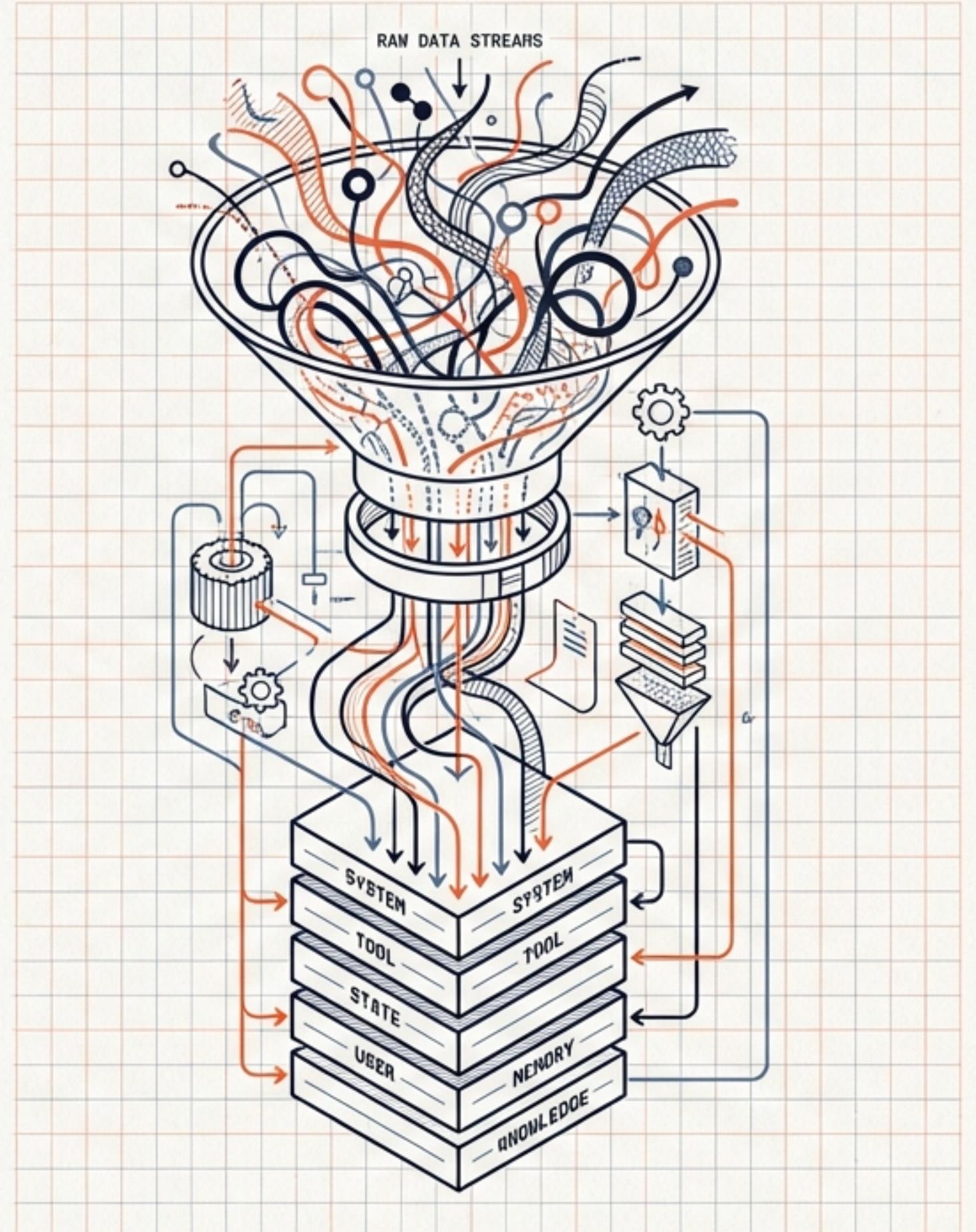


AI 에이전트를 위한 컨텍스트 엔지니어링

Context Engineering for AI Agents

LangChain과 Manus가 제안하는
LLM 에이전트 아키텍처 및 최적화 심층 가이드

Based on the Masterclass by Lance (LangChain) & Peet (Manus)



핵심 문제: 컨텍스트 폭발과 성능 저하 (Context Explosion & Rot)

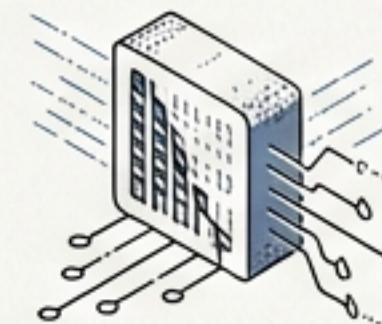
The Loop Paradox (루프의 역설)

에이전트는 도구 호출(Tool calls)과 관찰(Observations)을 반복하며 메시지 기록을 무제한으로 증가시킵니다. (예: Manus의 경우 일반적인 작업에 약 50회의 도구 호출 발생).



