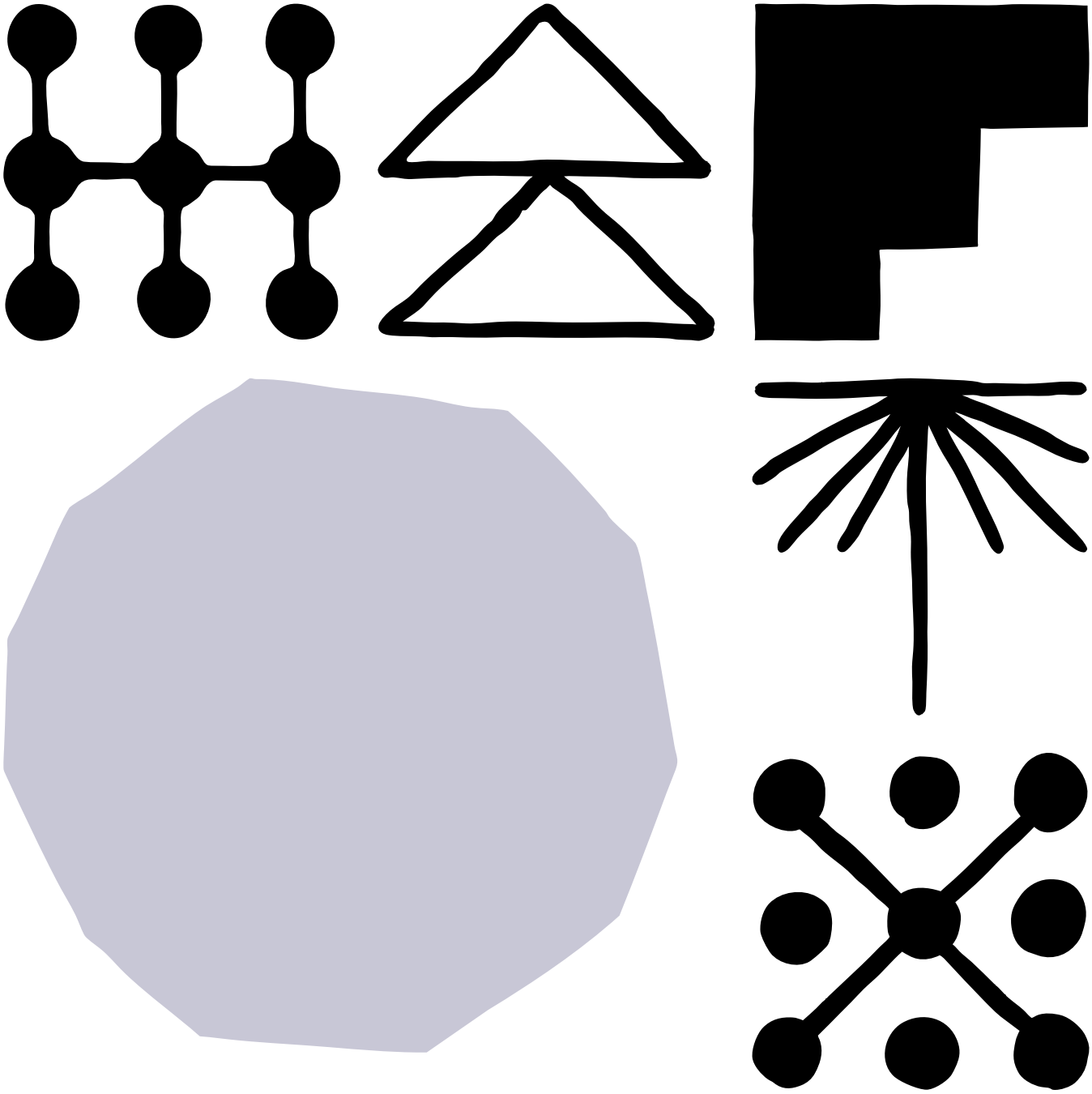


AI 에이전트를 위한 효과적인 컨텍스트 엔지니어링

출간됨2025년 9월 29일

컨텍스트는 AI 에이전트에게 중요하지만 한정된 자원입니다. 이 글에서는 AI 에이전트를 구동하는 컨텍스트를 효과적으로 큐레이션하고 관리하는 전략을 살펴봅니다.

응용 AI 분야에서 몇 년간 프롬프트 엔지니어링이 주목받은 후, 새로운 용어가 주목받고 있습니다: **컨텍스트 엔지니어링** 언어 모델을 활용한 구축은 프롬프트에 적합한 단어와 구문을 찾는 것에서 벗어나, "어떤 맥락 구성이 우리 모델의 원하는 행동을 생성할 가능성이 가장 높은가?"라는 더 광범위한 질문에 답하는 것으로 변화하고 있습니다.

맥락은 대규모 언어 모델(LLM)에서 샘플링할 때 포함되는 토큰 집합을 의미합니다. **엔지니어링**당면한 문제는 원하는 결과를 일관되게 달성하기 위해 LLM의 본질적인 제약 조건에 대해 해당 토큰들의 효용을 최적화하는 것입니다. LLM을 효과적으로 다루려면 종종 *맥락적 사고* 가 필요합니다. 즉, 주어진 시점에서 LLM이 사용할 수 있는 전체적인 상태와 그 상태가 야기할 수 있는 잠재적 행동들을 고려하는 것입니다.

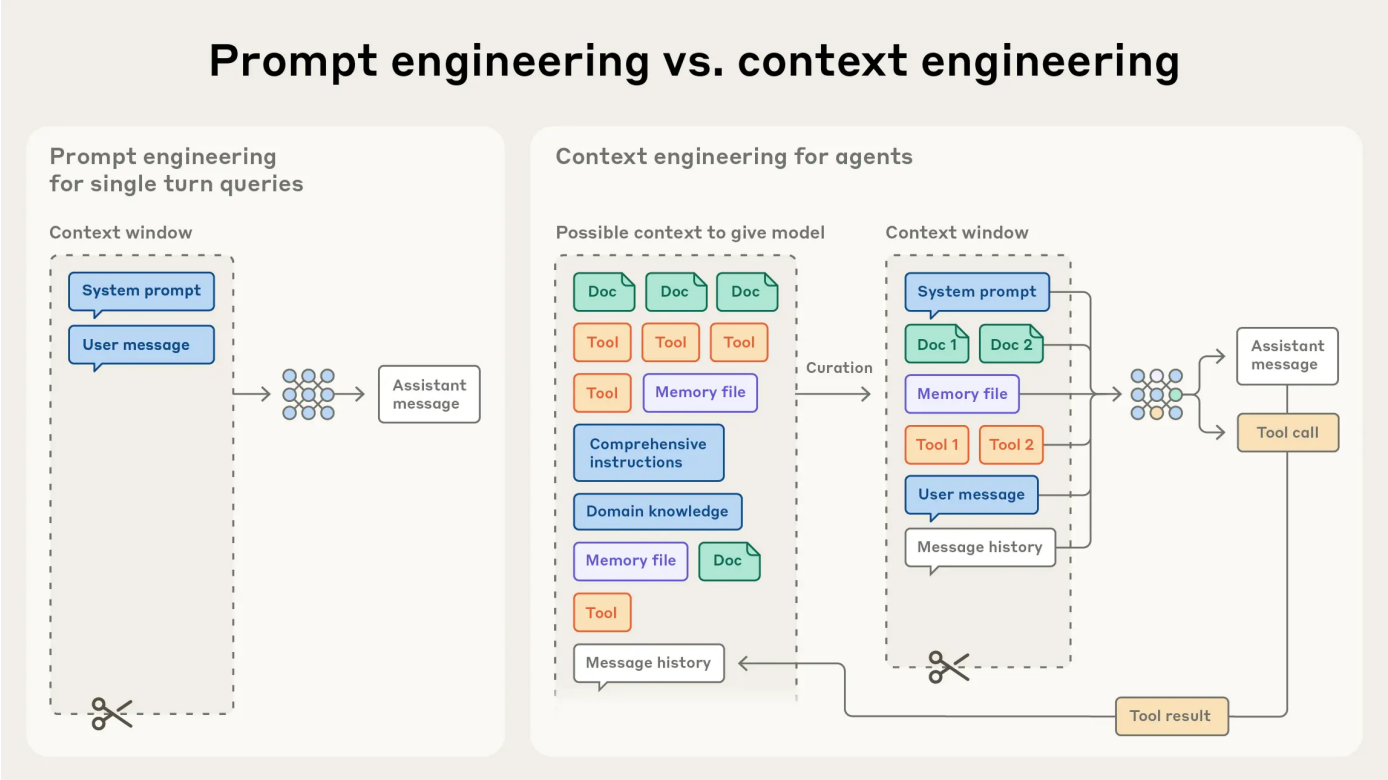
이 글에서는 맥락 엔지니어링이라는 새로운 기술을 탐구하고, 조종 가능하고 효과적인 에이전트를 구축하기 위한 정제된 멘탈 모델을 제시하겠습니다.

