덧붙인다. **[정답]**

D인 이유: 다음 사용자 메시지에 상태 업데이트를 prefix로 주입하면 대화 흐름을 방해하지 않고 자연스러운 대화 경계에서 실시간 context를 주입할 수 있습니다. system prompt 수정(A)은 session 재구성이 필요하거나 architecture상 번거롭습니다. 합성 사용자 메시지(B)는 자연스러운 대화 흐름을 깨고 attribution을 혼란스럽게 할 수 있습니다. 매 턴 tool 호출 강제(C)는 이벤트가 드문 경우 낭비입니다.

문제 74(시나리오: Conversational AI Architecture Patterns)

상황: 사용자가 "파티 장소를 예약해줘" 같은 요청을 자주 보냅니다. assistant는 4개 이상의 clarification 질문을 하며, 이로 인해 35% 이탈이 발생합니다.

trade-off를 가장 잘 개선하는 접근은 무엇인가요?

- A) 숨겨진 기본값으로 진행한다.

- B) 모든 clarification 질문을 하나의 복합 메시지로 묻는다.
- C) 가정을 명시적으로 밝히고 진행하되 수정을 초대한다. **[정답]**
- D) 구조화된 intake form을 사용한다.

C인 이유: 가정을 명시적으로 밝히고 진행하면 사용자는 즉시 유용한 응답을 받으면서도 잘못된 가정을 수정할 수 있습니다. 숨겨진 기본값(A)은 사용자가 무엇이 가정되었는지 알 수 없게 합니다. 복합 질문 목록(B)은 여전히 사용자에게 선행 노력을 요구합니다. structured 양식(D)은 더 많은 마찰을 추가해 이탈 감소 목표와 반대입니다.

문제 75(시나리오: Conversational AI Architecture Patterns)

상황: assistant는 contractor persona system prompt를 사용합니다. 초기 턴은 규칙을 따르지만 7턴째에는 일반적인 조언을 제공합니다. 대화 길이는 2,500token에 불과합니다.

가장 가능성이 높은 원인은 무엇인가요?

- A) system prompt는 초기 동작만 설정한다.
- B) 턴이 누적되면서 모델 주의가 약해진다.
- C) 누적된 assistant 응답이 system prompt 영향을 희석한다. **[정답]**
- D) system prompt가 한 번만 전송된다.

C인 이유: assistant 응답이 대화 기록에 누적되면서 system prompt의 행동 제약을 반영하는 텍스트 비율이 증가하는 assistant 생성 콘텐츠에 비해 줄어듭니다. 모델은 system prompt보다 자신의 이전 출력에 점점 더 패턴 매칭하면서 짧은 token 길이에서도 drift가 누적됩니다. system prompt는 모든 API 호출에 포함되며(D는 단독 설명으로 거짓), 모델 주의 저하(B)는 2,500token에서 작동하는 설명이 아닙니다.

문제 76(시나리오: Conversational AI Architecture Patterns)

상황: 사용자가 "보고서 좀 도와줄 수 있어?" 같은 모호한 요청을 합니다. assistant는 여러 질문(어떤 보고서? 어떤 도움? 마감일?)으로 응답하며, 이로 인해 40% 이탈이 발생합니다.

가장 좋은 해결책은 무엇인가요?

- A) 합리적인 가정을 하고, 이를 명시적으로 밝히며, 조정할 수 있다고 제안한다. **[정답]**
- B) 응답 전에 작은 모델로 모호성을 classification한다.
- C) 가정을 밝히지 않고 사전 정의된 해석을 사용한다.
- D) assistant를 턴당 하나의 clarification 질문으로 제한한다.

A인 이유: 합리적인 명시적 가정으로 진행하면 사용자가 정보를 알고 통제권을 유지하면서도 왕복을 완전히 제거할 수 있습니다. 조용한 사전 정의 해석(C)은 응답이 의도와 맞지 않을 때 사용자를 혼란스럽게 합니다. 한 질문 제한(D)은 여전히 여러 턴의 왕복을 요구합니다. 작은 classification 모델(B)은 핵심 UX 문제를 해결하지 못하면서 지연과 인프라 복잡성을 추가합니다.

