Writing Effective Tools for LLM Agents—Using LLM Agents

Anthropic Engineering 문서 요약 (2025)

📋 개요

Model Context Protocol (MCP)는 LLM 에이전트에게 수백 개의 도구를 제공하여 실제 작업을 해결할 수 있게 합니다. 이 문서는 Anthropic이 다양한 에이전트 AI 시스템에서 성능을 개선하기 위해 사용한 가장 효과적인 기술들을 설명합니다.

핵심 메시지:

> "Tools are a new kind of software which reflects a contract between deterministic systems and non-deterministic agents."

🎯 Tool이란 무엇인가?

전통적 소프트웨어 vs 에이전트 도구

측면	전통적 소프트웨어	에이전트 도구
시스템 타입	결정론적 (Deterministic)	비결정론적 (Non-deterministic)
계약 형태	결정론적 ↔ 결정론적	결정론적 ↔ 비결정론적
예측 가능성	동일 입력 → 동일 출력	동일 입력 → 다양한 응답
예시	`getWeather("NYC")`	사용자: "우산 가져가야 해?"

전통적 함수 호출

```
// 항상 정확히 동일한 방식으로 실행
getWeather("NYC")
```

에이전트의 도구 사용

사용자가 "우산을 가져가야 하나요?"라고 물으면:

- ✅ 날씨 도구 호출 가능
- ✅ 일반 지식으로 답변 가능
- ✅ 위치에 대한 명확한 질문 먼저 가능
- ❌ 환각(hallucination) 가능
- ❌ 도구 사용 방법을 파악하지 못할 수 있음

새로운 패러다임

전통적 접근:

- 다른 개발자나 시스템을 위해 함수와 API 작성

에이전트 도구 접근:

- 에이전트를 위해 도구와 MCP 서버 설계
- 에이전트가 다양한 성공 전략을 추구할 수 있는 표면적 증가
- 목표: 광범위한 작업 해결 능력 향상

놀라운 발견:

> "The tools that are most 'ergonomic' for agents also end up being surprisingly intuitive to grasp as humans."

🛠️ 도구를 작성하는 방법

3단계 반복 프로세스

```
1. 프로토타입 구축 (Building a Prototype)
   ↓
2. 평가 실행 (Running an Evaluation)
   ↓
3. 에이전트와 협업 (Collaborating with Agents)
   ↓
   (반복)
```

1️⃣ 프로토타입 구축

시작하기 전에

> "It can be difficult to anticipate which tools agents will find ergonomic and which tools they won't without getting hands-on yourself."

Claude Code로 빠른 프로토타입

제공할 것들:

- 소프트웨어 라이브러리 문서
- API/SDK 문서 (MCP SDK 포함)
- LLM 친화적 문서 (예: `llms.txt` 파일)
- 예시: Anthropic API llms.txt

도구 연결 방법

1) Local MCP Server

```
# Claude Code 연결
claude mcp add <n> <command> [args...]

# Claude Desktop 연결
Settings > Developer
```

2) Desktop Extension (DXT)

```
Settings > Extensions
```

3) Anthropic API 직접 호출

- 프로그래밍 방식 테스팅

초기 테스트

직접 테스트:

- 거친 부분 식별
- 사용 사례와 프롬프트에 대한 직관 구축

사용자 피드백 수집:

- 도구가 활성화해야 하는 사용 사례 이해
- 실제 프롬프트 패턴 파악

2️⃣ 평가 실행

전체 프로세스

Tool Evaluation Cookbook에서 전체 프로세스 확인 가능

A. 평가 작업 생성 (Generating Evaluation Tasks)

강력한 평가 작업의 특징

✅ **좋은 예시:**

1. **복잡한 다단계 작업**

   ```
   "다음 주에 Jane과 최신 Acme Corp 프로젝트에 대해 논의할
   미팅을 잡아줘. 마지막 프로젝트 기획 미팅 노트를 첨부하고
   회의실을 예약해줘."
   ```