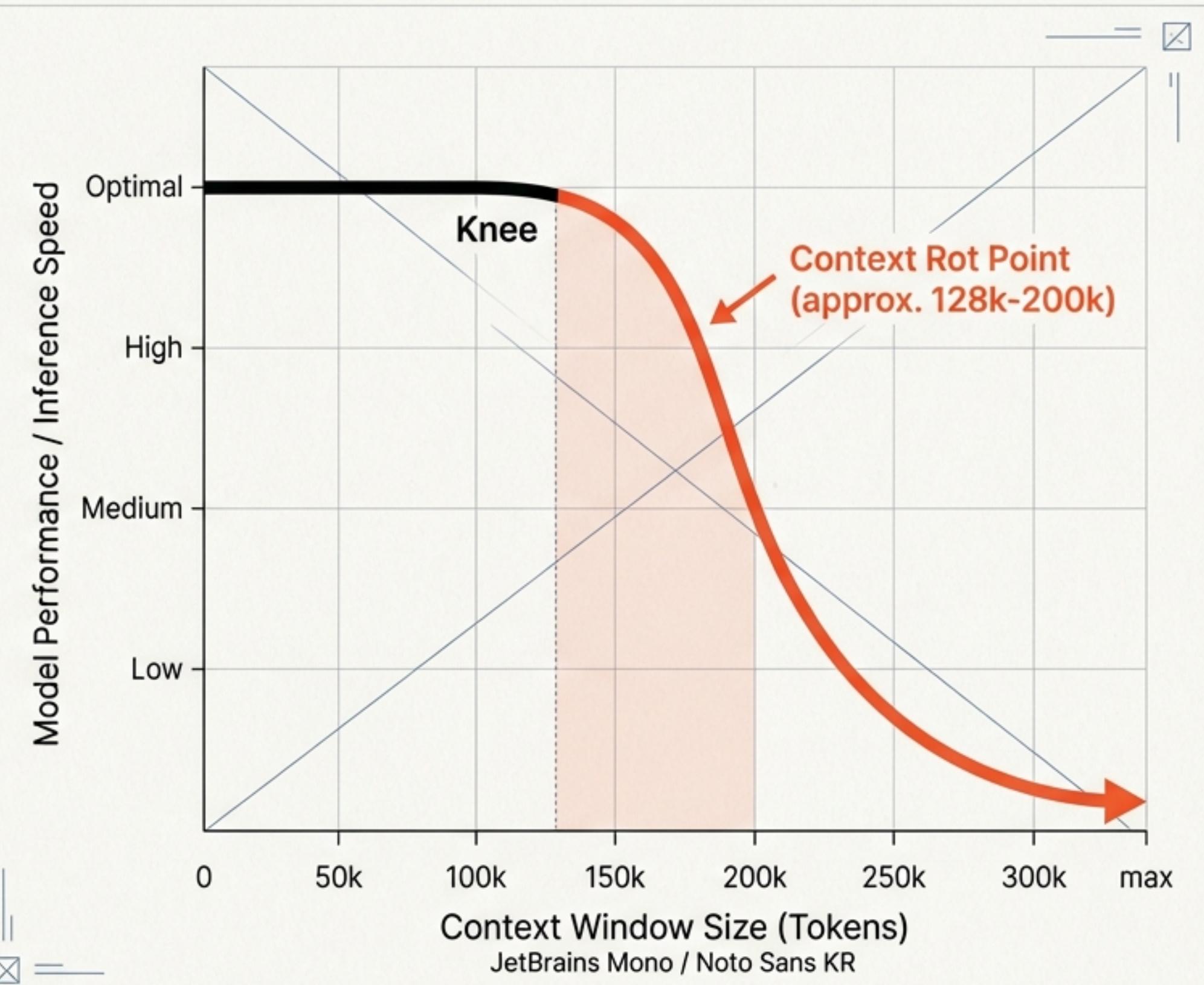
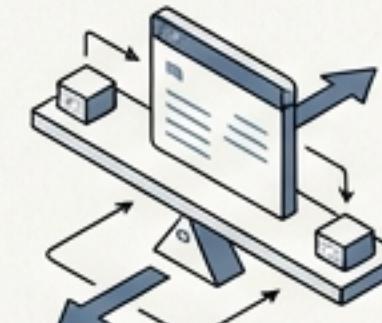
Context Rot (컨텍스트 부패)

컨텍스트가 길어질수록 모델의 추론 속도가 느려지고, 반복적인 출력이 발생하며, 지시 사항을 망각하는 성능 저하가 발생합니다.



The Goal

긴 실행 시간(Long-running)을 유지하면서도 윈도우를 깨끗하게 유지 하는 '적절한 정보 채우기(Filling the window)'의 기술.



솔루션 프레임워크: 컨텍스트 엔지니어링의 5가지 기둥



1. Offloading (오프로딩)

JetBrains Mono

токен 비중이 높은 데이터를
프롬프트 밖으로 이동

2. Reduction (축소)

Noto Sans KR

기록을 요약하거나
가지치기(Pruning)하여
길이 단축

3. Retrieval (검색)

Noto Sans KR

필요한 데이터만
온디맨드(On-demand)
로 호출

4. Isolation (격리)

Noto Sans KR

하위 에이전트(Sub-
agents)로 컨텍스트 분리

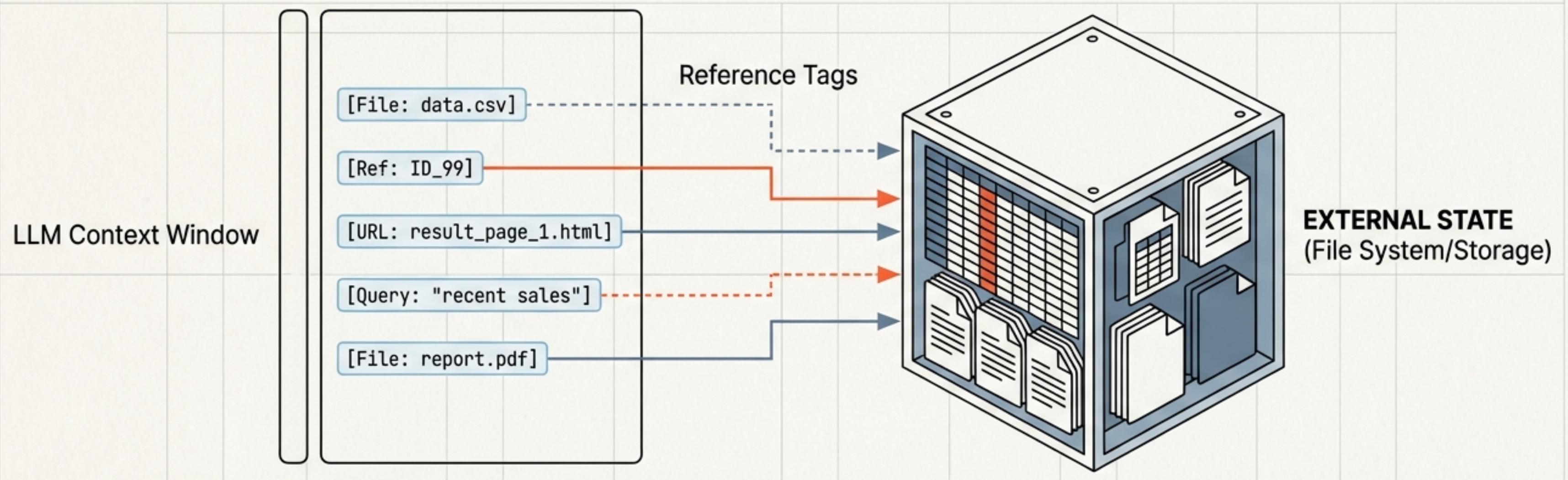
5. Caching (캐싱)