실습 과제

과제 1: escalation 로직을 갖춘 멀티 tool Agent

목표: tool 통합, structured error 처리, escalation을 포함한 Agent 루프를 설계합니다.

단계:

1. 상세 설명이 있는 MCP tool 3-4개 정의(tool 선택 테스트를 위해 유사한 tool 2개 포함)
2. `stop_reason`(`"tool_use"` / `"end_turn"`)을 확인하는 Agent 루프 구현
3. structured error 응답 추가: `errorCategory`, `isRetryable`, description
4. 임계값을 초과하는 작업을 차단하고 escalation으로 routing하는 interceptor hook 구현
5. 다면적 요청으로 테스트

도메인: 1(Agent architecture), 2(tool과 MCP), 5(context와 안정성)

과제 2: 팀 개발을 위한 Claude Code 구성

목표: CLAUDE.md, 사용자 지정 명령, 경로별 규칙, MCP 서버를 구성합니다.

단계:

1. 범용 표준을 포함한 프로젝트 수준 CLAUDE.md 생성
2. 서로 다른 코드 영역을 위한 YAML frontmatter가 있는 `.claude/rules/` 파일 생성(`paths: ["src/api/**/*"]`, `paths: ["**/*.test.*"]`)
3. `context: fork`와 `allowed-tools`가 있는 프로젝트 Skill을 `.claude/skills/` 아래 생성
4. 환경 변수를 사용하는 MCP 서버를 `.mcp.json`에 구성하고 `~/.claude.json`에 개인 override 추가
5. 서로 다른 복잡도의 작업에서 계획 모드와 직접 실행 테스트

도메인: 3(Claude Code 구성), 2(tool과 MCP)

과제 3: structured data extraction pipeline

목표: JSON schema, structured output을 위한 `tool_use`, validation/retry 루프, batch 처리.

단계:

1. JSON schema가 있는 extraction tool 정의(필수/선택 필드, "other"가 있는 enum, nullable 필드)
2. validation 루프 구축: 오류 시 문서, 잘못된 extraction, 구체적 validation 오류와 함께 retry
3. 서로 다른 구조의 문서를 위한 few-shot 예시 추가
4. Message Batches API를 통한 batch 처리 사용: 문서 100개, `custom_id`로 실패 처리
5. 사람 리뷰로 routing: 필드별 confidence 점수, 문서 유형별 분석

도메인: 4(Prompt Engineering), 5(context와 안정성)

과제 4: multi-agent research pipeline 설계와 디버깅

목표: subagent orchestration, context 전달, 오류 전파, source 추적이 포함된 종합.

단계:

1. 2개 이상의 subagent를 가진 Coordinator(`allowedTools`에 `"Task"` 포함, context는 prompt에 명시적으로 전달)
2. 단일 응답에서 여러 `Task` 호출로 subagent를 병렬 실행
3. 구조화된 subagent 출력 요구: claim, quote, source URL, publication date
4. subagent 타임아웃 시뮬레이션: structured error context를 Coordinator에게 반환하고 부분 결과로 계속 진행
5. 충돌 데이터로 테스트: 두 값을 모두 source와 함께 보존하고, 확인된 발견과 논쟁적 발견을 분리

도메인: 1(Agent architecture), 2(tool과 MCP), 5(context와 안정성)

