Effective Context Engineering for AI Agents

Anthropic Engineering 문서 요약 (2025)

📋 개요

Context Engineering은 Prompt Engineering의 자연스러운 진화입니다. AI 에이전트가 다중 턴 추론과 더 긴 작업을 처리하면서, 단순히 올바른 프롬프트를 작성하는 것을 넘어 **모델이 추론 중에 보는 전체 정보 세트를 관리**하는 것이 핵심이 되었습니다.

핵심 메시지:

> "Context engineering is the natural progression of prompt engineering."

근본적 질문:

> "What configuration of context is most likely to generate our model's desired behavior?"

🎯 Prompt Engineering vs Context Engineering

정의 비교

측면	Prompt Engineering	Context Engineering
초점	지시사항 작성 방법 (How)	정보 큐레이션 (What)
범위	주로 시스템 프롬프트	전체 컨텍스트 상태
시간성	일회성, 이산적	반복적, 순환적
대상	단일 작업 최적화	다중 턴 에이전트 관리
문제	"무엇을 할까?"	"어떤 리소스를 줄까?"

비유로 이해하기

Prompt Engineering:

> "무엇을 할지 말해주기" (Telling someone "what to do")

예시:

```
"이 문서를 재무 지표에 초점을 맞춰 3개 bullet point로 요약해줘."
```

- 명확한 지시사항이면 모델이 잘 수행

Context Engineering:

> "어떤 리소스를 줄지 결정하기" (Deciding "what resources to give them")

예시:

```
지시사항만이 아니라 결정해야 할 것:
- 전체 문서를 보여줄까?
- 마지막 3개 섹션만?
- 도구가 이미 준비한 요약본?
```

- 잘못 선택된 컨텍스트는 완벽한 프롬프트에도 혼란 야기

🧠 Context의 정의와 중요성

Context란?

기술적 정의:

> "The set of tokens included when sampling from a large-language model (LLM)."

포괄적 정의:
Context는 추론 중 모델이 보는 모든 정보:

- System instructions (시스템 지시사항)
- Tools (도구)
- Model Context Protocol (MCP)

- External data (외부 데이터)
- Message history (메시지 기록)
- Memory files (메모리 파일)
- Domain knowledge (도메인 지식)
- Tool outputs (도구 출력)

Engineering 문제

최적화 목표:

> "Optimizing the utility of those tokens against the inherent constraints of LLMs in order to consistently achieve a desired outcome."

핵심 과제:

- 제한된 컨텍스트 윈도우
- 끊임없이 진화하는 정보 우주
- 각 턴마다 순환적으로 정제해야 하는 데이터

⚠️ 왜 Context Engineering이 필요한가?

1. 제한된 주의력 (Limited Attention)

인간과의 유사성:

> "LLMs behave like humans: they can't recall everything if overloaded."

잘못된 가정:

```
더 많은 토큰 ≠ 더 높은 정확도
```

현실:

- 정보가 많을수록 집중하고 세부사항을 정확히 기억하기 어려움
- Transformer 아키텍처의 한계: 모든 토큰이 다른 모든 토큰에 "주의"를 기울여야 함
- 컨텍스트가 커질수록 주의력이 빠르게 부담

2. Context Rot (컨텍스트 부패)

정의:

> "As context length grows, retrieval precision falls."

문제 상황:

```
100페이지 로그 추가
→ 중요한 단일 세부사항이 숨겨짐
→ 검색 정밀도 하락
```

메커니즘:

- 컨텍스트 윈도우를 단순히 늘린다고 성능 보장 안 됨
- 정보가 많아질수록 "신호 대 잡음" 비율 저하
- 관련 정보를 찾기 어려워짐

3. 진화하는 작업 (Evolving Tasks)

에이전트 루프의 특성:

```
에이전트 루프 실행
→ 새로운 데이터 생성
→ 도구 출력 누적
→ 컨텍스트에 정보 축적
```

Context Engineering 없이:

- 윈도우가 잡음으로 가득 참
- 관련 정보가 묻힘
- 에이전트 성능 저하

4. 초기 LLM vs 현대 에이전트

초기 LLM (2022-2023):

- 대부분 일회성 분류 또는 텍스트 생성
- 프롬프트 엔지니어링이 주요 작업
- 단순한 채팅 상호작용

현대 에이전트 (2024-2025):

- 다중 턴 추론 (Multiple turns of inference)
- 긴 시간 지평 (Longer time horizons)
- 자율적 작업 수행
- 외부 도구 사용
- 복잡한 다단계 작업

필요한 변화:

> "We need strategies for managing the entire context state."