Noto Sans KR

KV Cache를 활용한
비용과 지연 시간 최적화

Deep Dive I: 컨텍스트 오프로딩 (Context Offloading)

채팅 기록(Chat History)을 데이터 저장소가 아닌 '참조'의 공간으로 활용



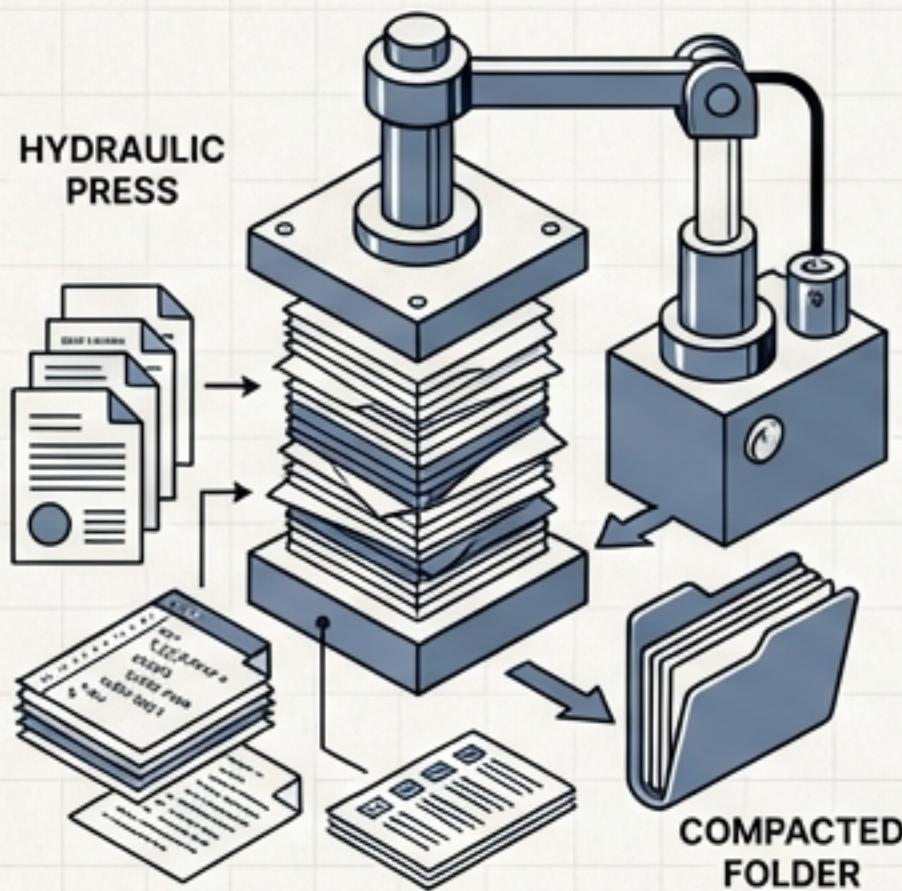
전략 (Strategy): 웹 검색 결과나 파일 읽기 결과와 같이 토큰 소모가 큰 출력물(Payload)을 채팅 기록에 남기지 않고 파일 시스템이나 외부 상태(State)로 내보냅니다.

Manus의 통찰:

- 에이전트의 거의 모든 작업은 **가역적(Reversible)**이거나 **오프로드 가능합니다**.
- **참조(Reference)만 남기기:** 에이전트에게는 원본 데이터 대신 'File Path', 'URL', 'Query'와 같은 고유 식별자만 전달해도 충분합니다.
- 에이전트는 필요할 때 해당 식별자를 통해 데이터를 다시 읽어올 수 있습니다.

Deep Dive II: 컨텍스트 축소 (Compaction vs. Summarization)

1. Compaction (압축 - 가역적)



Compaction: 외부에서 복구 가능한 정보(파일 내용 등)를 제거하고 메타데이터(파일 경로)만 남기는 방식.

Strategy: 가장 오래된 50%의 기록은 압축하되, 최근 도구 사용(Tool usage) 기록은 퓨샷(Few-shot) 학습을 위해 전체 포맷을 유지.

2. Summarization (요약 - 비가역적)



Summarization: 컨텍스트가 한계에 도달했을 때 정보를 영구적으로 압축.