   - 여러 도구 호출 필요
   - 컨텍스트 연결 필요
   - 실제 업무와 유사

2. **조사 및 분석**

   ```
   "고객 ID 9182가 단일 구매 시도에 대해 세 번 청구되었다고
   보고했어. 모든 관련 로그 항목을 찾고 동일한 문제의 영향을
   받은 다른 고객이 있는지 확인해줘."
   ```

   - 문제 진단 필요
   - 데이터 분석 필요
   - 영향 범위 파악 필요

3. **의사 결정 지원**

```
"고객 Sarah Chen이 방금 취소 요청을 제출했어.
유지 제안을 준비해줘. 다음을 결정해:
(1) 왜 떠나는지
(2) 어떤 유지 제안이 가장 설득력 있을지
(3) 제안하기 전에 알아야 할 위험 요소"
```

- 다차원적 분석 필요
- 전략적 사고 필요
- 리스크 평가 필요

❌ **약한 예시:**

1. **지나치게 단순**

```
"다음 주에 jane@acme.corp와 미팅 잡아줘."
```

- 단일 도구 호출
- 컨텍스트 불필요
- 복잡성 부족

2. **기계적인 명령**

```
"결제 로그에서 purchase_complete와 customer_id=9182를 검색해."
```

- 사람이 할 법하지 않은 지시
- 도구 특정 용어 사용
- 사고 과정 없음

3. **정보만 요청**

```
"고객 ID 45892의 취소 요청을 찾아줘."
```

- 단순 검색만
- 분석 불필요
- 의사 결정 없음

실제 환경 기반

- 실제 사용 사례에서 영감

- 현실적인 데이터 소스 및 서비스 기반
- 내부 지식 베이스, 마이크로서비스 등
- **피할 것:** 지나치게 단순하거나 표면적인 "샌드박스" 환경

검증 가능한 응답

- 각 프롬프트는 검증 가능한 응답/결과와 쌍을 이룸
- 단순: 정확한 문자열 비교
- 고급: Claude를 판사로 활용

검증자 설계 주의사항:

- ❌ 지나치게 엄격한 검증자 피하기
- ❌ 형식, 구두점, 유효한 대안 표현 차이로 정답 거부하지 않기

선택적: 예상 도구 호출 지정

- 에이전트가 호출해야 하는 도구 지정 가능
- 각 도구의 목적 파악 성공 여부 측정
- **주의:** 과도하게 지정하거나 전략에 과적합하지 않기
- 올바른 작업 해결에 여러 유효한 경로가 있을 수 있음

B. 평가 실행 (Running the Evaluation)

프로그래밍 방식 실행

```
# 직접 LLM API 호출 사용
# 간단한 에이전트 루프
while not task_complete:
    # LLM API 호출
    response = call_llm(prompt, tools)

    # 도구 호출
    if response.has_tool_calls:
        tool_results = execute_tools(response.tool_calls)

    # 다음 반복
```

구조:

- `while` 루프로 LLM API와 도구 호출 교대
- 각 평가 작업당 하나의 루프

- 각 평가 에이전트에게 단일 작업 프롬프트와 도구 제공

시스템 프롬프트 권장사항

포함할 것:

1. **구조화된 응답 블록** (검증용)
2. **추론 블록** (Reasoning)
3. **피드백 블록** (Feedback)

중요:

> "Instructing agents to output these before tool call and response blocks may increase LLMs' effective intelligence by triggering chain-of-thought (CoT) behaviors."

Claude 사용 시:

- Interleaved Thinking 활성화
- "즉시 사용 가능한" 유사 기능
- 에이전트가 특정 도구를 호출하거나 호출하지 않는 이유 조사
- 도구 설명 및 스펙의 개선 영역 강조

수집할 메트릭

메트릭	목적	활용
최상위 정확도	작업 성공률	전체 성능 측정
개별 도구 호출 시간	도구 효율성	병목 지점 식별
전체 작업 런타임	총 성능	최적화 필요 영역
총 도구 호출 횟수	효율성	통합 기회 발견
총 토큰 소비	비용 및 효율성	컨텍스트 최적화
도구 오류	신뢰성	문제 영역 식별

도구 호출 추적의 가치:

- 에이전트가 추구하는 일반적 워크플로우 드러냄
- 도구 통합 기회 제공

C. 결과 분석 (Analyzing Results)

에이전트를 파트너로 활용

에이전트가 찾아낼 수 있는 것:

- 모순되는 도구 설명
- 비효율적인 도구 구현
- 혼란스러운 도구 스키마

중요한 주의사항:

> "What agents omit in their feedback and responses can often be more important than what they include."