💡 Context Engineering의 핵심 원칙

1. System Prompts: Goldilocks Zone (골디락스 존)

두 가지 극단 실패 모드

❌ 극단 1: 과도하게 경직됨 (Too Rigid)

```
# 복잡하고 취약한 하드코딩된 로직
if user_query.contains("weather"):
    if location == "NYC":
        use_weather_tool_nyc()
    elif location == "LA":
        use_weather_tool_la()
    # ... 수백 개의 if-else
```

문제:

- 취약성 (Fragility)
- 유지보수 복잡성 증가
- 확장 불가능
- 예상치 못한 상황 처리 불가

❌ 극단 2: 과도하게 모호함 (Too Vague)

```
"사용자를 도와줘."
"최선을 다해봐."
"적절히 행동해."
```

문제:

- 구체적 신호 부족
- 잘못된 공유 컨텍스트 가정
- 일관되지 않은 행동
- 예측 불가능한 결과

✅ 최적의 고도 (Optimal Altitude)

균형잡힌 접근:

> "Specific enough to guide behavior effectively, yet flexible enough to provide the model with strong heuristics."

좋은 예시:

```
"사용자가 수치 계산을 요청하면 Calculator 도구를 사용하세요.
텍스트 조회는 KnowledgeBase 도구를 사용하세요."
```

특징:

- 명확한 가이드라인
- 유연한 적용
- 강력한 휴리스틱 제공
- 확장 가능

작성 원칙:

- 극도로 명확하게 (Extremely clear)
- 단순하고 직접적인 언어 (Simple, direct language)
- 적절한 고도에서 아이디어 제시 (Right altitude)
- 모듈화를 위한 구조화된 섹션

구조화 예시:

```
<instructions>
[핵심 지시사항]
</instructions>

<tools>
[도구 사용 가이드]
</tools>

## Output format
[출력 형식 설명]
```

반복 접근:

- 최소 버전으로 시작
- 테스트 결과 기반 반복
- 점진적 개선

---

2. Just-in-Time (JIT) Context Loading

패러다임 전환

전통적 RAG (Retrieval-Augmented Generation):

```
시작 시 데이터 사전 로드
→ 정적 컨텍스트
→ 모든 가능한 정보 포함
→ 대부분 사용되지 않음
```

JIT 전략:

```
필요한 시점에 동적으로 검색
→ 도구를 사용한 런타임 페칭
→ 가장 관련성 높은 데이터만
→ 추론에 필요한 정확한 순간
```

JIT의 핵심 특징

1) 자율적, 동적 컨텍스트 관리

- 에이전트가 무엇이 필요한지 결정
- 필요한 시점에만 검색

- 불필요한 정보 배제

2) 런타임 페칭 (Runtime Fetching)

```
# 에이전트가 도구를 사용하여 검색
search_tool(query="Q4 revenue")
read_file(path="reports/2024-Q4.pdf")
api_call(endpoint="/customer/12345")
```

도구 유형:

- File paths (파일 경로)
- Queries (쿼리)
- APIs (API 호출)

3) 효율성과 인지 개선

- 메모리 효율성 극적 향상
- 유연성 증대
- 인간의 외부 조직 시스템 모방
    - 파일 시스템
    - 북마크
    - 메모 시스템

4) 하이브리드 검색

Claude Code 예시:

```
정적 사전 로드 데이터 (Static Pre-loaded)
+
JIT 동적 검색 (JIT Dynamic Retrieval)
=
최적의 속도와 다양성
```

장점:

- 빠른 접근을 위한 공통 컨텍스트 사전 로드
- 특정 필요에 대한 동적 검색
- 최상의 성능

Engineering 도전과제

필요한 것:

- 신중한 도구 설계
- 사려 깊은 엔지니어링

방지해야 할 것:

- 에이전트의 도구 오용
- Dead-end 추적
- 컨텍스트 낭비

예시 문제:

```
❌ 나쁜 도구 설계:
list_all_files() → 10,000개 파일 반환
→ 컨텍스트 폭발
→ 에이전트 혼란

✅ 좋은 도구 설계:
search_files(query, limit=10)
→ 관련 파일 10개만 반환
→ 효율적 컨텍스트 사용
```

---

3. Tools: Distinct and Minimal

명확한 도구 경계

원칙:

> "Keep tools distinct."