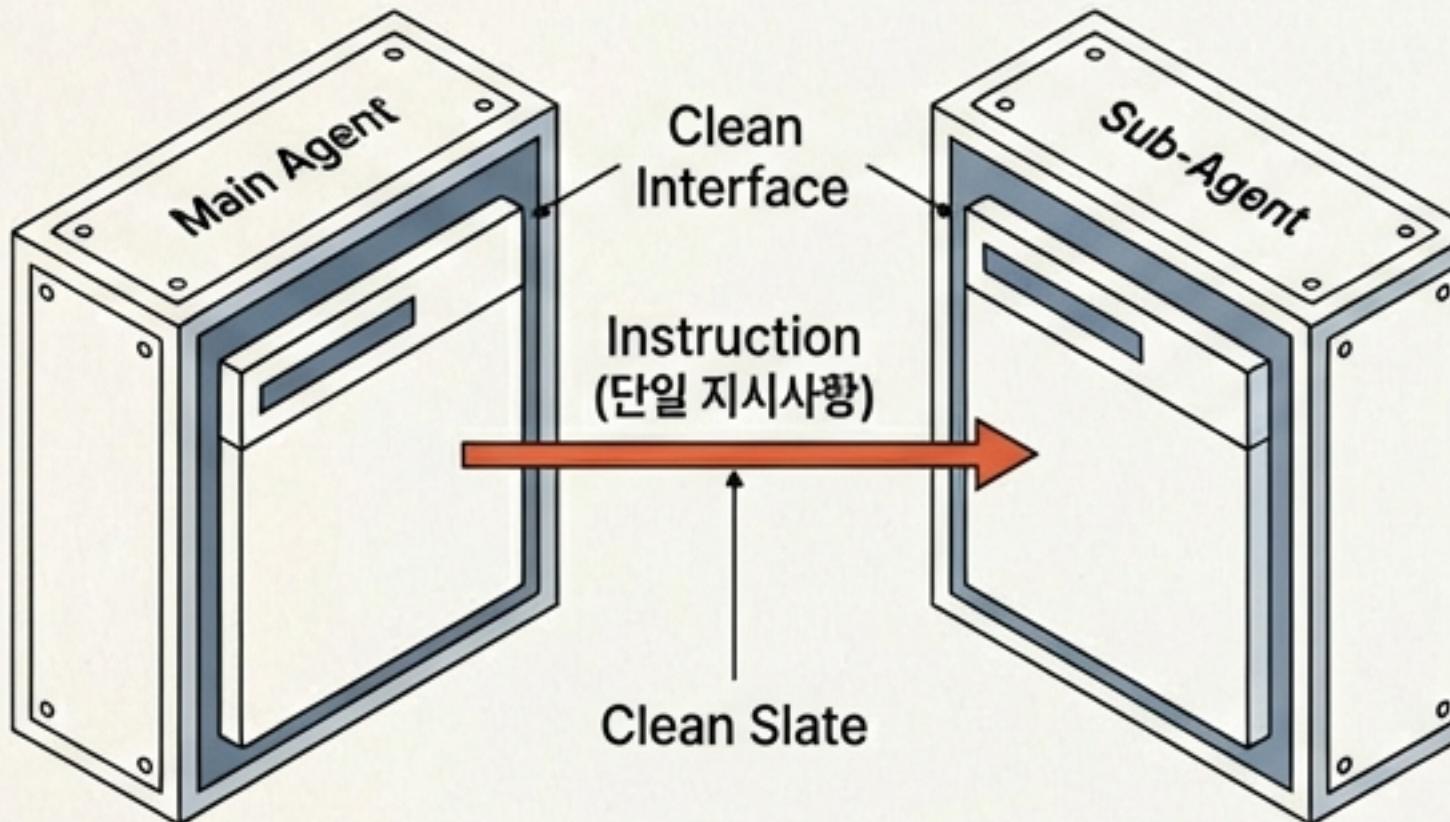
Strategy: 자유 형식(Free-form) 텍스트 요약은 위험. 반드시 구조화된 스키마(Structured Schema)를 사용하여 놓치는 정보가 없보가 없도록 강제 (예: 수정된 파일 목록, 사용자 목표).

⚠ IRREVERSIBLE DATA LOSS RISK

Deep Dive III: 컨텍스트 격리 (Context Isolation)

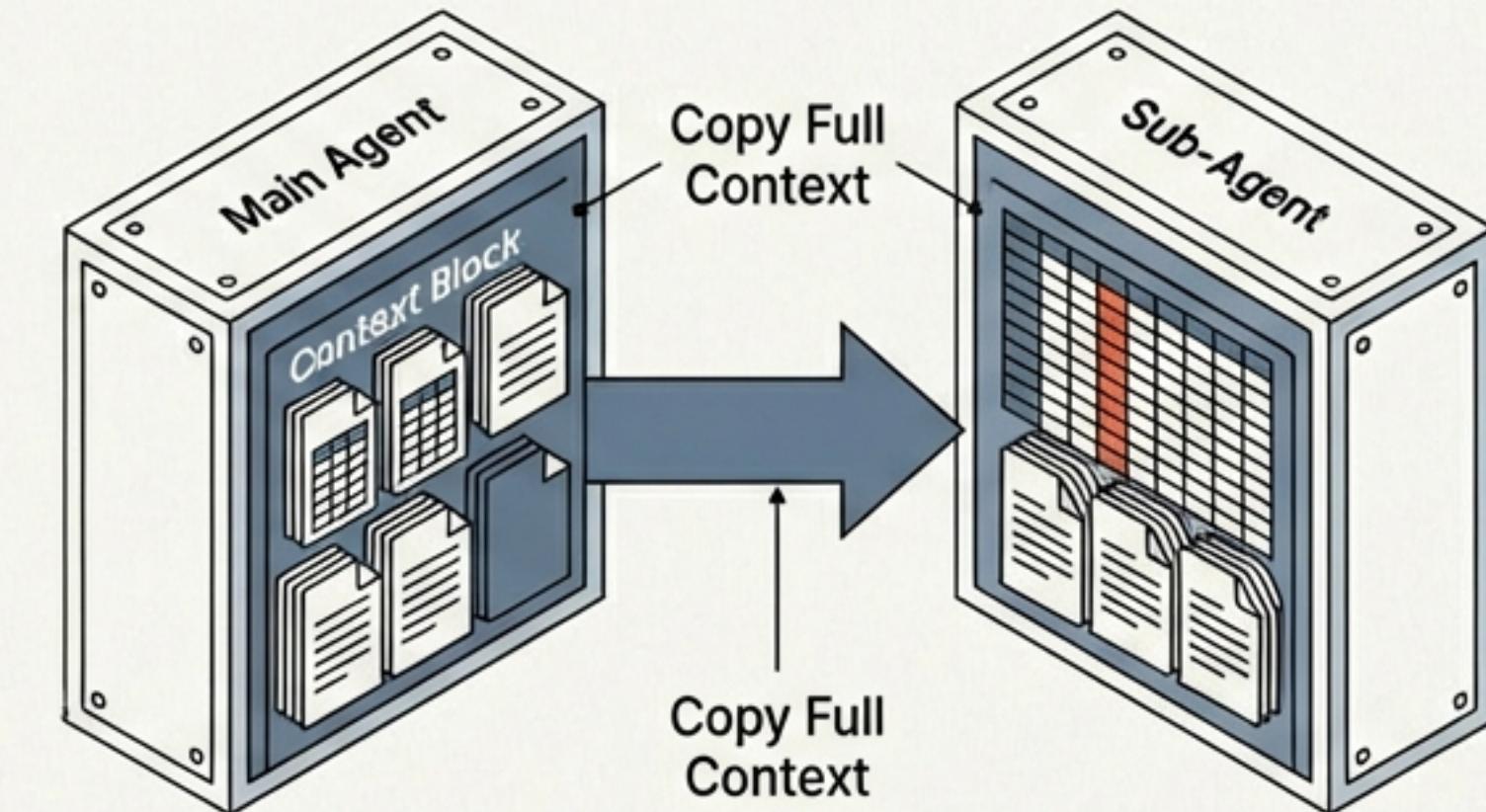
“메모리를 공유하여 통신하지 말고, 통신하여 메모리를 공유하라.” (The Go Philosophy)