부록: 기술과 개념

기술	핵심 측면
Claude Agent SDK	AgentDefinition, Agent 루프, `stop_reason`, hook(PostToolUse), Task를 통한 subagent 생성, `allowedTools`
Model Context Protocol(MCP)	MCP 서버, tool, resource, `isError`, tool 설명, `.mcp.json`, 환경 변수
Claude Code	CLAUDE.md 계층, glob 패턴이 있는 `.claude/rules/`, `.claude/commands/`, SKILL.md가 있는 `.claude/skills/`, 계획 모드, `/compact`, `--resume`, `fork_session`
Claude Code CLI	비대화형 모드를 위한 `-p` / `--print`, `--output-format json`, `--json-schema`
Claude API	JSON schema를 사용한 `tool_use`, `tool_choice`("auto"/"any"/forced), `stop_reason`, `max_tokens`, system prompt
Message Batches API	50% 비용 절감, 최대 24시간 window, `custom_id`, 멀티턴 tool 호출 없음
JSON Schema	필수 vs 선택, nullable 필드, enum 타입, "other" + detail, strict mode
Pydantic	schema validation, 의미 오류, validation/retry 루프
built-in tool	Read, Write, Edit, Bash, Grep, Glob — 목적과 선택 기준
Few-shot prompting	모호한 상황을 위한 목표 예시, 새 패턴으로 일반화
prompt chaining	집중된 패스로 순차 분해
context window	token 예산, 점진적 summary, "lost in the middle", 스크래치패드 파일
session 관리	Resume, `fork_session`, 이름 있는 session, context 격리

confidence calibration	필드 수준 점수, 라벨링 세트 기반 calibration, 계층화 sampling

범위 제외 주제

다음 인접 주제는 시험에 **출제되지 않습니다**.

- Claude 모델 fine-tuning 또는 사용자 지정 모델 학습
- Claude API 인증, billing, 계정 관리
- 특정 프로그래밍 언어나 프레임워크의 상세 구현(tool/schema 구성에 필요한 범위를 넘어서는 내용)
- MCP 서버 배포 또는 호스팅(인프라, 네트워킹, 컨테이너 orchestration)
- Claude의 내부 architecture, 학습 과정, 모델 가중치
- Constitutional AI, RLHF, 안전 학습 방법론
- Embedding 모델 또는 vector database 구현 세부사항
- Computer use(브라우저 자동화, 데스크톱 상호작용)
- 이미지 분석 기능(Vision)
- Streaming API 또는 server-sent events
- Rate limiting, quota, 상세 API 비용 계산
- OAuth, API 키 rotation, 인증 프로토콜 세부사항
- Cloud provider별 구성(AWS, GCP, Azure)
- 성능 benchmark 또는 모델 비교 지표
- Prompt caching 구현 세부사항(존재를 아는 것 이상)
- token 카운팅 알고리즘 또는 토크나이제이션 세부사항

준비 권장사항

1. **Claude Agent SDK로 Agent를 구축하세요** — tool 호출, 오류 처리, session 관리를 포함한 전체 Agent 루프를 구현합니다. subagent와 명시적 context 전달을 연습하세요.
2. **실제 프로젝트에 Claude Code를 구성하세요** — CLAUDE.md 계층, `.claude/rules/`의 경로별 규칙, `context: fork`와 `allowed-tools`가 있는 Skill, MCP 서버 통합을 사용합니다.
3. **MCP tool을 설계하고 테스트하세요** — 유사 tool을 구분하는 설명을 작성하고, 범주와 retry 플래그가 있는 structured error를 반환하며, 모호한 사용자 요청에 대해 테스트합니다.
4. **데이터 extraction pipeline을 구축하세요** — JSON schema가 있는 `tool_use`, validation/retry 루프, 선택/nullable 필드, Message Batches API를 통한 batch 처리를 사용합니다.
5. **Prompt Engineering을 연습하세요** — 모호한 시나리오를 위한 few-shot 예시, 명시적 리뷰 기준, 큰 코드 리뷰를 위한 다중 패스 architecture를 추가합니다.
6. **context 관리 패턴을 학습하세요** — 상세 출력에서 사실을 extraction하고, 스크래치패드 파일을 사용하며, context 한도를 처리하기 위해 발견 작업을 subagent에게 위임합니다.

7. **escalation과 human-in-the-loop를 이해하세요** — 언제 escalation할지(정책 공백, 명시적 사용자 요청, 진행 불가)와 confidence 기반 routing workflow를 익힙니다.
8. **실제 시험 전에 연습 시험을 보세요.** 같은 시나리오와 형식을 사용합니다.