컨텍스트 엔지니어링 vs. 프롬프트 엔지니어링

Anthropic에서는 컨텍스트 엔지니어링을 프롬프트 엔지니어링의 자연스러운 발전으로 봅니다. 프롬프트 엔지니어링은 최적의 결과를 위해 LLM 지시사항을 작성하고 구성하는 방법을 의미합니다(개요와 유용한 프롬프트 엔지니어링 전략은 **저희 문서**를 참조하세요). **컨텍스트 엔지니어링**LLM 추론 중에 프롬프트 외에 도달할 수 있는 다른 모든 정보를 포함하여 최적의 토큰(정보) 집합을 큐레이션하고 유지하기 위한 전략 집합을 의미합니다.

LLM을 활용한 엔지니어링 초기에는 프롬프팅이 AI 엔지니어링 작업의 가장 큰 구성 요소였습니다. 일상적인 채팅 상호작용 외의 대부분의 사용 사례가 원샷 분류나 텍스트 생성 작업에 최적화된 프롬프트를 필요로 했기 때문입니다. 용어가 시사하듯이, 프롬프트 엔지니어링의 주요 초점은 효과적인 프롬프트, 특히 시스템 프롬프트를 작성하는 방법에 있었습니다. 하지만 여러 차례의 추론과 더 긴 시간 범위에 걸쳐 작동하는 더 유능한 에이전트를 엔지니어링하는 방향으로 나아가면서, 전체 컨텍스트 상태(시스템 지침, 도구,**모델 컨텍스트 프로토콜**(MCP), 외부 데이터, 메시지 기록 등).

루프에서 실행되는 에이전트는 점점 더 많은 데이터를 생성합니다 *할 수 있었다* 다음 추론 턴에 관련이 있어야 하며, 이 정보는 순환적으로 정제되어야 합니다. 컨텍스트 엔지니어링은**예술과 과학**끊임없이 진화하는 가능한 정보의 우주에서 제한된 컨텍스트 윈도우에 무엇을 포함시킬지 선별하는.



프롬프트 작성이라는 개별적인 작업과 달리, 컨텍스트 엔지니어링은 반복적이며 모델에 무엇을 전달할지 결정할 때마다 큐레이션 단계가 발생합니다.

유능한 에이전트 구축에 컨텍스트 엔지니어링이 중요한 이유

속도와 점점 더 많은 양의 데이터를 관리하는 능력에도 불구하고, 우리는 LLM이 인간과 마찬가지로 특정 지점에서 집중력을 잃거나 혼란을 경험한다는 것을 관찰했습니다. 건초더미에서 바늘 찾기 스타일 벤치마킹에 대한 연구들은 다음 개념을 밝혀냈습니다:**컨텍스트 부패**: 컨텍스트 윈도우의 토큰 수가 증가할수록, 해당 컨텍스트에서 정보를 정확하게 기억하는 모델의 능력이 감소합니다.

일부 모델들이 다른 모델들보다 더 완만한 성능 저하를 보이지만, 이러한 특성은 모든 모델에서 나타납니다. 따라서 컨텍스트는 한계효용이 체감하는 유한한 자원으로 취급되어야 합니다. **제한된 작업 기억 용량을 가진**LLM은 대량의 맥락을 파싱할 때 사용하는 "주의 예산"을 가지고 있습니다. 새로운 토큰이 도입될 때마다 이 예산이 일정량 소모되므로, LLM에 제공되는 토큰을 신중하게 선별해야 할 필요성이 증가합니다.

이러한 주의 부족은 LLM의 아키텍처 제약에서 비롯됩니다. LLM은 **트랜스포머 아키텍처**를 기반으로 하며, 이는 모든 토큰이 전체 컨텍스트에서 **다른 모든 토큰에 주의를 기울일 수 있게** 합니다. 이로 인해 n개의 토큰에 대해 n^2 개의 쌍별 관계가 생성됩니다.

컨텍스트 길이가 증가함에 따라 모델이 이러한 쌍별 관계를 포착하는 능력이 희석되어, 컨텍스트 크기와 주의 집중 사이에 자연스러운 긴장 관계가 생성됩니다. 또한 모델은 일반적으로 짧은 시퀀스가 긴 시퀀스보다 더 흔한 훈련 데이터 분포에서 주의 패턴을 개발합니다. 이는 모델이 컨텍스트 전반의 의존성에 대한 경험이 적고, 이를 위한 특화된 매개변수도 더 적다는 것을 의미합니다.