1. By Communicating



하위 에이전트에게 깨끗한 상태(Clean Slate)에서
명확한 단일 지시사항만 전달. (Cloud Code 스타일)

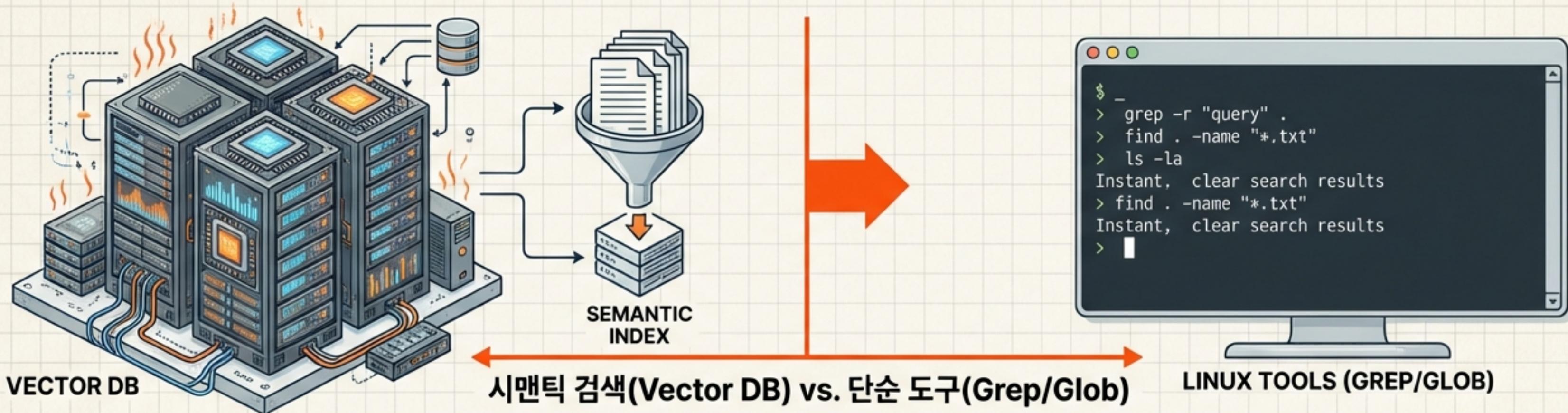
2. By Sharing Memory



하위 에이전트가 상위 에이전트의 전체 컨텍스트를 공유
받음. (Deep Research 스타일)

⚠ NOTE: 비용 높음 / KV Cache 재사용 불가

Deep Dive IV: 검색 전략 (Retrieval - Index vs. File System)

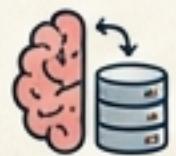


- Manus & Cursor Approach:



샌드박스/라이브 세션 (Live Session)

- Vector DB 구축 시간 부족. 'grep', 'glob', 'find'와 같은 표준 리눅스 도구 사용. (즉각적, 정확함, 모델 친화적)



장기 기억/엔터프라이즈 (Long-term)

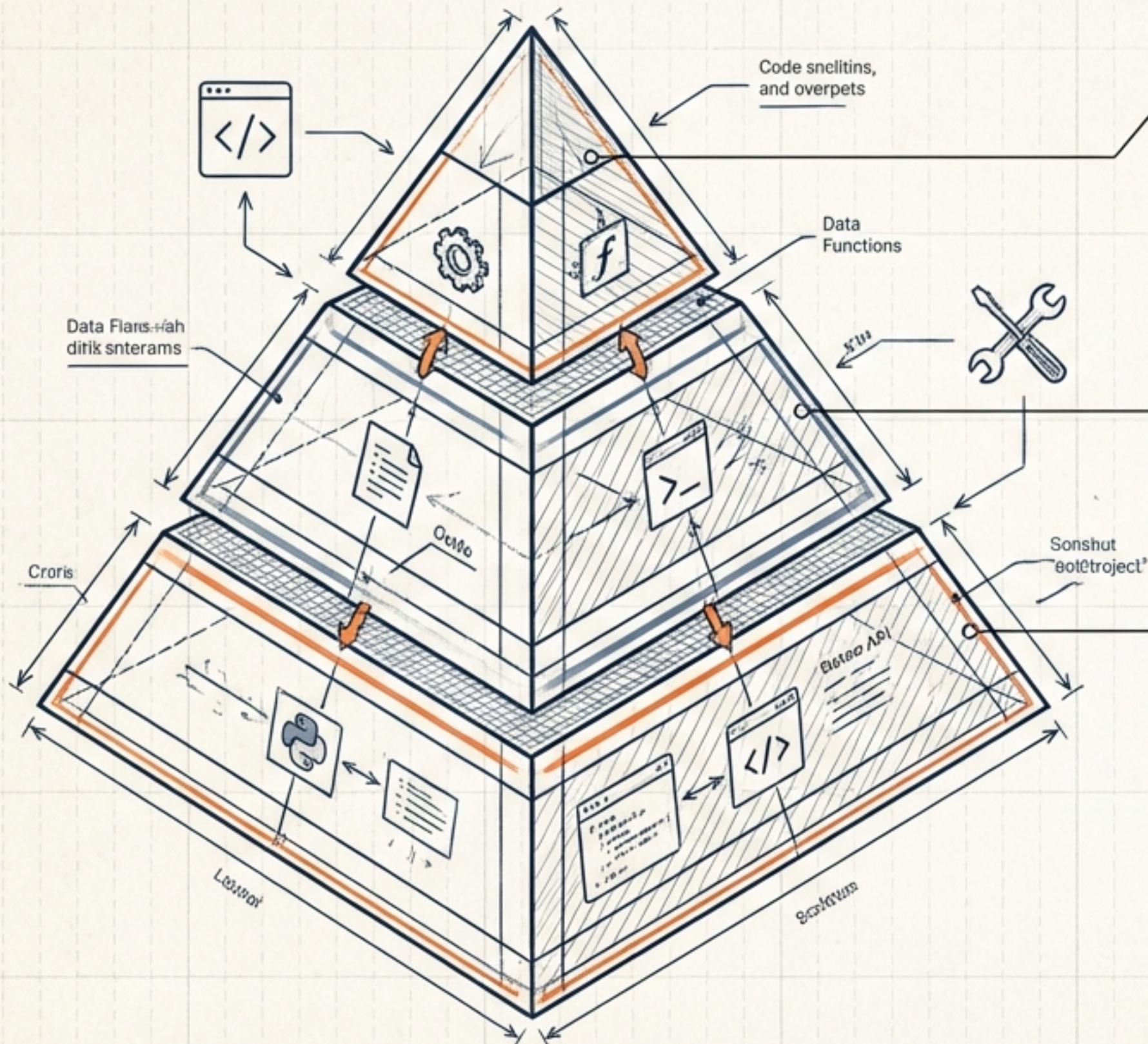
- 방대한 문서나 코드를 다룰 때만 **Vector Index** 사용.