❌ **나쁜 예:**

```
fetch_news_api()
get_news_articles()
```

- 두 도구가 모두 뉴스 가져옴
- 중복 기능
- 에이전트 혼란

✅ **좋은 예:**

```
search_news(query, date_range)  # 검색
fetch_article(article_id)       # 특정 아티클
```

- 명확히 구분된 목적
- 각각의 역할 명확
- 효율적 선택 가능

도구가 컨텍스트에 미치는 영향

도구 설명이 컨텍스트에 로드됨:

- 각 도구 설명 = 토큰 소비
- 많은 도구 = 많은 토큰
- 중복 도구 = 낭비된 컨텍스트

최적화 전략:

- 필수 도구만 포함
- 명확한 설명 작성
- 통합 가능한 것은 통합
- 네임스페이스로 조직화

---

4. External Data: Curated and Filtered

데이터 선택의 중요성

목표:

> "Maximize useful signal while minimizing noise."

질문:

```
이 정보가 에이전트의 다음 결정에 도움이 되는가?
```

큐레이션 전략:

1) 관련성 필터링

```
# 모든 고객 데이터 대신
relevant_customer_data = filter_by_relevance(
    customer_id=current_customer,
    context=conversation_topic,
    recency=last_30_days
)
```

2) 요약 및 압축

```
# 100페이지 문서 대신
document_summary = summarize(
    document=full_document,
    focus=query_topic,
    max_length=500_tokens
)
```

3) 계층적 접근

```
레벨 1: 고수준 요약 (항상 포함)
레벨 2: 중요 세부사항 (필요시 포함)
레벨 3: 전체 데이터 (명시적 요청시만)
```

---

5. Message History: Strategic Summarization

긴 대화의 문제

문제:

```
Turn 1: 100 tokens
Turn 2: 150 tokens
Turn 3: 200 tokens
...
Turn 50: 누적 10,000+ tokens
```

컨텍스트 윈도우 소진:

- 이전 대화가 대부분의 컨텍스트 차지
- 새로운 정보 입력 공간 부족
- 에이전트 성능 저하

해결 전략

1) 주기적 요약

```
if conversation_length > threshold:
    summary = summarize_conversation(
        messages=messages[:-5],  # 최근 5개 제외
        focus=key_topics
    )
    messages = [summary] + messages[-5:]  # 요약 + 최근 5개
```

2) 중요도 기반 유지

```
# 중요한 메시지 우선 유지
important_messages = filter_by_importance(messages)
recent_messages = messages[-10:]
context = important_messages + recent_messages
```

3) 외부 메모리 활용

```
# 장기 메모리 저장
save_to_memory(
    topic="project_requirements",
    content=key_decisions
)

# 필요시 검색
if needs_context:
    retrieve_from_memory(topic="project_requirements")
```

---

6. Tool Outputs: Efficient Processing

도구 출력의 누적 문제

문제 상황:

```
Tool 1: 5,000 tokens 출력
Tool 2: 3,000 tokens 출력
Tool 3: 7,000 tokens 출력
→ 총 15,000 tokens가 컨텍스트 차지
```

해결 전략:

1) Response Format 제어

```
# 이전 문서에서 다룬 개념
search_tool(
    query="user data",
    response_format="concise"  # 핵심만
)
```

2) 도구 내 필터링

```
def search_logs(query, max_results=10):
    # 도구 내부에서 필터링
    all_results = database.search(query)
    relevant = filter_most_relevant(all_results, query)
    return relevant[:max_results]  # 상위 10개만
```