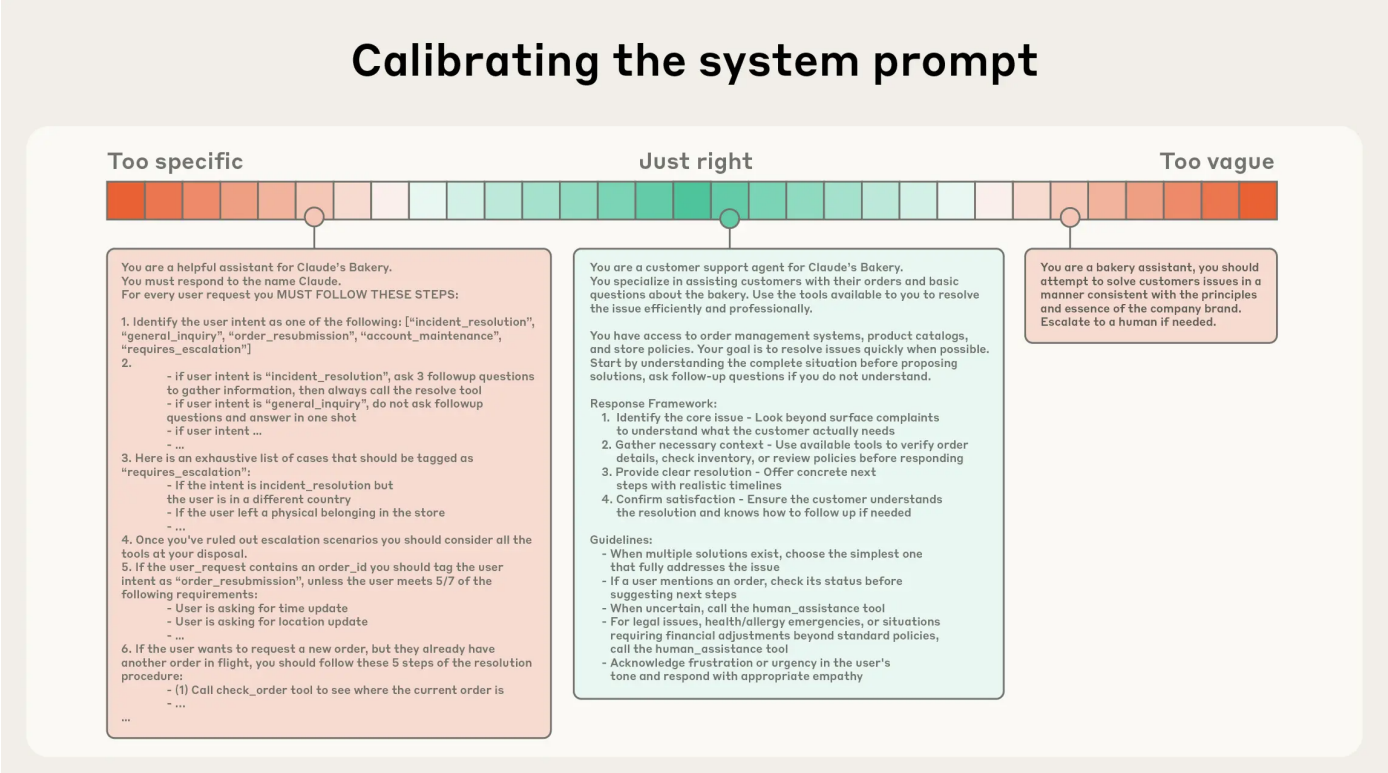
다음과 같은 기법들위치 인코딩 보간을 통해 모델이 원래 훈련된 더 작은 컨텍스트에 적응시켜 더 긴 시퀀스를 처리할 수 있게 하지만, 토큰 위치 이해에서는 어느 정도 성능 저하가 발생합니다. 이러한 요인들은 급격한 절벽이 아닌 성능 기울기를 만들어 냅니다: 모델은 더 긴 컨텍스트에서도 여전히 높은 성능을 유지하지만, 짧은 컨텍스트에서의 성능과 비교했을 때 정보 검색과 장거리 추론에서 정밀도가 감소할 수 있습니다.

이러한 현실은 유능한 에이전트를 구축하기 위해 신중한 컨텍스트 엔지니어링이 필수적임을 의미합니다.

효과적인 컨텍스트의 구조

LLM이 유한한 주의 예산에 제약을 받는다는 점을 고려할 때, *좋은* 컨텍스트 엔지니어링은 다음을 찾는 것을 의미합니다 *가장 작은 가능한* 원하는 결과의 가능성을 최대화하는 고신호 토큰 집합. 이 관행을 구현하는 것은 말하기는 쉽지만 실행하기는 훨씬 어렵습니다. 하지만 다음 섹션에서는 이 지침 원칙이 컨텍스트의 다양한 구성 요소에서 실제로 무엇을 의미하는지 설명합니다.

시스템 프롬프트매우 명확해야 하며 아이디어를 *적절한 고도* 에이전트에게 적절한 고도는 두 가지 일반적인 실패 모드 사이의 골디락스 존입니다. 한 극단에서는 엔지니어들이 정확한 에이전트 행동을 이끌어내기 위해 복잡하고 취약한 로직을 프롬프트에 하드코딩하는 것을 볼 수 있습니다. 이러한 접근 방식은 취약성을 만들고 시간이 지남에 따라 유지보수 복잡성을 증가시킵니다. 다른 극단에서는 엔지니어들이 때때로 모호하고 고수준의 지침을 제공하여 LLM에게 원하는 출력에 대한 구체적인 신호를 주지 못하거나 공유된 맥락을 잘못 가정하기도 합니다. 최적의 고도는 균형을 이룹니다: 행동을 효과적으로 안내할 만큼 구체적이면서도, 모델에게 행동을 안내할 강력한 휴리스틱을 제공할 만큼 유연합니다.



스펙트럼의 한쪽 끝에서는 취약한 if-else 하드코딩된 프롬프트를 볼 수 있고, 다른 쪽 끝에서는 지나치게 일반적이거나 공유된 맥락을 잘못 가정하는 프롬프트를 볼 수 있습니다.

우리는 프롬프트를 별개의 섹션들로 구성하고(`,` , `## 도구 안내` , `## 출력 설명` 등과 같이) XML 태깅이나 마크다운 헤더와 같은 기법을 사용하여 이러한 섹션들을 구분하는 것을 권장합니다. 다만 모델이 더욱 능력을 갖추게 됨에 따라 프롬프트의 정확한 형식은 덜 중요해지고 있을 것입니다.

시스템 프롬프트를 어떻게 구성하기로 결정하든, 기대하는 행동을 완전히 설명하는 최소한의 정보 집합을 목표로 해야 합니다. (최소한이라는 것이 반드시 짧다는 의미는 아닙니다. 에이전트가 원하는 행동을 준수하도록 하기 위해서는 여전히 충분한 정보를 미리 제공해야 합니다.) 가장 좋은 방법은 사용 가능한 최고의 모델로 최소한의 프롬프트를 테스트하여 작업에서 어떻게 수행되는지 확인한 다음, 초기 테스트 중에 발견된 실패 모드를 바탕으로 성능을 개선하기 위해 명확한 지침과 예시를 추가하는 것입니다.

도구에이전트가 환경과 상호작용하고 작업하면서 새로운 추가 컨텍스트를 가져올 수 있게 해줍니다. 도구는 에이전트와 그들의 정보/행동 공간 사이의 계약을 정의하기 때문에, 토큰 효율적인 정보를 반환하고 효율적인 에이전트 행동을 장려함으로써 효율성을 촉진하는 것이 매우 중요합니다.

에서AI 에이전트를 위한 도구 작성하기 - AI 에이전트와 함께, 우리는 LLM이 잘 이해할 수 있고 기능적 중복이 최소화된 도구를 구축하는 것에 대해 논의했습니다. 잘 설계된 코드베이스의 함수들과 마찬가지로, 도구들은 자체 완결적이고, 오류에 견고하며, 의도된 용도에 대해 극도로 명확해야 합니다. 입력 매개변수들도 마찬가지로 설명적이고, 모호하지 않으며, 모델의 고유한 강점을 활용해야 합니다.

우리가 보는 가장 일반적인 실패 모드 중 하나는 너무 많은 기능을 포함하거나 어떤 도구를 사용해야 할지에 대한 모호한 결정 지점을 만드는 비대한 도구 세트입니다. 인간 엔지니어가 주어진 상황에서 어떤 도구를 사용해야 할지 명확히 말할 수 없다면, AI 에이전트가 더 잘할 것이라고 기대할 수는 없습니다. 나중에 논의하겠지만, 에이전트를 위한 최소한의 실행 가능한 도구 세트를 선별하는 것은 긴 상호작용에서 더 안정적인 유지보수와 컨텍스트 정리로 이어질 수도 있습니다.

예시를 제공하는 것, 즉 퓨샷 프롬프팅(few-shot prompting)은 우리가 계속해서 강력히 권장하는 잘 알려진 모범 사례입니다. 하지만 팀들은 종종 LLM이 특정 작업에서 따라야 할 모든 가능한 규칙을 설명하려는 시도로 프롬프트에 엣지 케이스들의 긴 목록을 집어넣곤 합니다. 우리는 이를 권장하지 않습니다. 대신, 에이전트의 예상되는 행동을 효과적으로 보여주는 다양하고 표준적인 예시들의 집합을 선별하여 작업할 것을 권장합니다. LLM에게 예시는 천 마디 말의 가치가 있는 "그림"입니다.