KEY INSIGHT

핵심: 에이전트가 '컴퓨터'를 사용한다는 점을 기억하십시오. 리눅스 파일 시스템 자체가 훌륭한 검색 인덱스입니다.

고급 아키텍처: 계층화된 행동 공간 (Layered Action Space)



- **Layer 1: Atomic Functions**

모델이 직접 호출하는 10~20개의 고정 함수 ('Read', 'Write', 'Shell').

Use Case: Schema Safety, Core Logic.

- **Layer 2: Sandbox Utilities**

Linux Shell, 'grep', 'ffmpeg', Custom Scripts, MCP Tools.

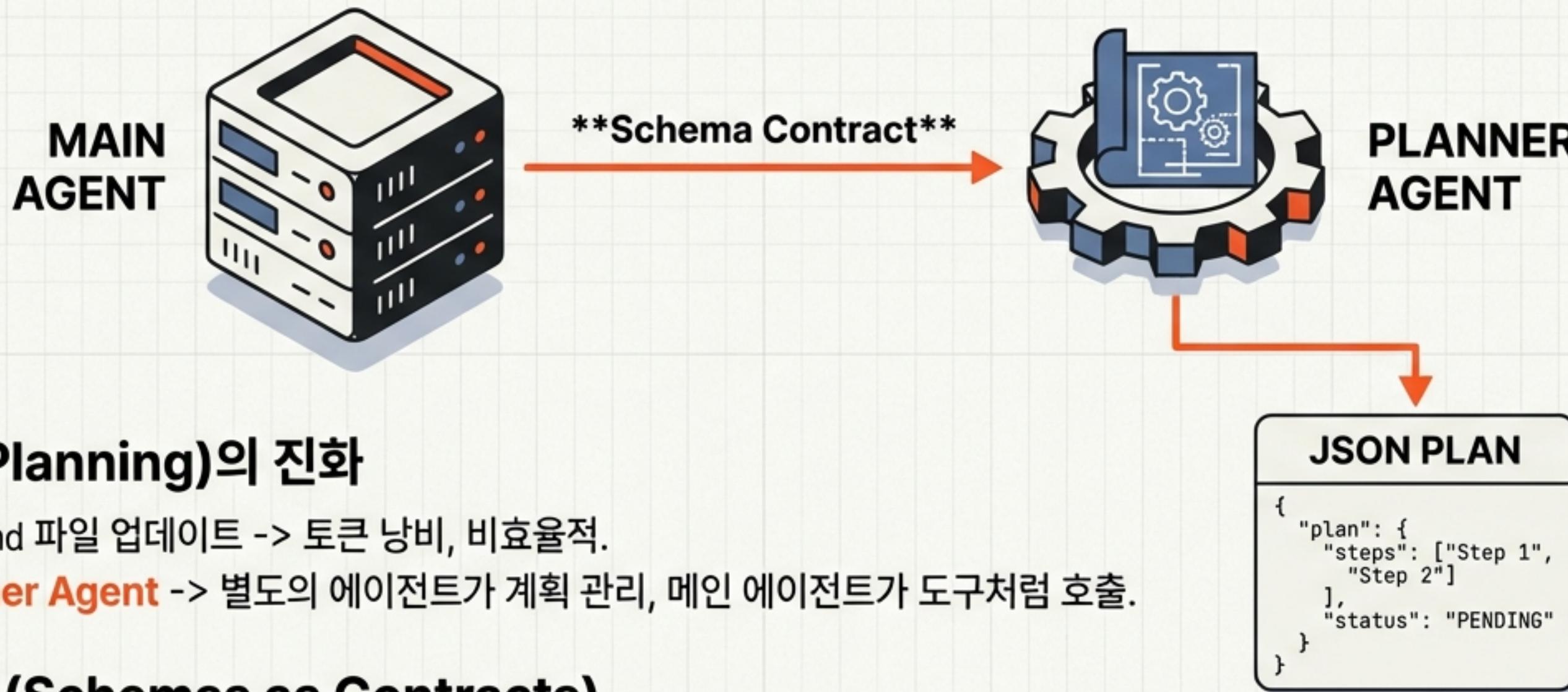
Use Case: 무한한 확장성 (CLI).

- **Layer 3: Packages & APIs**

Python 스크립트, 외부 API, Pandas 분석.