이유:

- LLM이 항상 **의도한 대로 말하지 않음**
- 에이전트가 올바른 답과 전략을 반드시 아는 것은 아님

분석 방법

1. 에이전트가 막히거나 혼란스러워하는 곳 관찰

```
어디서 실패했는가?
왜 특정 도구를 선택하지 않았는가?
무엇이 에이전트를 혼란스럽게 했는가?
```

2. 추론 및 피드백(CoT) 읽기

```
평가 에이전트의 추론 과정 검토
거친 부분 식별
논리적 점프나 혼란 찾기
```

3. 원시 트랜스크립트 검토

```
도구 호출 및 응답 포함
CoT에서 명시적으로 설명되지 않은 행동 포착
숨겨진 패턴 발견
```

4. 도구 호출 메트릭 분석

많은 중복 도구 호출:

- → 페이지네이션 조정 필요
- → 토큰 제한 매개변수 재설정

많은 도구 오류 (유효하지 않은 매개변수):

- → 더 명확한 도구 설명 필요
- → 더 나은 예시 필요

실제 사례 - Web Search Tool:

```
문제 발견:
Claude가 불필요하게 query 매개변수에 "2025"를 추가
→ 검색 결과 편향
→ 성능 저하

해결:
도구 설명 개선으로 올바른 방향 안내
```

5. 에이전트와 협업하여 도구 개선

프로세스:

```
1. 평가 에이전트의 트랜스크립트 연결
   ↓
2. Claude Code에 붙여넣기
   ↓
3. Claude가 트랜스크립트 분석
   ↓
4. Claude가 도구 리팩토링
   ↓
5. 도구 구현과 설명 일관성 유지
```

Claude의 전문성:

- 트랜스크립트 분석 전문가
- 여러 도구를 한 번에 리팩토링
- 새 변경 사항 적용 시 자체 일관성 보장

Anthropic의 경험:

> "Most of the advice in this post came from repeatedly optimizing our internal tool implementations with Claude Code."

평가 설정:

- 내부 워크스페이스 기반
- 실제 프로젝트, 문서, 메시지 포함
- 복잡성이 내부 워크플로우 반영

검증:

- Held-out test sets 사용
- "훈련" 평가에 과적합 방지

결과:

- 전문가가 수동으로 작성한 도구 구현 이상의 성능 개선
- Claude 자체가 생성한 도구보다도 추가 성능 개선

💡 효과적인 도구 작성 원칙

1. 에이전트에게 적합한 도구 선택

더 많은 도구 ≠ 더 나은 결과

일반적인 오류:

> "Tools that merely wrap existing software functionality or API endpoints—whether or not the tools are appropriate for agents."

근본적 차이: Affordances (행동 유도성)

측면	전통 소프트웨어	LLM 에이전트
메모리	저렴하고 풍부함	제한된 컨텍스트
정보 처리	한 번에 하나씩 효율적	토큰 단위로 순차 처리
검색 방식	반복문으로 빠른 검색	모든 항목을 읽어야 함

주소록 검색 예시

❌ **나쁜 접근 (전통 소프트웨어 패턴):**

```
# list_contacts 도구
def list_contacts():
    return all_contacts  # 모든 연락처 반환

# 에이전트가 해야 할 일:
# 1. 모든 연락처 받기
# 2. 토큰 단위로 하나씩 읽기 (비효율!)
# 3. 관련 없는 정보로 컨텍스트 낭비
```

인간 비유:

> "Imagine searching for a contact in your address book by reading each page from top-to-bottom—that is, via brute-force search."