컨텍스트의 다양한 구성 요소(시스템 프롬프트, 도구, 예시, 메시지 기록 등)에 대한 전반적인 지침은 신중하게 접근하여 컨텍스트를 유익하면서도 간결하게 유지하는 것입니다. 이제 런타임에서 컨텍스트를 동적으로 검색하는 방법에 대해 자세히 알아보겠습니다.

컨텍스트 검색과 에이전트 검색

에서효과적인 AI 에이전트 구축에서, 우리는 LLM 기반 워크플로우와 에이전트 간의 차이점을 강조했습니다. 그 게시물을 작성한 이후, 우리는간단한 정의에이전트용: LLM이 루프에서 자율적으로 도구를 사용합니다.

고객들과 함께 작업하면서, 우리는 이 분야가 이러한 간단한 패러다임으로 수렴하는 것을 목격했습니다. 기반 모델이 더욱 강력해짐에 따라 에이전트의 자율성 수준도 확장될 수 있습니다. 더 스마트한 모델은 에이전트가 미묘한 문제 공간을 독립적으로 탐색하고 오류로부터 복구할 수 있게 해줍니다.

이제 엔지니어들이 에이전트를 위한 컨텍스트 설계에 대해 생각하는 방식에 변화가 일어나고 있습니다. 오늘날 많은 AI 네이티브 애플리케이션들은 에이전트가 추론할 수 있도록 중요한 컨텍스트를 제공하기 위해 임베딩 기반의 사전 추론 시간 검색 방식을 어떤 형태로든 사용하고 있습니다. 이 분야가 더욱 에이전틱한 접근 방식으로 전환되면서, 팀들이 이러한 검색 시스템을 "적시" 컨텍스트 전략으로 보강하는 것을 점점 더 많이 보게 됩니다.

모든 관련 데이터를 미리 전처리하는 대신, "적시" 접근 방식으로 구축된 에이전트는 경량 식별자(파일 경로, 저장된 쿼리, 웹 링크 등)를 유지하고 이러한 참조를 사용하여 도구를 통해 런타임에 데이터를 컨텍스트로 동적으로 로드합니다. Anthropic의 에이전트 코딩 솔루션Claude Code이 접근 방식을 사용하여 대규모 데이터베이스에 대한 복잡한 데이터 분석을 수행합니다. 모델은 전체 데이터 객체를 컨텍스트에 로드하지 않고도 타겟팅된 쿼리를 작성하고, 결과를 저장하며, head와 tail 같은 Bash 명령어를 활용하여 대용량 데이터를 분석할 수 있습니다. 이러한 접근 방식은 인간의 인지 과정을 반영합니다: 우리는 일반적으로 전체 정보 말뭉치를 암기하지 않고, 대신 파일 시스템, 받은편지함, 북마크와 같은 외부 조직화 및 인덱싱 시스템을 도입하여 필요에 따라 관련 정보를 검색합니다.

저장 효율성을 넘어서, 이러한 참조의 메타데이터는 명시적으로 제공되든 직관적이든 관계없이 행동을 효율적으로 개선하는 메커니즘을 제공합니다. 파일 시스템에서 작동하는 에이전트에게 `test_utils.py` 폴더에 있는 `테스트` 라는 이름의 파일의 존재는 `src/core_logic/` 에 위치한 동일한 이름의 파일과는 다른 목적을 의미합니다. 폴더 계층 구조, 명명 규칙, 타임스탬프는 모두 인간과 에이전트가 정보를 언제 어떻게 활용해야 하는지 이해하는 데 도움이 되는 중요한 신호를 제공합니다.

에이전트가 자율적으로 탐색하고 데이터를 검색할 수 있게 하는 것은 점진적 공개(progressive disclosure)도 가능하게 합니다. 즉, 에이전트가 탐색을 통해 관련 맥락을 점진적으로 발견할 수 있게 해줍니다. 각 상호작용은 다음 결정에 정보를 제공하는 맥락을 생성합니다: 파일 크기는 복잡성을 시사하고, 명명 규칙은 목적을 암시하며, 타임스탬프는 관련성의 지표가 될 수 있습니다. 에이전트는 층층이 이해를 구축하며, 작업 메모리에는 필요한 것만 유지하고 추가적인 지속성을 위해 메모 작성 전략을 활용할 수 있습니다. 이러한 자체 관리되는 맥락 창은 에이전트가 포괄적이지만 잠재적으로 무관할 수 있는 정보에 압도되기보다는 관련성 있는 하위 집합에 집중할 수 있게 해줍니다.

물론 트레이드오프가 있습니다: 런타임 탐색은 미리 계산된 데이터를 검색하는 것보다 느립니다. 뿐만 아니라, LLM이 정보 환경을 효과적으로 탐색할 수 있는 적절한 도구와 휴리스틱을 갖도록 하려면 신중하고 사려 깊은 엔지니어링이 필요합니다. 적절한 가이드 없이는 에이전트가 도구를 잘못 사용하거나, 막다른 길을 쫓거나, 핵심 정보를 식별하지 못해 컨텍스트를 낭비할 수 있습니다.

특정 상황에서는 가장 효과적인 에이전트가 하이브리드 전략을 사용할 수 있습니다. 즉, 속도를 위해 일부 데이터를 미리 검색하고, 재량에 따라 추가적인 자율 탐색을 수행하는 것입니다. '적절한' 자율성 수준에 대한 결정 경계는 작업에 따라 달라집니다. Claude Code는 이러한 하이브리드 모델을 사용하는 에이전트입니다:CLAUDE.md파일들이 처음에 단순하게 컨텍스트에 투입되는 반면, glob과 grep 같은 기본 도구들은 환경을 탐색하고 필요할 때 파일을 검색할 수 있게 해주어, 오래된 인덱싱과 복잡한 구문 트리의 문제들을 효과적으로 우회합니다.

하이브리드 전략은 법률이나 금융 업무와 같이 덜 동적인 콘텐츠를 다루는 상황에 더 적합할 수 있습니다. 모델 능력이 향상됨에 따라, 에이전트 설계는 지능적인 모델이 지능적으로 행동하도록 하는 방향으로 발전할 것이며, 인간의 큐레이션은 점진적으로 줄어들 것입니다. 이 분야의 빠른 발전 속도를 고려할 때, "작동하는 가장 간단한 방법을 사용하라"는 것이 Claude를 기반으로 에이전트를 구축하는 팀들에게 여전히 최고의 조언이 될 것입니다.

장기 과제를 위한 컨텍스트 엔지니어링

장기 과제는 에이전트가 토큰 수가 LLM의 컨텍스트 윈도우를 초과하는 일련의 행동에서 일관성, 맥락, 그리고 목표 지향적 행동을 유지하도록 요구합니다. 대규모 코드베이스 마이그레이션이나 포괄적인 연구 프로젝트와 같이 수십 분에서 여러 시간의 지속적인 작업에 걸친 과제의 경우, 에이전트는 컨텍스트 윈도우 크기 제한을 해결하기 위한 특수한 기법이 필요합니다.

더 큰 컨텍스트 윈도우를 기다리는 것이 명백한 전략처럼 보일 수 있습니다. 하지만 가까운 미래에는 모든 크기의 컨텍스트 윈도우가 컨텍스트 오염과 정보 관련성 문제에 여전히 영향을 받을 가능성이 높습니다—적어도 가장 강력한 에이전트 성능이 요구되는 상황에서는 말입니다. 에이전트가 확장된 시간 범위에서 효과적으로 작동할 수 있도록 하기 위해, 우리는 이러한 컨텍스트 오염 제약을 직접적으로 해결하는 몇 가지 기법을 개발했습니다: 압축, 구조화된 노트 작성, 그리고 멀티 에이전트 아키텍처입니다.