Use Case: 복잡한 로직 및 데이터 처리.

고급 아키텍처: 도구로서의 에이전트 (Agent as a Tool)



계획 수립 (Planning)의 진화

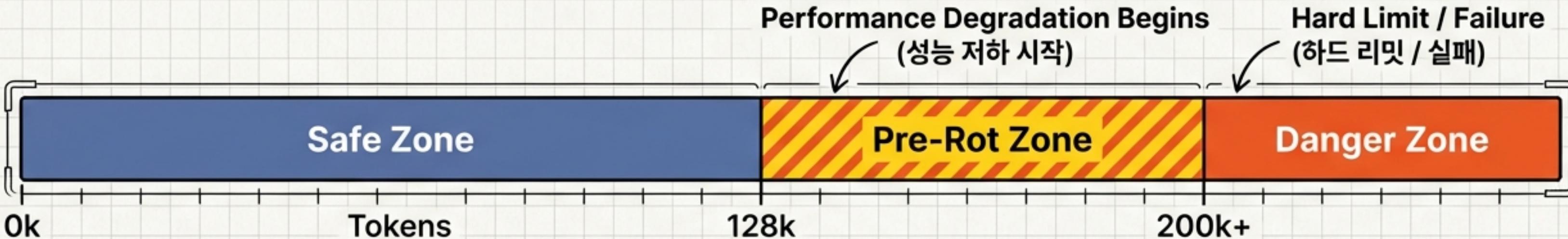
- ✗ Old: todo.md 파일 업데이트 -> 토큰 낭비, 비효율적.
- ✓ New: Planner Agent -> 별도의 에이전트가 계획 관리, 메인 에이전트가 도구처럼 호출.

스키마 계약 (Schemas as Contracts)

메인 에이전트와 서브 에이전트 간의 통신은 자연어가 아닌 **엄격한 출력 스키마(Output Schema)**를 통해 이루어집니다.
`Submit Result` 도구와 Constraint Decoding을 활용하여 형식을 강제합니다.



최적화 전략: 'Pre-Rot' 임계값 관리



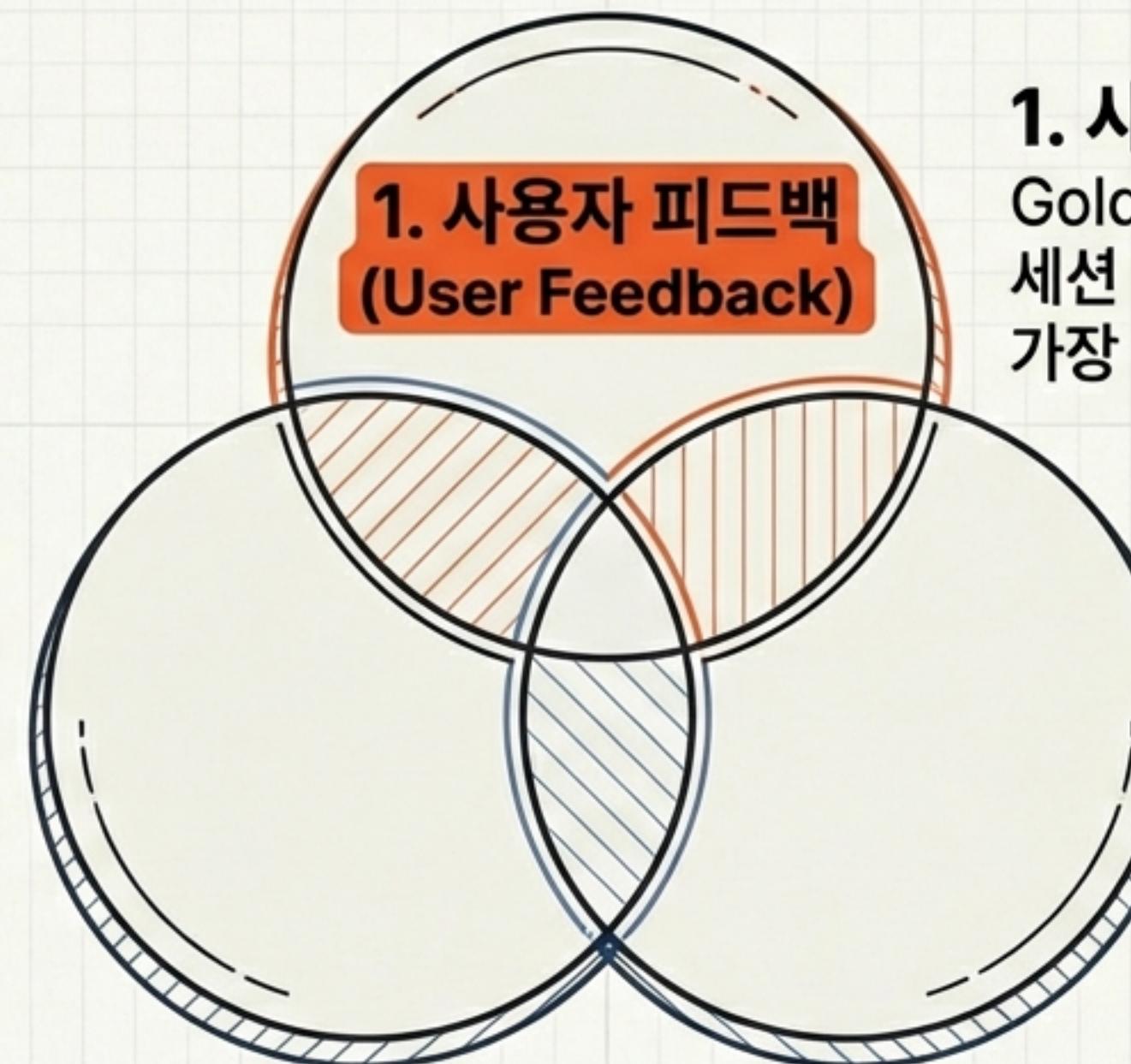
1. 컨텍스트 길이 모니터링

- ↗⚠ 2. Pre-Rot 임계값 도달 시 (128k~)
- └ 3. Compaction (압축) 우선 실행 [가역적]
- └ 4. Summarization (요약) 실행 [비가역적, 최후 수단]

Note: 최신 툴 호출 기록은 항상 원본 유지 (Few-shot 보호)

평가 전략: 벤치마크를 넘어서 (Beyond Benchmarks)

Manus의 3단계 평가 시스템 (The Trinity of Evals)



1. 사용자 피드백

Gold Standard.