✅ **좋은 접근 (에이전트 친화적):**

```
# search_contacts 도구
def search_contacts(query: str):
    return relevant_contacts  # 관련 연락처만 반환

# 또는
# message_contact 도구
def message_contact(name: str, message: str):
    # 검색과 메시징을 한 번에 처리
```

인간 비유:

> "Skip to the relevant page first (perhaps finding it alphabetically)."

권장 접근법

시작:

- 소수의 사려 깊은 도구
- 특정 고영향 워크플로우 타겟
- 평가 작업과 일치

확장:

- 거기서부터 스케일업

도구 통합 (Consolidation)

도구는 기능을 통합할 수 있음:

- 여러 개별 작업 처리 (내부에서)
- 여러 API 호출 처리 (내부에서)
- 관련 메타데이터로 응답 강화
- 자주 연결되는 다단계 작업을 단일 도구 호출로 처리

예시 1: 이벤트 스케줄링

```
❌ 나쁜 접근:
- list_users
- list_events
- create_event

✅ 좋은 접근:
- schedule_event
  (가용성 찾기 + 이벤트 스케줄링)
```

예시 2: 로그 분석

```
❌ 나쁜 접근:
- read_logs
  (모든 로그 반환)

✅ 좋은 접근:
- search_logs
  (관련 로그 라인 + 주변 컨텍스트만 반환)
```

예시 3: 고객 컨텍스트

```
❌ 나쁜 접근:
- get_customer_by_id
- list_transactions
- list_notes

✅ 좋은 접근:
- get_customer_context
  (고객의 모든 최근 & 관련 정보를 한 번에 컴파일)
```

명확한 목적

각 도구는 명확하고 구별되는 목적을 가져야 함:

도구가 해야 할 것:

- 인간이 동일한 리소스에 접근할 때 작업을 세분화하고 해결하는 방식과 동일하게 에이전트가 작업 세분화 및 해결 가능
- 중간 출력으로 소비되었을 컨텍스트 감소

너무 많거나 중복되는 도구:

- 에이전트가 효율적인 전략 추구를 방해
- 혼란 야기

결론:

> "Careful, selective planning of the tools you build (or don't build) can really pay off."

---

2. 도구 네임스페이스 (Namespacing)

문제 상황

- AI 에이전트가 수십 개의 MCP 서버에 접근
- 수백 개의 다른 도구 (다른 개발자의 도구 포함)
- 기능이 겹치거나 목적이 모호한 도구
- → 에이전트가 어떤 도구를 사용할지 혼란

해결책: Namespacing

정의:

> "Grouping related tools under common prefixes"

MCP 클라이언트:

- 기본적으로 때때로 이 작업 수행

네이밍 전략:

1) 서비스별 네임스페이스

```
asana_search
jira_search
slack_search
```

2) 리소스별 네임스페이스

```
asana_projects_search
asana_users_search
asana_tasks_search
```

효과:

- 많은 도구 간의 경계 명확화
- 에이전트가 적절한 시기에 올바른 도구 선택

Prefix vs Suffix 네이밍

발견:

- Prefix 기반 vs Suffix 기반 선택이 도구 사용 평가에 **비사소한 영향**
- LLM마다 효과 다름

권장:

> "Choose a naming scheme according to your own evaluations."

이점 요약

네임스페이싱의 효과:

1. **도구 및 도구 설명 수 감소**
    - 에이전트 컨텍스트에 로드되는 정보 감소
2. **에이전트 계산 오프로드**
    - 에이전트 컨텍스트 → 도구 호출 자체로 이동
3. **에이전트 실수 위험 감소**
    - 잘못된 도구 호출 감소
    - 잘못된 매개변수로 올바른 도구 호출 감소
    - 너무 적은 도구 호출 감소
    - 도구 응답 잘못 처리 감소

---

3. 도구에서 의미 있는 컨텍스트 반환

고신호 정보만 반환

원칙:

> "Tool implementations should take care to return only high signal information back to agents."