압축

압축은 컨텍스트 윈도우 한계에 근접한 대화를 요약하고, 그 요약으로 새로운 컨텍스트 윈도우를 다시 시작하는 방법입니다. 압축은 일반적으로 더 나은 장기 일관성을 위한 컨텍스트 엔지니어링의 첫 번째 수단으로 사용됩니다. 본질적으로 압축은 컨텍스트 윈도우의 내용을 고품질로 압축하여 에이전트가 최소한의 성능 저하로 계속 작업할 수 있게 합니다.

Claude Code에서는 예를 들어, 메시지 기록을 모델에 전달하여 가장 중요한 세부사항을 요약하고 압축하는 방식으로 이를 구현합니다. 모델은 중복된 도구 출력이나 메시지는 버리면서 아키텍처 결정사항, 해결되지 않은 버그, 구현 세부사항을 보존합니다. 그러면 에이전트는 이 압축된 컨텍스트와 가장 최근에 접근한 5개 파일을 함께 사용하여 작업을 계속할 수 있습니다. 사용자는 컨텍스트 윈도우 제한을 걱정하지 않고도 연속성을 얻을 수 있습니다.

압축의 기술은 무엇을 유지할지와 무엇을 버릴지를 선택하는 데 있습니다. 지나치게 공격적인 압축은 미묘하지만 중요한 맥락을 잃을 수 있으며, 그 중요성은 나중에야 명확해지기 때문입니다. 압축 시스템을 구현하는 엔지니어들에게는 복잡한 에이전트 추적에서 프롬프트를 신중하게 조정할 것을 권장합니다. 먼저 재현율을 최대화하여 압축 프롬프트가 추적에서 모든 관련 정보를 포착하도록 한 다음, 불필요한 내용을 제거하여 정밀도를 향상시키는 방향으로 반복 개선하세요.

불필요한 콘텐츠의 쉬운 예시는 도구 호출과 결과를 정리하는 것입니다. 메시지 기록 깊숙이 도구가 호출된 후에는 에이전트가 원시 결과를 다시 볼 필요가 있을까요? 가장 안전하고 가벼운 압축 형태 중 하나는 도구 결과 정리로, 최근에 Claude Developer Platform의 기능.

구조화된 노트 작성

구조화된 노트 작성, 또는 에이전트 메모리는 에이전트가 컨텍스트 윈도우 외부의 메모리에 지속적으로 저장되는 노트를 정기적으로 작성하는 기법입니다. 이러한 노트들은 나중에 컨텍스트 윈도우로 다시 불러와집니다.

이 전략은 최소한의 오버헤드로 지속적인 메모리를 제공합니다. Claude Code가 할 일 목록을 만들거나, 사용자 정의 에이전트가 NOTES.md 파일을 유지하는 것처럼, 이 간단한 패턴을 통해 에이전트는 복잡한 작업에서 진행 상황을 추적하고, 수십 번의 도구 호출에서 잃어버릴 수 있는 중요한 컨텍스트와 종속성을 유지할 수 있습니다.

포켓몬을 플레이하는 Claude메모리가 비코딩 영역에서 에이전트 능력을 어떻게 변화시키는지 보여줍니다. 에이전트는 수천 개의 게임 단계에 걸쳐 정확한 집계를 유지합니다—"지난 1,234단계 동안 1번 도로에서 포켓몬을 훈련시켰고, 피카츄가 목표인 10레벨 중 8레벨을 얻었다"와 같은 목표를 추적합니다. 메모리 구조에 대한 어떤 프롬프트도 없이, 탐험한 지역의 지도를 개발하고, 잠금 해제한 주요 성취를 기억하며, 다양한 상대에게 어떤 공격이 가장 효과적인지 학습하는 데 도움이 되는 전투 전략의 전략적 메모를 유지합니다.

컨텍스트가 재설정된 후, 에이전트는 자신의 노트를 읽고 수 시간에 걸친 훈련 시퀀스나 던전 탐험을 계속합니다. 요약 단계 전반에 걸친 이러한 일관성은 LLM의 컨텍스트 윈도우에만 모든 정보를 유지할 때는 불가능했던 장기적 전략을 가능하게 합니다.

우리의 Sonnet 4.5 출시의 일환으로, 파일 기반 시스템을 통해 컨텍스트 윈도우 밖에서 정보를 저장하고 참조하는 것을 더 쉽게 만드는 메모리 도구를 Claude Developer Platform에서 공개 베타로 출시했습니다. 이를 통해 에이전트는 시간이 지남에 따라 지식 베이스를 구축하고, 세션 간에 프로젝트 상태를 유지하며, 모든 것을 컨텍스트에 유지하지 않고도 이전 작업을 참조할 수 있습니다.

하위 에이전트 아키텍처

서브 에이전트 아키텍처는 컨텍스트 제한을 해결하는 또 다른 방법을 제공합니다. 하나의 에이전트가 전체 프로젝트에 걸쳐 상태를 유지하려고 시도하는 대신, 전문화된 서브 에이전트들이 깔끔한 컨텍스트 윈도우로 집중된 작업을 처리할 수 있습니다. 메인 에이전트는 고수준 계획으로 조정하는 반면, 서브 에이전트들은 심층적인 기술 작업을 수행하거나 도구를 사용하여 관련 정보를 찾습니다. 각 서브 에이전트는 수만 개 이상의 토큰을 사용하여 광범위하게 탐색할 수 있지만, 작업의 압축되고 정제된 요약만을 반환합니다(보통 1,000-2,000 토큰).

이 접근 방식은 관심사의 명확한 분리를 달성합니다. 세부적인 검색 컨텍스트는 하위 에이전트 내에서 격리된 상태로 유지되고, 리드 에이전트는 결과를 종합하고 분석하는 데 집중합니다. How we built our multi-agent research system에서 논의된 이 패턴은 복잡한 연구 작업에서 단일 에이전트 시스템보다 상당한 개선을 보여주었습니다.

이러한 접근 방식 간의 선택은 작업 특성에 따라 달라집니다. 예를 들어:

- 압축은 광범위한 상호작용이 필요한 작업에서 대화의 흐름을 유지합니다;
- 노트 작성은 명확한 이정표가 있는 반복적 개발에 뛰어납니다;
- 멀티 에이전트 아키텍처는 병렬 탐색이 효과를 발휘하는 복잡한 연구 및 분석을 처리합니다.

모델이 계속 개선되더라도, 확장된 상호작용에서 일관성을 유지하는 과제는 더 효과적인 에이전트를 구축하는 데 여전히 핵심적인 요소로 남을 것입니다.

결론

컨텍스트 엔지니어링은 LLM으로 구축하는 방식의 근본적인 변화를 나타냅니다. 모델이 더욱 강력해질수록, 완벽한 프롬프트를 만드는 것만이 과제가 아니라—각 단계에서 모델의 제한된 주의 예산에 어떤 정보가 들어가는지를 신중하게 선별하는 것이 과제입니다. 장기간 작업을 위한 압축을 구현하든, 토큰 효율적인 도구를 설계하든, 에이전트가 환경을 적시에 탐색할 수 있도록 하든, 지침 원칙은 동일합니다: 원하는 결과의 가능성을 최대화하는 가장 작은 고신호 토큰 집합을 찾는 것입니다.

우리가 설명한 기법들은 모델이 개선됨에 따라 계속 발전할 것입니다. 이미 더 똑똑한 모델들은 덜 규범적인 엔지니어링을 필요로 하여 에이전트가 더 많은 자율성을 가지고 작동할 수 있게 한다는 것을 보고 있습니다. 하지만 능력이 확장되더라도 컨텍스트를 귀중하고 유한한 자원으로 취급하는 것은 신뢰할 수 있고 효과적인 에이전트를 구축하는 데 여전히 핵심적일 것입니다.