세션 종료 후 1~5점 별점.
가장 신뢰할 수 있는 지표.

2. 내부 검증 세트 (Internal Verifiable Sets)

실행 결과의 성공/실패가
명확한 작업(Execution-based)
자동화 테스트.

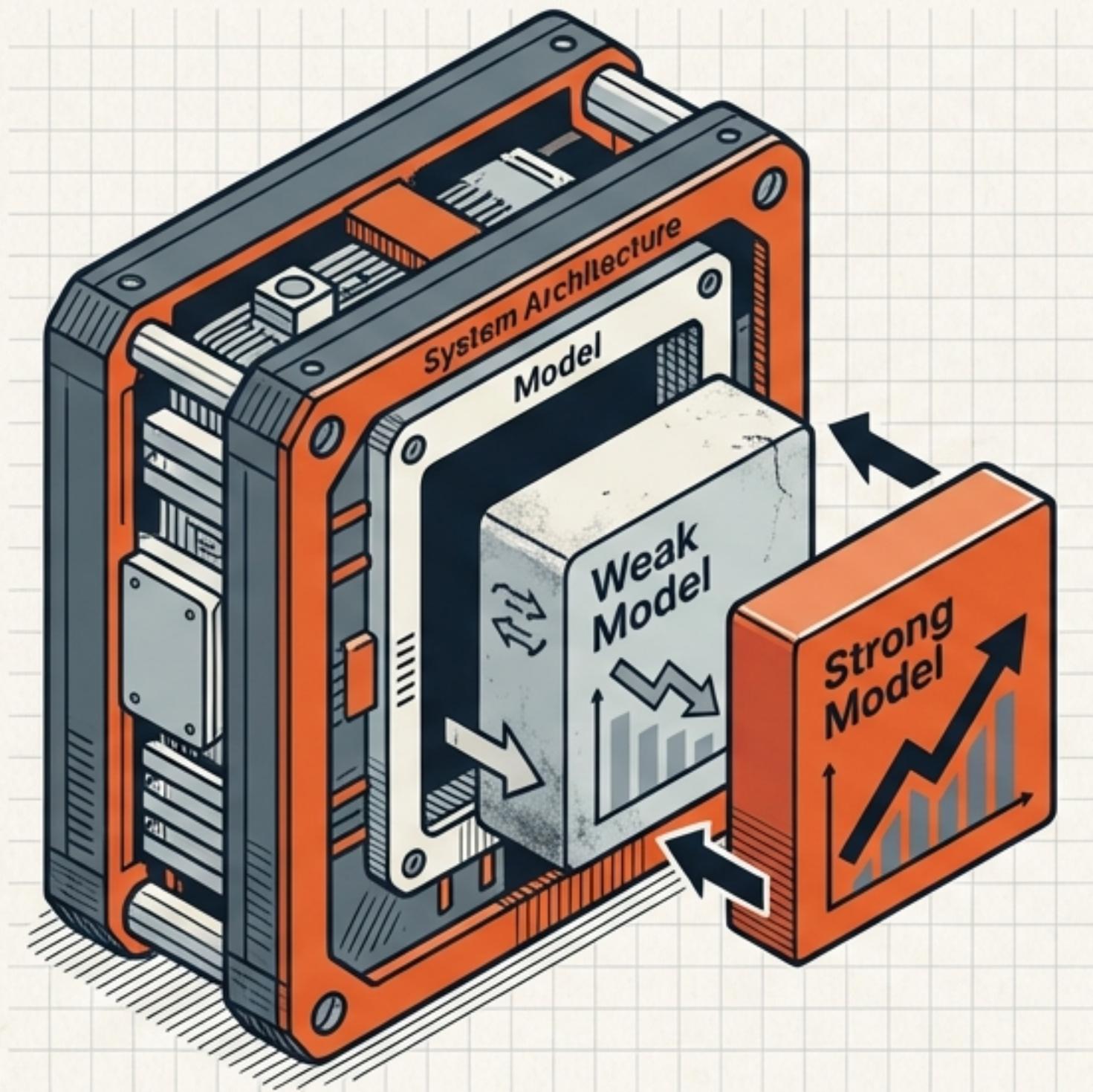
3. 인간 리뷰 (Human/Intern Review)

UI, 시각화 등 ‘미적 감각’이나
‘뉘앙스’가 필요한 영역의
Vibe Check.

Note: 공개 벤치마크(GAIA 등)는 실제 사용자 유ти리티와 괴리가 큼.



미래 대비 전략: 모델 독립성 (Model Independence)



아키텍처 > 모델: 특정 모델에 과도하게 최적화하지 마십시오. 모델은 계속 변합니다.

테스트 전략: 아키텍처의 견고함을 검증하기 위해 **약한 모델(Weaker models)**과 **강한 모델 (Strong models)**을 교차 테스트하십시오.

목표: 약한 모델에서도 아키텍처의 도움으로 성능을 낸다면, 강한 모델 도입 시 즉각적인 성능 향상이 가능합니다.

제언: 자체 모델 학습(RL)보다 범용 모델 + 강력한 컨텍스트 엔지니어링의 조합이 효율적.

“컨텍스트 엔지니어링은 윈도우를 정보로 채우는 것이 아니라,
다음 단계에 딱 필요한 정보만 남기는 예술입니다.”

Build Less, Understand More

Manus가 경험한 가장 큰 성능 향상은 복잡한 레이어를 추가했을 때가 아니라,
불필요한 트릭을 제거하고 모델의 능력(예: 리눅스 도구)을 신뢰했을 때
발생했습니다. 오버 엔지니어링을 피하십시오.



엔지니어를 위한 실행 체크리스트 (Actionable Checklist)