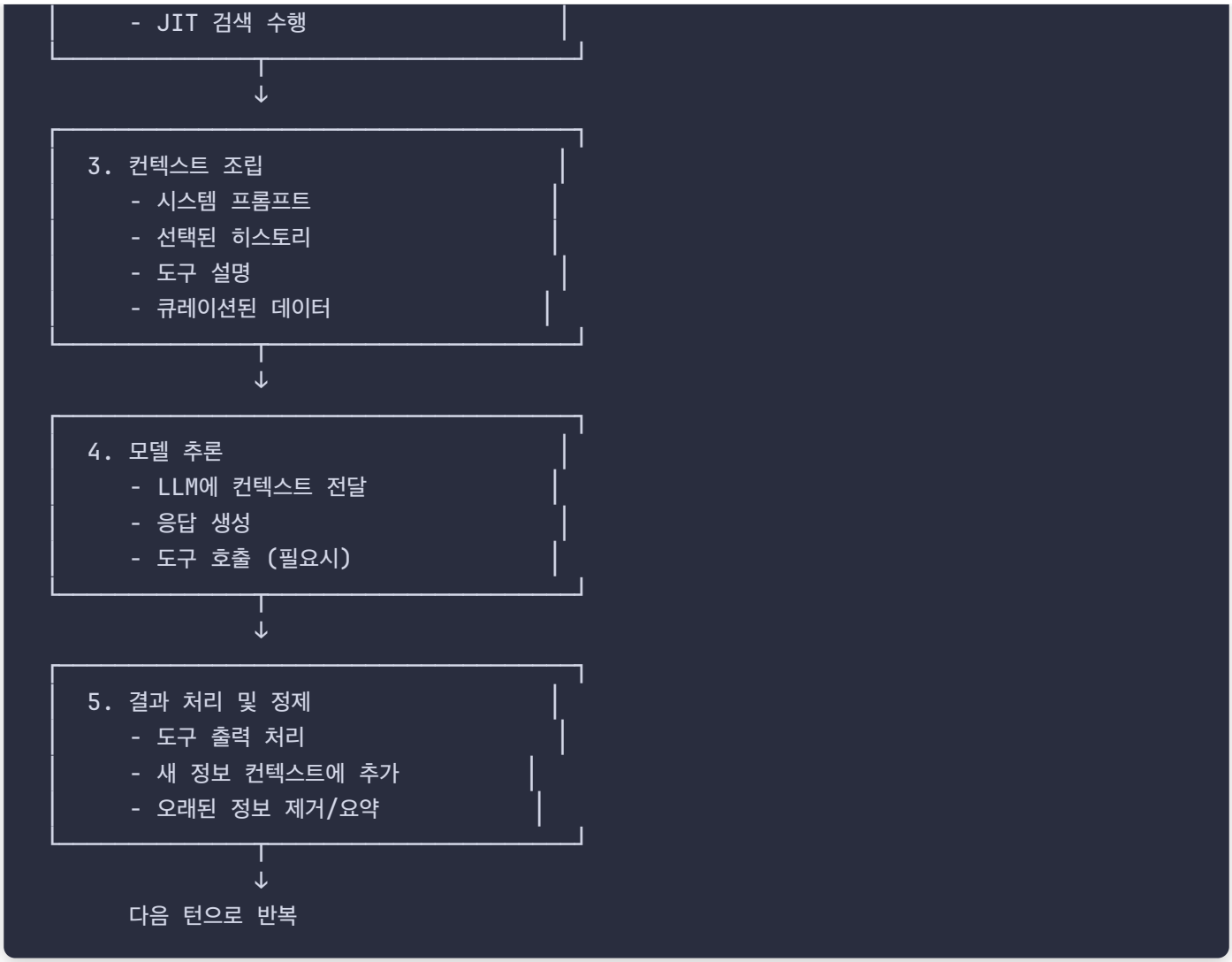
3) 점진적 세부화

```
# 1단계: 개요
overview = get_overview()

# 2단계: 필요시 세부사항
if needs_details:
    details = get_details(specific_id)
```

🔄 Context Engineering 워크플로우

반복적 큐레이션 프로세스



핵심:

> "Context engineering is iterative and the curation phase happens each time we decide what to pass to the model."

🎓 장기 작업을 위한 고급 기법

1. Checkpointing (체크포인팅)

개념:

```
작업 Phase 1 완료
→ 상태 저장 (checkpoint)
→ 컨텍스트 압축/요약
→ Phase 2 시작 (새로운 컨텍스트)
```

구현:

```
class TaskCheckpoint:
    def __init__(self):
        self.phase_summaries = []
        self.key_decisions = []
        self.current_state = {}

    def save_checkpoint(self, phase_name):
        summary = summarize_phase(self.current_state)
        self.phase_summaries.append({
            'phase': phase_name,
            'summary': summary,
            'timestamp': now()
        })
        # 컨텍스트 정리
        self.current_state = compact_state(self.current_state)
```

장점:

- 긴 작업에서 컨텍스트 오버플로우 방지
- 진행 상황 추적
- 필요시 이전 상태로 복원 가능

2. Hierarchical Context (계층적 컨텍스트)

구조:

```
레벨 0: 핵심 목표 및 제약사항 (항상 포함)
  ↓
레벨 1: 현재 Phase 요약 (항상 포함)
  ↓
레벨 2: 최근 작업 히스토리 (선택적)
  ↓
레벨 3: 상세 데이터 (필요시 JIT)
```

예시:

```
context = {
    'core': {  # 항상 포함
        'goal': "Build user authentication system",
        'constraints': ["Use OAuth 2.0", "GDPR compliant"]
    },
```

```
    'phase': {  # 항상 포함
        'current': "Implementing login flow",
        'progress': "60% complete"
    },
    'recent': {  # 최근 10개 작업
        'actions': recent_actions[-10:]
    },
    'details': {  # JIT 검색
        'retrieve_on_demand': True
    }
}
```

3. Context Compaction (컨텍스트 압축)

시점:

- 컨텍스트가 임계값에 도달
- Phase 전환 시
- 중요하지 않은 정보 누적 시

방법:

```
def compact_context(context, target_size):
    # 1. 중요도 평가
    importance_scores = score_importance(context)

    # 2. 낮은 중요도 항목 요약
    low_priority = [item for item in context
                    if importance_scores[item] < threshold]
    summary = summarize(low_priority)

    # 3. 새 컨텍스트 구성
    compacted = {
        'high_priority': high_priority_items,
        'summary': summary,
        'current_focus': current_items
    }

    return compacted
```

📊 Context Engineering 평가

측정 메트릭

메트릭	설명	목표
Context Utilization	사용된 컨텍스트 / 전체 컨텍스트	70-90%
Signal-to-Noise Ratio	관련 정보 / 전체 정보	최대화
Retrieval Precision	필요한 정보를 얼마나 정확히 검색?	>95%
Context Efficiency	작업당 평균 토큰 사용	최소화
Task Completion Rate	컨텍스트 최적화 후 성공률	증가 추세

A/B 테스트 전략

비교 항목:

```
Control: 전통적 RAG (모든 문서 사전 로드)
Treatment: JIT Context Engineering

측정:
- 작업 완료 시간
- 토큰 사용량
- 정확도
- 사용자 만족도
```

🚀 실전 적용 가이드

Step 1: 현재 상태 평가

체크리스트:

- ☐ 현재 평균 컨텍스트 사용량은?
- ☐ 컨텍스트 오버플로우 빈도는?
- ☐ 관련 없는 정보 비율은?
- ☐ 에이전트가 막히는 주요 지점은?

Step 2: 시스템 프롬프트 개선

Action Items:

```
1. 현재 프롬프트 분석
   - 너무 경직적인가?
   - 너무 모호한가?

2. Goldilocks Zone 찾기
   - 명확한 가이드라인 추가
   - 유연성 확보
   - 강력한 휴리스틱 제공

3. 구조화
   - <instructions> 섹션
   - <tools> 섹션
   - ## Output format 섹션
```

Step 3: JIT 검색 구현

우선순위 작업:

```
1. 고빈도 데이터 소스 식별
2. 검색 도구 구축
   - search_documents(query)
   - fetch_customer_data(id)
   - get_relevant_logs(timeframe)
3. 캐싱 전략 수립
4. 도구 설명 최적화
```

Step 4: 도구 정리

작업:

```
1. 중복 도구 식별 및 통합
2. 도구 설명 명확화
3. response_format 매개변수 추가
4. 토큰 제한 구현
```

Step 5: 메시지 히스토리 관리

구현:

```python
class ContextManager:
    def __init__(self, max_tokens=100000):
```

```
        self.max_tokens = max_tokens
        self.messages = []
        self.summaries = []

    def add_message(self, message):
        self.messages.append(message)
        if self.token_count() > self.max_tokens:
            self.compact()

    def compact(self):
        # 오래된 메시지 요약
        old_messages = self.messages[:-10]
        summary = summarize(old_messages)
        self.summaries.append(summary)
        self.messages = self.messages[-10:]
```

Step 6: 모니터링 및 반복

모니터링 대시보드:

```
- 컨텍스트 사용률 추세
- 토큰 소비 패턴
- 작업 성공률
- 평균 응답 시간
- 도구 호출 빈도
```

💡 Best Practices 요약

✅ DO (해야 할 것)