오늘 Claude Developer Platform에서 컨텍스트 엔지니어링을 시작하고, 다음을 통해 유용한 팁과 모범 사례에 액세스하세요메모리 및 컨텍스트 관리쿡북.

감사의 말

Anthropic의 Applied AI 팀이 작성: Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, Jeremy Hadfield, 그리고 팀 멤버 Rafi Ayub, Hannah Moran, Cal Rueb, Connor Jennings의 기여. 지원해 주신 Molly Vorwerck, Stuart Ritchie, Maggie Vo에게 특별한 감사를 드립니다.

Effective context engineering for AI agents

Published Sep 29, 2025

Context is a critical but finite resource for AI agents. In this post, we explore strategies for effectively curating and managing the context that powers them.

After a few years of prompt engineering being the focus of attention in applied AI, a new term has come to prominence: **context engineering**. Building with language models is becoming less about finding the right words and phrases for your prompts, and more about answering the broader question of "what configuration of context is most likely to generate our model's desired behavior?"

Context refers to the set of tokens included when sampling from a large-language model (LLM). The **engineering** problem at hand is optimizing the utility of those tokens against the inherent constraints of LLMs in order to consistently achieve a desired outcome. Effectively wrangling LLMs often requires *thinking in context* — in other words: considering the holistic state available to the LLM at any given time and what potential behaviors that state might yield.

In this post, we'll explore the emerging art of context engineering and offer a refined mental model for building steerable, effective agents.

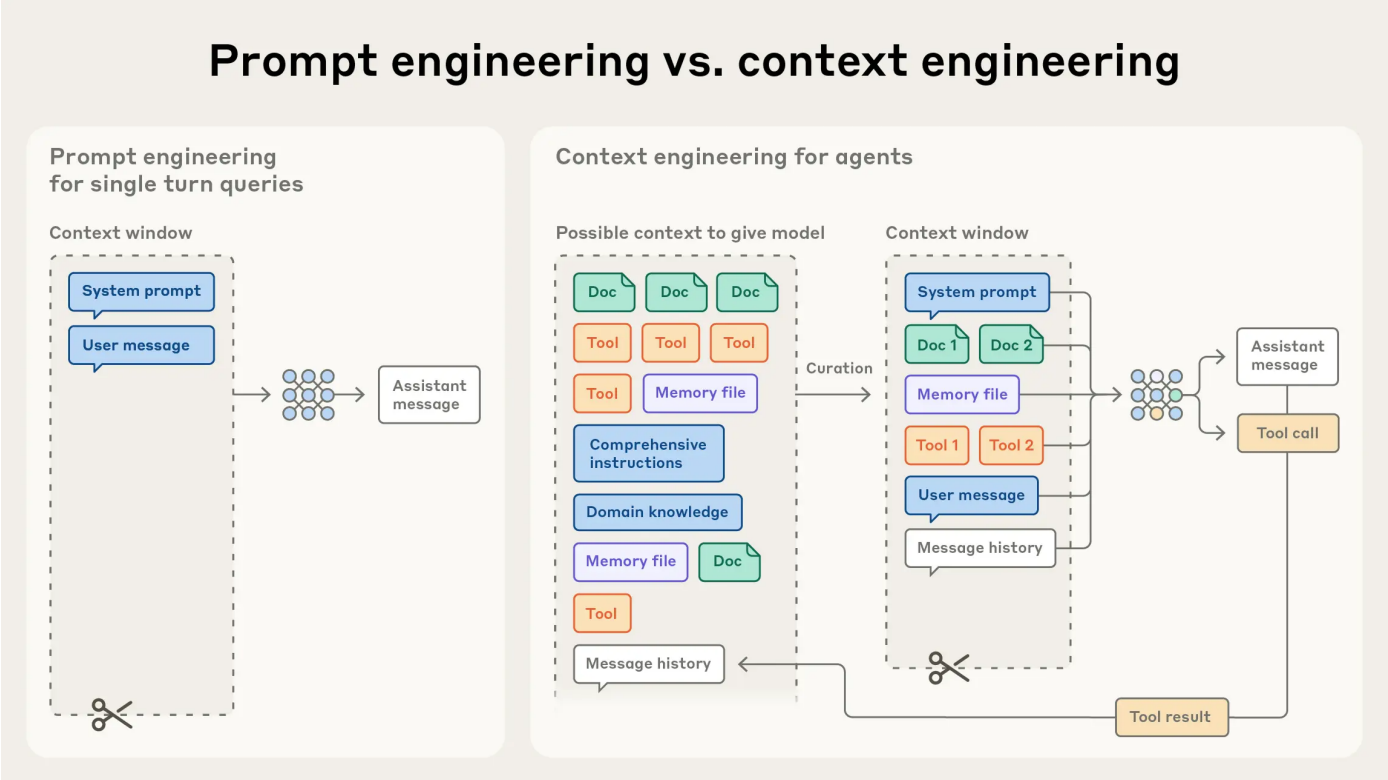
Context engineering vs. prompt engineering

At Anthropic, we view context engineering as the natural progression of prompt engineering. Prompt engineering refers to methods for writing and organizing LLM instructions for optimal outcomes (see our docs for an overview and useful prompt engineering strategies). **Context engineering** refers to the set of strategies for curating and maintaining the optimal set of tokens (information) during LLM inference,

including all the other information that may land there outside of the prompts.

In the early days of engineering with LLMs, prompting was the biggest component of AI engineering work, as the majority of use cases outside of everyday chat interactions required prompts optimized for one-shot classification or text generation tasks. As the term implies, the primary focus of prompt engineering is how to write effective prompts, particularly system prompts. However, as we move towards engineering more capable agents that operate over multiple turns of inference and longer time horizons, we need strategies for managing the entire context state (system instructions, tools, Model Context Protocol (MCP), external data, message history, etc).

An agent running in a loop generates more and more data that *could* be relevant for the next turn of inference, and this information must be cyclically refined. Context engineering is the art and science of curating what will go into the limited context window from that constantly evolving universe of possible information.



In contrast to the discrete task of writing a prompt, context engineering is iterative and the curation phase happens each time we decide what to pass to the model.

Why context engineering is important to building capable agents

Despite their speed and ability to manage larger and larger volumes of data, we've observed that LLMs, like humans, lose focus or experience confusion at a certain point. Studies on needle-in-a-haystack style benchmarking have uncovered the concept of **context rot**: as the number of tokens in the context window increases, the model's ability to accurately recall information from that context decreases.

While some models exhibit more gentle degradation than others, this characteristic emerges across all models. Context, therefore, must be treated as a finite resource with diminishing marginal returns. Like humans, who have **limited working memory capacity**, LLMs have an "attention budget" that they draw on when parsing large volumes of context. Every new token introduced depletes this budget by some amount, increasing the need to carefully curate the tokens available to the LLM.

This attention scarcity stems from architectural constraints of LLMs. LLMs are based on the **transformer architecture**, which enables every token to **attend to every other token** across the entire context. This results in n^2 pairwise relationships for n tokens.

As its context length increases, a model's ability to capture these pairwise relationships gets stretched thin, creating a natural tension between context size and attention focus. Additionally, models develop their attention patterns from training data distributions where shorter sequences are typically more common than longer ones. This means models have less experience with, and fewer specialized parameters for, context-wide dependencies.

Techniques like **position encoding interpolation** allow models to handle longer sequences by adapting them to the originally trained smaller context, though with some degradation in token position understanding. These factors create a performance gradient rather than a hard cliff: models remain highly capable at longer contexts but may show reduced precision for information retrieval and long-range reasoning compared to their performance on shorter contexts.