우선순위:

- ✅ 컨텍스트 관련성 (Contextual relevance)
- ✅ 유연성 (Flexibility)

피할 것:

- ❌ 저수준 기술 식별자

예시:

❌ 피할 것	✅ 선호할 것
`uuid`	`name`
`256px_image_url`	`image_url`
`mime_type`	`file_type`

이유:

- `name`, `image_url`, `file_type` 같은 필드가 에이전트의 다운스트림 액션과 응답에 직접 정보 제공 가능성 훨씬 높음

자연어 식별자의 힘

발견:

> "Agents tend to grapple with natural language names, terms, or identifiers significantly more successfully than they do with cryptic identifiers."

개선 사례:

```
❌ Before: UUID 사용
user_id: "a7f3d9e2-8c4b-11eb-8dcd-0242ac130003"

✅ After: 의미 있는 식별자
user_id: 12345
user_name: "Jane Doe"
```

효과:

- Claude의 검색 작업 정밀도 크게 향상
- 환각(hallucination) 감소

유연성과 효율성의 균형: response_format

문제:

- 일부 경우 에이전트가 두 가지 모두 필요
    - 자연어 식별자 (사람 친화적)
    - 기술 식별자 (다운스트림 도구 호출용)

해결책: response_format enum 매개변수

```
enum ResponseFormat {
   DETAILED = "detailed",   // 모든 정보 포함
   CONCISE = "concise"      // 핵심 정보만
}
```

사용 예시:

```
# 초기 검색
search_user(name='jane', response_format='concise')
→ 자연어 정보만 반환

# 후속 작업
send_message(id=12345, response_format='detailed')
→ 기술 식별자 포함 반환
```

실제 예시: Slack 도구

Detailed 응답 (206 토큰):

```
{
  "channel_id": "C1234567890",
  "channel_name": "general",
  "thread_ts": "1234567890.123456",
  "user_id": "U0987654321",
  "user_name": "Jane Doe",
  "message": "Meeting at 3pm",
  "timestamp": "2025-01-15T15:00:00Z",
  "reactions": [...],
  "replies_count": 5
}
```

Concise 응답 (72 토큰):

```
{
  "channel": "general",
  "user": "Jane Doe",
  "message": "Meeting at 3pm",
  "time": "3:00 PM"
}
```

토큰 절약: ~66% (206 → 72 토큰)

설계 고려사항:

- `thread_ts`는 스레드 답글 가져오기에 필요
- `thread_ts`와 기타 ID는 "detailed"에서 검색 가능
- "concise"는 스레드 내용만 반환, ID 제외
- 추가 도구 호출 필요 시에만 "detailed" 사용

GraphQL 스타일 확장

더 큰 유연성:

```
enum ResponseFormat {
  DETAILED = "detailed",
  CONCISE = "concise",
  MINIMAL = "minimal",
  CUSTOM = "custom"  // 특정 필드 선택
}
```

GraphQL 유사:

- 정확히 원하는 정보 조각 선택 가능

응답 구조 형식

선택지:

- XML
- JSON
- Markdown

중요:

> "There is no one-size-fits-all solution."

이유:

- LLM은 next-token prediction으로 훈련됨
- 훈련 데이터와 일치하는 형식에서 더 나은 성능
- 최적 응답 구조는 작업과 에이전트에 따라 크게 다름

권장:

> "Select the best response structure based on your own evaluation."

---

4. 토큰 효율성을 위한 도구 응답 최적화

컨텍스트의 질과 양

모두 중요:

- ✅ 컨텍스트 품질 최적화
- ✅ 컨텍스트 양 최적화

구현 기법

권장 기능 조합:

1. **Pagination (페이지네이션)**
2. **Range Selection (범위 선택)**
3. **Filtering (필터링)**
4. **Truncation (잘라내기)**

+ 합리적 기본 매개변수 값

Claude Code 예시:

- 기본적으로 도구 응답을 **25,000 토큰**으로 제한

미래 전망:

> "We expect the effective context length of agents to grow over time, but the need for context-efficient tools to remain."