1. **시스템 프롬프트**
    - 명확하고 구조화
    - Goldilocks zone 유지
    - 모듈화
2. **컨텍스트 큐레이션**
    - JIT 검색 우선
    - 신호 대 잡음 최대화
    - 주기적 압축
3. **도구 설계**

- 명확한 경계
- 효율적 출력
- 토큰 제한

4. **메시지 관리**
    - 전략적 요약
    - 중요도 기반 유지
    - 외부 메모리 활용
5. **모니터링**
    - 지속적 측정
    - A/B 테스트
    - 반복 개선

❌ DON'T (하지 말아야 할 것)

1. **과도한 정보**
    - "혹시 몰라" 모든 데이터 포함
    - 사전 필터링 없이 전체 문서 로드
    - 무제한 도구 출력
2. **경직된 프롬프트**
    - 복잡한 if-else 체인
    - 하드코딩된 로직
    - 확장 불가능한 규칙
3. **모호한 지시**
    - "최선을 다해"
    - "적절히 판단해"
    - 구체적 가이드라인 없음
4. **컨텍스트 방치**
    - 무한정 누적
    - 압축 없음
    - 관리 전략 없음
5. **측정 부재**
    - 눈대중 최적화

    - 데이터 없는 결정
    - 반복 없음

🌟 미래 전망

Context Engineering의 진화

현재 (2025):

- 수동 큐레이션
- 규칙 기반 압축
- 개발자 중심 최적화

가까운 미래:

- 자동 컨텍스트 최적화
- AI가 자신의 컨텍스트 관리
- 적응형 압축 전략

장기 비전:

- 무한 컨텍스트 (이론적)
- 완벽한 정보 검색
- 인간 수준 컨텍스트 관리

산업 영향

자율 에이전트:

> "Context engineering will be critical for autonomous agents trusted to perform complex tasks without errors."

도메인 전문가:

- 헬스케어: 낮은 실수 허용도
- 금융: 정밀한 컴플라이언스
- 법률: 정확한 참조 필요

기술 부채 제거:

- 조직 특정 IT 인프라

- 레거시 시스템 이해
- 복잡한 코드베이스 탐색

Enterprise AI:

> "Organizations transition from pilots to production-scale deployments — prompt engineering cannot deliver the accuracy, memory, or governance required in complex environments on its own."

📚 핵심 인사이트

1. **패러다임 전환**
    - Prompt Engineering → Context Engineering
    - "무엇을 할까?" → "무엇을 줄까?"
    - 일회성 → 반복적
2. **제한된 리소스**
    - Context는 critical하지만 finite
    - 더 많은 토큰 ≠ 더 나은 성능
    - 큐레이션이 핵심
3. **JIT의 힘**
    - 정적 → 동적
    - 사전 로드 → 필요시 검색
    - 모든 것 → 필요한 것만
4. **Goldilocks Zone**
    - 너무 경직 ❌
    - 너무 모호 ❌
    - 적절한 균형 ✅
5. **반복적 큐레이션**
    - 매 턴마다 재평가
    - 지속적 최적화
    - 데이터 기반 개선

마지막 말:

> "Context engineering is not just 'prompting 2.0'. It is a discipline of curation. The future of reliable AI agents will depend on how effectively we balance instructions (prompting) with resources (context)."

🎯 실행 체크리스트

즉시 시작 (오늘)

- ☐ 현재 컨텍스트 사용량 측정
- ☐ 시스템 프롬프트 Goldilocks zone 평가
- ☐ 중복 도구 식별

단기 (1주일)

- ☐ JIT 검색 도구 1-2개 구현
- ☐ 메시지 히스토리 압축 전략 수립
- ☐ 모니터링 대시보드 설정

중기 (1개월)

- ☐ 전체 도구 스택 최적화
- ☐ A/B 테스트 실행
- ☐ 계층적 컨텍스트 구조 구현

장기 (3개월)

- ☐ 자동 컨텍스트 최적화 시스템
- ☐ 포괄적 평가 프레임워크
- ☐ 팀 전체 Best Practices 문서화

핵심을 기억하세요:
Context Engineering은 에이전트가 보는 정보를 최적화하는 예술이자 과학입니다. 올바른 정보를 올바른 시간에 제공하는 것이 성공의 열쇠입니다.