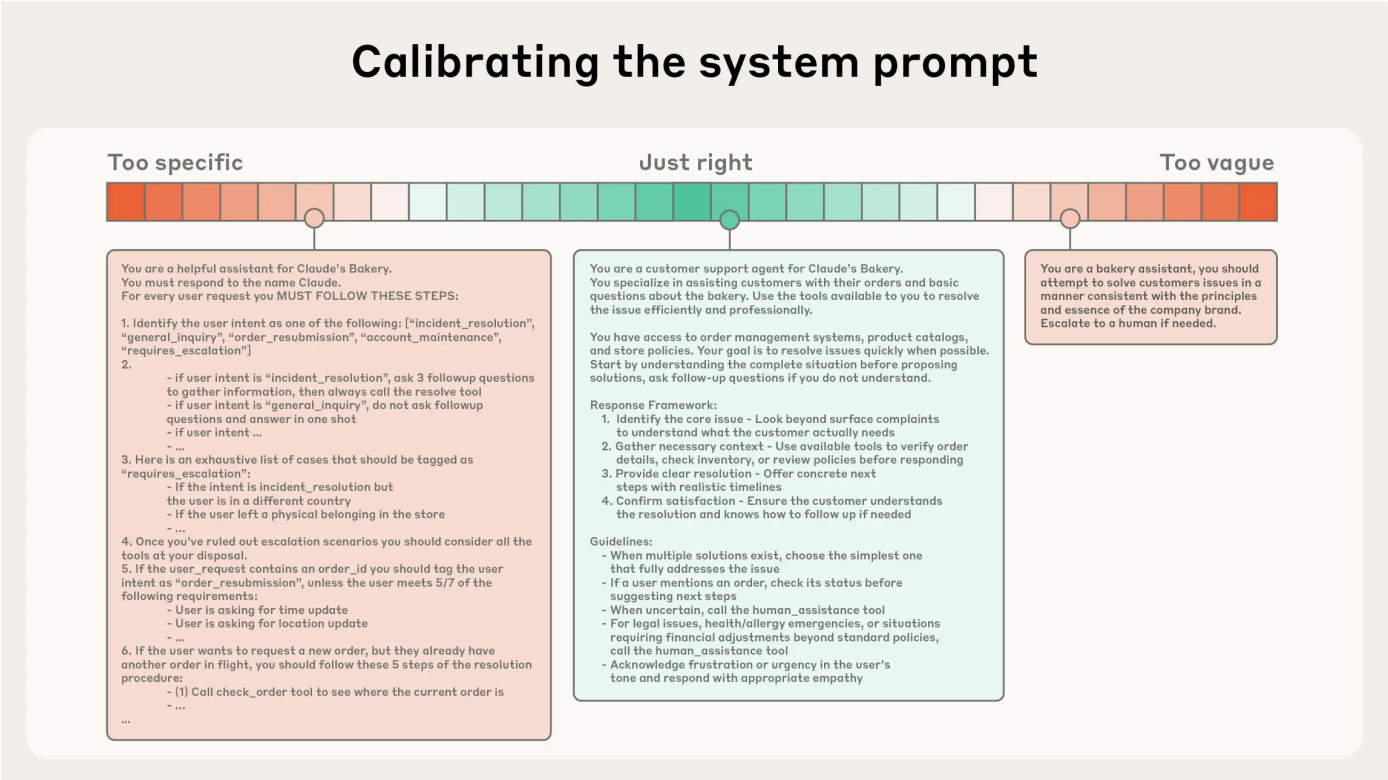
These realities mean that thoughtful context engineering is essential for building capable agents.

The anatomy of effective context

Given that LLMs are constrained by a finite attention budget, *good* context engineering means finding the *smallest possible* set of high-signal tokens that maximize the likelihood of some desired outcome. Implementing this practice is much easier said than done, but in the following section, we outline what this guiding principle means in practice across the different components of context.

System prompts should be extremely clear and use simple, direct language that presents ideas at the *right altitude* for the agent. The right altitude is the Goldilocks zone between two common failure modes. At one extreme, we see engineers hardcoding complex, brittle logic in their prompts to elicit exact agentic behavior. This approach creates fragility and increases maintenance complexity over time. At the other extreme, engineers sometimes provide vague, high-level guidance that fails to give the LLM concrete signals for

desired outputs or falsely assumes shared context. The optimal altitude strikes a balance: specific enough to guide behavior effectively, yet flexible enough to provide the model with strong heuristics to guide behavior.



At one end of the spectrum, we see brittle if-else hardcoded prompts, and at the other end we see prompts that are overly general or falsely assume shared context.

We recommend organizing prompts into distinct sections (like `<background_information>`, `<instructions>`, `## Tool guidance`, `## Output description`, etc) and using techniques like XML tagging or Markdown headers to delineate these sections, although the exact formatting of prompts is likely becoming less important as models become more capable.

Regardless of how you decide to structure your system prompt, you should be striving for the minimal set of information that fully outlines your expected behavior. (Note that minimal does not necessarily mean short; you still need to give the agent sufficient information up front to ensure it adheres to the desired behavior.) It's best to start by testing a minimal prompt with the best model available to see how it performs on your task, and then add clear instructions and examples to improve performance based on failure modes found during initial testing.

Tools allow agents to operate with their environment and pull in new, additional context as they work. Because tools define the contract between agents and their information/action space, it's extremely important that tools promote efficiency, both by returning information that is token efficient and by encouraging efficient agent behaviors.

In Writing tools for AI agents – with AI agents, we discussed building tools that are well understood by LLMs and have minimal overlap in functionality. Similar to the functions of a well-designed codebase, tools should be self-contained, robust to error, and extremely clear with respect to their intended use. Input parameters should similarly be descriptive, unambiguous, and play to the inherent strengths of the model.

One of the most common failure modes we see is bloated tool sets that cover too much functionality or lead to ambiguous decision points about which tool to use. If a human engineer can't definitively say which tool should be used in a given situation, an AI agent can't be expected to do better. As we'll discuss later, curating a minimal viable set of tools for the agent can also lead to more reliable maintenance and pruning of context over long interactions.

Providing examples, otherwise known as few-shot prompting, is a well known best practice that we continue to strongly advise. However, teams will often stuff a laundry list of edge cases into a prompt in an attempt to articulate every possible rule the LLM should follow for a particular task. We do not recommend this. Instead, we recommend working to curate a set of diverse, canonical examples that effectively portray the expected behavior of the agent. For an LLM, examples are the "pictures" worth a thousand words.

Our overall guidance across the different components of context (system prompts, tools, examples, message history, etc) is to be thoughtful and keep your context informative, yet tight. Now let's dive into dynamically retrieving context at runtime.

Context retrieval and agentic search

In Building effective AI agents, we highlighted the differences between LLM-based workflows and agents. Since we wrote that post, we've gravitated towards a simple definition for agents: LLMs autonomously using tools in a loop.

Working alongside our customers, we've seen the field converging on this simple paradigm. As the underlying models become more capable, the level of autonomy of agents can scale: smarter models allow agents to independently navigate nuanced problem spaces and recover from errors.

We're now seeing a shift in how engineers think about designing context for agents. Today, many AI-native applications employ some form of embedding-based pre-inference time retrieval to surface important context for the agent to reason over. As the field transitions to more agentic approaches, we increasingly see teams augmenting these retrieval systems with "just in time" context strategies.