Truncation 구현 시 가이던스

응답을 잘라낼 경우:

- 유용한 지침으로 에이전트 안내
- 더 토큰 효율적인 전략 권장

예시 권장 전략:

```
"단일 광범위 검색 대신 여러 개의 작고 타겟팅된 검색을 수행하세요."
```

에러 응답 프롬프트 엔지니어링

도구 호출 오류 발생 시 (예: 입력 검증):

❌ **불친절한 에러:**

```
Error: INVALID_PARAMETER
Code: 400
```

✅ **친절한 에러:**

```
Error: 'user_id' 매개변수가 필요합니다.

제공된 것: name="Jane"
필요한 것: user_id (숫자, 예: 12345)

힌트: search_user 도구를 먼저 사용하여 user_id를 찾으세요.
```

친절한 에러 특징:

- 구체적 (Specific)
- 실행 가능 (Actionable)
- 개선 방향 명확

피할 것:

- 불투명한 에러 코드
- 스택 트레이스만

시각적 예시

Truncated 응답:

```
{
  "results": [...],
  "truncated": true,
  "message": "결과가 25,000 토큰으로 제한되었습니다.
              더 구체적인 검색을 위해 필터를 사용하세요.",
  "total_results": 1000,
  "showing": 50
}
```

---

5. 도구 설명 프롬프트 엔지니어링

가장 효과적인 개선 방법

핵심:

> "One of the most effective methods for improving tools: prompt-engineering your tool descriptions and specs."

이유:

- 도구 설명과 스펙이 에이전트 컨텍스트에 로드됨
- 효과적인 도구 호출 행동으로 에이전트를 집단적으로 조종 가능

작성 시 생각해야 할 것

질문:

> "How would you describe your tool to a new hire on your team?"

고려사항:

- 암묵적으로 가져오는 컨텍스트
    - 전문 쿼리 형식
    - 틈새 용어 정의
    - 기본 리소스 간의 관계
- **→ 명시적으로 만들기**

명확성과 명확한 데이터 모델

모호성 피하기:

- 예상 입력과 출력을 명확히 설명

- 엄격한 데이터 모델로 강제

매개변수 네이밍:

```
❌ 나쁜 예:
user: str  # 모호함 - 이름? ID? 이메일?

✅ 좋은 예:
user_id: int  # 명확함
user_name: str  # 명확함
user_email: str  # 명확함
```

평가로 영향 측정

중요성:

> "With your evaluation you can measure the impact of your prompt engineering with greater confidence."

효과:

> "Even small refinements to tool descriptions can yield dramatic improvements."

실제 사례: SWE-bench Verified

- Claude Sonnet 3.5가 state-of-the-art 성능 달성
- 도구 설명의 정밀한 개선 후
- 에러율 극적 감소
- 작업 완료율 향상

추가 리소스

Best Practices for Tool Definitions:

- Developer Guide

Claude용 도구 구축 시:

- System Prompt
  - 도구가 어떻게 동적으로 Claude의 시스템 프롬프트에 로드되는지 이해

MCP 서버용 도구 작성 시:

- Tool Annotations

- 오픈 월드 접근 필요 도구 공개
- 파괴적 변경을 수행하는 도구 공개

🎓 Anthropic의 실제 경험

반복적, 평가 주도 프로세스

Anthropic이 이 포스트의 조언을 얻은 방법:

> "Repeatedly optimizing our internal tool implementations with Claude Code."

평가 환경:

- 내부 워크스페이스 기반
- 내부 워크플로우의 복잡성 반영
- 실제 프로젝트, 문서, 메시지 포함

검증 방법:

- Held-out test sets 의존
- "훈련" 평가에 과적합 방지

놀라운 발견:

- "전문가" 도구 구현 이상의 추가 성능 개선 추출 가능
- 수동 작성 (연구원) vs Claude 생성 → 둘 다 개선 가능

평가 결과: Human vs Claude-Optimized

Slack MCP Server:

- Claude 최적화 버전이 인간 작성 버전보다 우수한 테스트 정확도

Asana MCP Server:

- Claude 최적화 버전이 인간 작성 버전보다 우수한 테스트 정확도

🚀 미래 전망

소프트웨어 개발 패러다임 전환

기존:

- 예측 가능한, 결정론적 패턴

새로운:

- 비결정론적 패턴
- 에이전트를 위한 도구 설계

일관된 패턴

효과적인 도구의 특징:

1. **의도적이고 명확하게 정의됨**
    - 목적이 명확
    - 설명이 상세
    - 매개변수가 명확
2. **에이전트 컨텍스트를 신중하게 사용**
    - 고신호 정보만 반환
    - 토큰 효율적
    - 적절한 상세도
3. **다양한 워크플로우에서 결합 가능**
    - 모듈화
    - 재사용 가능
    - 명확한 경계
4. **에이전트가 실제 작업을 직관적으로 해결 가능**
    - 자연스러운 인터페이스
    - 인간의 문제 해결 방식과 일치

진화하는 메커니즘

예상되는 변화:

- MCP 프로토콜 업데이트
- 기본 LLM 업그레이드
- 에이전트-세계 상호작용 메커니즘 진화

대응 방법:

> "With a systematic, evaluation-driven approach to improving tools for agents, we can ensure that as agents become more capable, the tools they use will evolve alongside them."

📊 핵심 체크리스트

✅ 도구 설계 체크리스트

- ☐ **명확한 목적**: 각 도구가 구별되는 목적을 가지는가?
- ☐ **에이전트 친화적**: 인간의 문제 해결 방식과 일치하는가?
- ☐ **적절한 네임스페이싱**: 도구 간 경계가 명확한가?
- ☐ **고신호 응답**: 관련 정보만 반환하는가?
- ☐ **토큰 효율적**: 페이지네이션/필터링/잘라내기를 구현했는가?
- ☐ **명확한 설명**: 새 팀원에게 설명하듯 작성했는가?
- ☐ **명확한 매개변수**: 매개변수 이름이 명확한가?

✅ 평가 체크리스트

- ☐ **실제 작업 기반**: 평가 작업이 실제 사용 사례를 반영하는가?
- ☐ **복잡성**: 충분히 복잡하여 도구를 제대로 테스트하는가?
- ☐ **검증 가능**: 응답이 검증 가능한가?
- ☐ **메트릭 추적**: 정확도, 런타임, 토큰, 에러를 추적하는가?
- ☐ **Held-out 테스트**: 과적합 방지를 위한 테스트 세트가 있는가?

✅ 개선 체크리스트

- ☐ **에이전트 협업**: Claude Code로 도구 분석 및 개선하는가?
- ☐ **반복 프로세스**: 평가 → 분석 → 개선을 반복하는가?
- ☐ **CoT 분석**: 에이전트의 추론 과정을 검토하는가?
- ☐ **트랜스크립트 검토**: 원시 도구 호출과 응답을 확인하는가?
- ☐ **메트릭 분석**: 패턴과 개선 기회를 찾는가?

💡 핵심 인사이트 요약

1. **도구는 새로운 소프트웨어**
    - 결정론적 시스템과 비결정론적 에이전트 간의 계약
2. **에이전트를 위한 설계**
    - 함수/API를 작성하듯 하지 말 것
    - 에이전트의 affordances 고려
3. **평가 주도 개발**
    - 프로토타입 → 평가 → 협업 → 개선
    - 반복이 핵심
4. **에이전트와 협업**
    - Claude Code로 도구 분석 및 개선
    - 에이전트가 더 나은 도구를 만들도록 도움
5. **의도적 도구 선택**
    - 더 많은 도구 ≠ 더 나은 결과
    - 소수의 사려 깊은 도구 〉 많은 일반 도구
6. **컨텍스트가 귀중함**
    - 고신호 정보만 반환
    - 토큰 효율성 최적화
7. **설명이 중요함**
    - 작은 개선이 큰 영향
    - 새 팀원에게 설명하듯 작성

마지막 말:

> "The tools that are most 'ergonomic' for agents also end up being surprisingly intuitive to grasp as humans."

에이전트에게 좋은 도구는 인간에게도 직관적입니다.