Rather than pre-processing all relevant data up front, agents built with the "just in time" approach maintain lightweight identifiers (file paths, stored queries, web links, etc.) and use these references to dynamically load data into context at runtime using tools. Anthropic's agentic coding solution Claude Code uses this approach to perform complex data analysis over large databases. The model can write targeted queries, store results, and leverage Bash commands like head and tail to analyze large volumes of data without ever

loading the full data objects into context. This approach mirrors human cognition: we generally don't memorize entire corpuses of information, but rather introduce external organization and indexing systems like file systems, inboxes, and bookmarks to retrieve relevant information on demand.

Beyond storage efficiency, the metadata of these references provides a mechanism to efficiently refine behavior, whether explicitly provided or intuitive. To an agent operating in a file system, the presence of a file named `test_utils.py` in a `tests` folder implies a different purpose than a file with the same name located in `src/core_logic/` Folder hierarchies, naming conventions, and timestamps all provide important signals that help both humans and agents understand how and when to utilize information.

Letting agents navigate and retrieve data autonomously also enables progressive disclosure—in other words, allows agents to incrementally discover relevant context through exploration. Each interaction yields context that informs the next decision: file sizes suggest complexity; naming conventions hint at purpose; timestamps can be a proxy for relevance. Agents can assemble understanding layer by layer, maintaining only what's necessary in working memory and leveraging note-taking strategies for additional persistence. This self-managed context window keeps the agent focused on relevant subsets rather than drowning in exhaustive but potentially irrelevant information.

Of course, there's a trade-off: runtime exploration is slower than retrieving pre-computed data. Not only that, but opinionated and thoughtful engineering is required to ensure that an LLM has the right tools and heuristics for effectively navigating its information landscape. Without proper guidance, an agent can waste context by misusing tools, chasing dead-ends, or failing to identify key information.

In certain settings, the most effective agents might employ a hybrid strategy, retrieving some data up front for speed, and pursuing further autonomous exploration at its discretion. The decision boundary for the 'right' level of autonomy depends on the task. Claude Code is an agent that employs this hybrid model: CLAUDE.md files are naively dropped into context up front, while primitives like glob and grep allow it to navigate its environment and retrieve files just-in-time, effectively bypassing the issues of stale indexing and complex syntax trees.

The hybrid strategy might be better suited for contexts with less dynamic content, such as legal or finance work. As model capabilities improve, agentic design will trend towards letting intelligent models act intelligently, with progressively less human curation. Given the rapid pace of progress in the field, "do the simplest thing that works" will likely remain our best advice for teams building agents on top of Claude.

Context engineering for long-horizon tasks

Long-horizon tasks require agents to maintain coherence, context, and goal-directed behavior over sequences of actions where the token count exceeds the LLM's context window. For tasks that span tens of minutes to multiple hours of continuous work, like large codebase migrations or comprehensive research projects, agents require specialized techniques to work around the context window size limitation.

Waiting for larger context windows might seem like an obvious tactic. But it's likely that for the foreseeable future, context windows of all sizes will be subject to context pollution and information relevance concerns—at least for situations where the strongest agent performance is desired. To enable agents to work effectively across extended time horizons, we've developed a few techniques that address these context pollution constraints directly: compaction, structured note-taking, and multi-agent architectures.

Compaction

Compaction is the practice of taking a conversation nearing the context window limit, summarizing its contents, and reinitiating a new context window with the summary. Compaction typically serves as the first lever in context engineering to drive better long-term coherence. At its core, compaction distills the contents of a context window in a high-fidelity manner, enabling the agent to continue with minimal performance degradation.

In Claude Code, for example, we implement this by passing the message history to the model to summarize and compress the most critical details. The model preserves architectural decisions, unresolved bugs, and implementation details while discarding redundant tool outputs or messages. The agent can then continue with this compressed context plus the five most recently accessed files. Users get continuity without worrying about context window limitations.

The art of compaction lies in the selection of what to keep versus what to discard, as overly aggressive compaction can result in the loss of subtle but critical context whose importance only becomes apparent later. For engineers implementing compaction systems, we recommend carefully tuning your prompt on complex agent traces. Start by maximizing recall to ensure your compaction prompt captures every relevant piece of information from the trace, then iterate to improve precision by eliminating superfluous content.

An example of low-hanging superfluous content is clearing tool calls and results - once a tool has been called deep in the message history, why would the agent need to see the raw result again? One of the safest lightest touch forms of compaction is tool result clearing, most recently launched as a feature on the Claude Developer Platform.

Structured note-taking

Structured note-taking, or agentic memory, is a technique where the agent regularly writes notes persisted to memory outside of the context window. These notes get pulled back into the context window at later times.

This strategy provides persistent memory with minimal overhead. Like Claude Code creating a to-do list, or your custom agent maintaining a NOTES.md file, this simple pattern allows the agent to track progress across complex tasks, maintaining critical context and dependencies that would otherwise be lost across dozens of tool calls.

Claude playing Pokémon demonstrates how memory transforms agent capabilities in non-coding domains. The agent maintains precise tallies across thousands of game steps—tracking objectives like "for the last 1,234 steps I've been training my Pokémon in Route 1, Pikachu has gained 8 levels toward the target of 10." Without any prompting about memory structure, it develops maps of explored regions, remembers which key achievements it has unlocked, and maintains strategic notes of combat strategies that help it learn which attacks work best against different opponents.

After context resets, the agent reads its own notes and continues multi-hour training sequences or dungeon explorations. This coherence across summarization steps enables long-horizon strategies that would be impossible when keeping all the information in the LLM's context window alone.

As part of our Sonnet 4.5 launch, we released a memory tool in public beta on the Claude Developer Platform that makes it easier to store and consult information outside the context window through a file-based system. This allows agents to build up knowledge bases over time, maintain project state across sessions, and reference previous work without keeping everything in context.

Sub-agent architectures

Sub-agent architectures provide another way around context limitations. Rather than one agent attempting to maintain state across an entire project, specialized sub-agents can handle focused tasks with clean context windows. The main agent coordinates with a high-level plan while subagents perform deep technical work or use tools to find relevant information. Each subagent might explore extensively, using tens of thousands of tokens or more, but returns only a condensed, distilled summary of its work (often 1,000-2,000 tokens).

This approach achieves a clear separation of concerns—the detailed search context remains isolated within sub-agents, while the lead agent focuses on synthesizing and analyzing the results. This pattern, discussed in How we built our multi-agent research system, showed a substantial improvement over single-agent systems on complex research tasks.

The choice between these approaches depends on task characteristics. For example:

- Compaction maintains conversational flow for tasks requiring extensive back-and-forth;
- Note-taking excels for iterative development with clear milestones;
- Multi-agent architectures handle complex research and analysis where parallel exploration pays dividends.

Even as models continue to improve, the challenge of maintaining coherence across extended interactions will remain central to building more effective agents.

Conclusion

Context engineering represents a fundamental shift in how we build with LLMs. As models become more capable, the challenge isn't just crafting the perfect prompt—it's thoughtfully curating what information enters the model's limited attention budget at each step. Whether you're implementing compaction for long-horizon tasks, designing token-efficient tools, or enabling agents to explore their environment just-in-time, the guiding principle remains the same: find the smallest set of high-signal tokens that maximize the likelihood of your desired outcome.

The techniques we've outlined will continue evolving as models improve. We're already seeing that smarter models require less prescriptive engineering, allowing agents to operate with more autonomy. But even as capabilities scale, treating context as a precious, finite resource will remain central to building reliable, effective agents.

Get started with context engineering in the Claude Developer Platform today, and access helpful tips and best practices via our memory and context management cookbook.

Acknowledgements

Written by Anthropic's Applied AI team: Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield, with contributions from team members Rafi Ayub, Hannah Moran, Cal Rueb, and Connor Jennings. Special thanks to Molly Vorwerck, Stuart Ritchie, and Maggie Vo for their support.